ВИЛЬЯМС
OSBORNE
McGraw Hill

Протестировано при подготовке последней версии Visual Studio .Net

Полный справочник

по C#

"Герберт Шилдт написал книгу, которую должен иметь каждый, кто программирует на C#."
— Прашант Шридхаран, управляющий выпуском C#-продуктов, Microsoft

БЕСПЛАТНЫЙ КОД В INTERNET

Исчерпывающее описание средств языка C#

Подробное рассмотрение возможностей основных библиотек классов C#

Множество примеров с комментариями

Герберт Шилдт

Автор бестселлеров по программированию: продано более 3 миллионов его книг!

Полный справочник по C#

Герберт Шилдт

Издательский дом "Вильямс"
Москва ♦ Санкт-Петербург ♦ Киев
2004

ББК 32.973.26-018.2.75
Ш57
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *Н.М. Ручко*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Шилдт, Герберт.

Ш57 Полный справочник по C#. : Пер. с англ. — М. : Издательский дом "Вильямс", 2004. — 752 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0563-X (рус.)

В этом полном справочнике по C# — новому языку программирования, разработанному специально для среды .NET, — описаны все основные аспекты языка: типы данных, операторы, управляющие инструкции, классы, интерфейсы, делегаты, индексаторы, события, указатели и директивы препроцессора. Подробно описаны возможности основных библиотек классов C#.

Автор справочника — общепризнанный авторитет в области программирования на языках C и C++, Java и C# — включил в книгу полезные советы и сотни примеров с комментариями, которые удовлетворят как начинающих программистов, так и опытных специалистов. Этот справочник обязан иметь под рукой каждый, кто программирует на C#.

ББК 32.973.26-018.2.75

ISBN 5-8459-0563-X (рус.)
ISBN 0-07-213485-2 (англ.)

Оглавление

Содержание

Об авторе

Герберт Шилдт (Herbert Schildt) — всемирно известный автор книг по программированию и крупный специалист в области таких языков, как C, C++, Java и C#. Продано свыше 3 миллионов экземпляров его книг. Они переведены на множество языков. Шилдт — автор таких бестселлеров, как *Полный справочник по C, Полный справочник по C++, C++: A Beginner's Guide, C++ from the Ground Up, Java 2: A Beginner's Guide* и *Windows 2000 Programming from the Ground Up*. Шилдт — обладатель степени магистра в области вычислительной техники (университет шт. Иллинойс). Телефон его консультационного отдела: (217) 586-4683.

Введение

Программисты — такие люди, которым всегда чего-то не хватает: мы без конца ищем способы повышения быстродействия программ, их эффективности и переносимости. Зачастую мы требуем слишком многого от инструментов, с которыми работаем, особенно, когда это касается языков программирования. Хотя таких языков существует великое множество, но только некоторые из них по-настоящему сильны. Эффективность языка заключается в его мощности и одновременно — в гибкости. Синтаксис языка должен быть лаконичным, но ясным. Он должен способствовать созданию корректного кода и предоставлять реальные возможности, а не ультрамодные (и, как правило, тупиковые) решения. Наконец, мощный язык должен иметь одно нематериальное качество: вызывать ощущение гармонии. Как раз таким языком программирования и является C#.

Созданный компанией Microsoft для поддержки среды .NET Framework, язык C# опирается на богатое наследие в области программирования. Его главным архитектором был ведущий специалист в этой области — Андерс Хейлсберг (Anders Hejlsberg). C# — прямой потомок двух самых успешных в мире компьютерных языков: C и C++. От C он унаследовал синтаксис, ключевые слова и операторы. Он позволяет построить и усовершенствовать объектную модель, определенную в C++. Кроме того, C# близко связан с другим очень успешным языком: Java. Имея общее происхождение, но различаясь во многих важных аспектах, C# и Java — это скорее "двоюродные братья". Например, они оба поддерживают программирование распределенных систем и оба используют промежуточный код для достижения переносимости, но различаются при этом в деталях реализации.

Опираясь на мощный фундамент, который составляют унаследованные характеристики, C# содержит ряд важных новшеств, поднимающих искусство программирования на новую ступень. Например, в состав элементов языка C# включены такие понятия, как делегаты (представители), свойства, индексаторы и события. Добавлен также синтаксис, который поддерживает атрибуты; упрощено создание компонентов за счет исключения проблем, связанных с COM (Component Object Model — модель компонентных объектов Microsoft — стандартный механизм, включающий интерфейсы, с помощью которых объекты предоставляют свои службы другим объектам). И еще. Подобно Java язык C# предлагает средства динамического обнаружения ошибок, обеспечения безопасности и управляемого выполнения программ. Но, в отличие от Java, C# дает программистам доступ к указателям. Таким образом, C# сочетает первозданную мощь C++ с типовой безопасностью Java, которая обеспечивается наличием механизма контроля типов (type checking) и корректным использованием шаблонных классов (template class). Более того, язык C# отличается тем, что компромисс между мощью и надежностью тщательно сбалансирован и практически прозрачен (не заметен для пользователя или программы).

На протяжении всей истории развития вычислительной техники эволюция языков программирования означала изменение вычислительной среды, способа мышления программистов и самого подхода к программированию. Язык C# не является исключением. В непрекращающемся процессе усовершенствования, адаптации и внедрения нововведений C# в настоящее время находится на переднем крае. Это — язык, игнорировать существование которого не может ни один профессиональный программист.

Структура книги

При изложении материала о языке C# труднее всего заставить себя поставить точку. Сам по себе язык C# очень большой, а библиотека классов C# еще больше. Чтобы облегчить читателю овладение таким огромным объемом материала, книга была разделена на три части.

- Часть I, *Язык C#*.
- Часть II, *Библиотека языка C#*.
- Часть III, *Применение языка C#*.

Часть I содержит исчерпывающее описание языка C#. Это самая большая часть книги, в которой описаны ключевые слова, синтаксис и средства программирования, определенные в самом языке, а также организация ввода-вывода данных, обработка файлов и директивы препроцессора.

В части II исследуются возможности библиотеки классов C#. Одной из ее составляющих является библиотека классов среды .NET Framework. Она просто поражает своими размерами. Поскольку ограниченный объем книги не позволяет охватить библиотеку классов среды .NET Framework полностью, в части II акцент делается на корневой библиотеке, относящейся к пространству имен `System`. Именно эта часть библиотеки особым образом связана с C#. Кроме того, здесь описаны коллекции, организация многопоточной обработки и сетевые возможности. Эти разделы библиотеки будет использовать практически каждый, кто программирует на C#.

Часть III содержит примеры применения C#. В главе 24 продемонстрировано создание программных компонентов, а в главе 25 описано создание Windows-приложений с использованием библиотеки Windows Forms. В главе 26 показан процесс разработки программы синтаксического анализа числовых выражений методом рекурсивного спуска (recursive descent parser).

Книга для всех программистов

Для работы с этой книгой опыта в области программирования не требуется. Если же вы знакомы с C++ или Java, то с освоением C# у вас не будет проблем, поскольку у C# много общего с этими языками. Если вы не считаете себя опытным программистом, книга поможет изучить C#, но для этого придется тщательно разобраться в примерах, приведенных в каждой главе.

Программное обеспечение

Чтобы скомпилировать и выполнить программы из этой книги, необходимо установить на своем компьютере пакет Visual Studio .Net 7 (или более позднюю версию), а также оболочку .NET Framework.

Программный код — из Web-пространства

Исходный код всех программ, приведенных в книге, можно загрузить с Web-сайта с адресом: `www.osborne.com`.

Что еще почитать

Книга *Полный справочник по C#* — это "ключ" к серии книг по программированию, написанных Гербертом Шилдтом. Ниже перечислены те из них, которые могут представлять для вас интерес.

Новичкам в программировании на C# стоит обратиться к книге

- *C#: A Beginner's Guide.*

Тем, кто желает подробнее изучить язык C++, будут интересны следующие книги:

- *C++: A Beginner's Guide*
- *Полный справочник по C++*
- *Teach Yourself C++*
- *C++ from the Ground Up*
- *STL* Programming *from the Ground Up*
- The C/C++ Programming Annotated Archives

Тем, кто интересуется программированием на языке Java, мы рекомендуем такие книги:

- *Java 2: A Beginner's Guide*
- *Полный справочник по Java*
- *Java 2: Programmer's Reference*

Если вы интересуетесь языком C, который является фундаментом всех современных языков программирования, обратитесь к книгам

- *Полный справочник по C*
- *Teach Yourself C*

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

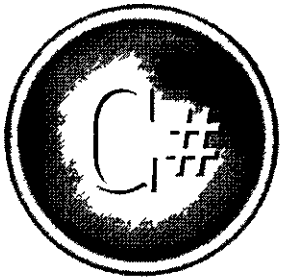
Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail	info@williamspublishing.com
WWW	http://www.williamspublishing.com

Информация для писем из:

России	115419, Москва, а/я 783
Украины	03150, Киев, а/я 152

Полный справочник по

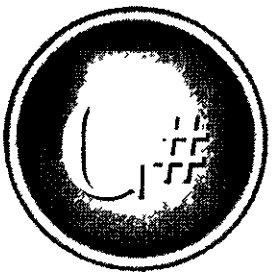


Часть I

Язык C#

В части I описаны элементы языка C#. ключевые слова, синтаксис и операторы. Кроме того, здесь рассмотрены основные инструменты программирования C# (например, способы организации ввода-вывода и средства получения информации о типе), которые тесно связаны с языком C#.

Полный справочник по



Глава 1

Создание языка C#

Язык C# — это очередная ступень бесконечной эволюции языков программирования. Его создание вызвано процессом усовершенствования и адаптации, который определял разработку компьютерных языков в течение последних лет. Подобно всем успешным языкам, которые увидели свет раньше, C# опирается на прошлые достижения постоянно развивающегося искусства программирования.

В языке C# (созданном компанией Microsoft для поддержки среды .NET Framework) проверенные временем средства усовершенствованы с помощью самых современных технологий. C# предоставляет очень удобный и эффективный способ написания программ для современной среды вычислительной обработки данных, которая включает операционную систему Windows, Internet, компоненты и пр. В процессе становления язык C# переопределил весь "ландшафт" программирования.

Назначение этой главы — рассмотреть C# в исторической среде, исследовать мотивы его создания и конструктивные особенности, а также степень влияния на него других языков программирования. Описана связь C# со средой .NET Framework.

Генеалогическое дерево C#

Компьютерные языки существуют не в вакууме. Они связаны друг с другом, причем на каждый новый язык в той или иной форме влияют его предшественники. В процессе такого "перекрестного опыления" средства из одного языка адаптируются другим, удачная новинка интегрируется в существующий контекст, а отжившая конструкция отбрасывается за ненадобностью. Примерно так и происходит эволюция компьютерных языков и развитие искусства программирования. Не избежал подобной участи и C#.

Языку C# "досталось" богатое наследство. Он — прямой потомок двух самых успешных языков программирования (C и C++) и тесно связан с не менее успешным языком Java. Понимание природы этих взаимосвязей крайне важно для понимания C#. Поэтому знакомство с C# мы начнем с рассмотрения исторической среды этих трех языков.

Язык C, или начало современной эпохи программирования

Начало современной эпохи программирования отмечено созданием языка C. Он был разработан Дэнисом Ритчи (Dennis Ritchie) в 1970-х годах для компьютера PDP-11 компании DEC (Digital Equipment Corporation), в котором использовалась операционная система UNIX. Несмотря на то что некоторые известные языки программирования, в особенности Pascal, достигли к тому времени значительного развития и признания, именно язык C определил направление сегодняшнего программирования.

Язык C вырос из кризиса программного обеспечения 1960-х годов и революционного перехода к *структурному программированию*. До структурного программирования многие программисты испытывали трудности при написании больших программ, поскольку обозначилась тенденция вырождения программной логики и появления так называемого "спагетти-кода" (spaghetti code) с большим размером процедур и интенсивным использованием оператора перехода `goto`. Такие программы были весьма трудны для изучения и модификаций. В структурных языках программирования эта проблема решалась посредством добавления точно определенных управляющих конструкций, вызова подпрограмм с локальными переменными и других усовершенствований. Структурные языки позволили писать довольно большие программы в приемлемые сроки.

Хотя в то время уже существовали другие структурные языки, C был первым языком, в котором удачно сочетались мощь, элегантность, гибкость и выразительность. Его лаконичный и к тому же простой в применении синтаксис в совокупности с философией, подразумевающей возложение ответственности на программиста (а не на язык), быстро завоевал множество сторонников. С точки зрения сегодняшнего дня, этот язык, возможно, несколько трудноват для понимания, но программистам того времени он показался порывом свежего ветра, которого они так долго ждали. В результате C стал самым популярным структурным языком программирования 1980-х годов.

Но многоуважаемый язык C имел ограничения. Одним из его недостатков была невозможность справиться с большими программами. Если проект достигал определенного размера, то дальнейшая его поддержка и развитие были связаны с определенными трудностями. Местоположение этой "точки насыщения" зависело от конкретной программы, программиста и используемых им средств, но вероятность ее достижения очень возрастала, когда количество строк в программе приближалось к 5 000.

Создание ООП и C++

К концу 1970-х размер проектов стал приближаться к критическому, при превышении которого методика структурного программирования и язык C "опускали руки". Поэтому стали появляться новые подходы к программированию, позволяющие решить эту проблему. Один из них получил название *объектно-ориентированного программирования* (ООП). Используя ООП, программист мог справляться с программами гораздо большего размера, чем прежде. Но вся беда состояла в том, что C, самый популярный на то время язык, не поддерживал ООП. Желание работать с объектно-ориентированной версией языка C в конце концов и привело к созданию C++.

Язык C++ был разработан Бьярни Страуструпом (Bjarne Stroustrup) в компании Bell Laboratories (Муррей Хил, Нью-Джерси), и годом создания считается 1979-й. Первоначально создатель нового языка назвал его "C с классами", но в 1983 году это имя было изменено на C++. C++ полностью включает элементы языка C. Таким образом, C можно считать фундаментом, на котором построен C++. Большинство дополнений, которые Страуструп внес в C, были предназначены для поддержки объектно-ориентированного программирования. По сути, C++ — это объектно-ориентированная версия языка C. Возводя "здание" C++ на фундаменте C, Страуструп обеспечил плавный переход многих программистов на "рельсы" ООП. Вместо необходимости изучать совершенно новый язык, C-программисту достаточно было освоить лишь новые средства, позволяющие использовать преимущества объектно-ориентированной методики.

На протяжении 1980-х годов C++ интенсивно развивался и к началу 1990-х уже был готов для широкого использования. Рост его популярности носил взрывоподобный характер, и к концу этого десятилетия он стал самым широко используемым языком программирования. В наши дни язык C++ по-прежнему имеет неоспоримое превосходство при разработке высокопроизводительных программ системного уровня.

Важно понимать, что создание C++ не было попыткой изобрести совершенно новый язык программирования. Это было своего рода усовершенствование и без того очень успешного языка. Такой подход к разработке языков (взять за основу существующий язык и поднять его на новую ступень развития) дал начало тенденции, которая продолжает жить и сегодня.

Internet и появление языка Java

Следующей ступенью на лестнице прогресса языков программирования стал язык Java, который первоначально назывался Oak (в переводе с англ. "дуб"). Работа над его созданием началась в 1991 году в компании Sun Microsystems. Основной движущей силой разработки Java был Джеймс Гослинг (James Gosling). В его рабочую группу входили Патрик Нотон (Patrick Naughton), Крис Уортс (Chris Warth), Эд Фрэнк (Ed Frank) и Майк Шеридан (Mike Sheridan).

Java — это структурный объектно-ориентированный язык программирования, синтаксис и основополагающие принципы которого "родом" из C++. Своими новаторскими аспектами Java обязан не столько прогрессу в искусстве программирования (хотя и это имело место), сколько изменениям в компьютерной среде. Еще до наступления эры Internet большинство программ писалось, компилировалось и предназначалось для выполнения с использованием определенного процессора и под управлением конкретной операционной системы. Несмотря на то что программисты всегда старались делать свои программы так, чтобы их можно было применять неоднократно, возможность легко переносить программу из одной среды в другую не была еще достигнута, к тому же проблема переносимости постоянно отодвигалась, решались же более насущные проблемы. Однако с появлением всемирной сети Internet, в которой оказались связанными различные типы процессоров и операционных систем, старая проблема переносимости заявила о себе уже в полный голос. Для ее решения понадобился новый язык программирования, и им стал Java.

Интересно отметить, что, хотя единственным наиболее важным аспектом Java (и причиной быстрого признания) является возможность создавать на нем межплатформенный (совместимый с несколькими операционными средами) переносимый программный код, исходным импульсом для возникновения Java стала не сеть Internet, а настоятельная потребность в не зависящем от платформы языке, который можно было бы использовать в процессе создания программного обеспечения для встроенных контроллеров. В 1993 году стало очевидным, что проблемы межплатформенной переносимости, четко проявившиеся при создании кода для встроенных контроллеров, также оказались весьма актуальными при попытке написать код для Internet. Ведь Internet — это безбрежная компьютерная среда, в которой "обитает" множество компьютеров различных типов. И оказалось, что одни и те же методы решения проблемы переносимости в малых масштабах можно успешно применить и к гораздо большим, т.е. в Internet.

В Java переносимость достигается посредством преобразования исходного кода программы в промежуточный код, именуемый *байт-кодом* (bytecode), т.е. машинно-независимый код, генерируемый Java-компилятором. Байт-код выполняется виртуальной машиной Java (Java Virtual Machine — JVM) — специальной операционной системой. Следовательно, Java-программа могла бы работать в любой среде, где доступна JVM. А поскольку JVM относительно проста для реализации, она быстро стала доступной для большого количества сред.

Использование Java-программами байт-кода радикально отличало их от C- и C++-программ, которые почти всегда компилировались для получения исполняемого машинного кода. Машинный код связан с конкретным процессором и операционной системой. Поэтому, если C/C++-программу нужно выполнить в другой системе, ее необходимо перекомпилировать в машинный код, соответствующий этой среде. Следовательно, чтобы создать C/C++-программу, предназначенную для выполнения в различных средах, необходимо иметь несколько различных исполняемых (машинных) версий этой программы. Это было непрактично и дорого. И наоборот, использование для выполнения Java-программ промежуточного языка было элегантным и рентабельным решением. Именно это решение было адаптировано для языка C#.

Как уже упоминалось, Java — потомок C и C++. Его синтаксис основан на синтаксисе C, а объектная модель — продукт эволюции объектной модели C++. Хотя Java-код несовместим с C или C++ ни снизу вверх, ни сверху вниз, его синтаксис так похож на синтаксис языка C, что толпы C/C++-программистов могли с минимальными усилиями переходить к программированию на Java. Более того, поскольку язык Java строился на существующей парадигме (и усовершенствовал ее), Джеймсу Гослингу ничто не мешало сосредоточить внимание на новых возможностях этого языка. Подобно тому как Страуструпу не нужно было "изобретать колесо" при создании C++, так и Гослингу при разработке Java не было необходимости создавать совершенно новый язык программирования. Более того, создание Java показало, что языки C и C++ — прекрасный "субстрат" для "выращивания" новых компьютерных языков.

Создание C#

Разработчики Java успешно решили многие проблемы, связанные с переносимостью в среде Internet, но далеко не все. Одна из них — *межъязыковая возможность взаимодействия* (cross-language interoperability) программных и аппаратных изделий разных поставщиков, или *многоязыковое программирование* (mixed-language programming). В случае решения этой проблемы программы, написанные на разных языках, могли бы успешно работать одна с другой. Такое взаимодействие необходимо для создания больших систем с распределенным программным обеспечением (ПО), а также для программирования компонентов ПО, поскольку самым ценным является компонент, который можно использовать в широком диапазоне компьютерных языков и операционных сред.

Кроме того, в Java не достигнута полная интеграция с платформой Windows. Хотя Java-программы могут выполняться в среде Windows (при условии установки виртуальной машины Java), Java и Windows не являются прочно связанными средами. А поскольку Windows — это наиболее широко используемая операционная система в мире, то отсутствие прямой поддержки Windows — серьезный недостаток Java.

Чтобы удовлетворить эти потребности, Microsoft разработала язык C#. C# был создан в конце 1990-х годов и стал частью общей .NET-стратегии Microsoft. Впервые он увидел свет в качестве α-версии в середине 2000 года. Главным архитектором C# был Андерс Хейлсберг (Anders Hejlsberg) — один из ведущих специалистов в области языков программирования, получивший признание во всем мире. Достаточно сказать, что в 1980-х он был автором весьма успешного продукта Turbo Pascal, изящная реализация которого установила стандарт для всех будущих компиляторов.

C# непосредственно связан с C, C++ и Java. И это не случайно. Эти три языка — самые популярные и самые любимые языки программирования в мире. Более того, почти все профессиональные программисты сегодня знают C и C++, и большинство знает Java. Поскольку C# построен на прочном, понятном фундаменте, то переход от этих "фундаментальных" языков к "надстройке" происходит без особых усилий со стороны программистов. Так как Андерс Хейлсберг не собирался изобретать свое "колесо", он сосредоточился на введении усовершенствований и новшеств.

Генеалогическое дерево C# показано на рис. 1.1. "Дедушкой" C# является язык C. От C язык C# унаследовал синтаксис, многие ключевые слова и операторы. Кроме того, C# построен на улучшенной объектной модели, определенной в C++. Если вы знаете C или C++, то с C# вы сразу станете друзьями.

C# и Java связаны между собой несколько сложнее. Как упоминалось выше, Java также является потомком C и C++. У него тоже общий с ними синтаксис и сходная объектная модель. Подобно Java C# предназначен для создания переносимого кода. Однако C# — не потомок Java. Скорее C# и Java можно считать двоюродными братьями, имеющими общих предков, но получившими от родителей разные наборы

"генов". Если вы знаете язык Java, то вам будут знакомы многие понятия C#. И наоборот, если в будущем вам придется изучать Java, то, познакомившись с C#, вам не придется осваивать многие средства Java.

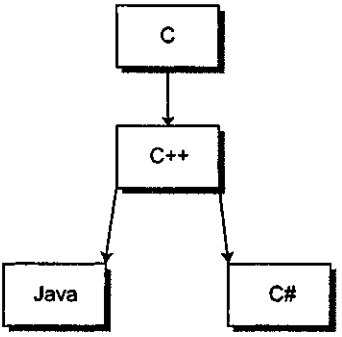


Рис. 1.1. Генеалогическое дерево C#

C# содержит множество новых средств, которые описаны в этой книге. Самые важные из них связаны со встроенной поддержкой программных компонентов. Именно наличие встроенных средств написания программных компонентов и позволило C# называться *компонентно-ориентированным языком*. Например, C# включает средства, которые напрямую поддерживают составные части компонентов: свойства, методы и события. Все же самым важным качеством компонентно-ориентированного языка является его способность работать в среде многоязыкового программирования.

Связь C# с оболочкой .NET Framework

Несмотря на то что C# — самодостаточный компьютерный язык, у него особые взаимоотношения со средой .NET Framework. И на это есть две причины. Во-первых, C# изначально разработан компанией Microsoft для создания кода, выполняющегося в среде .NET Framework. Во-вторых, в этой среде определены библиотеки, используемые языком C#. И хотя можно отделить C# от .NET Framework, эти две среды тесно связаны, поэтому очень важно иметь общее представление о .NET Framework и понимать, почему эта среда столь важна для C#.

О среде .NET Framework

Оболочка .NET Framework определяет среду для разработки и выполнения сильно распределенных приложений, основанных на использовании компонентных объектов. Она позволяет "мирно сосуществовать" различным языкам программирования и обеспечивает безопасность, переносимость программ и общую модель программирования для платформы Windows. Важно при этом понимать, что .NET Framework по своему существу не ограничена применением в Windows, т.е. программы, написанные для нее, можно затем переносить в среды, отличные от Windows.

Связь среды .NET Framework с C# обусловлена наличием двух очень важных средств. Одно из них, *Common Language Runtime* (CLR), представляет собой систему, которая управляет выполнением пользовательских программ. CLR — это составная часть .NET Framework, которая делает программы переносимыми, поддерживает многоязыковое программирование и обеспечивает безопасность.

Второе средство, *библиотека классов* .NET-оболочки, предоставляет программам доступ к среде выполнения. Например, если вам нужно выполнить операцию ввода-вывода, скажем, отобразить что-либо на экране, то для этого необходимо использовать .NET-библиотеку классов. Если вы — новичок в программировании, термин *класс* вам может быть незнаком. Ниже вы найдете подробное объяснение этого понятия, а пока ограничимся кратким его определением: класс — это объектно-ориентированная конструкция, с помощью которой организуются программы. Если программа ограничивается использованием средств, определенных .NET-библиотекой классов, она может выполняться везде (т.е. в любой среде), где поддерживается .NET-система. Поскольку C# автоматически использует .NET-библиотеку классов, C#-программы автоматически переносимы во все .NET-среды.

Функционирование системы CLR

Система CLR управляет выполнением .NET-кода. Вот как это происходит. В результате компиляции C#-программы получается не исполняемый код, а файл, который содержит специальный псевдокод, именуемый *промежуточным языком Microsoft* (*Microsoft Intermediate Language* — MSIL). MSIL определяет набор переносимых инструкций, которые не зависят от типа процессора. По сути, MSIL определяет переносимость ассемблера. И хотя концептуально MSIL подобен байт-коду Java, это не одно и то же.

Цель CLR-системы — при выполнении программы перевести ее промежуточный код в исполняемый. Таким образом, программа, подвергнутая MSIL-компиляции, может быть выполнена в любой среде, для которой реализована CLR-система. В этом частично и состоит способность среды .NET Framework добиваться переносимости программ.

Код, написанный на промежуточном языке Microsoft, переводится в исполняемый с помощью *JIT-компилятора*. "JIT" — сокр. от выражения "just-in-time", означающего выполнение точно к нужному моменту (так обозначается стратегия принятия решений в самый последний подходящий для этого момент в целях обеспечения их максимальной точности). Этот процесс работает следующим образом. При выполнении .NET-программы CLR-система активизирует JIT-компилятор, который преобразует MSIL-код в ее "родной" код на требуемой основе, поскольку необходимо сохранить каждую часть программы. Таким образом, C#-программа в действительности выполняется в виде "родного" кода, несмотря на то, что первоначально она была скомпилирована в MSIL-код. Это значит, что программа будет выполнена практически так же быстро, как если бы она с самого начала была скомпилирована с получением "родного" кода, но с "добавлением" преимуществ переносимости от преобразования в MSIL-код.

В результате компиляции C#-программы помимо MSIL-кода образуются и *метаданные* (metadata). Они описывают данные, используемые программой, и позволяют коду взаимодействовать с другим кодом. Метаданные содержатся в том же файле, где хранится MSIL-код.

Сравнение управляемого кода с неуправляемым

В общем случае при написании C#-программы создается код, называемый *управляемым* (managed code). Управляемый код выполняется под управлением CLR-системы. У такого выполнения в результате есть как определенные ограничения, так и немалые достоинства. К числу ограничений относится необходимость иметь, во-

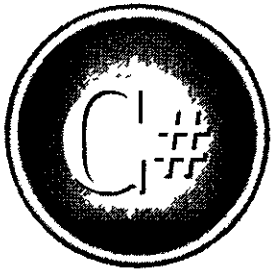
первых, специальный компилятор, который должен создавать MSIL-файл, предназначенный для работы под управлением CLR-системы, и, во-вторых, этот компилятор должен использовать библиотеки среды .NET Framework. Достоинства же управляемого кода — современные методы управления памятью, возможность использовать различные языки, улучшенная безопасность, поддержка управления версиями и четкая организация взаимодействия программных компонентов.

Что же понимается под неуправляемым кодом? Все Windows-программы до создания среды .NET Framework использовали неуправляемый код, который не выполняется CLR-системой. Управляемый и неуправляемый код могут работать вместе, поэтому факт создания C#-компилятором управляемого кода отнюдь не ограничивает его возможность выполняться совместно с ранее созданными программами.

Спецификация универсального языка

Несмотря на то что управляемый код обладает достоинствами, предоставляемыми CLR-системой, но если он используется другими программами, написанными на иных языках, то для достижения максимального удобства и простоты использования он должен соответствовать *спецификации универсального языка* (Common Language Specification — CLS). Эта спецификация описывает набор свойств, которыми одновременно должны обладать различные языки. Соответствие CLS-спецификации особенно важно при создании программных компонентов, которые предназначены для использования программами, написанными на других языках. CLS-спецификация включает подмножество *системы поддержки общих типов* (Common Type System — CTS). CTS-система определяет правила в отношении типов данных. Безусловно, C# поддерживает как CLS-, так и CTS-спецификации.

Полный справочник по



Глава 2

Обзор элементов языка C#

Самым трудным в изучении языка программирования, безусловно, является то, что ни один его элемент не существует изолированно от других. Компоненты языка работают вместе, можно сказать, в дружном "коллективе". Такая тесная взаимосвязь усложняет рассмотрение одного аспекта C# без рассмотрения других. Зачастую обсуждение одного средства предусматривает предварительное знакомство с другим. Для преодоления подобных трудностей в этой главе приводится краткое описание таких элементов C#, как общая форма C#-программы, основные инструкции управления и операторы. При этом мы не будем углубляться в детали, а сосредоточимся на общих концепциях создания C#-программы. Большинство затронутых здесь тем более подробно рассматриваются в остальных главах части I.

Объектно-ориентированное программирование

Главное в языке C# — реализация принципов *объектно-ориентированного программирования* (ООП). Объектно-ориентированная методика неотделима от C#, и все C#-программы в какой-то степени имеют объектную ориентацию. Поэтому, прежде чем приступать к написанию даже простой C#-программы, необходимо понять основные принципы ООП.

ООП — это мощный "рычаг", позволяющий усовершенствовать процесс программирования. С момента изобретения компьютера методы программирования менялись много раз и причем коренным образом, но в основном, с целью адаптации к непрерывному повышению сложности программ. Например, программирование для первых компьютеров осуществлялось посредством набора машинных инструкций (в двоичном коде) на передней панели компьютера. Этот метод работал до тех пор, пока длина программы не превышала нескольких сот инструкций. С ростом программ был изобретен язык ассемблер, который позволил программисту писать гораздо большие и более сложные программы, используя символическое представление машинных инструкций. По мере роста программ появились языки высокого уровня (например, FORTRAN и COBOL), которые позволили программистам справляться с возрастающей сложностью программ. Когда эти первые компьютерные языки начали приближаться к критическому состоянию, было изобретено структурное программирование.

Каждая веха в развитии программирования характеризовалась созданием методов и средств, позволяющих программисту писать все более сложные программы. С каждым шагом на этом пути изобретался новый метод, который, опираясь на самые удачные элементы предыдущих методов, вносил что-то свое, осуществляя таким образом прогресс в области программирования в целом. Примерно по такой схеме развития инструментария для программистов "дошло дело" и до объектно-ориентированного программирования. Его появлению способствовал тот факт, что реализация многих проектов начала серьезно стопориться, поскольку структурный подход уже не справлялся с поставленными задачами. Нужен был новый способ преодоления сложности программ, и решением этой проблемы стало объектно-ориентированное программирование.

Объектно-ориентированное программирование вобрало в себя лучшие идеи структурного программирования и объединило их с новыми концепциями. В результате появился более совершенный способ организации программы. Если говорить в самых общих чертах, программу можно организовать одним из двух способов: опираясь либо на код (т.е. на действия, или на то, что происходит в программе), либо на данные (т.е. на то, что подвергается определенному воздействию). При использовании исключительно методов структурного программирования программы обычно организовывались с опорой на действия. Такой подход можно представить себе в виде кода, воздействующего на данные.

Объектно-ориентированные программы работают совсем по-другому. Они организованы вокруг данных, а ключевой принцип такой организации гласит: именно данные должны управлять доступом к коду. В объектно-ориентированном языке программист определяет данные и код, который разрешен для выполнения действий над этими данными. Таким образом, тип данных точно определяет операции, которые могут быть к ним применены.

Для поддержки принципов объектно-ориентированного программирования все ООП-языки, включая C#, имеют три характерных черты: инкапсуляцию, полиморфизм и наследование.

Инкапсуляция

Инкапсуляция — это механизм программирования, который связывает код (действия) и данные, которыми он манипулирует, и при этом предохраняет их от вмешательства извне и неправильного использования. В объектно-ориентированном языке код и данные можно связать таким образом, что будет создан автономный *черный ящик*. Внутри этого ящика находятся все необходимые данные и код. При таком связывании кода и данных создается *объект*. Другими словами, объект — это элемент, который поддерживает инкапсуляцию.

Код, данные или обе эти составляющие объекта могут быть *закрытыми* внутри него или *открытыми*. Закрытый код или закрытые данные известны лишь остальной части этого объекта и доступны только ей. Это означает, что к закрытому коду или данным не может получить доступ никакая другая часть программы, существующая вне этого объекта. Если код или данные являются открытыми, к ним (несмотря на то, что они определены внутри объекта) могут получить доступ другие части программы. Как правило, открытые части объекта используются для обеспечения управляемого интерфейса с закрытыми элементами.

Основной единицей инкапсуляции в C# является *класс*. Класс определяет форму объекта. Он задает как данные, так и код, который будет оперировать этими данными. В C# класс используется для создания объектов. Объекты — это экземпляры класса. Таким образом, класс — это по сути набор шаблонных элементов, которые показывают, как построить объект.

Код и данные, которые составляют класс, называются *членами* класса. Данные, определенные в классе, называются *переменными экземпляра* (instance variable), а код, который оперирует этими данными, — *методами-членами* (member method), или просто *методами*. "Метод" — это термин, применяемый в C# для обозначения подпрограммы. Если вы знакомы с языками C или C++, то, вероятно, догадываетесь о том, что то, что C#-программист называет методом, C/C++-программист назовет *функцией*. А поскольку C# — прямой потомок C++, термин "функция" также приемлемо использовать, когда речь идет о C#-методе.

Полиморфизм

Полиморфизм (от греческого слова *polymorphism*, означающего "много форм") — это качество, которое позволяет одному интерфейсу получать доступ к целому классу действий. Простым примером полиморфизма может послужить руль автомобиля. Руль (интерфейс) остается рулем независимо от того, какой тип рулевого механизма используется в автомобиле. Другими словами, руль работает одинаково в любом случае: оснащен ли ваш автомобиль рулевым управлением прямого действия, рулевым управлением с усилителем или реечным управлением. Таким образом, поворот руля влево заставит автомобиль поехать влево независимо от типа используемого в нем рулевого управления. Достоинство такого *единообразного интерфейса* состоит, безусловно, в том, что, если вы знаете, как обращаться с рулем, вы сможете водить автомобиль любого типа.

Тот же принцип можно применить и к программированию. Рассмотрим, например, *стек* (stack), т.е. область памяти, функционирующую по принципу "последним пришел — первым обслужен". Предположим, вы пишете программу, для которой нужно организовать три различных типа стека. Один стек предназначен для целочисленных значений, второй — для значений с плавающей точкой, а третий — для символов. В этом случае для реализации каждого стека используется один и тот же алгоритм, несмотря на различие в типах сохраняемых данных. В случае не объектно-ориентированного языка вам пришлось бы создать три набора "стековых" подпрограмм, имеющих различные имена. Но благодаря полиморфизму в среде C# можно создать один общий набор "стековых" подпрограмм, который обрабатывает все три типа стека. Иными словами, зная, как использовать один стек, можно использовать все остальные.

Концепцию полиморфизма часто выражают такой фразой: "один интерфейс — много методов". Это означает, что для выполнения группы подобных действий можно разработать общий интерфейс. Полиморфизм позволяет понизить степень сложности программы, предоставляя программисту возможность использовать один и тот же интерфейс для задания *общего класса действий*. Конкретное (нужное в том или ином случае) действие (метод) выбирается компилятором. Программисту нет необходимости делать это вручную. Его задача — правильно использовать общий интерфейс.

Наследование

Наследование — это процесс, благодаря которому один объект может приобретать свойства другого. Благодаря наследованию поддерживается концепция иерархической классификации. В виде управляемой иерархической (нисходящей) классификации организуется большинство областей знаний. Например, яблоки *Красный Делишес* являются частью классификации *яблоки*, которая в свою очередь является частью класса *фрукты*, а тот — частью еще большего класса *пища*. Таким образом, класс *пища* обладает определенными качествами (съедобность, питательность и пр.), которые применимы и к подклассу *фрукты*. Помимо этих качеств, класс *фрукты* имеет специфические характеристики (сочность, сладость и пр.), которые отличают их от других пищевых продуктов. В классе *яблоки* определяются качества, специфичные для яблок (растут на деревьях, не тропические и пр.). Класс *Красный Делишес* наследует качества всех предыдущих классов и при этом определяет качества, которые являются уникальными для этого сорта яблок.

Если не использовать иерархическое представление признаков, для каждого объекта пришлось бы в явной форме определить все присущие ему характеристики. Но благодаря наследованию объекту нужно доопределить только те качества, которые делают его уникальным внутри его класса, поскольку он (объект) наследует общие атрибуты своего родителя. Следовательно, именно механизм наследования позволяет одному объекту представлять конкретный экземпляр более общего класса.

Первая простая программа

Настало время рассмотреть реальную C#-программу.

```
/*
   Это простая C#-программа.

   Назовем ее Example.cs.
*/
```

```
using System;

class Example {

  // Любая C#-программа начинается с вызова метода Main().
  public static void Main() {
    Console.WriteLine("Простая C#-программа.");
  }
}
```

На момент написания этой книги единственной доступной средой разработки C#-программ была Visual Studio .NET. Эта среда позволяет отредактировать, скомпилировать и выполнить C#-программу, причем это можно сделать двумя способами: с помощью компилятора командной строки `csc.exe` или интегрированной среды разработки (Integrated Development Environment — IDE). Здесь описаны оба способа. (Если вы используете иной компилятор, следуйте инструкциям, приведенным в сопроводительной документации.)

Использование компилятора командной строки `csc.exe`

Несмотря на то что в случае коммерческих проектов вы, по всей вероятности, будете работать в интегрированной среде разработки Visual Studio, использование C#-компилятора командной строки — самый простой способ скомпилировать и выполнить примеры программ, приведенные в этой книге. Для создания и запуска программ с помощью C#-компилятора командной строки необходимо выполнить следующие действия.

1. Ввести текст программы, используя любой текстовый редактор.
2. Скомпилировать программу.
3. Выполнить программу.

Ввод текста программы

Программы, представленные в этой книге, можно загрузить с Web-сайта компании Osborne с адресом: `www.osborne.com`. Но при желании вы можете ввести текст программ вручную. В этом случае необходимо использовать какой-нибудь текстовый редактор, например Notepad. Однако помните, что при вводе текста программ должны быть созданы исключительно текстовые файлы, а не файлы с форматированием, используемым при текстовой обработке, поскольку информация о форматировании помешает работе C#-компилятора. Введя текст приведенной выше программы, назовите соответствующий файл `Example.cs`.

Компилирование программы

Чтобы скомпилировать программу, запустите C#-компилятор, `csc.exe`, указав в командной строке имя исходного файла.

```
C:\>csc Example.cs
```

Компилятор `csc` создаст файл с именем `Example.exe`, который будет содержать MSIL-версию этой программы. Хотя MSIL-код не является выполняемым, тем не менее он содержится в `exe`-файле. При попытке выполнить файл `Example.exe` система Common Language Runtime автоматически вызовет JIT-компилятор. Однако имейте в виду: если вы попытаетесь выполнить `Example.exe` (или любой другой `exe`-файл, содержащий MSIL-код) на компьютере, в котором не установлена среда .NET Framework, программа выполнена не будет ввиду отсутствия системы CLR.

На заметку

Перед запуском компилятора `csc.exe` вам, возможно, придется выполнить пакетный файл `vcvars32.bat`, который обычно расположен в папке `//Program Files/Microsoft Visual Studio .NET/Vc7/Bin`. В качестве альтернативного варианта можно перейти в режим работы по приглашению на ввод команды. Для C# этот режим инициализируется выбором команды Microsoft Visual Studio .NET⇨Visual Studio .NET Command Prompt⇨Visual Studio .NET Tools из меню Пуск⇨Программы, активизируемом на панели задач.

Выполнение программы

Для выполнения программы достаточно ввести ее имя в командную строку.

```
C:\>Example
```

При выполнении этой программы на экране отобразится следующая информация:

```
Простая C#-программа.
```

Использование Visual Studio IDE

Теперь можно обратиться к версии 7 пакета Visual Studio .NET, поскольку Visual Studio IDE позволяет компилировать C#-программы. Чтобы отредактировать, скомпилировать и выполнить C#-программу с помощью интегрированной среды разработки пакета Visual Studio (версия 7), выполните следующие действия. (Если вы работаете с другой версией пакета Visual Studio, возможно, придется следовать другим инструкциям.)

1. Создайте новый (пустой) C#-проект, выполнив команду File⇨New⇨Project (Файл⇨Создать⇨Проект).
2. Среди представленных типов проектов (на панели Project Types (Типы проектов)) выберите вариант Visual C# Projects (Проекты Visual C#), а затем (как показано на рис. 2.1) на панели Templates (Шаблоны) — шаблон Empty Project (Пустой проект).

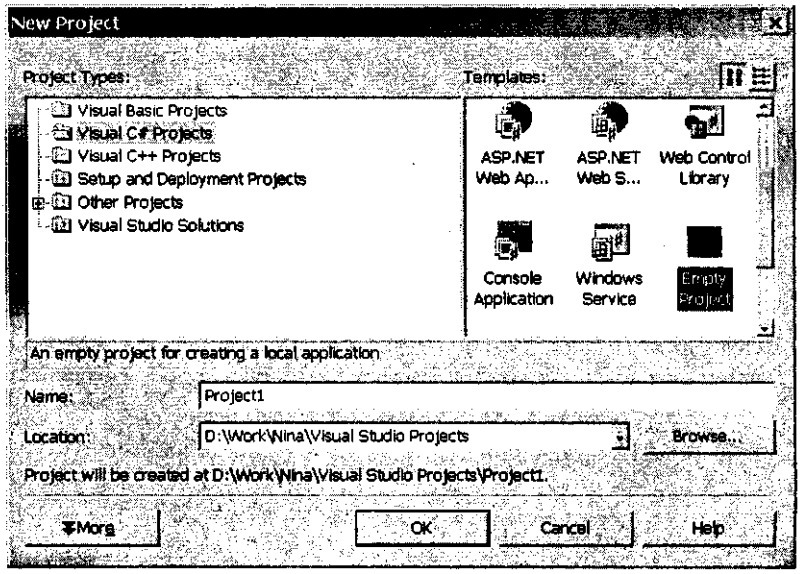


Рис. 2.1. Окно New Project (Создать проект)

3. Создав проект, щелкните правой кнопкой мыши на имени проекта в окне Solution Explorer (Проводник решений) Затем, используя всплывающее контекстное меню, выберите команду Add⇨Add New Item (Добавить⇨Добавить новый элемент) При этом экран должен выглядеть, как показано на рис 2 2
4. В открывшемся диалоговом окне Add New Item (Добавить новый элемент) на панели Categories (Категории) выберите вариант Local Project Items (Элементы локальных проектов), а на панели Templates — шаблон Code File (Файл с текстом программы) При этом экран должен выглядеть, как показано на рис 2 3

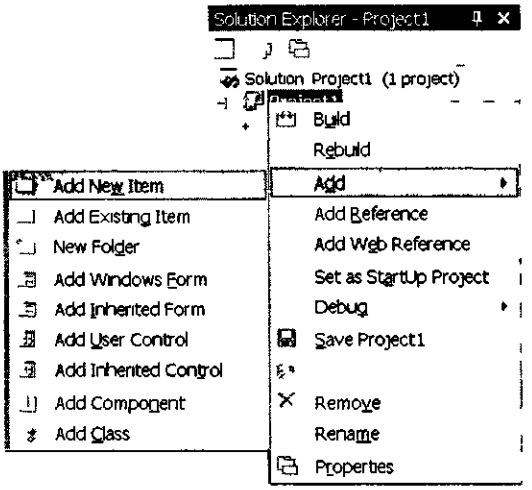


Рис 2 2 Контекстные меню при выборе команды Add⇨Add New Item

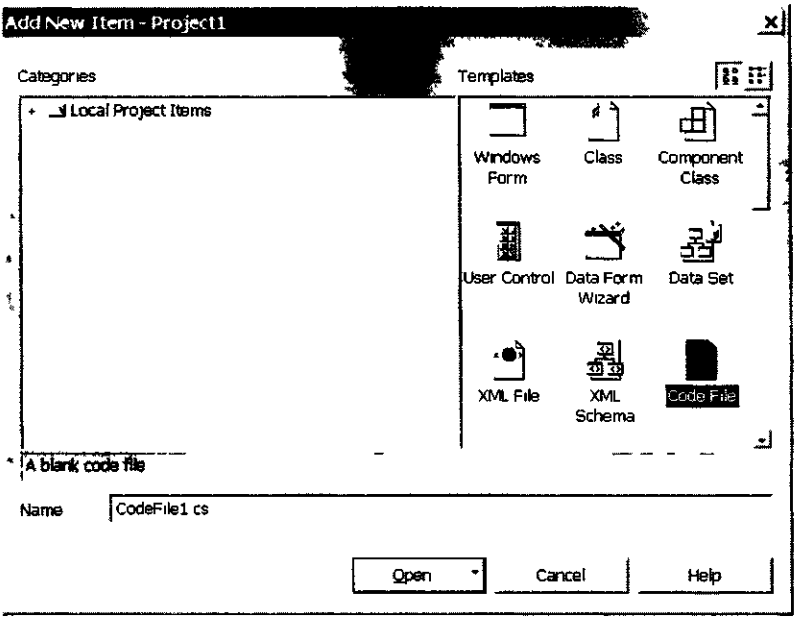


Рис 2 3 Диалоговое окно Add New Item

5. Введите текст программы и сохраните файл под именем `Example.cs`. (Помните, что программы, представленные в этой книге, можно загрузить с Web-сайта компании Osborne с адресом: `www.osborne.com`.) После этого экран должен выглядеть, как показано на рис. 2.4.

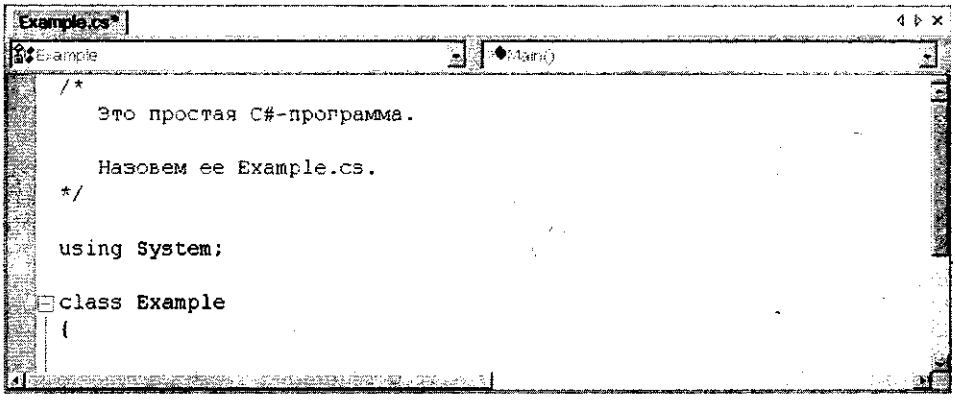


Рис. 2.4. Окно проекта `Example.cs`

6. Скомпилируйте программу с помощью команды Build⇨Build Solution (Построить⇨Построить решение).
7. Выполните программу с помощью команды Debug⇨Start Without Debugging (Отладка⇨Начать выполнение без отладки).

После выполнения этой программы вы должны увидеть окно, показанное на рис. 2.5.

Рис. 2.5. Диалоговое окно Add New Item

На заметку

Чтобы скомпилировать и выполнить примеры программ, представленные в этой книге, нет необходимости для каждой программы создавать новый проект. Можно использовать тот же самый C#-проект. Просто удалите текущий файл и добавьте новый. Затем перекомпилируйте его и выполните.

Как уже упоминалось, короткие программы (из первой части книги) проще компилировать и выполнять, используя компилятор командной строки, но окончательный выбор, конечно, за вами.

"Разбор полетов", или первый пример программы "под микроскопом"

Несмотря на миниатюрные размеры, программа `Example.cs` включает ряд ключевых средств, которые применимы ко всем C#-программам. Поэтому имеет смысл подробно рассмотреть каждую часть программы, начиная с имени.

В отличие от некоторых языков программирования (например, Java), в которых имя программного файла имеет очень большое значение, C#-программа может иметь любое имя. С таким же успехом вы могли бы назвать первую в этой книге программу не `Example.cs`, а, скажем, `Sample.cs`, `Test.cs` или даже `X.cs`.

По соглашению для исходных файлов C#-программ используется расширение `.cs`, и этому соглашению вы должны следовать безоговорочно. Многие программисты называют файл программы по имени основного класса, определенного в этом файле. Поэтому (как вы, вероятно, догадались) и было выбрано имя `Example.cs`. Поскольку имена C#-программ могут быть произвольными, для большинства примеров программ в этой книге имена вообще не указаны. Это значит, что вы можете называть их по своему вкусу.

Первая наша программа начинается со следующих строк.

```
/*
   Это простая C#-программа.

   Назовем ее Example.cs.
*/
```

Эти строки образуют *комментарий*. Подобно большинству других языков программирования C# позволяет вводить в исходный файл программы комментарии, содержимое которых компилятор игнорирует. С помощью комментариев описываются или разъясняются действия, выполняемые в программе, и эти разъяснения предназначаются для тех, кто будет читать исходный код. В данном случае в комментарии дается общая характеристика программы и предлагается назвать этот исходный файл именем `Example.cs`. Конечно, в реальных приложениях комментарии используются для разъяснения особенностей работы отдельных частей программы или конкретных действий программных средств.

В C# поддерживается три стиля комментариев. Первый, показанный в начале рассматриваемой программы, называется *многострочным*. Комментарий этого типа должен начинаться символами `/*` и заканчиваться ими же, но в обратном порядке (`*/`). Все, что находится между этими парами символов, компилятор игнорирует. Комментарий этого типа, как следует из его названия, может занимать несколько строк.

Рассмотрим следующую строку программы.

```
using System;
```

Эта строка означает, что программа использует пространство имен `System`. В C# *пространство имен* (namespace) определяет декларативную область. Подробнее о пространствах имен мы поговорим позже, а пока ограничимся тем, что заявленное пространство имен позволяет хранить одно множество имен отдельно от другого. Другими словами, имена, объявленные в одном пространстве имен, не будут конфликтовать с такими же именами, объявленными в другом. В нашей программе используется пространство имен `System`, которое зарезервировано для элементов, связанных с библиотекой классов среды .NET Framework, используемой языком C#. Ключевое слово

using — это своего рода заявление о том, что программа использует имена в заданном пространстве имен.

Перейдем к следующей строке программы.

```
class Example {
```

В этой строке используется ключевое слово class, которое объявляет об определении нового класса. Как упоминалось выше, в C# класс — это базовая единица инкапсуляции. Example — имя определяемого класса. Определение класса заключено между открывающей ({) и закрывающей (}) фигурными скобками. Таким образом, элементы, расположенные между этими двумя фигурными скобками, являются членами класса. Пока мы не будем углубляться в детали определения класса, но отметим, что в C# работа программы протекает именно внутри класса. Это одна из причин, по которой все C#-программы являются объектно-ориентированными.

Очередная строка в нашей программе представляет собой *однострочный комментарий.*

```
// Любая C#-программа начинается с вызова метода Main().
```

Так выглядит комментарий второго типа, поддерживаемый в C#. Однострочный комментарий начинается с пары символов // и заканчивается в конце строки. Как правило, программисты используют многострочные комментарии для подробных и потому более пространных разъяснений, а однострочные — для кратких (построчных) описаний происходящего в программе.

Следующей строке стоит уделить особое внимание.

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

В этой строке начинается определение метода Main(). Как упоминалось выше, в C# подпрограмма называется *методом* (method). Предшествующий этой строке однострочный комментарий подчеркивает, что именно с этой строки и будет начато выполнение программы. Все C#-приложения начинают выполняться с вызова метода Main(). (Для сравнения: выполнение C/C++-программ начинается с обращения к функции main().) В полном описании каждой части этой строки сейчас большого смысла нет, поскольку это требует глубокого понимания других средств языка C#. Но так как эта строка кода включена во многие примеры программ этой книги, придется кратко на ней остановиться.

Ключевое слово public представляет собой *спецификатор доступа* (access specifier). Спецификатор доступа определяет, *как* другие части программы могут получать доступ к члену класса. Если объявление члена класса включает ключевое слово public, значит, к этому члену можно получить доступ с помощью кода, расположенного вне класса, в котором этот член объявлен. (Противоположным по значению ключевому слову public является слово private, которое не допускает использования соответствующего члена класса кодом, определенным вне его (члена) класса.) В данном случае метод Main() объявляется как public-метод, поскольку при запуске этой программы он будет вызываться внешним (по отношению к его классу) кодом (а именно операционной системой).

На заметку

На момент написания книги в документации на язык C# не было обозначено требования, чтобы Main() объявлялся как public-метод. Однако именно таким образом оформлены подобные объявления в примерах, включенных в описание пакета Visual Studio .NET. Этот способ объявления предпочитают использовать многие C#-программисты. Поэтому и в настоящей книге метод Main() объявляется с использованием спецификатора public. Но вы не должны удивляться, встретив несколько иной способ объявления метода Main().

Ключевое слово static позволяет реализовать вызов метода Main() еще до создания объекта соответствующего класса. Это — очень важный момент, поскольку метод

Main() вызывается при запуске программы. Ключевое слово void просто сообщает компилятору о том, что метод Main() не возвращает значения. Как будет показано выше, методы могут возвращать значения. Пустые круглые скобки после имени Main() говорят о том, что методу не передается никакой информации. Как будет показано выше, методу Main() (или любому другому) можно передавать данные, которые будут им обрабатываться. Рассматриваемую строку венчает символ открывающей фигурной скобки ({), который служит признаком начала тела метода Main(). Между открывающей и закрывающей фигурными скобками и должен находиться весь код, составляющий тело метода.

Рассмотрим следующую строку программы. Обратите внимание на то, что она принадлежит телу метода Main().

```
Console.WriteLine("Простая C#-программа.");
```

Здесь реализован вывод на экран текстовой строки "Простая C#-программа." и следующего за ней символа новой строки. Сам вывод осуществляется встроенным методом WriteLine(). В данном случае на экране будет отображена строка, переданная методу. Передаваемая методу информация называется *аргументом*. Помимо текстовых строк, метод WriteLine() может отображать и данные других типов. Console — это имя встроенного класса, который поддерживает консольные операции ввода-вывода данных. Связав класс Console с методом WriteLine(), вы тем самым сообщаете компилятору, что WriteLine() — член класса Console. Тот факт, что в C# для определения консольного вывода данных используется некоторый объект, является еще одним свидетельством объектно-ориентированной природы этого языка программирования.

Обратите внимание на то, что инструкция, содержащая вызов метода WriteLine(), завершается точкой с запятой, как и рассмотренная выше инструкция программы using System. В C# точкой с запятой завершаются все инструкции. Если же вы встречаете строки программы, которые не оканчиваются точкой с запятой, значит, они попросту не являются инструкциями.

Первая в программе закрывающая фигурная скобка (}) завершает метод Main(), а вторая — определение класса Example.

И еще. В языке C# различаются прописные и строчные буквы. Игнорирование этого факта может вызвать серьезные проблемы. Например, если случайно вместо имени Main ввести имя main или вместо WriteLine ввести writeline, то рассмотренная ниже программа сразу же станет некорректной. Более того, хотя C#-компилятор компилирует классы, которые не содержат метода Main(), у него нет возможности выполнить их. Поэтому, даже если вы введете имя Main с опечаткой (main), компилятор все равно скомпилирует программу. Но затем вы получите сообщение об ошибке, уведомляющее о том, что в файле Example.exe не определена точка входа.

Обработка синтаксических ошибок

Введите, скомпилируйте и выполните рассмотренную выше программу (если вы еще не сделали этого). При вводе в компьютер текста программы вручную очень легко сделать случайную опечатку. К счастью, при попытке скомпилировать некорректно введенную программу компилятор сообщит о наличии в ней *синтаксической ошибки (ошибок)*. При этом C#-компилятор попытается найти какой-либо смысл в предоставленном ему исходном коде независимо от того, что вы ему "подсунули". Поэтому ошибка, о которой "просигналил" компилятор, может не всегда отражать истинную причину проблемы. Например, если в предыдущей программе случайно опустить от-

крывающую фигурную скобку после имени метода Main(), компилятор IDE сгенерирует последовательность "обнаруженных" ошибок, показанную на рис 2 6 (аналогичный отчет об ошибках генерируется и в результате вызова компилятора командной строки csc)

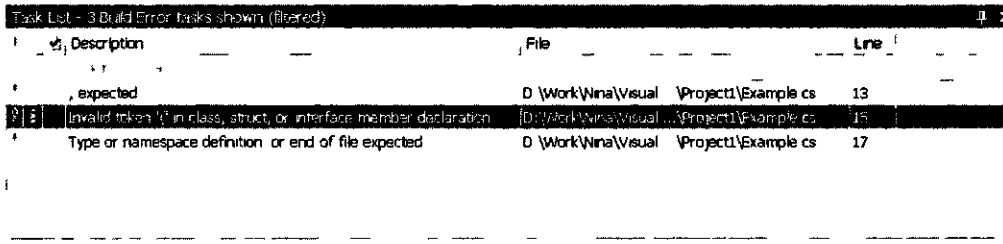


Рис 2 6 Отчет об ошибках

Очевидно, первое сообщение об ошибке неверно, поскольку пропущена не точка с запятой, а фигурная скобка Следующие два сообщения вообще могут лишь сбить с толку

Цель этих рассуждений — доказать, что если в программе присутствует синтаксическая ошибка, то не все сообщения компилятора следует принимать "за чистую монету" Они могут легко ввести в заблуждение Чтобы найти ошибку, нужно при анализе сообщений, генерируемых компилятором, научиться "ясновидению" Кроме того, делу может помочь просмотр нескольких последних строк кода, непосредственно предшествующих строке с "обнаруженной" ошибкой Ведь иногда компилятор начинает "чуять недоброе" только через несколько строк после реального местоположения ошибки

Небольшая вариация на тему первой программы

Несмотря на то что инструкцию

```
using System;
```

используют все программы в этой книге, в первой программе без нее можно обойтись Однако не стоит забывать, что она предоставляет большое удобство для программиста Причина же ее необязательности состоит в том, что в C# можно всегда *полностью определить* имя с использованием пространства имен, которому оно принадлежит Например, программную строку

```
Console.WriteLine("Простая C#-программа.");
```

можно заменить следующей

```
System.Console.WriteLine("Простая C#-программа.");
```

Таким образом, первую программу можно переписать следующим образом

```
// Эта версия не включает инструкцию using System.

class Example {

  // Любая C#-программа начинается с вызова метода Main().
  public static void Main() {

    // Здесь имя Console.WriteLine полностью определено.
    System.Console.WriteLine("Простая C#-программа.");
  }
}
```

Поскольку довольно утомительно указывать пространство имен `System` везде, где используется его член, большинство C#-программистов включают в начало программ инструкцию `using System` (эта участь постигла и все программы, представленные в этой книге). Однако важно понимать, что при необходимости можно полностью определить любое имя, явно указав пространство имен, которому оно принадлежит.

Вторая простая программа

При использовании любого языка программирования нет более важной инструкции, чем присваивание переменной значения. *Переменная* — это именованная область памяти, которой может быть присвоено определенное значение. Во время выполнения программы значение переменной может меняться. Другими словами, содержимое переменной не фиксировано, а изменяемо.

Следующая программа создает две переменные с именами `x` и `y`.

```
// Эта программа демонстрирует работу с переменными.

using System;

class Example2 {
  public static void Main() {
    int x; // Здесь объявляется переменная.
    int y; // Здесь объявляется еще одна переменная.

    x = 100; // Здесь переменной x присваивается 100.

    Console.WriteLine("x содержит " + x);

    y = x / 2;

    Console.Write("y содержит x / 2: ");
    Console.WriteLine(y);
  }
}
```

При выполнении этой программы вы должны увидеть следующий результат:

```
x содержит 100
y содержит x / 2: 50
```

Что же нового в этой программе? Итак, инструкция

```
int x; // Здесь объявляется переменная.
```

объявляет переменную с именем `x` целочисленного типа. В C# все переменные должны быть объявлены до их использования. В объявлении переменной помимо ее имени необходимо указать, значения какого типа она может хранить. Тем самым объявляется *тип переменной*. В данном случае переменная `x` может хранить целочисленные значения, т.е. целые числа. В C# для объявления переменной целочисленного типа достаточно поставить перед ее именем ключевое слово `int`. Таким образом, инструкция `int x;` объявляет переменную `x` типа `int`.

Следующая строка программы объявляет вторую переменную с именем `y`.

```
int y; // Здесь объявляется еще одна переменная.
```

Обратите внимание на то, что здесь используется тот же формат объявления, что и в предыдущей строке кода. Разница состоит лишь в именах объявляемых переменных.

В общем случае, чтобы объявить переменную, необходимо использовать инструкцию следующего формата:

тип имя_переменной;

Здесь с помощью элемента *тип* задается тип объявляемой переменной, а с помощью элемента *имя_переменной* — ее имя. Помимо типа `int`, C# поддерживает и другие типы данных.

Следующая строка кода присваивает переменной x значение 100.

```
x = 100; // Здесь переменной x присваивается 100.
```

В C# *оператор присваивания* представляется одиночным знаком равенства (=). Его действие заключается в копировании значения, расположенного справа от оператора, в переменную, указанную слева от него.

Следующая строка кода выводит значение переменной x, предваряя его текстовой строкой `"x содержит "`.

```
Console.WriteLine("x содержит " + x);
```

В этой инструкции знак "плюс" означает не операцию сложения, а последовательное отображение заданной текстовой строки и значения переменной x. В общем случае, используя оператор "+", можно в одной инструкции вызова метода `WriteLine()` сформировать сцепление элементов в нужном количестве.

Следующая строка кода присваивает переменной y значение переменной x, разделенное на 2.

```
y = x / 2;
```

При выполнении этой строки программы значение переменной x делится на 2, а затем полученный результат сохраняется в переменной y. Таким образом, переменная y будет содержать значение 50. Значение переменной x при этом не изменится. Подобно большинству языков программирования, C# поддерживает полный диапазон арифметических операторов, включая следующие:

+	Сложение
-	Вычитание
*	Умножение
/	Деление

Рассмотрим две строки программы:

```
Console.Write("y содержит x / 2: ");
Console.WriteLine(y);
```

Здесь сразу два новых момента. Во-первых, для отображения строки `"y содержит x / 2: "` используется не уже знакомый нам метод `WriteLine()`, а встроенный метод `Write()`. В этом случае выводимая текстовая строка не сопровождается символом новой строки. Это означает, что выполнение очередной операции вывода данных будет начинаться на той же строке. Таким образом, метод `Write()` аналогичен методу `WriteLine()`, но с той лишь разницей, что после каждого вызова он не выводит символ новой строки. Во-вторых, обратите внимание на то, что в обращении к методу `WriteLine()` переменная y используется самостоятельно, т.е. без текстового сопровождения. Эта инструкция служит демонстрацией того, что как `WriteLine()`, так и `Write()` можно использовать для вывода значений любых встроенных C#-типов.

Необходимо также отметить, что с помощью одной инструкции можно объявить сразу две или больше переменных. Для этого достаточно разделить их имена запятыми. Например, в рассмотренной программе переменные x и y можно было объявить следующим образом:

```
int x, y; // Объявление обеих переменных в одной инструкции.
```

Другие типы данных

В предыдущей программе мы использовали переменную типа int. Однако в переменной типа int можно хранить только целые числа. Следовательно, ее нельзя использовать для обработки дробной части числа. Например, в int-переменной может содержаться значение 18, но отнюдь не значение 18,3. К счастью, тип данных int — не единственный тип, определенный в C#. Для обработки чисел с дробной частью в C# предусмотрены два типа данных с плавающей точкой, float и double, которые представляют значения с обычной и удвоенной точностью, соответственно. (Тип double пользуется у программистов "повышенным спросом".)

Для объявления переменной типа double используйте инструкцию, подобную следующей:

```
double result;
```

Здесь result — это имя переменной типа double. Поскольку переменная result имеет тип double, она может хранить такие значения, как 122,23, 0,034 или −19,0.

Чтобы лучше понять различие между типами данных int и double, рассмотрим следующую программу:

```
/*
   Эта программа иллюстрирует различие между
   типами int и double.
*/

using System;

class Example3 {
  public static void Main() {
    int ivar;     // Объявляем переменную типа int.
    double dvar;  // Объявляем переменную типа double.

    ivar = 100;   // Присваиваем переменной ivar
                  // значение 100.

    dvar = 100.0; // Присваиваем переменной dvar
                  // значение 100.0.

    Console.WriteLine(
            "Исходное значение переменной ivar: " + ivar);
    Console.WriteLine(
            "Исходное значение переменной dvar: " + dvar);

    Console.WriteLine(); // Выводим пустую строку.

    // Теперь делим оба значения на 3.
    ivar = ivar / 3;
    dvar = dvar / 3.0;

    Console.WriteLine("ivar после деления: " + ivar);
    Console.WriteLine("dvar после деления: " + dvar);
  }
}
```

Вот как выглядит результат выполнения этой программы:

```
Исходное значение переменной ivar: 100
Исходное значение переменной dvar: 100
```

```
ivar после деления: 33
dvar после деления: 33.3333333333333
```

Как видите, при делении переменной `ivar` на 3 выполняется операция целочисленного деления, результат которой равен 33, т.е. дробная часть отбрасывается. Но при делении переменной `dvar` на 3 дробная часть сохраняется.

Итак, если нужно определить в программе значение с плавающей точкой, необходимо включить в его представление десятичную точку. Если этого не сделать, оно будет интерпретироваться как целое число. Например, в C# число `100` рассматривается как целое, а число `100.0` — как значение с плавающей точкой.

Обратите внимание на то, что для вывода пустой строки достаточно вызвать метод `WriteLine` без аргументов.

Тип данных с плавающей точкой обычно используется при работе с реальными значениями, т.е. когда необходимо учитывать дробную часть каждого числа. Например, следующая программа вычисляет площадь круга, используя для π значение 3,1416.

```
// Вычисляем площадь круга.

using System;

class Circle {
  static void Main() {
    double radius;
    double area;

    radius = 10.0;
    area = radius * radius * 3.1416;

    Console.WriteLine("Площадь равна " + area);
  }
}
```

Результат выполнения этой программы таков:

```
Площадь равна 314.16
```

Очевидно, что вычисление площади круга не может быть вычислено с удовлетворительным результатом без использования данных с плавающей точкой.

Первое знакомство с инструкциями управления

Инструкции внутри метода выполняются последовательно, можно сказать, сверху вниз. Но такой ход выполнения можно изменить с помощью различных инструкций управления, поддерживаемых в C#. Подробно инструкции управления будут рассмотрены ниже, а пока мы кратко познакомимся с двумя из них, поскольку они используются в примерах программ, приведенных в этом разделе.

Инструкция `if`

С помощью инструкции `if` можно организовать избирательное выполнение части программы. Действие инструкции `if` в C# во многом подобно действию одноименной инструкции в любом другом языке программирования. Что касается языков C, C++ и Java, то здесь налицо полная идентичность. Вот как выглядит простейшая форма записи этой инструкции:

```
if(условие) инструкция;
```

Здесь элемент *условие* представляет собой булево выражение (которое приводится к значению ИСТИНА или ЛОЖЬ). Заданная *инструкция* будет выполнена, если *условие* окажется истинным. В противном случае (если *условие* окажется ложным) заданная *инструкция* игнорируется. Рассмотрим следующий пример:

```
if(10 < 11) Console.WriteLine("10 меньше 11");
```

В данном случае число 10 действительно меньше 11, т.е. условное выражение истинно, поэтому метод `WriteLine()` будет вызван. Рассмотрим другой пример:

```
if(10 < 9) Console.WriteLine("Этот текст выведен не будет.");
```

Здесь же число 10 никак не меньше 9, поэтому вызов метода `WriteLine()` не произойдет.

В C# определен полный комплект операторов отношения, которые можно использовать в условных выражениях. Вот их список:

`<`	Меньше
`<=`	Меньше или равно
`>`	Больше
`>=`	Больше или равно
`==`	Равно
`!=`	Не равно

Следующая программа иллюстрирует использование инструкции `if`.

```
// Демонстрация использования инструкции if.

using System;

class IfDemo {
  public static void Main() {
    int a, b, c;

    a = 2;
    b = 3;

    if(a < b) Console.WriteLine("a меньше b");

    // Следующая инструкция ничего не отобразит на экране.
    if(a == b) Console.WriteLine(
                        "Этого текста никто не увидит.");
      Console.WriteLine();

    c = a - b; // Переменная c содержит -1.

    Console.WriteLine("c содержит -1");
    if(c >= 0) Console.WriteLine(
                              "Значение c неотрицательно");
    if(c < 0) Console.WriteLine("Значение c отрицательно");

    Console.WriteLine();

    c = b - a; // Теперь переменная c содержит 1.
    Console.WriteLine("c содержит 1");
    if(c >= 0) Console.WriteLine(
                              "Значение c неотрицательно");
    if(c < 0) Console.WriteLine("Значение c отрицательно");

  }
}
```

Результат выполнения этой программы имеет такой вид:

```
a меньше b

c содержит -1
Значение c отрицательно

c содержит 1
Значение c неотрицательно
```

Обратите внимание вот на что. В строке

```
int a, b, c;
```

объявляются сразу три переменных за счет использования списка элементов, разделенных запятой. Как упоминалось выше, две или больше переменных одинакового типа можно объявить в одной инструкции, отделив их имена запятыми.

Цикл `for`

Чтобы многократно выполнить последовательность программных инструкций, необходимо организовать *цикл*. В языке C# циклические конструкции представлены в богатом ассортименте. В этом разделе мы рассмотрим цикл `for`. Если вы знакомы с C, C++ или Java, то вам будет приятно узнать, что цикл `for` в C# работает точно так же, как в этих языках. Простейшая форма записи цикла `for` имеет следующий вид.

`for(`*инициализация*`;` *условие*`;` *итерация*`)` *инструкция*`;`

В самой общей форме элемент *инициализация* устанавливает управляющую переменную цикла равной некоторому начальному значению. Элемент *условие* представляет собой булево выражение, в котором тестируется значение управляющей переменной цикла. Если результат этого тестирования истинен, цикл `for` выполняется еще раз, в противном случае его выполнение прекращается. Элемент *итерация* — это выражение, которое определяет, как изменяется значение управляющей переменной цикла после каждой итерации. Рассмотрим небольшую программу, в которой иллюстрируется цикл `for`.

```
// Демонстрация цикла for.

using System;

class ForDemo {
  public static void Main() {
    int count;

    for(count = 0; count < 5; count = count+1)
      Console.WriteLine("Это счет: " + count);

    Console.WriteLine("Готово!");
  }
}
```

Вот как выглядит результат выполнения этой программы:

```
Это счет: 0
Это счет: 1
Это счет: 2
Это счет: 3
Это счет: 4
Готово!
```

В этой программе управляющей переменной цикла является `count`. В выражении инициализации цикла `for` она устанавливается равной нулю. В начале каждой итерации (включая первую) выполняется проверка условия `count < 5`. Если результат этой проверки окажется истинным, выполнится инструкция вывода строки `WriteLine()`, а после нее — итерационное выражение цикла. Этот процесс будет продолжаться до тех пор, пока проверка условия не даст в результате значение ЛОЖЬ, после чего выполнение программы возобновится с инструкции, расположенной за циклом.

Интересно отметить, что в C#-программах, написанных профессиональными программистами, редко можно встретить итерационное выражение цикла в том виде, в каком оно представлено в рассматриваемой программе. Другими словами, вряд ли вы увидите инструкции, подобные следующей:

```
count = count + 1;
```

Дело в том, что C# включает специальный *оператор инкремента*, который позволяет более эффективно выполнить операцию увеличения значения на единицу. Оператор инкремента обозначается двумя последовательными знаками "плюс" (++). С его помощью предыдущую инструкцию можно переписать следующим образом:

```
count++;
```

Следовательно, начало цикла `for` в предыдущей программе опытный программист оформил бы так:

```
for(count = 0; count < 5; count++)
```

Если вам захочется выполнить предыдущую программу, используя оператор инкремента, вы убедитесь, что результат останется прежним.

В C# также предусмотрен *оператор декремента* (`--`). Нетрудно догадаться, что этот оператор уменьшает значение операнда на единицу.

Использование блоков кода

Не менее важным, чем инструкции управления, элементом языка C# является *программный блок*. Программный блок представляет собой группирование двух или более инструкций. Такое группирование инструкций реализуется посредством их заключения между открывающей и закрывающей фигурными скобками. После создания блок кода становится логическим элементом программы, который можно использовать в любом ее месте, где может находиться одна инструкция. Например, блок может быть частью `if`- или `for`-инструкций. Рассмотрим следующую `if`-инструкцию:

```
if(w < h) {
  v = w * h;
  w = 0;
}
```

Здесь сравниваются значения переменных `w` и `h`, и если оказывается, что `w < h`, то будут выполнены обе инструкции внутри блока. Следовательно, две инструкции в блоке образуют логический элемент, в результате чего одна инструкция не может быть выполнена без выполнения другой. Важно то, что, если нужно логически связать две или более инструкций, это легко реализуется созданием программного блока. Именно благодаря блокам можно упростить код реализации многих алгоритмов и повысить эффективность их выполнения.

Рассмотрим программу, в которой программный блок используется для предотвращения деления на нуль.

```
// Демонстрация использования программного блока.
```

```
using System;

class BlockDemo {
  public static void Main() {
    int i, j, d;

    i = 5;
    j = 10;

    // Эта if-инструкция управляет программным
    // блоком, а не одной инструкцией.
    if(i != 0) {
      Console.WriteLine("i не равно нулю");
      d = j / i;
      Console.WriteLine("j / i равно " + d);
    }
  }
}
```

Результат выполнения этой программы имеет следующий вид:

```
i не равно нулю
j / i равно 2
```

В этом случае `if`-инструкция управляет программным блоком, а не просто одной инструкцией. Если управляющее условие `if`-инструкции окажется истинным (а оно таким здесь и является), будут выполнены все три инструкции, составляющие этот блок. Проведите небольшой эксперимент. Замените в этой программе инструкцию

```
i = 5;
```

инструкцией

```
i = 0;
```

и сравните результат выполнения нового варианта программы со старым.

А вот еще один пример. На этот раз программный блок используется для вычисления суммы чисел от 1 до 10 и их произведения.

```
// Вычисляем сумму и произведение чисел от 1 до 10.

using System;

class ProdSum {
  static void Main() {
    int prod;
    int sum;
    int i;

    sum = 0;
    prod = 1;

    for(i=1; i <= 10; i++) {
      sum = sum + i;
      prod = prod * i;
    }
    Console.WriteLine("Сумма равна " + sum);
    Console.WriteLine("Произведение равно " + prod);

  }
}
```

В результате выполнения программы получаем следующее:

```
Сумма равна 55
Произведение равно 3628800
```

Здесь (благодаря блоку) в одном цикле вычисляется как сумма чисел, так и их произведение. Без этого средства языка пришлось бы использовать два отдельных `for`-цикла.

И еще. Программные блоки не снижают динамику выполнения программ. Другими словами, наличие фигурных скобок (`{` и `}`) не означает дополнительных затрат времени на выполнение программы. Наоборот, благодаря способности блоков кода упрощать программирование алгоритмов, повышается скорость и эффективность выполнения программ в целом.

Использование точки с запятой и оформление текста программы

В C# точка с запятой означает конец инструкции, т.е. каждая отдельная инструкция должна оканчиваться точкой с запятой.

Как вы уже знаете, блок — это набор логически связанных инструкций, заключенный между открывающей и закрывающей фигурными скобками. Поскольку каждая из инструкций блока завершается точкой с запятой, то признаком завершения самого блока является закрывающая фигурная скобка (а не точка с запятой).

C# не распознает конец строки как конец инструкции; признаком конца инструкции служит только точка с запятой, поэтому расположение инструкции в строке не имеет значения. Например, в C# следующий фрагмент кода

```
x = y;
y = y + 1;
Console.WriteLine(x + " " + y);
```

абсолютно идентичен представленному в виде одной строке.

```
x = y;  y = y + 1;  Console.WriteLine(x + " " + y);
```

Более того, различные элементы инструкции можно расположить на отдельных строках. Например, следующая запись инструкции абсолютно приемлема.

```
Console.WriteLine("Это длинная текстовая строка" +
                  x + y + z +
                  "другие данные, подлежащие выводу");
```

Подобное разбиение длинных программных строк часто позволяет сделать программу более читабельной.

Использование отступов

Глядя на текст предыдущих программ, вы, вероятно, заметили, что некоторые инструкции записаны с отступами от левого края. C# — это язык, допускающий свободную форму записи инструкций, т.е. не имеет значения, как расположены инструкции на строке относительно друг друга. Однако у программистов выработался определенный стиль оформления программ, который позволяет сделать программу максимально читабельной. Программы, представленные в этой книге, оформлены с соблюдением этого стиля, что рекомендуется делать и вам. Согласно этому стилю, после каждой открывающей фигурной скобки следует делать отступ (в виде нескольких пробелов), а после каждой закрывающей фигурной скобки — возвращаться назад (к предыдущему уровню отступа). Для некоторых инструкций рекомендуется делать дополнительный отступ, но об этом речь впереди.

Ключевые слова C#

В языке C# на данный момент определено 77 ключевых слов, которые перечислены в табл. 2.1. Эти ключевые слова (в сочетании с синтаксисом операторов и разделителей) образуют определение языка C#. Ключевые слова нельзя использовать в качестве имен переменных, классов или методов.

Таблица 2.1. Ключевые слова C#

abstract	event	new	struct
as	explicit	null	switch
base	extern	object	this
bool	false	operator	throw
break	finally	out	true
byte	fixed	override	try
case	float	params	typeof
catch	for	private	uint
char	foreach	protected	ulong
checked	goto	public	unchecked
class	if	readonly	unsafe
const	implicit	ref	ushort
continue	in	return	using
decimal	int	sbyte	virtual
default	interface	sealed	volatile
delegate	internal	short	void
do	is	sizeof	while
double	lock	stackalloc	
else	long	static	
enum	namespace	string	

Идентификаторы

В C# идентификатор представляет собой имя, присвоенное методу, переменной или иному элементу, определенному пользователем. Идентификаторы могут состоять из одного или нескольких символов. Имена переменных должны начинаться с буквы или символа подчеркивания. Последующим символом может быть буква, цифра и символ подчеркивания. Символ подчеркивания можно использовать для улучшения читабельности имени переменной, например `line_count`. В C# прописные и строчные буквы *воспринимаются* как различные символы, т.е. `myvar` и `MyVar` — это разные имена. Вот несколько примеров допустимых идентификаторов.

```
Test      x        y2        MaxLoad
up        _top     my_var    sample23
```

Помните, что идентификатор не должен начинаться с цифры. Например, `12x` — недопустимый идентификатор. Конечно, вы вольны называть переменные и другие программные элементы по своему усмотрению, но обычно идентификатор отражает назначение или смысловую характеристику элемента, которому он принадлежит.

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. Теперь самое время рассмотреть программу, в которой используется @-идентификатор.

```
// Демонстрируем использование @-идентификатора.

using System;

class IdTest {
  static void Main() {
    int @if; // Используем if в качестве идентификатора.

    for(@if = 0; @if < 10; @if++)
      Console.WriteLine("@if равно " + @if);
  }
}
```

Результат выполнения этой программы доказывает, что `@if` действительно интерпретируется как идентификатор.

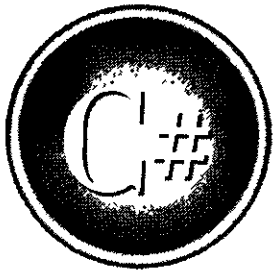
```
@if равно 0
@if равно 1
@if равно 2
@if равно 3
@if равно 4
@if равно 5
@if равно 6
@if равно 7
@if равно 8
@if равно 9
```

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.

Библиотеки классов C#

В примерах программ, представленных в этой главе, использовано два встроенных C#-метода — `WriteLine()` и `Write()`. Как упоминалось выше, эти методы — члены класса `Console`, который является частью пространства имен `System`, определенного в библиотеках классов среды .NET Framework. Вы уже знаете, что C#-среда опирается на библиотеки классов среды .NET Framework, что позволяет ей обеспечить поддержку операций ввода-вывода, обработку строк, сетевые возможности и графические интерфейсы пользователя (GUIs). Таким образом, C# в целом — это объединение самого языка C# (его языковых элементов) и классов .NET-стандарта. Как будет показано ниже, библиотеки классов существенно повышают функциональность C#-программы. Чтобы стать профессиональным C#-программистом, важно научиться эффективно использовать эти стандартные классы. В части I мы познакомимся с элементами библиотечных классов .NET-стандарта, а детали .NET-библиотеки описаны в части II.

Полный справочник по



Глава 3

Типы данных, литералы и переменные

В этой главе рассматриваются три основных элемента языка C#: типы данных, литералы и переменные. В общем случае типы данных определяют класс задач, к которым они могут быть применены. В C# предусмотрен богатый набор встроенных типов данных, что позволяет использовать этот язык для широкого диапазона приложений. Программист может создавать переменные нужного ему типа и определять константы, которые в языке C# называются *литералами*.

О важности типов данных

Типы данных имеют в C# особое значение, поскольку C# — строго типизированный язык. Это значит, что все операции проверяются компилятором на соответствие типов. Некорректные операции не компилируются. Таким образом, контроль типов способствует предотвращению ошибок и повышает надежность программ. Для обеспечения контроля типов необходимо, чтобы все переменные, выражения и значения имели определенный тип. Например, в языке не допускается, чтобы переменная не имела типа. Более того, тип значения определяет, какие операции разрешено выполнять с таким значением. Операция, разрешенная для одного типа, может быть недопустимой для другого.

Типы значений в C#

C# содержит две категории встроенных типов данных: *типы значений* и *ссылочные типы*. Ссылочные типы определяются в классах, но о классах речь еще впереди. Ядро языка C# составляют 13 типов, перечисленных в табл. 3.1. Это — встроенные типы, которые определяются ключевыми словами C# и доступны для использования в любой C#-программе.

Термин "тип значения" применяется к переменным, которые непосредственно содержат значения. (Для сравнения: переменные ссылочных типов содержат ссылки на реальные значения.) Таким образом, типы значений в C# во многом подобны типам данных, определенным в других языках программирования (например, C++). Типы значений также называют *простыми типами*.

В C# строго определяется диапазон и поведение каждого типа значения. Исходя из требований переносимости, C# на этот счет не допускает никаких компромиссов. Например, тип `int` должен быть одинаковым во всех средах выполнения. Поэтому при переходе на другую платформу не должна возникать необходимость переделки кода. Несмотря на то что строгое задание размерных характеристик типов значений может вызвать в некоторых средах небольшие потери производительности, ради достижения переносимости с ними необходимо смириться.

Таблица 3.1. Типы значений в C#

Ключевое слово	*Тип*
`bool`	Логический, или булев, представляет значения ИСТИНА/ЛОЖЬ
`byte`	8-разрядный целочисленный без знака
`char`	Символьный
`decimal`	Числовой тип для финансовых вычислений
`double`	С плавающей точкой двойной точности

Окончание табл. 3.1

Ключевое слово	*Тип*
`float`	С плавающей точкой
`int`	Целочисленный
`long`	Тип для представления длинного целого числа
`sbyte`	8-разрядный целочисленный со знаком
`short`	Тип для представления короткого целого числа
`uint`	Целочисленный без знака
`ulong`	Тип для представления длинного целого числа без знака
`ushort`	Тип для представления короткого целого числа без знака

Целочисленные типы

В C# определено девять целочисленных типов: `char`, `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` и `ulong`. Однако тип `char` в основном используется для представления символов (подробнее рассматривается ниже в этой главе). Остальные восемь целочисленных типов предназначены для числовой обработки данных. Размер значений в битах и диапазоны представления для каждого из этих восьми типов приведены в табл. 3.2.

Таблица 3.2. Характеристики целочисленных типов

Тип	*Размер в битах*	*Диапазон*
`byte`	8	0–255
`sbyte`	8	-128–127
`short`	16	-32 768–32 767
`ushort`	16	0–65 535
`int`	32	-2 147 483 648–2 147 483 647
`uint`	32	0–4 294 967 295
`long`	64	-9 223 372 036 854 775 808–9 223 372 036 854 775 807
`ulong`	64	0–18 446 744 073 709 551 615

Согласно этой таблице, в C# определены обе версии всех целочисленных типов: как со знаком, так и без него. Различие между этими версиями заключается в способе интерпретации старшего разряда. Если в программе задано целочисленное значение со знаком, то компилятор сгенерирует код, в котором предусматривается, что старший разряд такого значения используется в качестве *флага знака* (sign flag). Если флаг знака равен нулю, значит, число положительно, если он равен 1, то число отрицательно. Отрицательные числа почти всегда представляются в виде дополнения до двух. Для получения дополнительного кода сначала все разряды числа, за исключением знакового, инвертируются, а затем результат инвертирования суммируется с единицей. Наконец, флаг знака устанавливается равным единице.

Целочисленные значения со знаком широко используются во многих алгоритмах, но следует понимать, что по абсолютному значению (по модулю) они составляют только половину от своих "родственников" со знаком. Например, вот как выглядит в двоичном коде число 32 767 в качестве значения типа `short`:

```
01111111 11111111
```

Если старший разряд этого значения со знаком установить равным 1, оно будет интерпретироваться как −1 (с использованием формата дополнения до двух). То же значение, но объявленное с типом ushort (и с "единичным" старшим разрядом), будет равно числу 65 535.

Вероятно, самым популярным целочисленным типом является int. Переменные типа int часто используются для управления циклами, индексации массивов и в математических вычислениях общего назначения. Если нужно обрабатывать величины, диапазон которых превышает диапазон, допустимый для типа int, у вас есть несколько возможных вариантов для выбора. Если вы не предполагаете обрабатывать отрицательные числа, можно воспользоваться типом uint (в этом случае диапазон представления чисел увеличится вдвое). Для работы с большими числами со знаком, возможно, подойдет тип long, без знака — тип ulong. Рассмотрим, например, программу, которая вычисляет расстояние от Земли до Солнца в дюймах. Поскольку результат будет представлять собой довольно большое число, для его хранения в этой программе используется переменная типа long.

```
// Вычисляем расстояние от Земли до Солнца в дюймах.

using System;

class Inches {
  public static void Main() {
    long inches;
    long miles;

    miles = 93000000; // 93 000 000 миль до Солнца

    // 5 280 футов в миле, 12 дюймов в футе
    inches = miles * 5280 * 12;

    Console.WriteLine("Расстояние от Земли до Солнца: " +
                      inches + " дюймов.");

  }
}
```

Вот как выглядит результат выполнения этой программы:

```
Расстояние от Земли до Солнца: 5892480000000 дюймов.
```

Ясно, что полученный результат невозможно было бы сохранить в переменной типа int или uint.

Самый маленький диапазон представления имеют целочисленные типы byte и sbyte. Тип byte предназначен для хранения чисел без знака от 0 до 255. Переменные типа byte особенно полезны для обработки двоичных данных, например, полученных в результате опроса некоторого датчика. Для небольших целых чисел со знаком используется тип sbyte. Приведем пример применения типа byte для управления циклом for, который вычисляет сумму чисел от 1 до 100.

```
// Использование типа byte.

using System;

class Use_byte {
  public static void Main() {
    byte x;
    int sum;
```

```
    sum = 0;
    for(x = 1; x <= 100; x++)
      sum = sum + x;

    Console.WriteLine("Сумма чисел от 1 до 100 равна "
                      + sum);
  }
}
```

Результат выполнения этой программы таков:

```
Сумма чисел от 1 до 100 равна 5050
```

Поскольку цикл `for` повторяется от 0 до 100 раз, что не выходит за пределы диапазона типа `byte`, нет необходимости использовать для управления циклом переменную с типом, допускающим более широкий диапазон представления чисел.

Если вам понадобится целочисленный тип с диапазоном представления, большим, чем у `byte` и `sbyte`, но меньшим, чем у `int` или `uint`, используйте тип `short` или `ushort`.

Типы для представления чисел с плавающей точкой

Типы с плавающей точкой могут представлять числа с дробными компонентами. Таких типов только два: `float` и `double`. Для значений типа `float` используется 32 бита, которые позволяют представить числа в диапазоне от 1,5E−45 до 3,4E+38. Для значений же типа `double` (т.е. удвоенной точности) используется 64 бита, позволяющие представить числа в диапазоне от 5E−324 до 1,7E+308.

Из этих двух типов с плавающей точкой гораздо большей популярностью у программистов пользуется тип `double`. Одной из основных причин этого является использование `double`-значений множеством математических функций библиотеки классов C# (библиотека .NET Framework). Например, метод `Sqrt()`, который определен в стандартном классе `System.Math`, возвращает `double`-значение, являющееся квадратным корнем из `double`-аргумента. В следующей программе метод `Sqrt()` используется для вычисления радиуса круга, исходя из его площади.

```
// Находим радиус круга по его площади.

using System;

class FindRadius {
  public static void Main() {
    double r;
    double area;

    area = 10.0;

    r = Math.Sqrt(area / 3.1416);

    Console.WriteLine("Радиус равен " + r);
  }
}
```

Результат выполнения этой программы таков:

```
Радиус равен 1.78412203012729
```

В этой программе стоит обратить ваше внимание на то, как вызывается метод `Sqrt()`: его имени предшествует имя `Math`. И здесь нет ничего удивительного, ведь, как уже упоминалось выше, метод `Sqrt()` — член класса `Math`. (Точно также при вызове метода `WriteLine()` его имени предшествовало имя класса `Console`.) Однако необходимо отметить, что не все стандартные методы вызываются посредством первоначального указания имени их "родного" класса.

В следующей программе демонстрируется несколько тригонометрических функций, которые являются частью математической библиотеки C#. Они также оперируют данными типа `double`. Программа отображает значения синуса, косинуса и тангенса для углов (измеряемых в радианах) от 0,1 до 1,0.

```
// Демонстрируем использование методов Math.Sin(),
// Math.Cos() и Math.Tan().

using System;

class Trigonometry {
  public static void Main() {
    double theta; // Угол задан в радианах.

    for(theta = 0.1; theta <= 1.0; theta = theta + 0.1) {
      Console.WriteLine("Синус угла " + theta +
                        "  равен " + Math.Sin(theta));
      Console.WriteLine("Косинус угла " + theta +
                        "  равен " + Math.Cos(theta));
      Console.WriteLine("Тангенс угла " + theta +
                        "  равен " + Math.Tan(theta));
      Console.WriteLine();
    }

  }
}
```

Вот как выглядит часть результатов выполнения этой программы:

```
Синус угла 0.1  равен 0.0998334166468282
Косинус угла 0.1  равен 0.995004165278026
Тангенс угла 0.1  равен 0.100334672085451

Синус угла 0.2  равен 0.198669330795061
Косинус угла 0.2  равен 0.980066577841242
Тангенс угла 0.2  равен 0.202710035508673

Синус угла 0.3  равен 0.29552020666134
Косинус угла 0.3  равен 0.955336489125606
Тангенс угла 0.3  равен 0.309336249609623
```

Для вычисления синуса, косинуса и тангенса используются стандартные библиотечные методы `Math.Sin()`, `Math.Cos()` и `Math.Tan()`. Подобно методу `Math.Sqrt()` эти тригонометрические методы вызываются с аргументом типа `double` и возвращают результат типа `double`. При этом углы должны быть заданы в радианах.

Тип decimal

Возможно, самым интересным в C# числовым типом является тип `decimal`, который предназначен для выполнения вычислений, связанных с денежными единицами. Тип `decimal` для представления значений в диапазоне от 1E−28 до 7,9E+28 использу-

ет 128 бит. Применение обычной арифметики с плавающей точкой к десятичным значениям чревато ошибками округления. Во избежание этих ошибок и предусмотрен тип decimal, который способен точно представить до 28 десятичных разрядов (в некоторых случаях до 29). Способность представлять десятичные значения без ошибок округления делает этот тип особенно полезным для вычислений в денежной сфере.

Рассмотрим программу, которая использует тип decimal для вычисления цены со скидкой на основе заданных значений исходной цены и процента скидки.

```
// Использование типа decimal для вычисления скидки.

using System;

class UseDecimal {
  public static void Main() {
    decimal price;
    decimal discount;
    decimal discounted_price;

    // Вычисляем цену со скидкой.
    price = 19.95m;
    discount = 0.15m; // Ставка дисконта равна 15%.

    discounted_price = price - ( price * discount);

    Console.WriteLine("Цена со скидкой: $" +
                      discounted_price);
  }
}
```

Результат работы этой программы выглядит так:

```
Цена со скидкой: $16.9575
```

Обратите внимание на то, что задание decimal-констант сопровождается наличием суффикса m. Без него эти константы интерпретировались бы как стандартные константы с плавающей точкой, несовместимые с типом данных decimal. При этом переменной типа decimal можно присвоить любое целочисленное значение (например, 10) без использования суффикса m. (О задании числовых констант мы еще поговорим более подробно позже в этой главе.)

А вот еще один пример использования типа decimal. В следующей программе вычисляется будущая стоимость капиталовложений, которые имеют фиксированную годовую процентную ставку для прибыли на инвестированный капитал.

```
/*
   Использование типа decimal для вычисления будущей
   стоимости капиталовложений.
*/

using System;

class FutVal {
  public static void Main() {
    decimal amount;
    decimal rate_of_return;
    int years, i;

    amount = 1000.0M;
    rate_of_return = 0.07M;
    years = 10;
```

```
    Console.WriteLine("Исходный вклад: $" + amount);
    Console.WriteLine("Норма прибыли: " + rate_of_return);
    Console.WriteLine("Через " + years + " лет");

    for(i = 0; i < years; i++)
      amount = amount + (amount * rate_of_return);

    Console.WriteLine("Будущая стоимость равна $" +
                      amount);
  }
}
```

Результаты работы этой программы имеют такой вид:

```
Исходный вклад: $1000
Норма прибыли: 0.07
Через 10 лет
Будущая стоимость равна $1967.15135728956532249
```

Обратите внимание на (даже излишнюю) точность результата! Ниже в этой главе вы узнаете, как форматировать результат, чтобы он выглядел более привлекательно.

Символы

В C# символы представляются не 8-разрядными величинами, как в других языках программирования (например, C++), а 16-разрядными. Для представления символов в C# используется *Unicode* (уникод), 16-разрядный стандарт кодирования символов, позволяющий представлять алфавиты всех существующих в мире языков. Хотя во многих языках (например, в английском, немецком, французском) алфавиты относительно невелики, существуют языки (например, китайский), построенные на очень больших наборах символов, которые нельзя представить восьмью битами. Чтобы можно было "охватить" символьные наборы всех языков, требуются 16-разрядные значения. Таким образом, в C# `char` — это 16-разрядный тип без знака, который позволяет представлять значения в диапазоне 0–65 535. Стандартный 8-разрядный набор символов ASCII составляет лишь подмножество Unicode с диапазоном 0–127. Таким образом, ASCII-символы — это действительные C#-символы.

Символьной переменной можно присвоить значение, заключив соответствующий символ в одинарные кавычки. Например, чтобы присвоить значение буквы `X` переменной `ch`, нужно выполнить следующие инструкции:

```
char ch;
ch = 'X';
```

Чтобы вывести `char`-значение, хранимое в переменной `ch`, можно использовать метод `WriteLine()`.

```
Console.WriteLine("Это ch: " + ch);
```

Хотя тип `char` определяется в C# как целочисленный, его нельзя свободно смешивать с целыми числами во всех случаях без разбору. Все дело в том, что автоматического преобразования целочисленных значений в значения типа `char` не существует. Например, следующий фрагмент программы содержит ошибку.

```
char ch;
ch = 10;  // Ошибка, это работать не будет.
```

Поскольку 10 — целое число, оно не может быть автоматически преобразовано в значение типа `char`. При попытке скомпилировать этот код вы получите сообщение об ошибке. Ниже в этой главе мы рассмотрим "обходной путь", позволяющий обойти это ограничение.

Тип bool

Тип bool представляет значения ИСТИНА/ЛОЖЬ, которые в C# определяются зарезервированными словами true и false. Таким образом, переменная или выражение типа bool будет иметь одно из этих двух значений. В C# не определено ни одно преобразование значения типа bool в целочисленное значение. Например, число 1 не преобразуется в значение true, а число 0 — в значение false.

Рассмотрим использование типа bool на примере следующей программы:

```
// Демонстрация использования значений типа bool.

using System;

class BoolDemo {
  public static void Main() {
    bool b;

    b = false;
    Console.WriteLine("b содержит " + b);
    b = true;
    Console.WriteLine("b содержит " + b);

    // Значение типа bool может управлять if-инструкцией.
    if(b) Console.WriteLine("Эта инструкция выполняется.");

    b = false;
    if(b) Console.WriteLine(
                    "Эта инструкция не выполняется.");

    // Оператор отношения возвращает результат типа bool.
    Console.WriteLine("10 > 9 равно " + (10 > 9));
  }
}
```

Эта программа генерирует следующий результат:

```
b содержит False
b содержит True
Эта инструкция выполняется.
10 > 9 равно True
```

Итак, что интересного в этой программе? Во-первых, при выводе bool-значения методом WriteLine() отображаются значения True или False. Во-вторых, одного значения bool-переменной вполне достаточно для управления if-инструкцией, т.е. нет необходимости в написании if-инструкции такого вида.

```
if(b == true) ...
```

В-третьих, результатом выполнения оператора отношения (например, оператора "<") является bool-значение. Поэтому результат выражения 10 > 9 приводит к отображению значения True. При использовании выражения 10 > 9, как видно в этой программе, потребовался дополнительный набор круглых скобок, поскольку оператор "+" имеет более высокий приоритет, чем оператор ">".

О некоторых вариантах вывода данных

До сих пор при выводе данных с помощью метода `WriteLine()` они отображались с использованием стандартного формата, определенного в C#. Однако в C# предусмотрен и более высокоорганизованный механизм форматирования, который позволяет более тонко управлять отображением данных. Несмотря на то что форматированный ввод-вывод подробно описывается далее, нам не обойтись без рассмотрения некоторых деталей уже сейчас. Они позволят сделать результаты, выводимые программой, более читабельными и привлекательными. Однако не забывайте, что в этом разделе описана только малая часть средств форматирования, поддерживаемых в C#.

При выводе списка данных его элементы необходимо разделять знаками "плюс". Вот пример:

```
Console.WriteLine(
   "Вы заказали " + 2 + " предмета по $" + 3 + " каждый.");
```

Несмотря на определенные достоинства такого способа вывода данных, он не дает никаких "рычагов" управления их форматированием. Например, выводя значение с плавающей точкой, вы не сможете управлять количеством отображаемых десятичных разрядов. Рассмотрим следующую инструкцию:

```
Console.WriteLine(
               "При делении 10/3 получаем: " + 10.0/3.0);
```

При ее выполнении увидим на экране такой результат:

```
При делении 10/3 получаем: 3.33333333333333
```

Результат, представленный с таким количеством десятичных разрядов, годится при решении одних задач и совершенно неприемлем в других случаях. Например, в денежных расчетах обычно ограничиваются отображением только двух десятичных разрядов.

Для управления форматированием числовых данных необходимо использовать вторую форму метода `WriteLine()`, которая позволяет ввести информацию о форматировании.

```
WriteLine("строка_форматирования",
                              arg0, arg1, ... , argN);
```

В этой версии метода `WriteLine()` передаваемые ему аргументы разделяются запятыми, а не знаками "+". Элемент *строка_форматирования* содержит две составляющие: "постоянную" и "переменную". Постоянная составляющая представляет собой печатные символы, отображаемые "как есть", а переменная состоит из *спецификаторов формата*. Общая форма записи спецификатора формата имеет следующий вид:

```
{номер_аргумента, ширина: формат}
```

Здесь элемент *номер_аргумента* определяет порядковый номер отображаемого аргумента (начиная с нулевого). С помощью элемента *ширина* указывается минимальная ширина поля, а формат задается элементом *формат*.

Если при выполнении метода `WriteLine()` в строке форматирования встречается спецификатор формата, вместо него подставляется (и отображается) аргумент, соответствующий заданному элементу *номер_аргумента*. Таким образом, элементы *номер_аргумента* указывают позицию спецификации в строке форматирования, которая определяет, где именно должны быть отображены соответствующие данные. Элементы *ширина* и *формат* указывать необязательно. Следовательно, спецификатор формата `{0}` означает *arg0*, `{1}` означает *arg1* и т.д.

Теперь рассмотрим простой пример. При выполнении инструкции

```
Console.WriteLine("В феврале {0} или {1} дней.", 28, 29);
```

будет сгенерирован следующий результат:

```
В феврале 28 или 29 дней.
```

Как видите, вместо спецификатора {0} было подставлено значение 28, а вместо спецификатора {1} — значение 29. Таким образом, внутри строки форматирования спецификаторы формата идентифицируют местоположение последовательно заданных аргументов (в данном случае это числа 28 и 29). Обратите также внимание на то, что составные части выводимого результата разделены не знаками "+", а запятыми.

А теперь "сыграем" вариацию на тему предыдущей инструкции, указав в спецификаторах формата минимальную ширину поля.

```
Console.WriteLine(
            "В феврале {0,10} или {1,5} дней.", 28, 29);
```

Вот как будет выглядеть результат ее выполнения:

```
В феврале         28 или    29 дней.
```

Нетрудно убедиться, что при выводе значений аргументов были добавлены пробелы, заполняющие неиспользуемые части полей. Обратите внимание на то, что второй элемент спецификатора формата означает *минимальную* ширину поля. Другими словами, при необходимости это значение может быть превышено.

Безусловно, аргументы, связанные с командой форматирования, необязательно должны быть константами. Например, в следующей программе отображается таблица результатов возведения ряда чисел в квадрат и куб.

```
// Использование команд форматирования.

using System;

class DisplayOptions {
  public static void Main() {
    int i;

    Console.WriteLine("Число\tКвадрат\tКуб");

    for(i = 1; i < 10; i++)
      Console.WriteLine("{0}\t{1}\t{2}",
                        i, i*i, i*i*i);
  }
}
```

Вот как выглядит результат выполнения этой программы:

```
Число   Квадрат Куб
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
```

В предыдущих примерах программ выводимые значения не форматировались. Конечно же, спецификаторы формата позволяют управлять характером их отображения. Обычно форматируются десятичные значения и значения с плавающей точкой. Самый простой способ задать формат — описать шаблон, которым будет пользоваться метод WriteLine(). Для этого рассмотрим пример форматирования с помощью символов "#", отмечающих позиции цифр. При этом можно указать расположение деся-

тичной точки и запятых, которые используются в качестве разделителей групп разрядов. Выше мы приводили пример отображения частного от деления числа 10 на 3. Теперь рассмотрим еще один вариант вывода результата выполнения этой арифметической операции.

```
Console.WriteLine(
        "При делении 10/3 получаем: {0:#.##}", 10.0/3.0);
```

Теперь результат выглядит по-другому:

```
При делении 10/3 получаем: 3.33
```

В этом примере шаблон имеет вид `#.##`, что для метода `WriteLine()` служит указанием отобразить лишь два десятичных разряда. Но важно понимать, что при необходимости слева от десятичной точки будет отображено столько цифр, сколько потребуется, чтобы не исказить значение.

А вот еще пример. При выполнении инструкции

```
Console.WriteLine("{0:###,###.##}", 123456.56);
```

будет сгенерирован следующий результат:

```
123,456.56
```

Если нужно отобразить значение в формате представления долларов и центов, используйте спецификатор формата `C`. Вот пример:

```
decimal balance;

balance = 12323.09m;
Console.WriteLine("Текущий баланс равен {0:C}, balance);
```

Результат выполнения этой последовательности инструкций выглядит так:

```
Текущий баланс равен $12,323.09
```

Формат `C` можно использовать для улучшения представления результата выполнения программы вычисления цены со скидкой, которая рассматривалась выше.

```
/*
   Использование спецификатора формата C для вывода
   значений в виде долларов и центов.
*/

using System;

class UseDecimal {
  public static void Main() {
    decimal price;
    decimal discount;
    decimal discounted_price;

    // Вычисляем цену со скидкой.
    price = 19.95m;
    discount = 0.15m; // Ставка дисконта равна 15%.

    discounted_price = price - ( price * discount);

    Console.WriteLine("Цена со скидкой: {0:C}",
                      discounted_price);
  }
}
```

Посмотрите, как теперь выглядит результат выполнения программы, и сравните его с предыдущим:

```
Цена со скидкой: $16.96
```

Литералы

В C# *литералами* называются фиксированные значения, представленные в понятной форме. Например, число `100` — это литерал. Литералы также называют *константами*. По большей части применение литералов происходит на интуитивном уровне, и поэтому мы без особых пояснений использовали их в той или иной форме во всех предыдущих примерах программ. Теперь настало время объяснить их формально.

C#-литералы могут иметь любой тип значений. Способ их представления зависит от их типа. Как упоминалось выше, символьные константы заключаются между двумя одинарными кавычками. Например, как `'a'`, так и `'%'` — символьные константы.

Целочисленные литералы задаются как числа без дробной части. Например, `10` и `-100` — это целочисленные константы. Константы с плавающей точкой должны обязательно иметь десятичную точку, а за ней — дробную часть числа. Примером константы с плавающей точкой может служить число `11.123`. Для вещественных чисел C# позволяет также использовать экспоненциальное представление (в виде мантиссы и порядка).

Поскольку C# — строго типизированный язык, литералы в нем также имеют тип. Отсюда сразу возникает вопрос: как определить тип литерала? Например, какой тип имеют такие литералы, как `12`, `123987` или `0.23`? К счастью, C# определяет несколько простых правил, позволяющих ответить на эти вопросы.

Во-первых, что касается целочисленных литералов, то им присваивается наименьший целочисленный тип, который сможет его хранить, начиная с типа `int`. Таким образом, целочисленный литерал, в зависимости от конкретного значения, может иметь тип `int`, `uint`, `long` или `ulong`. Во-вторых, все литералы с плавающей точкой имеют тип `double`.

Если тип, задаваемый по умолчанию в языке C#, не соответствует вашим намерениям в отношении типа конкретного литерала, вы можете явно определить его с помощью нужного суффикса. Чтобы задать литерал типа `long`, присоедините к его концу букву `l` или `L`. Например, если значение `12` автоматически приобретает тип `int`, но значение `12L` имеет тип `long`. Чтобы определить целочисленное значение без знака, используйте суффикс `u` или `U`. Так, если значение `100` имеет тип `int`, но значение `100U` — тип `uint`. Для задания длинного целого без знака используйте суффикс `ul` или `UL` (например, значение `987654UL` будет иметь тип `ulong`).

Чтобы задать литерал типа `float`, используйте суффикс `f` или `F` (например, `10.19F`).

Чтобы задать литерал типа `decimal`, используйте суффикс `m` или `M` (например, `9.95M`).

Несмотря на то что целочисленные литералы создают `int`-, `uint`-, `long`- или `ulong`-значения по умолчанию, их тем не менее можно присваивать переменным типа `byte`, `sbyte`, `short` или `ushort`, если, конечно, они могут быть представлены соответствующим типом.

Шестнадцатеричные литералы

Вероятно, вам известно, что в программировании вместо десятичной иногда удобнее использовать систему счисления по основанию 16, которая называется *шестнадцатеричной*. В ней используются цифры от 0 до 9 и буквы от A до F, которые служат для обозначения шестнадцатеричных "цифр" 10, 11, 12, 13, 14 и 15. Например, число 10 в шестнадцатеричной системе равно десятичному числу 16. Язык C#, как и многие другие языки программирования, позволяет задавать целочисленные константы в ше-

стнадцатеричном формате. Шестнадцатеричный литерал должен начинаться с пары символов 0x (нуля и буквы "x"). Приведем несколько примеров.

```
count = 0xFF;  // 255 в десятичной системе
incr = 0x1a;   // 26 в десятичной системе
```

Управляющие последовательности символов

Среди множества символьных констант, образующихся в результате заключения символов в одинарные кавычки, помимо печатных символов есть такие (например, символ возврата каретки), которые создают проблему при использовании текстовых редакторов. Некоторые символы, например одинарная или двойная кавычка, имеют в C# специальное значение, поэтому их нельзя использовать непосредственно. По этим причинам в C# предусмотрено несколько управляющих последовательностей символов (ESC-последовательностей), перечисленных в табл. 3.3. Эти последовательности используются вместо символов, которых они представляют.

Например, следующая инструкция присваивает переменной ch символ табуляции:

```
ch = '\t';
```

А эта инструкция присваивает переменной ch символ одинарной кавычки:

```
ch = '\'';
```

Таблица 3.3. Управляющие последовательности символов

ESC-последовательность	*Описание*
\a	Звуковой сигнал (звонок)
\b	Возврат на одну позицию
\f	Подача страницы (для перехода к началу следующей страницы)
\n	Новая строка
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\0	Нуль-символ
\'	Одинарная кавычка (апостроф)
\"	Двойная кавычка
\\	Обратная косая черта

Строковые литералы

Язык C# поддерживает еще один тип литерала: строковый. *Строка* — это набор символов, заключенных в двойные кавычки. Например, фрагмент кода

```
"Это тест"
```

представляет собой строку. В предыдущих фрагментах программ (а именно в инструкциях вызова метода WriteLine()) вы видели другие примеры строк.

Помимо обычных символов, строковый литерал может содержать одну или несколько управляющих последовательностей. Рассмотрим, например, следующую программу. В ней используются такие ESC-последовательности, как \n, \t и \".

```
// Использование ESC-последовательностей в строках.

using System;
```

```
class StrDemo {
 public static void Main() {
   Console.WriteLine(
          "Первая строка\nВторая строка\nТретья строка");
   Console.WriteLine("Один\tДва\tТри");
   Console.WriteLine("Четыре\tПять\tШесть");

   // Вставляем кавычки.
   Console.WriteLine("\"Зачем?\", спросил он.");
 }
}
```

Вот что получаем в результате:

```
Первая строка
Вторая строка
Третья строка
Один    Два     Три
Четыре  Пять    Шесть
"Зачем?", спросил он.
```

Обратите внимание на то, как управляющая последовательность \n используется для перехода на новую строку, благодаря чему не нужно многократно вызывать метод WriteLine() для организации выводимых данных на нескольких строках. В те позиции, где необходимо сделать переход на новую строку, достаточно вставить ESC-последовательность \n. Обратите также внимание на то, как в выводимых строках обеспечивается наличие двойных кавычек (с помощью ESC-последовательности \").

Помимо формы только что описанного строкового литерала можно также определить *буквальный* (verbatim) строковый литерал. Буквальный строковый литерал 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 или более строк. Таким образом, можно переходить на новую строку, использовать табуляцию и пр., не прибегая к помощи управляющих последовательностей. Единственное исключение составляет двойная кавычка ("). Чтобы получить в выходных данных двойную кавычку, в буквальном строковом литерале необходимо использовать две подряд двойные кавычки (""). А теперь обратимся к программе, в которой демонстрируется использование буквального строкового литерала.

```
// Демонстрация буквальных строковых литералов.

using System;

class Verbatim {
  public static void Main() {
    Console.WriteLine(@"Это буквальный
строковый литерал,
который занимает несколько строк.
");
    Console.WriteLine(@"А теперь воспользуемся табуляцией:
1  2   3   4
5  6   7   8
");
    Console.WriteLine(
          @"Отзыв программиста: ""Мне нравится C#.""");
  }
}
```

Вот что сгенерирует эта программа:

```
Это буквальный
строковый литерал,
который занимает несколько строк.

А теперь воспользуемся табуляцией:
1       2       3       4
5       6       7       8

Отзыв программиста: "Мне нравится C#."
```

Здесь важно отметить, что буквальные строковые литералы отображаются точно так, как они введены в программе. Они позволяют программисту так формировать выходные данные, как они будут отображены на экране. Но в случае многострочного вывода переход на следующую строку нарушит систему формирования отступов в программе. Поэтому буквальные строковые литералы не слишком часто используются в программах этой книги, хотя во многих случаях форматирования данных они оказываются хорошим подспорьем.

И последнее. Не путайте строки с символами. Символьный литерал (например, `'X'`) представляет одиночную букву типа `char`. А строка, хотя и содержащая всего одну букву (например, `"X"`), это все-таки строка.

Рассмотрим переменные поближе

Как вы узнали в главе 2, для объявления переменной необходимо использовать инструкцию следующего формата:

тип имя_переменной;

Здесь с помощью элемента *тип* задается тип объявляемой переменной, а с помощью элемента *имя_переменной* — ее имя. Можно объявить переменную любого допустимого типа. При создании переменной создается экземпляр соответствующего типа. Таким образом, возможности переменной определяются ее типом. Например, переменную типа `bool` нельзя использовать для хранения значений с плавающей точкой. Более того, тип переменной невозможно изменить во время ее существования. Например, переменную типа `int` нельзя преобразовать в переменную типа `char`.

Все переменные в C# должны быть объявлены до их использования. Это — требование компилятора, поскольку, прежде чем скомпилировать надлежащим образом инструкцию, в которой используется переменная, он должен "знать" тип содержащейся в ней информации. "Знание" типа также позволяет C# осуществлять строгий контроль типов.

Помимо типов переменные различаются и другими качествами. Например, переменные, которые мы использовали в примерах программ до сих пор, называются *локальными*, поскольку они объявляются внутри метода.

Инициализация переменной

Переменная до использования должна получить значение. Это можно сделать с помощью инструкции присваивания. Можно также присвоить переменной начальное значение одновременно с ее объявлением. Для этого достаточно после имени переменной поставить знак равенства и указать присваиваемое значение. Общий формат инициализации переменной имеет такой вид:

тип имя_переменной = значение;

Здесь, как нетрудно догадаться, элемент *значение* — это начальное значение, которая получает переменная при создании. Значение инициализации должно соответствовать заданному типу переменной.

Вот несколько примеров:

```
int count = 10;   // Присваиваем переменной count
                  // начальное значение 10.
char ch = 'X';    // Инициализируем ch буквой X.
float f = 1.2F    // Переменная f инициализируется
                  // числом 1.2.
```

При объявлении двух или более переменных одного типа с помощью списка (с разделением элементов списка запятыми) одной или нескольким из этих переменных можно присвоить начальные значения. Например, в инструкции

```
int a, b = 8, c = 19, d;  // Переменные b и c
                          // инициализируются числами.
```

Динамическая инициализация

Хотя в предыдущих примерах в качестве инициализаторов были использованы только константы, C# позволяет инициализировать переменные динамически, с помощью любого выражения, действительного на момент объявления переменной. Рассмотрим, например, короткую программу, которая вычисляет гипотенузу прямоугольного треугольника, заданного длинами двух противоположных сторон.

```
// Демонстрация динамической инициализации.

using System;

class DynInit {
  public static void Main() {
    double s1 = 4.0, s2 = 5.0; // Длины сторон.

    // Динамически инициализируем переменную hypot.
    double hypot = Math.Sqrt( (s1 * s1) + (s2 * s2) );

    Console.Write("Гипотенуза треугольника со сторонами " +
                  s1 + " и " + s2 + " равна ");

    Console.WriteLine("{0:#.###}.", hypot);

  }
}
```

Результат выполнения этой программы имеет такой вид:

```
Гипотенуза треугольника со сторонами 4 и 5 равна 6.403.
```

Здесь объявлены три локальные переменные: `s1`, `s2` и `hypot`. Первые две (`s1` и `s2`) инициализируются константами, а третья, `hypot`, инициализируется динамически результатом вычисления гипотенузы по двум катетам. Обратите внимание на то, что инициализация включает вызов метода `Math.Sqrt()`. Как уже было сказано, для инициализации переменной можно использовать любое выражение, действительное на момент ее объявления. Поскольку вызов метода `Math.Sqrt()` (как и любого другого библиотечного метода) действителен в этой точке программы, его вполне можно использовать для инициализации переменной `hypot`. Здесь важно то, что в выражении инициализации можно использовать любой элемент, действительный на момент инициализации, включая вызовы методов, другие переменные или литералы.

Область видимости и время существования переменных

До сих пор все переменные, с которыми мы имели дело, объявлялись в начале метода `Main()`. Однако в C# разрешается объявлять переменные внутри любого блока. Блок начинается открывающей, а завершается закрывающей фигурными скобками. Любой блок определяет *область объявления*, или *область видимости* (scope) объектов. Таким образом, при создании блока создается и новая область видимости, которая определяет, какие объекты видимы для других частей программы. Область видимости также определяет время существования этих объектов.

Самыми важными в C# являются области видимости, которые определены классом и методом. Область видимости класса (и переменные, объявленные внутри нее) мы рассмотрим позже, когда доберемся до описания классов, а пока затронем области видимости, определяемые методами.

Область видимости, определяемая методом, начинается с открывающей фигурной скобки. Но если метод имеет параметры, они также относятся к области видимости метода.

Как правило, переменные, объявленные в некоторой области видимости, невидимы (т.е. недоступны) для кода, который определяется вне этой области видимости. Таким образом, при объявлении переменной внутри области видимости вы локализируете ее и защищаете от неправомочного доступа и/или модификации. Эти правила области видимости обеспечивают основу для инкапсуляции.

Области видимости могут быть вложенными. Например, при каждом создании программного блока создается новая вложенная область видимости. В этом случае внешняя область включает внутреннюю. Это означает, что объекты, объявленные внутри внешней области, будут видимы для кода внутренней области. Но обратное утверждение неверно: объекты, объявленные во внутренней области, невидимы вне ее.

Чтобы лучше понять суть вложенных областей видимости, рассмотрим следующую программу:

```
// Демонстрация области видимости блока.

using System;

class ScopeDemo {
  public static void Main() {
    int x; // Переменная x известна всему коду в пределах
           // метода Main().

    x = 10;
    if(x == 10) { // Начало новой области видимости.
      int y = 20; // Переменная y известна только
                  // этому блоку.

      // Здесь известны обе переменные x и y.
      Console.WriteLine("x и y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Ошибка! Переменная y здесь неизвестна.

    // Переменная x здесь известна.
    Console.WriteLine("Значение x равно " + x);
  }
}
```

Как утверждается в комментариях, переменная x объявляется в начале области видимости метода Main() и потому доступна всему последующему коду метода. Внутри блока инструкции if объявляется переменная y. А поскольку блок определяет область видимости, то переменная y видима только коду внутри этого блока. Поэтому, находясь вне этого блока, программная строка

```
// y = 100; // Ошибка! Переменная y здесь неизвестна.
```

оформлена как комментарий. Если убрать символ комментария, компилятор выдаст сообщение об ошибке, поскольку переменная y невидима вне if-блока. Переменную x можно свободно использовать и внутри if-блока, поскольку внутренний код этого блока (т.е. код во вложенной области видимости) имеет доступ к переменным, объявленным вне его.

Внутри блока переменные можно объявлять в любой точке, но действительными они становятся только после объявления. Таким образом, если объявить переменную в начале метода, она будет доступна всему коду этого метода. И наоборот, если объявить переменную в конце метода, она будет попросту бесполезной ввиду отсутствия кода, который мог бы ее использовать.

Переменные создаются после входа в их область видимости, а разрушаются при выходе из нее. Это означает, что переменная не будет хранить значение за пределами области видимости. Таким образом, переменная, объявленная внутри некоторого метода, не будет хранить значение между вызовами этого метода. И точно так же переменная, объявленная внутри некоторого блока, потеряет свое значение по завершении этого блока. Следовательно, время существования переменной ограничивается ее областью видимости.

Если объявление переменной включает инициализатор, такая переменная будет повторно инициализироваться при каждом входе в блок, в котором она объявляется. Рассмотрим, например, следующую программу:

```
// Демонстрация времени существования переменной.

using System;

class VarInitDemo {
  public static void Main() {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1; // Переменная y инициализируется при
                  // каждом входе в программный блок.
      Console.WriteLine("Значение y равно: " + y); // Здесь
                                        // всегда выводится -1.
      y = 100;
      Console.WriteLine("Теперь значение y равно: " + y);
    }
  }
}
```

Вот какие результаты генерирует эта программа:

```
Значение y равно: -1
Теперь значение y равно: 100
Значение y равно: -1
Теперь значение y равно: 100
Значение y равно: -1
Теперь значение y равно: 100
```

Как видите, при каждом входе в цикл `for` переменная `y` неизменно принимает значение −1. Несмотря на последующее присваивание ей значения 100, она это значение теряет.

В правилах действия областей видимости есть одна деталь: хотя блоки могут быть вложенными, ни одна переменная, объявленная во внутренней области видимости, не может иметь имя, совпадающее с именем переменной, объявленной во внешней области видимости. Например, следующая программа из-за попытки объявить две отдельные переменные с одинаковыми именами скомпилирована не будет.

```
/*
   Здесь делается попытка объявить переменную во
   внутренней области видимости с таким же именем, как у
   переменной, определенной во внешней области видимости.

   *** Эта программа не будет скомпилирована. ***
*/

using System;

class NestVar {
  public static void Main() {
    int count;

    for(count = 0; count < 10; count = count+1) {
      Console.WriteLine("This is count: " + count);

      int count; // Неверно!!!
      for(count = 0; count < 2; count++)
        Console.WriteLine("В этой программе есть ошибка!");
    }
  }
}
```

Если вы до этого программировали на C/C++, вам должно быть известно, что на имена, объявляемые во внутренней области видимости, никакие ограничения не накладываются. Таким образом, в языках C/C++ объявление переменной `count` внутри блока внешнего цикла `for` было бы совершенно законным. Однако при всей своей законности такое объявление скрывает внешнюю переменную. Поэтому разработчики C#, зная, что подобное *сокрытие имен* может легко привести к ошибкам программирования, решили запретить его.

Преобразование и приведение типов

В программировании переменной одного типа часто присваивается значение переменной другого типа. Например, как показано в следующем фрагменте программы, мы могли бы присвоить переменной типа `float` значение типа `int`.

```
int i;
float f;

i = 10;
f = i;  // float-переменной присваивается int-значение.
```

Если в инструкции присваивания смешиваются совместимые типы, значение с правой стороны (от оператора присваивания) автоматически преобразуется в значение "левостороннего" типа. Таким образом, в предыдущем фрагменте программы значе-

ние, хранимое в int-переменной i, преобразуется в значение типа float, а затем присваивается переменной f. Но, поскольку в C# не все типы совместимы и действует строгий контроль типов, не все преобразования типов разрешены в неявном виде. Например, типы bool и int несовместимы. Тем не менее с помощью операции *приведения типов* все-таки возможно выполнить преобразование между несовместимыми типами. Приведение типов — это выполнение преобразования типов в явном виде.

Автоматическое преобразование типов

При присвоении значения одного типа данных переменной другого типа будет выполнено автоматическое преобразование типов, если

- эти два типа совместимы;
- тип приемника больше (т.е. имеет больший диапазон представления чисел), чем тип источника.

При соблюдении этих двух условий выполняется *преобразование с расширением*, или *расширяющее преобразование*. Например, тип int — достаточно "большой" тип, чтобы сохранить любое допустимое значение типа byte, а поскольку как int, так и byte — целочисленные типы, здесь может быть применено автоматические преобразование.

Для расширяющих преобразований числовые типы, включая целочисленные и с плавающей точкой, совместимы один с другим. Например, следующая программа совершенно легальна, поскольку преобразование типов из long в double является расширяющим, которое выполняется автоматически.

```
// Демонстрация автоматического преобразования типов
// из long в double.

using System;

class LtoD {
  public static void Main() {
    long L;
    double D;

    L = 100123285L;
    D = L;

    Console.WriteLine("L и D: " + L + " " + D);
  }
}
```

Несмотря на возможность автоматического преобразования типов из long в double, обратное преобразование типов (из double в long) автоматически не выполняется, поскольку это преобразование не является расширяющим. Таким образом, следующая версия предыдущей программы недопустима:

```
// *** Эта программа не будет скомпилирована. ***

using System;

class LtoD {
  public static void Main() {
    long L;
    double D;

    D = 100123285.0;
    L = D; // Неверно!!!
```

```
        Console.WriteLine("L и D: " + L + " " + D);

    }
}
```

Помимо только что описанных ограничений не существует автоматического преобразования между типом `decimal` и `float` (или `double`), а также из числовых типов в тип `char` (или `bool`). Кроме того, несовместимы и типы `char` и `bool` .

Приведение несовместимых типов

Несмотря на большую пользу автоматического преобразования типов оно не в состоянии удовлетворить все нужды программирования, поскольку реализуется только при расширяющем преобразовании между совместимыми типами. Во всех остальных случаях приходится применять приведение к типу. *Приведение к типу* — это явно заданная инструкция компилятору преобразовать один тип в другой. Инструкция приведения записывается в следующей общей форме:

(*тип_приемника*) *выражение*

Здесь элемент *тип_приемника* определяет тип для преобразования заданного выражения. Например, если вам нужно, чтобы выражение `x/y` имело тип `int`, напишите следующие программные инструкции:

```
double x, y;
// ...
(int) (x / y);
```

В этом фрагменте кода, несмотря на то, что переменные `x` и `y` имеют тип `double`, результат вычисления заданного выражения приводится к типу `int`. Круглые скобки, в которые заключено выражение `x / y`, обязательны. В противном случае (без круглых скобок) операция приведения к типу `int` была бы применена только к значению переменной `x`, а не к результату деления. Для получения результата желаемого типа здесь не обойтись без операции приведения, поскольку автоматического преобразования из типа `double` в `int` не существует.

Если приведение приводит к *сужающему преобразованию*, возможна потеря информации. Например, в случае приведения типа `long` к типу `int` информация будет утеряна, если значение типа `long` больше максимально возможного числа, которое способен представить тип `int`, поскольку будут "усечены" старшие разряды `long`-значения. При выполнении операции приведения типа с плавающей точкой к целочисленному будет утеряна дробная часть простым ее отбрасыванием. Например, при присвоении переменной целочисленного типа числа 1,23 в действительности будет присвоено число 1. Дробная часть (0,23) будет утеряна.

В следующей программе демонстрируется ряд преобразований типов, которые требуют приведения типов, причем в некоторых ситуациях приведение вызывает потерю данных.

```
// Демонстрация приведения типов.

using System;

class CastDemo {
  public static void Main() {
    double x, y;
    byte b;
    int i;
    char ch;
    uint u;
```

```
    short s;
    long l;

    x = 10.0;
    y = 3.0;

    // Приведение типа double к типу int.
    i = (int) (x / y); // Дробная часть теряется.
    Console.WriteLine(
          "Целочисленный результат деления x / y: " + i);
    Console.WriteLine();

    // Приведение типа int к типу byte без потери данных.
    i = 255;
    b = (byte) i;
    Console.WriteLine("b после присваивания 255: " + b +
                      " -- без потери данных.");

    // Приведение типа int к типу byte с потерей данных
    i = 257;
    b = (byte) i;
    Console.WriteLine("b после присваивания 257: " + b +
                      " -- с потерей данных.");
    Console.WriteLine();

    // Приведение типа uint к типу short без потери данных.
    u = 32000;
    s = (short) u;
    Console.WriteLine("s после присваивания 32000: " + s +
                      " -- без потери данных.");

    // Приведение типа uint к типу short с потерей данных.
    u = 64000;
    s = (short) u;
    Console.WriteLine("s после присваивания 64000: " + s +
                      " -- с потерей данных.");
    Console.WriteLine();

    // Приведение типа long к типу uint без потери данных.
    l = 64000;
    u = (uint) l;
    Console.WriteLine("u после присваивания 64000: " + u +
                      " -- без потери данных.");

    // Приведение типа long к типу uint с потерей данных.
    l = -12;
    u = (uint) l;
    Console.WriteLine("u после присваивания -12: " + u +
                      " -- с потерей данных.");
    Console.WriteLine();

    // Приведение типа byte к типу char.
    b = 88; // ASCII-код для буквы X.
    ch = (char) b;
    Console.WriteLine("ch после присваивания 88: " + ch);
  }
}
```

Результаты выполнения этой демонстрационной программы имеют такой вид:

```
Целочисленный результат деления x / y: 3

b после присваивания 255: 255 -- без потери данных.
b после присваивания 257: 1 -- с потерей данных.

s после присваивания 32000: 32000 -- без потери данных.
s после присваивания 64000: -1536 -- с потерей данных.

u после присваивания 64000: 64000 -- без потери данных.
u после присваивания -12: 4294967284 -- с потерей данных.

ch после присваивания 88: X
```

Теперь рассмотрим каждую инструкцию присваивания отдельно. Приведение результата деления (`x / y`) к типу `int` приводит к усечению дробной части, т.е. к потере информации.

Однако никакой потери информации не происходит, если переменной `b` присваивается значение 255, поскольку переменная типа `byte` в состоянии хранить число 255. Но при попытке присвоить переменной `b` число 257 информация будет потеряна, поскольку число 257 находится за пределами диапазона представления чисел для типа `byte`. В обоих этих случаях без операции приведения типов не обойтись, поскольку автоматическое преобразование типа `int` в тип `byte` невозможно.

В случае присвоения переменной `s` типа `short` значения 32 000 (с помощью переменной `u` типа `uint`) данные не теряются, потому что `short`-переменная может хранить число 32 000. Но следующее присвоение уже не такое успешное, поскольку число 64 000 находится за пределами диапазона представления чисел для типа `short`, и эта ситуация сопровождается потерей данных. В обоих этих случаях без операции приведения типов также не обойтись, поскольку автоматическое преобразование типа `uint` в тип `short` невозможно.

Затем в программе переменной `u` (типа `uint`) присваивалось значение 64 000 (с помощью переменной `l` типа `long`). Эта инструкция была выполнена без потери данных, поскольку число 64 000 находится в пределах `uint`-диапазона. Но попытка присвоить той же переменной `u` число −12, конечно, привела к потере данных, так как тип `uint` не предназначен для хранения отрицательных чисел. И в этих обоих случаях без операции приведения типов не обойтись, поскольку автоматическое преобразование типа `long` в тип `uint` невозможно.

Наконец, присваивание `byte`-значения переменной типу `char` обходится "без жертв", т.е. без потери информации, но здесь также необходимо применять операцию приведения типов.

Преобразование типов в выражениях

Преобразование типов встречается не только в инструкциях присваивания, но и в выражениях. В выражениях можно смешивать различные типы данных, если они совместимы. Например, можно смешивать типы `short` и `long`, поскольку это числовые типы. При смешении различных типов в одном выражении все его составляющие преобразуются к одному типу, причем это происходит по мере перехода от одной операции к другой.

Преобразование типов выполняется на основе *правил продвижения* по "типовой" лестнице. Для бинарных операций действует следующий алгоритм.

ЕСЛИ один операнд имеет тип decimal, ТО и второй "возводится в ранг", т.е. "в тип" decimal (но если второй операнд имеет тип float или double, результат будет ошибочным).

ЕСЛИ один операнд имеет тип double, ТО и второй преобразуется в значение типа double.

ЕСЛИ один операнд имеет тип float, ТО и второй преобразуется в значение типа float.

ЕСЛИ один операнд имеет тип ulong, ТО и второй преобразуется в значение типа ulong (но если второй операнд имеет тип sbyte, short, int или long, результат будет ошибочным).

ЕСЛИ один операнд имеет тип long, ТО и второй преобразуется в значение типа long.

ЕСЛИ один операнд имеет тип uint, а второй имеет тип sbyte, short или int, ТО оба операнда преобразуются в значения типа long.

ЕСЛИ один операнд имеет тип uint, ТО и второй преобразуется в значение типа uint.

ИНАЧЕ оба операнда преобразуются в значения типа int.

Относительно правил продвижения по "типовой" лестнице необходимо сделать несколько замечаний. Во-первых не все типы можно смешивать в одном выражении. Например, не выполняется неявное преобразование значения типа float или double в значение типа decimal. Нельзя также смешивать тип ulong и целочисленный тип со знаком. Чтобы все-таки объединить эти несовместимые типы в одном выражении, необходимо использовать в явном виде операцию приведения типов.

Во-вторых, уделите особое внимание последнему правилу. Оно утверждает, что все операнды будут преобразованы в значения типа int, если не было применено ни одно их предыдущих правил. Следовательно, в выражении все char-, sbyte-, byte-, ushort- и short-значения будут в процессе вычислений преобразованы в значения типа int. Такое "поголовное" int-преобразование называют *целочисленным продвижением типа* (integer promotion). Следствием этого алгоритма является то, что результат всех арифметических операций будет иметь тип по "званию" не ниже int.

Важно понимать, что продвижение типов применяется только к значениям, используемым при вычислении выражения. Например, хотя значение переменной типа byte внутри выражения будет "подтянуто" до типа int, вне выражения эта переменная по-прежнему имеет тип byte. Правило продвижения типов действует только при вычислении выражения.

Однако продвижение типов может иметь неожиданные последствия. Например, предположим, что арифметическая операция включает два byte-значения. Тогда выполняется следующая последовательность действий. Сначала byte-операнды "подтягиваются" до типа int, затем вычисляется результат операции, который имеет тип int. Выходит, после выполнения операции над двумя byte-операндами вместо ожидаемого byte-результата мы получим int-значение. Именно такая неожиданность и может иметь место. А теперь рассмотрим такую программу:

```
// Сюрприз в результате продвижения типов!

using System;

class PromDemo {
  public static void Main() {
    byte b;
```

```
    b = 10;
    b = (byte) (b * b); // Необходимо приведение типов!!

    Console.WriteLine("b: "+ b);
  }
}
```

Кажется странным, что при присвоении результата произведения `b * b` переменной `b` необходимо выполнять операцию приведения типов. Дело в том, что в выражении `b * b` значение переменной `b` "подтягивается" до типа `int`, т.е. результат выражения `b * b` представляет собой `int`-значение, которое нельзя присвоить `byte`-переменной без приведения типов. Имейте это в виду, если вдруг получите сообщение об ошибке, где сказано о несовместимости типов для выражений, в которых, казалось бы, все в полном порядке.

Ситуация подобного рода встречается также при выполнении операций над операндами типа `char`. Например, в следующем фрагменте кода также необходимо "возвратить" результат к исходному типу из-за автоматического преобразования `char`-операндов к типу `int` внутри вычисляемого выражения.

```
char ch1 = 'a', ch2 = 'b';

ch1 = (char) (ch1 + ch2);
```

Без приведения типов результат сложения операндов `ch1` и `ch2` имел бы тип `int`, а `int`-значение невозможно присвоить `char`-переменной.

Продвижение типов также имеет место при выполнении унарных операций (например, с унарным минусом). Операнды унарных операций, тип которых по диапазону меньше типа `int` (т.е. `sbyte`-, `byte`-, `short`- и `ushort`-значения), "подтягиваются" к типу `int`. То же происходит с операндом типа `char`. Более того, при выполнении операции отрицания `uint`-значения результат приобретает тип `long`.

Приведение типов в выражениях

Операцию приведения типов можно применить не ко всему выражению, а к конкретной его части. Это позволяет более тонко управлять преобразованием типов при вычислении выражения. Рассмотрим, например, следующую программу. Она отображает значения квадратных корней из чисел от 1 до 10. Она также выводит по отдельности целую и дробную части каждого результата. Для этого в программе используется операция приведения типов, которая позволяет преобразовать результат вызова метода `Math.Sqrt()` в значение типа `int`.

```
// Приведение типов в выражениях.

using System;

class CastExpr {
  public static void Main() {
    double n;

     for(n = 1.0; n <= 10; n++) {
       Console.WriteLine(
                     "Квадратный корень из {0} равен {1}",
                     n, Math.Sqrt(n));

       Console.WriteLine("Целая часть числа: {0}" ,
                         (int) Math.Sqrt(n));

       Console.WriteLine(
```

```
                    "Дробная часть числа: {0}",
                    Math.Sqrt(n) - (int) Math.Sqrt(n) );
        Console.WriteLine();
    }

  }
}
```

Вот как выглядят результаты выполнения этой программы:

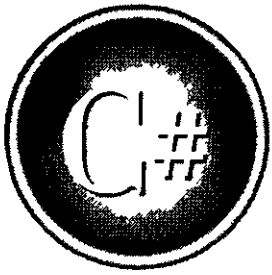
```
Квадратный корень из 1 равен 1
Целая часть числа: 1
Дробная часть числа: 0

Квадратный корень из 2 равен 1.4142135623731
Целая часть числа: 1
Дробная часть числа: 0.414213562373095

Квадратный корень из 3 равен 1.73205080756888
Целая часть числа: 1
Дробная часть числа: 0.732050807568877

Квадратный корень из 4 равен 2
Целая часть числа: 2
Дробная часть числа: 0

Квадратный корень из 5 равен 2.23606797749979
Целая часть числа: 2
Дробная часть числа: 0.23606797749979

Квадратный корень из 6 равен 2.44948974278318
Целая часть числа: 2
Дробная часть числа: 0.449489742783178

Квадратный корень из 7 равен 2.64575131106459
Целая часть числа: 2
Дробная часть числа: 0.645751311064591

Квадратный корень из 8 равен 2.82842712474619
Целая часть числа: 2
Дробная часть числа: 0.82842712474619

Квадратный корень из 9 равен 3
Целая часть числа: 3
Дробная часть числа: 0

Квадратный корень из 10 равен 3.16227766016838
Целая часть числа: 3
Дробная часть числа: 0.16227766016838
```

Как видно из результатов выполнения программы, приведение значения, возвращаемого методом `Math.Sqrt()`, к типу `int`, позволяет получить целую часть значения. А его дробную часть мы получаем в результате вычисления следующего выражения (если из вещественного числа вычесть его целую часть, то результат даст дробную часть исходного числа):

```
Math.Sqrt(n) - (int) Math.Sqrt(n)
```

Результат этого выражения имеет тип `double`. Здесь к типу `int` приводится только результат второго вызова метода `Math.Sqrt()`.

Полный справочник по



Глава 4

Операторы

В C# предусмотрен широкий набор операторов, которые дают в руки программисту мощные рычаги управления при создании разнообразнейших выражений и их вычислении. В C# имеется четыре общих класса операторов: арифметические, поразрядные, логические и операторы отношений. Помимо них в этой главе рассматриваются оператор присвоения и оператор ?. В C# определены также операторы для обработки специальных ситуаций, но их мы рассмотрим после изучения средств, к которым они применяются.

Арифметические операторы

В C# определены следующие арифметические операторы.

Оператор	*Действие*
+	Сложение
–	Вычитание, унарный минус
*	Умножение
/	Деление
%	Деление по модулю
--	Декремент
++	Инкремент

Действие C#-операторов +, –, * и / совпадает с действием аналогичных операторов в любом другом языке программирования (да и в алгебре, если уж на то пошло). Их можно применять к данным любого встроенного числового типа.

Хотя действия арифметических операторов хорошо известны всем читателям, существуют ситуации, которые наверняка потребуют специальных разъяснений. Прежде всего хочу напомнить, что после применения оператора деления (/) к целому числу остаток будет отброшен. Например, результат целочисленного деления 10/3 будет равен 3. Остаток от деления можно получить с помощью *оператора деления по модулю* (%). Этот оператор работает практически так же, как в других языках программирования: возвращает остаток от деления нацело. Например, 10 % 3 равно 1. В C# оператор % можно применить как к целочисленным типам, так и типам с плавающей точкой. Например, 10,0 % 3,0 также равно 1. (В языках C/C++ операции деления по модулю применимы только к целочисленным типам.) Использование оператора деления по модулю демонстрируется в следующей программе.

```
// Демонстрация использования оператора %.

using System;

class ModDemo {
  public static void Main() {
    int iresult, irem;
    double dresult, drem;

    iresult = 10 / 3;
    irem = 10 % 3;

    dresult = 10.0 / 3.0;
    drem = 10.0 % 3.0;
```

```
        Console.WriteLine(
              "Результат и остаток от деления 10 / 3: " +
              iresult + " " + irem);
        Console.WriteLine(
              "Результат и остаток от деления 10.0 / 3.0: " +
              dresult + " " + drem);
    }
}
```

Результат выполнения этой программы таков:

```
Результат и остаток от деления 10 / 3: 3 1
Результат и остаток от деления 10.0 / 3.0: 3.33333333333333 1
```

Как видите, оператор % генерирует остаток, равный 1, как при делении целочисленных значений, так и значений с плавающей точкой.

Инкремент и декремент

Операторы инкремента (++) и декремента (--) увеличивают и уменьшают значение операнда на единицу, соответственно. Как будет показано ниже, эти операторы обладают специальными свойствами, которые делают их весьма интересными для рассмотрения.

Итак, оператор инкремента выполняет сложение операнда с числом 1, а оператор декремента вычитает 1 из своего операнда. Это значит, что инструкция

```
x = x + 1;
```

аналогична такой инструкции:

```
x++;
```

Точно так же инструкция

```
x = x - 1;
```

аналогична такой инструкции:

```
x--;
```

Операторы инкремента и декремента могут стоять как перед своим операндом, так и после него. Например, инструкцию

```
x = x + 1;
```

можно переписать в виде префиксной формы

```
++x;  // Префиксная форма оператора инкремента.
```

или в виде постфиксной формы:

```
x++;  // Постфиксная форма оператора инкремента.
```

В предыдущем примере не имело значения, в какой форме был применен оператор инкремента: префиксной или постфиксной. Но если оператор инкремента или декремента используется как часть большего выражения, то форма его применения имеет важное значение. Если такой оператор применен в префиксной форме, то C# сначала выполнит эту операцию, чтобы операнд получил новое значение, которое затем будет использовано остальной частью выражения. Если же оператор применен в постфиксной форме, то C# использует в выражении его старое значение, а затем выполнит операцию, в результате которой операнд обретет новое значение. Рассмотрим следующий фрагмент кода:

```
x = 10;
y = ++x;
```

В этом случае переменная y будет установлена равной 11. Но если в этом коде префиксную форму записи заменить постфиксной, переменная y будет установлена равной 10:

```
x = 10;
y = x++;
```

В обоих случаях переменная x получит значение 11. Разница состоит лишь в том, в какой момент она станет равной 11 (до присвоения ее значения переменной y или после).

Для программиста очень важно иметь возможность управлять временем выполнения операции инкремента или декремента. Рассмотрим следующую программу, которая генерирует ряд чисел:

```
/*
   Демонстрация различия между префиксной и
   постфиксной формами оператора ++.
*/
using System;

class PrePostDemo {
  public static void Main() {
    int x, y;
    int i;

    x = 1;
    Console.WriteLine(
      "Ряд, построенный с помощью инструкции y = x + x++;");
    for(i = 0; i < 10; i++) {

      y = x + x++; // постфиксная форма оператора ++

      Console.WriteLine(y + " ");
    }
    Console.WriteLine();

    x = 1;
    Console.WriteLine(
      "Ряд, построенный с помощью инструкции y = x + ++x;");
    for(i = 0; i < 10; i++) {

      y = x + ++x; // префиксная форма оператора ++

      Console.WriteLine(y + " ");
    }
    Console.WriteLine();

  }
}
```

Вот как выглядит результат выполнения этой программы:

```
Ряд, построенный с помощью инструкции y = x + x++;
2
4
6
8
10
12
14
```

```
16
18
20

Ряд, построенный с помощью инструкции y = x + ++x;
3
5
7
9
11
13
15
17
19
21
```

Как видно из результатов работы этой программы, инструкция

```
y = x + x++;
```

сначала суммирует значения x и x, после чего присваивает результат переменной y. Только затем она инкрементирует переменную x. Но инструкция

```
y = x + ++x;
```

выполняется по-другому. Сначала она получает (и запоминает) исходное значение переменной x, затем инкрементирует его, суммирует новое значение с исходным, а результат суммирования присваивает переменной y. Нетрудно заметить, что простая замена элемента x++ элементом ++x меняет числовой ряд, генерируемый программой, с четного на нечетный.

И еще. Выражение

```
x + ++x;
```

на первый взгляд может показаться странным, но только не компилятору. Несмотря на стоящие рядом два оператора, компилятор позаботится о правильной последовательности их выполнения. Достаточно понимать, что в этом выражении значение переменной x суммируется с инкрементированным значением той же переменной x.

Операторы отношений и логические операторы

Операторы отношений оценивают по "двубальной системе" (ИСТИНА/ЛОЖЬ) отношения между двумя значениями, а логические определяют различные способы сочетания истинных и ложных значений. Поскольку операторы отношений генерируют ИСТИНА/ЛОЖЬ-результаты, то они часто выполняются с логическими операторами. Поэтому мы и рассматриваем их в одном разделе.

Итак, перечислим операторы отношений.

Оператор	*Значение*
==	Равно
!=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно

Приведем список логических операторов.

Оператор	Значение
&	И
|	ИЛИ
^	Исключающее ИЛИ
&&	Сокращенное И
||	Сокращенное ИЛИ
!	НЕ

Результат выполнения операторов отношений и логических операторов имеет тип `bool`.

В C# на равенство или неравенство можно сравнивать (соответственно, с помощью операторов == и !=) *все* объекты. Но такие операторы сравнения, как <, >, <= или >=, можно применять только к типам, которые поддерживают отношения упорядочения. Это значит, что все операторы отношений можно применять ко всем числовым типам. Однако значения типа `bool` можно сравнивать только на равенство или неравенство, поскольку значения `true` и `false` не упорядочиваются. Например, в C# сравнение `true > false` не имеет смысла.

Что касается логических операторов, то их операнды должны иметь тип `bool`, и результат логической операции всегда будет иметь тип `bool`. Логические операторы `&`, `|`, `^` и `!` выполняют базовые логические операции И, ИЛИ, исключающее ИЛИ и НЕ в соответствии со следующей таблицей истинности.

p	*q*	*p & q*	*p | q*	*p ^ q*	*!p*
false	false	false	false	false	true
true	false	false	true	true	false
false	true	false	true	true	true
true	true	true	true	false	false

Как видно из этой таблицы, операция "исключающее ИЛИ" сгенерирует результат ИСТИНА лишь в случае, если истинен только один из ее операндов.

Рассмотрим программу, которая демонстрирует использование операторов отношений совместно с логическими операторами.

```
// Демонстрация использования операторов отношений
// и логических операторов.

using System;

class RelLogOps {
  public static void Main() {
    int i, j;
    bool b1, b2;

    i = 10;
    j = 11;
    if(i < j) Console.WriteLine("i < j");
    if(i <= j) Console.WriteLine("i <= j");
    if(i != j) Console.WriteLine("i != j");
    if(i == j) Console.WriteLine("Это не будет выполнено.");
```

```
    if(i >= j) Console.WriteLine("Это не будет выполнено.");
    if(i > j) Console.WriteLine("Это не будет выполнено.");

    b1 = true;
    b2 = false;
    if(b1 & b2) Console.WriteLine("Это не будет выполнено.");
    if(!(b1 & b2)) Console.WriteLine("!(b1 & b2) -- ИСТИНА");
    if(b1 | b2) Console.WriteLine("b1 | b2 -- ИСТИНА");
    if(b1 ^ b2) Console.WriteLine("b1 ^ b2 -- ИСТИНА");
  }
}
```

Результат выполнения этой программы таков:

```
i < j
i <= j
i != j
!(b1 & b2) -- ИСТИНА
b1 | b2 -- ИСТИНА
b1 ^ b2 -- ИСТИНА
```

Рассмотренные выше логические операторы предназначены для выполнения самых распространенных логических операций. Однако существует ряд других операций, которые определяются правилами формальной логики. Их также можно выполнить с помощью логических операторов C#. Так, C# поддерживает набор логических операторов, на базе которых можно построить любую другую логическую операцию, например операцию *импликации*. Импликация — это логическая операция, результат которой будет ложным только в случае, когда левый операнд имеет значение ИСТИНА, а правый — ЛОЖЬ. (Операция импликации отражает идею о том, что истина не может подразумевать ложь.) Вот как выглядит таблица истинности для оператора импликации:

p	*q*	*Результат импликации p и q*
true	true	true
true	false	false
false	false	true
false	true	true

Операцию импликации можно создать, используя комбинацию операторов ! и |.

```
!p | q
```

Использование импликации демонстрируется в следующей программе:

```
// Создание оператора импликации в языке C#.
using System;

class Implication {
  public static void Main() {
    bool p=false, q=false;
    int i, j;

    for(i = 0; i < 2; i++) {
      for(j = 0; j < 2; j++) {
        if(i==0) p = true;
        if(i==1) p = false;
        if(j==0) q = true;
        if(j==1) q = false;
```

```
        Console.WriteLine(
                    "p равно " + p + ", q равно " + q);
        if(!p | q)
            Console.WriteLine("Результат импликации " +
                        p + " и " + q + " равен " + true);
        Console.WriteLine();
      }
    }
  }
}
```

Результат выполнения этой программы выглядит так:

```
p равно True, q равно True
Результат импликации True и True равен True

p равно True, q равно False

p равно False, q равно True
Результат импликации False и True равен True

p равно False, q равно False
Результат импликации False и False равен True
```

Сокращенные логические операторы

C# поддерживает специальные *сокращенные* (short-circuit) версии логических операторов И и ИЛИ, которые можно использовать для создания более эффективного кода. Вспомним, что, если в операции И один операнд имеет значение ЛОЖЬ, результат будет ложным независимо от того, какое значение имеет второй операнд. А если в операции ИЛИ один операнд имеет значение ИСТИНА, результат будет истинным независимо от того, какое значение имеет второй операнд. Таким образом, в этих двух случаях вычислять второй операнд не имеет смысла. Если не вычисляется один из операндов, тем самым экономится время и создается более эффективный код.

Сокращенный оператор И обозначается символом `&&`, а сокращенный оператор ИЛИ — символом `||` (их обычные версии обозначаются одинарными символами `&` и `|`, соответственно). Единственное различие между обычной и сокращенной версиями этих операторов состоит в том, что при использовании обычной операции всегда вычисляются оба операнда, в случае же сокращенной версии второй операнд вычисляется только при необходимости.

Рассмотрим программу, в которой демонстрируется использование сокращенного оператора И. Программа определяет, является ли значение переменной `d` множителем числа `n`. Здесь используется операция деления по модулю. Если остаток от деления `d / n` равен нулю, значит, `d` — множитель числа `n`. Чтобы не допустить ошибки деления на нуль, используется сокращенная форма оператора И.

```
// Демонстрация использования сокращенных операторов.

using System;

class SCops {
  public static void Main() {
    int n, d;

    n = 10;
    d = 2;
    if(d != 0 && (n % d) == 0)
```

```
      Console.WriteLine(d + " -- множитель числа " + n);

    d = 0; // Теперь установим d равным нулю.

    // Поскольку d равно нулю,
    // второй операнд не вычисляется.
    if(d != 0 && (n % d) == 0)
      Console.WriteLine(d + " -- множитель числа " + n);

    /* Теперь попробуем проделать то же самое без
       сокращенного оператора. Такая попытка приведет
       к ошибке (деление на нуль).  */
    if(d != 0 & (n % d) == 0)
      Console.WriteLine(d + " -- множитель числа " + n);
  }
}
```

Чтобы не допустить деления на нуль, в инструкции `if` сначала проверяется значение переменной `d` на равенство нулю. Если наши опасения окажутся ненапрасными, выполнение сокращенного оператора И на этом прекратится. В первой проверке, когда переменная `d` содержит число 2, операция деления по модулю выполняется. Вторая проверка (а ей предшествует принудительная установка переменной `d` нулем) показывает, что второй операнд вычислять не имеет смысла, поэтому деление на нуль опускается. Попытка заменить сокращенный оператор И обычным заставит вычислить оба оператора и, как следствие, приведет к ошибке деления на нуль.

Поскольку сокращенные формы операторов И и ИЛИ в некоторых случаях работают эффективнее своих обычных "коллег", читатель мог бы задать вполне резонный вопрос: "Почему бы компилятору C# вообще не отказаться от обычных форм этих операторов?". Дело в том, что иногда необходимо, чтобы вычислялись оба операнда, поскольку вас могут интересовать побочные эффекты вычислений. Чтобы прояснить ситуацию, рассмотрим следующую программу:

```
// Демонстрация важности побочных эффектов.

using System;

class SideEffects {
  public static void Main() {
    int i;

    i = 0;

    /* Здесь значение i инкрементируется, несмотря на то,
       что инструкция выполнена не будет. */
    if(false & (++i < 100))
       Console.WriteLine("Этот текст не будет выведен.");
    Console.WriteLine(
            "Инструкция if выполнена: " + i); // Отображает: 1

    /* В этом случае значение i не инкрементируется, поскольку
       сокращенный оператор И опускает инкрементирование. */
    if(false && (++i < 100))
      Console.WriteLine("Этот текст не будет выведен.");
    Console.WriteLine(
            "Инструкция if выполнена: " + i); // По-прежнему 1 !!
  }
}
```

Как поясняется в комментариях, в первой if-инструкции значение переменной i инкрементируется независимо от результата выполнения самой if-инструкции. Но при использовании сокращенной версии оператора И во второй if-инструкции значение переменной i не инкрементируется, поскольку первый операнд имеет значение false. Из этого примера вы должны извлечь следующий урок. Если в программе предполагается обязательное выполнение правого операнда операции И/ИЛИ, вы должны использовать полную, или обычную, форму этих операторов, а не сокращенную.

И еще одна терминологическая деталь. Сокращенный оператор И также называется *условным И*, а сокращенный ИЛИ — *условным ИЛИ*.

Оператор присваивания

С оператором присваивания мы "шапочно" познакомились в главе 2. Теперь пришло время для более официального знакомства. *Оператор присваивания* обозначается одинарным знаком равенства (=). Его роль в языке C# во многом такая же, как и в других языках программирования. Общая форма записи оператора присваивания имеет следующий вид.

```
переменная = выражение;
```

Здесь тип элемента *переменная* должен быть совместим с типом элемента *выражение*.

Оператор присваивания интересен тем, что позволяет создавать целую цепочку присвоений. Рассмотрим, например, следующий фрагмент кода.

```
int x, y, z;

x = y = z = 100;  // Устанавливаем переменные x, y
                  // и z равными 100.
```

В этом фрагменте значения переменных x, y и z устанавливаются равными 100 в одной инструкции. Эта инструкция успешно работает благодаря тому, что оператор присваивания генерирует значение правостороннего выражения. Это значит, что значение выражения z = 100 равно числу 100, которое затем присваивается переменной y, после чего в свою очередь присваивается переменной x. Использование цепочки присвоений — простой способ установить группу переменных равными одному (общему для всех) значению.

Составные операторы присваивания

В C# предусмотрены специальные составные операторы присваивания, которые упрощают программирование определенных инструкций присваивания. Лучше всего начать с примера. Рассмотрим следующую инструкцию:

```
x = x + 10;
```

Используя составной оператор присваивания, ее можно переписать в таком виде:

```
x += 10;
```

Пара операторов += служит указанием компилятору присвоить переменной x сумму текущего значения переменной x и числа 10. А вот еще один пример. Инструкция

```
x = x - 100;
```

аналогична такой:

```
x -= 100;
```

Обе эти инструкции присваивают переменной x ее прежнее значение, уменьшенное на 100.

Составные версии операторов присваивания существуют для всех бинарных операторов (т.е. для всех операторов, которые работают с двумя операндами). Общая форма их записи такова:

```
переменная op = выражение;
```

Здесь элемент *op* означает конкретный арифметический или логический оператор, объединяемый с оператором присваивания.

Возможны следующие варианты объединения операторов.

```
+=          -=          *=          /=
%=          &=          |=          ^=
```

Поскольку составные операторы присваивания выглядят короче своих несоставных эквивалентов, то составные версии часто называют *укороченными операторами присваивания.*

Составные операторы присваивания обладают двумя заметными достоинствами. Во-первых, они компактнее своих "длинных" эквивалентов. Во-вторых, их наличие приводит к созданию более эффективного кода (поскольку операнд в этом случае вычисляется только один раз). Поэтому в профессионально написанных C#-программах вы часто встретите именно составные операторы присваивания.

Поразрядные операторы

В C# предусмотрен набор поразрядных операторов, которые расширяют области приложения языка C#. Поразрядные операторы действуют непосредственно на разряды своих операндов. Они определены только для целочисленных операндов и не могут быть использованы для операндов типа `bool`, `float` или `double`.

Поразрядные операторы предназначены для тестирования, установки или сдвига битов (разрядов), из которых состоит целочисленное значение. Поразрядные операторы очень часто используются для решения широкого круга задач программирования системного уровня, например, при опросе информации о состоянии устройства или ее формировании. Поразрядные операторы перечислены в табл. 4.1.

Таблица 4.1. Поразрядные операторы

Оператор	*Значение*
&	Поразрядное И
|	Поразрядное ИЛИ
^	Поразрядное исключающее ИЛИ
>>	Сдвиг вправо
<<	Сдвиг влево
~	Дополнение до 1 (унарный оператор НЕ)

Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ

Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ обозначаются символами `&`, `|`, `^` и `~`, соответственно. Они выполняют те же операции, что и их логические эквиваленты, описанные выше. Различие состоит лишь в том, что поразрядные операции работают на побитовой основе. В следующей таблице показан результат выполнения каждой поразрядной операции для всех возможных сочетаний операндов (нулей и единиц).

p	q	p & q	p | q	p ^ q	~p
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

Поразрядный оператор И можно представить как способ подавления битовой информации. Это значит, что 0 в любом операнде обеспечит установку в 0 соответствующего бита результата. Вот пример:

```
    1101 0011
    1010 1010
&   ---------
    1000 0010
```

В следующей программе демонстрируется использование поразрядного оператора & для получения четных чисел из нечетных. Это реализуется посредством подавления (установки в нуль) младшего разряда числа. Например, число 9 в двоичном коде представляется числом 0000 1001. После обнуления младшего разряда получается число 8 (0000 1000 в двоичном коде).

```
//  Использование поразрядного оператора И для
//  "превращения" любого числа в четное.

using System;

class MakeEven {
  public static void Main() {
    ushort num;
    ushort i;

    for(i = 1; i <= 10; i++) {
      num = i;

      Console.WriteLine("num: " + num);

      num = (ushort) (num & 0xFFFE); // num & 1111 1110

      Console.WriteLine("num после сброса младшего бита: "
                        +  num + "\n");
    }
  }
}
```

Результат выполнения этой программы имеет следующий вид:

```
num: 1
num после сброса младшего бита: 0

num: 2
num после сброса младшего бита: 2

num: 3
num после сброса младшего бита: 2

num: 4
num после сброса младшего бита: 4

num: 5
```

```
num после сброса младшего бита: 4

num: 6
num после сброса младшего бита: 6

num: 7
num после сброса младшего бита: 6

num: 8
num после сброса младшего бита: 8

num: 9
num после сброса младшего бита: 8

num: 10
num после сброса младшего бита: 10
```

Значение `0xFFFE`, используемое в этой программе, в двоичном коде представляется числом 1111 1111 1111 1110. Таким образом, операция `num & 0xFFFE` оставляет все биты неизменными за исключением младшего, который устанавливается в нуль. Поэтому любое четное число, пройдя через это "чистилище", остается четным, а любое нечетное "выходит" из него уже четным (за счет уменьшения на единицу).

Оператор И также используется для определения значения разряда. Например, следующая программа определяет, является ли заданное число нечетным.

```
//  Использование поразрядного оператора И для
//  определения, является ли число нечетным.
using System;

class IsOdd {
  public static void Main() {
    ushort num;

    num = 10;

    if((num & 1) == 1)
      Console.WriteLine("Этот текст не будет отображен.");

    num = 11;

    if((num & 1) == 1)
      Console.WriteLine(num + " -- нечетное число.");

  }
}
```

Результат выполнения этой программы выглядит так:

```
11 -- нечетное число.
```

В обеих инструкциях `if` выполняется операция И для значения переменной `num` и числа 1. Если младший бит переменной `num` установлен (т.е. равен единице), результат операции `num & 1` также будет равен единице. В противном случае результат будет равен нулю. Условие инструкции `if` выполнится только в случае, если анализируемое число окажется нечетным.

Возможности поразрядного тестирования, которые предоставляет поразрядный оператор `&`, можно использовать для создания программы, которая отображает значение типа `byte` в двоичном формате. Рассмотрим один из возможных вариантов решения этой задачи.

```
// Отображение значений битов, составляющих байт.
using System;

class ShowBits {
  public static void Main() {
    int t;
    byte val;

    val = 123;
    for(t=128; t > 0; t = t/2) {
      if((val & t) != 0) Console.Write("1 ");
      if((val & t) == 0) Console.Write("0 ");
    }
  }
}
```

Вот как выглядит результат выполнения этой программы:

```
0 1 1 1 1 0 1 1
```

В цикле `for` с помощью поразрядного оператора И последовательно тестируется каждый бит переменной `val`. Если оказывается, что этот бит установлен, отображается цифра `1`, в противном случае — цифра `0`.

Поразрядный оператор ИЛИ, в противоположность поразрядному И, удобно использовать для установки нужных битов в единицу. При выполнении операции ИЛИ наличие в операнде бита, равного 1, означает, что в результате соответствующий бит также будет равен единице. Вот пример:

```
  1101 0011
| 1010 1010
  ---------
  1111 1011
```

С помощью поразрядного оператора ИЛИ рассмотренную выше программу получения четных чисел легко превратить в программу получения нечетных чисел.

```
//  Использование поразрядного оператора ИЛИ для
//  "превращения" любого числа в нечетное.
using System;

class MakeOdd {
  public static void Main() {
    ushort num;
    ushort i;

    for(i = 1; i <= 10; i++) {
      num = i;

      Console.WriteLine("num: " + num);

      num = (ushort) (num | 1); // num | 0000 0001

      Console.WriteLine(
                  "num после установки младшего бита: "
                  +  num + "\n");
    }
  }
}
```

Результат выполнения этого варианта программы таков:

```
num: 1
num после установки младшего бита: 1

num: 2
num после установки младшего бита: 3

num: 3
num после установки младшего бита: 3

num: 4
num после установки младшего бита: 5

num: 5
num после установки младшего бита: 5

num: 6
num после установки младшего бита: 7

num: 7
num после установки младшего бита: 7

num: 8
num после установки младшего бита: 9

num: 9
num после установки младшего бита: 9

num: 10
num после установки младшего бита: 11
```

Работа этой программы основана на выполнении поразрядной операции ИЛИ между каждым числом, генерируемым в цикле `for`, и числом 1, которое в двоичном коде представляется как 0000 0001. Таким образом, 1 — это значение, у которого установлен только один младший разряд. Если это значение является одним из операндов операции ИЛИ, то результат выполнения этой операции совпадет со вторым операндом за исключением его младшего разряда, который станет равным единице (а все остальные при этом не изменятся). Следовательно, любое четное число, "пройдя" через операцию ИЛИ, увеличится на единицу, т.е. станет нечетным.

Поразрядное исключающее ИЛИ (XOR) устанавливает в единицу бит результата только в том случае, если соответствующие биты операндов отличаются один от другого, т.е. не равны. Вот пример:

```
    0111 1111
    1011 1001
^   ---------
    1100 0110
```

Оператор XOR обладает одним интересным свойством, которое позволяет использовать его для кодирования сообщений. Если выполнить операцию XOR между значением `X` и значением `Y`, а затем снова выполнить операцию XOR между результатом первой операции и тем же значением `Y`, получим исходное значение `X`. Это значит, что после выполнения двух операций

```
R1 = X ^ Y;
R2 = R1 ^ Y;
```

значение `R2` совпадет со значением `X`. Таким образом, в результате выполнения двух последовательных операций XOR, использующих одно и то же значение (`Y`), получается исходное значение (`X`). Этот принцип можно использовать для создания простой

программы шифрования, в которой некоторое целочисленное значение — ключ — служит для кодирования и декодирования сообщения, состоящего из символов. Для шифрования сообщения операция исключающего ИЛИ применяется первый раз, а для его дешифровки — второй. Реализуем этот простой способ шифрования в следующей программе:

```
// Использование оператора XOR для шифрования
// и дешифрирования сообщения.

using System;

class Encode {
  public static void Main() {
    char ch1 = 'H';
    char ch2 = 'i';
    char ch3 = '!';

    int key = 88;

    Console.WriteLine("Исходное сообщение: " +
                      ch1 + ch2 + ch3);

    // Шифруем сообщение.
    ch1 = (char) (ch1 ^ key);
    ch2 = (char) (ch2 ^ key);
    ch3 = (char) (ch3 ^ key);

    Console.WriteLine("Зашифрованное сообщение: " +
                      ch1 + ch2 + ch3);

    // Дешифрируем сообщение.
    ch1 = (char) (ch1 ^ key);
    ch2 = (char) (ch2 ^ key);
    ch3 = (char) (ch3 ^ key);

    Console.WriteLine("Дешифрованное сообщение: " +
                      ch1 + ch2 + ch3);
  }
}
```

Вот как выглядит результат выполнения этой программы:

```
Исходное сообщение: Hi!
Зашифрованное сообщение: >1y
Дешифрованное сообщение: Hi!
```

Как видите, в результате выполнения двух операций XOR, использующих одно и то же значение ключа, получается исходное (дешифрованное) сообщение.

Унарный оператор HE (или оператор дополнения до 1) инвертирует состояние всех битов своего операнда. Например, если целочисленное значение (хранимое в переменной A), представляет собой двоичный код 1001 0110, то в результате операции ~A получим двоичный код 0110 1001.

В следующей программе демонстрируется использование оператора HE посредством отображения некоторого числа и его дополнения до 1 в двоичном коде.

```
// Демонстрация поразрядного оператора HE.
using System;

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

```
    sbyte b = -34;
    int t;

    for(t=128; t > 0; t = t/2) {
      if((b & t) != 0) Console.Write("1 ");
      if((b & t) == 0) Console.Write("0 ");
    }
    Console.WriteLine();

    // Инвертируем все биты.
    b = (sbyte) ~b;

    for(t=128; t > 0; t = t/2) {
      if((b & t) != 0) Console.Write("1 ");
      if((b & t) == 0) Console.Write("0 ");
    }
  }
}
```

Выполнение этой программы дает такой результат:

```
1 1 0 1 1 1 1 0
0 0 1 0 0 0 0 1
```

Операторы сдвига

В C# можно сдвигать значение влево или вправо на заданное число разрядов. Для это в C# определены следующие операторы поразрядного сдвига:

<< сдвиг влево;

>> сдвиг вправо.

Общий формат записи этих операторов такой:

значение << *число_битов*;
значение >> *число_битов*.

Здесь *значение* — это объект операции сдвига, а элемент *число_битов* указывает, на сколько разрядов должно быть сдвинуто *значение*.

При сдвиге влево на один разряд все биты, составляющее значение, сдвигаются влево на одну позицию, а в младший разряд записывается нуль. При сдвиге вправо все биты сдвигаются, соответственно, вправо. Если сдвигу вправо подвергается значение без знака, в старший разряд записывается нуль. Если же сдвигу вправо подвергается значение со знаком, значение знакового разряда сохраняется. Вспомните: отрицательные целые числа представляются установкой старшего разряда числа равным единице. Таким образом, если сдвигаемое значение отрицательно, при каждом сдвиге вправо в старший разряд записывается единица, а если положительно — нуль.

При сдвиге как вправо, так и влево крайние биты теряются. Следовательно, при этом выполняется нециклический сдвиг, и содержимое потерянного бита узнать невозможно.

Ниже приводится программа, которая наглядно иллюстрирует результат сдвигов влево и вправо. Значение, которое будет сдвигаться, устанавливается сначала равным единице, т.е. только младший разряд этого значения "на старте" равен 1, все остальные равны 0. После выполнения каждого из восьми сдвигов влево программа отображает младшие восемь разрядов нашего "подопытного" значения. Затем описанный процесс повторяется, но в зеркальном отображении. На этот раз перед началом сдвига в переменную `val` заносится не 1, а 128, что в двоичном коде представляется как 1000 0000. И, конечно же, теперь сдвиг выполняется не влево, а вправо.

```
// Демонстрация использования операторов сдвига << и >>.
using System;

class ShiftDemo {
  public static void Main() {
    int val = 1;
    int t;
    int i;

    for(i = 0; i < 8; i++) {
      for(t=128; t > 0; t = t/2) {
        if((val & t) != 0) Console.Write("1 ");
        if((val & t) == 0) Console.Write("0 ");
      }
      Console.WriteLine();
      val = val << 1; // Сдвиг влево.
    }
    Console.WriteLine();

    val = 128;
    for(i = 0; i < 8; i++) {
      for(t=128; t > 0; t = t/2) {
        if((val & t) != 0) Console.Write("1 ");
        if((val & t) == 0) Console.Write("0 ");
      }
      Console.WriteLine();
      val = val >> 1; // Сдвиг вправо.
    }
  }
}
```

Вот результаты выполнения этой программы:

```
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
```

Поскольку разряды представления двоичных чисел представляют собой степени числа 2, то операторы сдвига можно использовать в качестве быстрого способа умножения или деления чисел на 2. При сдвиге влево число удваивается. При сдвиге вправо число делится пополам. Конечно же, это будет справедливо до тех пор, пока с одного или другого конца не выдвинутся (и потеряются) значимые биты. Вот пример:

```
// Использование операторов сдвига для
// умножения и деления на 2.
```

```
using System;

class MultDiv {
  public static void Main() {
    int n;

    n = 10;

    Console.WriteLine("Значение переменной n: " + n);

    // Умножаем на 2.
    n = n << 1;
    Console.WriteLine(
        "Значение переменной n после n = n * 2: " + n);

    // Умножаем на 4.
    n = n << 2;
    Console.WriteLine(
        "Значение переменной n после n = n * 4: " + n);

    // Делим на 2.
    n = n >> 1;
    Console.WriteLine(
        "Значение переменной n после n = n / 2: " + n);

    // Делим на 4.
    n = n >> 2;
    Console.WriteLine(
        "Значение переменной n после n = n / 4: " + n);
    Console.WriteLine();

    // Устанавливаем n в исходное состояние.
    n = 10;
    Console.WriteLine("Значение переменной n: " + n);

    // Умножаем на 2, причем 30 раз.
    n = n << 30; // Увы: данные потеряны.
    Console.WriteLine(
        "Значение n после сдвига влево на 30 разрядов: " + n);

  }
}
```

Вот как выглядят результаты выполнения этой программы:

```
Значение переменной n: 10
Значение переменной n после n = n * 2: 20
Значение переменной n после n = n * 4: 80
Значение переменной n после n = n / 2: 40
Значение переменной n после n = n / 4: 10

Значение переменной n: 10
Значение n после сдвига влево на 30 разрядов: -2147483648
```

Обратите внимание на последнюю строку результатов выполнения программы. После сдвига числа 10 влево на 30 разрядов (т.е. после умножения на 2^{30}) информация будет потеряна, поскольку значение исходного числа было "выдвинуто" за пределы диапазона представления чисел, соответствующего типу `int`. В данном случае вы видите странное отрицательное значение, которое получилось в результате попадания

единицы в старший разряд числа, который для типа int используется в качестве знакового. Вследствие этого число стало интерпретироваться как отрицательное. Этот пример показывает, как необходима осторожность при использовании операторов сдвига для умножения или деления чисел на 2. (Чтобы вспомнить, чем отличается представление значений со знаком от представления значений без знака, обратитесь к главе 3.)

Поразрядные составные операторы присваивания

Все бинарные поразрядные операторы можно успешно объединять с оператором присваивания, образуя поразрядные составные операторы присваивания. Например, следующие две инструкции присваивают переменной x результат выполнения операции исключающего ИЛИ (XOR) с операндами x и 127.

```
x = x ^ 127;
x ^= 127;
```

Оператор ?

Одним из самых замечательных операторов C# является тернарный оператор ?. Оператор ? часто используется для замены определенных типов конструкций if-then-else. Оператор ? называется *тернарным*, поскольку он работает с тремя операторами. Его общий формат записи имеет такой вид:

Выражение1 ? *Выражение2* : *Выражение3*;

Здесь *Выражение1* должно иметь тип bool. Типы элементов *Выражение2* и *Выражение3* должны быть одинаковы. Обратите внимание на использование и расположение двоеточия.

Значение ?-выражения определяется следующим образом. Вычисляется *Выражение1*. Если оно оказывается истинным, вычисляется *Выражение2*, и результат его вычисления становится значением всего ?-выражения. Если результат вычисления элемента *Выражение1* оказывается ложным, значением всего ?-выражения становится результат вычисления элемента *Выражение3*. Рассмотрим пример, в котором переменной absval присваивается абсолютное значение переменной val.

```
absval =  val < 0 ? -val : val; // Получаем абсолютное
                                // значение val.
```

Здесь переменной absval присваивается значение переменной val, если оно больше или равно нулю. Если же значение переменной val отрицательно, переменной absval присваивается результат применения к ней операции "унарный минус", который будет представлять собой положительное значение.

Вот еще один пример использования оператора ?. В следующей программе выполняется деление числа 100 на разные числа, но попытка деления на нуль реализована не будет.

```
// Способ обойти деление на нуль с помощью оператора ?.
using System;

class NoZeroDiv {
  public static void Main() {
    int result;
    int i;
```

```
    for(i = -5; i < 6; i++) {
      result = i != 0 ? 100 / i : 0;
      if(i != 0)
        Console.WriteLine("100 / " + i + " равно " + result);
    }
  }
}
```

Посмотрите на результаты выполнения этой программы.

```
100 / -5 равно -20
100 / -4 равно -25
100 / -3 равно -33
100 / -2 равно -50
100 / -1 равно -100
100 / 1 равно 100
100 / 2 равно 50
100 / 3 равно 33
100 / 4 равно 25
100 / 5 равно 20
```

Обратите внимание на следующую строку из этой программы:

```
result = i != 0 ? 100 / i : 0;
```

Здесь переменной `result` присваивается результат деления числа 100 на значение переменной `i`. Однако это деление выполнится только в том случае, если `i` не равно нулю. В противном случае (при `i = 0`) переменной `result` будет присвоено нулевое значение.

В действительности совсем не обязательно присваивать переменной значение, генерируемое оператором `?`. Например, вы могли бы использовать это значение в качестве аргумента, передаваемого методу. Или возможен еще такой вариант. Если все выражения, принимаемые оператором `?`, имеют тип `bool`, то результат выполнения этого оператора можно использовать в качестве условного выражения в цикле или инструкции `if`. Рассмотрим, например, предыдущую программу, переписанную в более эффективном виде (результат ее выполнения аналогичен предыдущему).

```
// Способ обойти деление на нуль с помощью ?-оператора.
using System;

class NoZeroDiv2 {
  public static void Main() {
    int i;

    for(i = -5; i < 6; i++)
      if(i != 0 ? true : false)
        Console.WriteLine("100 / " + i +
                          " равно " + 100 / i);
  }
}
```

Обратите внимание на инструкцию `if`. Если значение переменной `i` равно нулю, результат проверки `if`-условия будет равен значению `false`, которое не допустит выполнения инструкции вывода, а значит, и деления на нуль. В противном случае деление (с выводом результата) будет иметь место.

Использование пробелов и круглых скобок

Любое выражение в C# для повышения читабельности может включать пробелы (или символы табуляции). Например, следующие два выражения совершенно одинаковы, но второе прочитать гораздо легче:

```
x=10/y*(127/x);
x = 10 / y * (127/x);
```

Круглые скобки (так же, как в алгебре) повышают приоритет операций, содержащихся внутри них. Использование избыточных или дополнительных круглых скобок не приведет к ошибке или замедлению вычисления выражения. Другими словами, от них не будет никакого вреда, но зато сколько пользы! Ведь они помогут прояснить (для вас самих в первую очередь, не говоря уже о тех, кому придется разбираться в этом без вас) точный порядок вычислений. Скажите, например, какое из следующих двух выражений легче понять?

```
x = y/3-34*temp+127;
X = (y/3) - (34*temp) + 127;
```

Приоритет операторов

В табл. 4.2 показан порядок выполнения C#-операторов (от высшего до самого низкого). Эта таблица включает несколько операторов, которые описаны далее.

Таблица 4.2. Приоритет C#-операторов

Наивысший
`( ) [ ] . ++(постфиксный) --(постфиксный) checked new sizeof typeof unchecked`
`! ~ Операторы приведения типа +(унарный) -(унарный) ++(префиксный) --(префиксный)`
`* / %`
`+ -`
`<< >>`
`< <= > >= is`
`== !=`
`&`
`^`
`|`
`&&`
`||`
`?:`
`= op=`
Низший

Полный справочник по

Глава 5

Инструкции управления

В этой главе рассматриваются инструкции управления ходом выполнения C#-программы. Существует три категории управляющих инструкций: *инструкции выбора* (`if`, `switch`), *итерационные инструкции* (состоящие из `for`-, `while`-, `do-while`- и `foreach`-циклов) и *инструкции перехода* (`break`, `continue`, `goto`, `return` и `throw`). За исключением инструкции `throw`, которая является частью C#-механизма обработки исключительных ситуаций (и рассматривается в главе 13), все остальные перечисленные выше инструкции описаны в этой главе.

Инструкция `if`

Инструкция `if` была представлена в главе 2, но здесь мы рассмотрим ее более детально. Полный формат ее записи такой:

```
if(условие) инструкция;
else инструкция;
```

Здесь под элементом *инструкция* понимается одна инструкция языка C#. Часть `else` необязательна. Вместо элемента *инструкция* может быть использован блок инструкций. В этом случае формат записи `if`-инструкции принимает такой вид:

```
if(условие)
{
    последовательность инструкций
}
else
{
    последовательность инструкций
}
```

Если элемент *условие*, который представляет собой условное выражение, при вычислении даст значение ИСТИНА, будет выполнена `if`-инструкция; в противном случае — `else`-инструкция (если таковая существует). Обе инструкции никогда не выполняются. Условное выражение, управляющее выполнением `if`-инструкции, должно иметь тип `bool`.

Рассмотрим простую программу, в которой используется `if-else`-инструкция для определения того, является число положительным или отрицательным.

```
// Программа определяет, является число положительным
// или отрицательным.

using System;

class PosNeg {
  public static void Main() {
    int i;

    for(i=-5; i <= 5; i++) {
      Console.Write("Тестирование " + i + ": ");

      if(i < 0) Console.WriteLine("Число отрицательно");
      else Console.WriteLine("Число положительно");
    }

  }
}
```

Результаты выполнения программы выглядят так:

```
Тестирование -5: Число отрицательно
Тестирование -4: Число отрицательно
Тестирование -3: Число отрицательно
Тестирование -2: Число отрицательно
Тестирование -1: Число отрицательно
Тестирование 0: Число положительно
Тестирование 1: Число положительно
Тестирование 2: Число положительно
Тестирование 3: Число положительно
Тестирование 4: Число положительно
Тестирование 5: Число положительно
```

Если оказывается, что в этом примере значение переменной `i` меньше нуля, выполняется `if`-инструкция (выводится "диагноз": `Число отрицательно`); в противном случае — `else`-инструкция (выводится "диагноз": `Число положительно`). Обе инструкции вместе ни при каких условиях выполнены не будут.

Вложенные `if`-инструкции

Вложенные `if`-инструкции образуются в том случае, если в качестве элемента *инструкция* (см. полный формат записи) используется другая `if`-инструкция. Вложенные `if`-инструкции очень популярны в программировании. Главное здесь — помнить, что `else`-инструкция всегда относится к ближайшей `if`-инструкции, которая находится внутри того же программного блока, но еще не связана ни с какой другой `else`-инструкцией. Вот пример:

```
if(i == 10) {
  if(j < 20) a = b;
  if(k > 100) c = d;
  else a = c; // Эта else-инструкция
              // относится к if(k > 100).
}
else a = d; // Эта else-инструкция относится к if(i == 10).
```

Как утверждается в комментариях, последняя `else`-инструкция не связана с инструкцией `if(j < 20)`, поскольку они не находятся в одном блоке (несмотря на то что эта `if`-инструкция — ближайшая, которая не имеет при себе "`else`-пары"). Внутренняя `else`-инструкция связана с инструкцией `if(k > 100)`, поскольку она — ближайшая и находится внутри того же блока.

В предыдущей программе нулевое значение тестируемой переменной интерпретировалось как положительное. Однако в некоторых приложениях нуль рассматривается как значение без знака. Поэтому в следующей версии программы, которая демонстрирует использование вложенных `else`-инструкций, нулю дается именно такая характеристика: "без знака".

```
// Программа определяет, является число положительным,
// отрицательным или нулем.

using System;

class PosNegZero {
  public static void Main() {
    int i;

    for(i=-5; i <= 5; i++) {
```

```
      Console.Write("Тестирование " + i + ": ");

      if(i < 0) Console.WriteLine("Число отрицательно");
      else if(i == 0) Console.WriteLine("Число без знака");
        else Console.WriteLine("Число положительно");
    }

  }
}
```

Вот какие получаем результаты:

```
Тестирование -5: Число отрицательно
Тестирование -4: Число отрицательно
Тестирование -3: Число отрицательно
Тестирование -2: Число отрицательно
Тестирование -1: Число отрицательно
Тестирование 0: Число без знака
Тестирование 1: Число положительно
Тестирование 2: Число положительно
Тестирование 3: Число положительно
Тестирование 4: Число положительно
Тестирование 5: Число положительно
```

Конструкция if-else-if

Очень распространенной в программировании конструкцией, в основе которой лежит вложенная if-инструкция, является "лестница" if-else-if. Ее можно представить в следующем виде:

```
if(условие)
  инструкция;
else if(условие)
  инструкция;
else if(условие)
  инструкция;
.
.
.
else
  инструкция;
```

Здесь под элементом *условие* понимается условное выражение. Условные выражения вычисляются сверху вниз. Как только в какой-нибудь ветви обнаружится истинный результат, будет выполнена инструкция, связанная с этой ветвью, а вся остальная "лестница" опускается. Если окажется, что ни одно из условий не является истинным, будет выполнена последняя else-инструкция (можно считать, что она выполняет роль условия, которое действует по умолчанию). Если последняя else-инструкция не задана, а все остальные оказались ложными, то вообще никакое действие не будет выполнено.

Работа if-else-if-"лестницы" демонстрируется в следующей программе, которая находит для заданного значения наименьший множитель (отличный от единицы и состоящий из одной цифры).

```
// Определение наименьшего множителя,
// состоящего из одной цифры.

using System;
```

```
class Ladder {
 public static void Main() {
   int num;

   for(num = 2; num < 12; num++) {
     if((num % 2) == 0)
       Console.WriteLine("Наименьший множитель числа " +
                         num + " равен 2.");
     else if((num % 3) == 0)
       Console.WriteLine("Наименьший множитель числа " +
                         num + " равен 3.");
     else if((num % 5) == 0)
       Console.WriteLine("Наименьший множитель числа " +
                         num + " равен 5.");
     else if((num % 7) == 0)
       Console.WriteLine("Наименьший множитель числа " +
                         num + " равен 7.");
     else
       Console.WriteLine(num +
                       " не делится на 2, 3, 5 или 7.");
   }
 }
}
```

Результаты выполнения этой программы выглядят так:

```
Наименьший множитель числа 2 равен 2.
Наименьший множитель числа 3 равен 3.
Наименьший множитель числа 4 равен 2.
Наименьший множитель числа 5 равен 5.
Наименьший множитель числа 6 равен 2.
Наименьший множитель числа 7 равен 7.
Наименьший множитель числа 8 равен 2.
Наименьший множитель числа 9 равен 3.
Наименьший множитель числа 10 равен 2.
11 не делится на 2, 3, 5 или 7.
```

Как видите, последняя `else`-инструкция выполняется только в том случае, если не выполнилась ни одна из предыдущих `if`-инструкций.

Инструкция `switch`

Второй инструкцией выбора является `switch`. Инструкция `switch` обеспечивает многонаправленное ветвление. Она позволяет делать выбор одной из множества альтернатив. Хотя многонаправленное тестирование можно реализовать с помощью последовательности вложенных `if`-инструкций, для многих ситуаций инструкция `switch` оказывается более эффективным решением. Она работает следующим образом. Значение выражения последовательно сравнивается с константами из заданного списка. При обнаружении совпадения для одного из условий сравнения выполняется последовательность инструкций, связанная с этим условием. Общий формат записи инструкции `switch` такой:

```
switch(выражение) {
  case константа1:
    последовательность инструкций
    break;
case константа2:
```

```
    последовательность инструкций
    break;
  case константа3:
    последовательность инструкций
    break;
.
.
.
  default:
    последовательность инструкций
    break;
}
```

Элемент *выражение* инструкции `switch` должен иметь целочисленный тип (например, `char`, `byte`, `short` или `int`) или тип `string` (о котором речь пойдет ниже в этой книге). Выражения, имеющие тип с плавающей точкой, не разрешены. Очень часто в качестве управляющего `switch`-выражения используется просто переменная; `case`-константы должны быть литералами, тип которых совместим с типом заданного выражения. При этом никакие две `case`-константы в одной `switch`-инструкции не могут иметь идентичных значений.

Последовательность инструкций `default`-ветви выполняется в том случае, если ни одна из заданных `case`-констант не совпадет с результатом вычисления `switch`-выражения. Ветвь `default` необязательна. Если она отсутствует, то при несовпадении результата выражения ни с одной из `case`-констант никакое действие выполнено не будет. Если такое совпадение все-таки обнаружится, будут выполнены инструкции, соответствующие данной `case`-ветви до тех пор, пока не встретится инструкция `break`.

Использование `switch`-инструкции демонстрируется в следующей программе.

```
// Демонстрация использования инструкции switch.

using System;

class SwitchDemo {
  public static void Main() {
    int i;

    for(i=0; i<10; i++)
      switch(i) {
        case 0:
          Console.WriteLine("i равно нулю.");
          break;
        case 1:
          Console.WriteLine("i равно единице.");
          break;
        case 2:
          Console.WriteLine("i равно двум.");
          break;
        case 3:
          Console.WriteLine("i равно трем.");
          break;
        case 4:
          Console.WriteLine("i равно четырем.");
          break;
        default:
          Console.WriteLine("i равно или больше пяти.");
          break;
```

```
      }

  }
}
```

Результаты выполнения этой программы:

```
i равно нулю.
i равно единице.
i равно двум.
i равно трем.
i равно четырем.
i равно или больше пяти.
i равно или больше пяти.
i равно или больше пяти.
i равно или больше пяти.
i равно или больше пяти.
```

Как видите, на каждой итерации цикла выполняются инструкции, соответствующие case-константе, которая совпадает с текущим значением переменной i. При этом все остальные инструкции опускаются. Если i равно или больше пяти, выполняется default-инструкция.

В предыдущем примере switch-инструкция управлялась переменной типа int. Но, как вы уже знаете, для управления в switch-инструкции можно использовать переменную любого целочисленного типа, в том числе и типа char. Перед вами пример использования в case-ветвях char-выражения и char-констант.

```
// Использование типа char для управления
// switch-инструкцией.

using System;

class SwitchDemo2 {
  public static void Main() {
    char ch;

    for(ch='A'; ch<= 'E'; ch++)
      switch(ch) {
        case 'A':
          Console.WriteLine("ch содержит A");
          break;
        case 'B':
          Console.WriteLine("ch содержит B");
          break;
        case 'C':
          Console.WriteLine("ch содержит C");
          break;
        case 'D':
          Console.WriteLine("ch содержит D");
          break;
        case 'E':
          Console.WriteLine("ch содержит E");
          break;
      }
  }
}
```

Результаты выполнения этой программы выглядят так:

```
ch содержит A
ch содержит B
```

```
ch содержит C
ch содержит D
ch содержит E
```

Обратите внимание на то, что в этом примере программы `default`-инструкция отсутствует. Не забывайте, что она необязательна. Если в ней нет необходимости, ее можно опустить.

В C# считается ошибкой, если последовательность инструкций, относящаяся к одной `case`-ветви, переходит в последовательность инструкций, связанную со следующей. Здесь должно действовать правило запрета на передачу управления вниз, на "провал", как говорят программисты. Поэтому `case`-последовательности чаще всего оканчиваются инструкцией `break`. ("Провала" можно избежать и другими способами, например с помощью инструкции `goto`, которая будет рассмотрена ниже в этой главе, но `break` — это самое распространенное средство от "провалов".) Инструкция `break`, завершающая последовательность `case`-инструкций, приводит к выходу из всей конструкции `switch` и передаче управления к следующей инструкции, находящейся вне конструкции `switch`. Последовательность инструкций `default`-ветви также не должна "проваливаться" и обычно завершается инструкцией `break`.

Правило запрета на "провальную" передачу управления вниз — одно из отличий языка C# от C, C++ и Java. В этих трех упомянутых языках `case`-инструкции имеют право "плавно" переходить в инструкции, соответствующие следующей `case`-ветви, т.е. "проваливаться" вниз. Разработчики языка C# обосновали запрет на "провал" двумя следующими причинами. Во-первых, это позволяет компилятору в целях оптимизации свободно менять порядок следования `case`-ветвей, что было бы невозможно, если бы одна `case`-последовательность могла беспрепятственно перетекать в следующую. Во-вторых, требование явного завершения `case`-последовательности специальной инструкцией исключит возможность случайного "провала", организованного программистом по недосмотру.

Несмотря на запрет "провальной" передачи управления от одной `case`-последовательности к следующей, можно, как показано в следующем примере, организовать программный код так, чтобы две или больше инструкций `case` ссылались на одну и ту же кодовую последовательность.

```
// "Пустые" case-инструкции могут "проваливаться".

using System;

class EmptyCasesCanFall {
  public static void Main() {
    int i;

    for(i=1; i < 5; i++)
      switch(i) {
        case 1:
        case 2:
        case 3: Console.WriteLine("i равно 1, 2 или 3");
          break;
        case 4: Console.WriteLine("i равно 4");
          break;
      }

  }
}
```

Результаты работы этой программы вполне ожидаемы:

```
i равно 1, 2 или 3
i равно 1, 2 или 3
i равно 1, 2 или 3
i равно 4
```

В этом примере, если переменная `i` содержит значение 1,2 или 3, то выполняется первая инструкция вызова метода `WriteLine()`, а если значение `i` равно 4, то — вторая. Показанное здесь пакетирование `case`-ветвей не нарушает правило запрета "провалов", поскольку все эти `case`-инструкции используют одну и ту же кодовую последовательность.

Такое `case`-пакетирование — распространенный способ совместного использования кода несколькими `case`-ветвями, позволяющий устранить ненужное дублирование кодовых последовательностей.

Вложенные инструкции `switch`

Инструкция `switch` может быть использована как часть `case`-последовательности внешней инструкции `switch`. В этом случае она называется вложенной инструкцией `switch`. Необходимо отметить, что `case`-константы внутренних и внешних инструкций `switch` могут иметь одинаковые значения, при этом никаких конфликтов не возникнет. Например, следующий фрагмент кода вполне допустим:

```
switch(ch1) {
  case 'A':
    Console.WriteLine(
       "Эта буква A - часть внешней инструкции switch.");
    switch(ch2) {
      case 'A':
        Console.WriteLine(
          "Эта буква A - часть внутренней инструкции switch.");
        break;
      case 'B':  // ...
    } // Конец внутренней инструкции switch.
    break;
  case 'B': // ...
```

Цикл `for`

Начиная с главы 2, мы уже использовали простую форму цикла `for`. В этой главе мы рассмотрим этот цикл более детально, и вы узнаете, насколько мощным и гибким средством программирования он является. Начнем с традиционных форм его использования. Итак, общий формат записи цикла `for` для повторного выполнения одной инструкции имеет следующий вид:

```
for(инициализация; условие; итерация) инструкция;
```

Если цикл `for` предназначен для повторного выполнения программного блока, то его общий формат выглядит так:

```
for(инициализация; условие; итерация)
{
  последовательность инструкций
}
```

Элемент *инициализация* обычно представляет собой инструкцию присваивания, которая устанавливает *управляющую переменную цикла* равной начальному значению.

Эта переменная действует в качестве счетчика, который управляет работой цикла. Элемент *условие* представляет собой выражение типа `bool`, в котором тестируется значение управляющей переменной цикла. Результат этого тестирования определяет, выполнится цикл `for` еще раз или нет. Элемент *итерация* — это выражение, которое определяет, как изменяется значение управляющей переменной цикла после каждой итерации. Обратите внимание на то, что все эти элементы цикла `for` должны отделяться точкой с запятой. Цикл `for` будет выполняться до тех пор, пока вычисление элемента *условие* дает истинный результат. Как только условие станет ложным, выполнение программы продолжится с инструкции, следующей за циклом `for`.

Управляющая переменная цикла `for` может изменяться как с положительным, так и с отрицательным приращением, причем величина этого приращения также может быть любой. Например, следующая программа выводит числа в диапазоне от 100 до −100 с декрементом, равным 5.

```
// Цикл for с отрицательным приращением
// управляющей переменной.

using System;

class DecrFor {
  public static void Main() {
    int x;

    for(x = 100; x > -100; x -= 5)
      Console.WriteLine(x);
  }
}
```

Важно понимать, что условное выражение всегда тестируется в начале выполнения цикла `for`. Это значит, что если первая же проверка условия даст значение ЛОЖЬ, код тела цикла не выполнится ни разу. Вот пример:

```
for(count=10; count < 5; count++)
  x += count; // Эта инструкция не будет выполнена вовсе.
```

Этот цикл никогда не выполнится, поскольку уже при входе в него значение его управляющей переменной `count` больше пяти. Это делает условное выражение (`count < 5`) ложным с самого начала. Поэтому даже одна итерация этого цикла не будет выполнена.

Цикл `for` особенно полезен в тех случаях, когда известно количество его повторений. Например, следующая программа использует два цикла `for` для отыскания простых чисел в диапазоне от 2 до 20. Если число не простое, программа отобразит его самый большой множитель.

```
/*
   Программа определения простых чисел.
   Если число не простое, программа отображает
   его самый большой множитель.
*/

using System;

class FindPrimes {
  public static void Main() {
    int num;
    int i;
    int factor;
    bool isprime;
    for(num = 2; num < 20; num++) {
```

```
      isprime = true;
      factor = 0;

      // Узнаем, делится ли num на i без остатка.
      for(i=2; i <= num/2; i++) {
        if((num % i) == 0) {
          // Если num делится на i без остатка,
          // значит num -- число не простое.
          isprime = false;
          factor = i;
        }
      }

      if(isprime)
        Console.WriteLine(num + " -- простое число.");
      else
        Console.WriteLine("Максимальный множитель числа " +
                          num + " равен " + factor);
    }
  }
}
```

Результаты выполнения этой программы имеют такой вид:

```
2 -- простое число.
3 -- простое число.
Максимальный множитель числа 4 равен 2
5 -- простое число.
Максимальный множитель числа 6 равен 3
7 -- простое число.
Максимальный множитель числа 8 равен 4
Максимальный множитель числа 9 равен 3
Максимальный множитель числа 10 равен 5
11 -- простое число.
Максимальный множитель числа 12 равен 6
13 -- простое число.
Максимальный множитель числа 14 равен 7
Максимальный множитель числа 15 равен 5
Максимальный множитель числа 16 равен 8
17 -- простое число.
Максимальный множитель числа 18 равен 9
19 -- простое число.
```

Вариации на тему цикла `for`

Цикл `for` — одна из наиболее гибких инструкций в C#, поскольку она позволяет получить широкий диапазон вариаций.

Использование нескольких управляющих переменных цикла

Для управления циклом `for` можно использовать две или больше переменных. В этом случае инструкции инициализации и итерации для каждой из этих переменных отделяются запятыми. Вот пример:

```
// Использование запятых в цикле for.

using System;

class Comma {
```

```
  public static void Main() {
    int i, j;

    for(i=0, j=10; i < j; i++, j--)
      Console.WriteLine("i и j: " + i + " " + j);
  }
}
```

Вот как выглядят результаты выполнения этой программы:

```
i и j: 0 10
i и j: 1 9
i и j: 2 8
i и j: 3 7
i и j: 4 6
```

Здесь запятыми отделяются две инструкции инициализации и два итерационных выражения. При входе в цикл инициализируются обе переменные — `i` и `j`. После выполнения каждой итерации цикла переменная `i` инкрементируется, а переменная `j` декрементируется. Использование нескольких управляющих переменных в цикле иногда позволяет упростить алгоритмы. В разделах инициализации и итерации цикла `for` можно использовать любое количество инструкций, но обычно их число не превышает двух.

Приведем пример практического использования двух управляющих переменных в цикле `for`. Рассмотрим программу, которая находит наибольший и наименьший множители числа (в данном случае числа 100). Обратите особое внимание на условие завершения цикла: оно включает обе управляющих переменных.

```
/*
   Использование запятых в цикле for для определения
   наибольшего и наименьшего множителей числа.
*/

using System;

class Comma {
  public static void Main() {
    int i, j;
    int smallest, largest;
    int num;

    num = 100;

    smallest = largest = 1;

    for(i=2, j=num/2; (i <= num/2) & (j >= 2); i++, j--) {

      if((smallest == 1) & ((num % i) == 0))
        smallest = i;

      if((largest == 1) & ((num % j) == 0))
        largest = j;

    }

    Console.WriteLine("Наибольший множитель: " + largest);
    Console.WriteLine("Наименьший множитель: " + smallest);
  }
}
```

Результаты выполнения этой программы выглядят так:

```
Наибольший множитель: 50
Наименьший множитель: 2
```

Благодаря использованию сразу двух управляющих переменных в одном цикле `for` можно найти как наибольший, так и наименьший множитель числа. Для определения наименьшего множителя используется управляющая переменная `i`. Первоначально она устанавливается равной числу 2 и инкрементируется до тех пор, пока ее значение не превысит половину исследуемого числа (оно хранится в переменной `num`). Для определения наибольшего множителя используется управляющая переменная `j`. Первоначально она устанавливается равной половине числа, хранимого в переменной `num`, и декрементируется до тех пор, пока ее значение не станет меньше двух. Цикл работает до тех пор, пока обе переменные — `i` и `j` — не достигнут своих конечных значений. По завершении цикла будут найдены оба множителя.

Условное выражение

Условным выражением, которое управляет циклом `for`, может быть любое допустимое выражение, генерирующее результат типа `bool`. Например, в следующей программе цикл управляется переменной `done`.

```
// Условием цикла может быть любое выражение типа bool.
using System;

class forDemo {
  public static void Main() {
    int i, j;
    bool done = false;

    for(i=0, j=100; !done; i++, j--) {

      if(i*i >= j) done = true;

      Console.WriteLine("i, j: " + i + " " + j);
    }

  }
}
```

А вот результаты выполнения этой программы:

```
i, j: 0 100
i, j: 1 99
i, j: 2 98
i, j: 3 97
i, j: 4 96
i, j: 5 95
i, j: 6 94
i, j: 7 93
i, j: 8 92
i, j: 9 91
i, j: 10 90
```

В этом примере цикл `for` повторяется до тех пор, пока `bool`-переменная `done` имеет значение `true`. Эта переменная устанавливается равной `true` внутри цикла, если квадрат значения переменной `i` больше значения переменной `j` или равен ему.

Отсутствие элементов в определении цикла

В C# разрешается опустить любой элемент заголовка цикла (инициализация, условие, итерация) или даже все сразу. Отсутствие некоторых элементов в определении цикла может дать интересный результат. Рассмотрим следующую программу:

```
// Составляющие части цикла for могут быть пустыми.

using System;

class Empty {
  public static void Main() {
    int i;

    for(i = 0; i < 10; ) {
      Console.WriteLine("Проход №" + i);
      i++; // Инкрементируем управляющую переменную цикла.
    }
  }
}
```

Здесь отсутствует выражение итерации цикла `for`. Вместо него инкрементирование управляющей переменной `i` выполняет инструкция, находящаяся внутри цикла. Это значит, что перед каждым повторением тела цикла выполняется только одно действие: значение переменной `i` сравнивается с числом 10. Но поскольку значение `i` инкрементируется внутри цикла, он функционирует нормально, отображая следующие результаты:

```
Проход №0
Проход №1
Проход №2
Проход №3
Проход №4
Проход №5
Проход №6
Проход №7
Проход №8
Проход №9
```

В следующем примере из определения цикла `for` удалена и часть инициализации управляющей переменной.

```
// Определение цикла for состоит из одного условия.

using System;

class Empty2 {
  public static void Main() {
    int i;

    i = 0; // Убираем из цикла раздел инициализации.
    for(; i < 10; ) {
      Console.WriteLine("Проход №" + i);
      i++; // Инкрементируем управляющую переменную цикла.
    }
  }
}
```

В этой версии переменная `i` инициализируется до входа в цикл `for`, а не в его заголовке. Обычно программисты предпочитают инициализировать управляющую переменную цикла внутри цикла `for`. К размещению выражения инициализации за

пределами цикла, как правило, прибегают только в том случае, когда начальное значение генерируется сложным процессом, который неудобно поместить в определение цикла.

Бесконечный цикл

Оставив пустым условное выражение цикла `for`, можно создать *бесконечный цикл* (цикл, который никогда не заканчивается). Например, в следующем фрагменте программы показан способ, который используют многие C#-программисты для создания бесконечного цикла.

```
for(;;) // Специально созданный бесконечный цикл.
{
  // ...
}
```

Этот цикл будет работать без конца. Несмотря на существование некоторых задач программирования (например, командных процессоров операционных систем), которые требуют наличия бесконечного цикла, большинство "бесконечных циклов" — это просто циклы со специальными требованиями к завершению. Ближе к концу этой главы будет показано, как завершить цикл такого типа. (Подсказка: с помощью инструкции `break`.)

Циклы без тела

В C# тело, связанное с циклом `for` (или каким-нибудь другим циклом), может быть пустым. Дело в том, что *пустая инструкция* синтаксически допустима. "Бестелесные" циклы часто оказываются полезными. Например, следующая программа использует "бестелесный" цикл для получения суммы чисел от 1 до 5.

```
// Тело цикла может быть пустым.

using System;

class Empty3 {
  public static void Main() {
    int i;
    int sum = 0;

    // Суммируем числа от 1 до 5.
    for(i = 1; i <= 5; sum += i++) ;

    Console.WriteLine("Сумма равна " + sum);
  }
}
```

Результат работы этой программы весьма лаконичен:

```
Сумма равна 15
```

Обратите внимание на то, что процесс суммирования полностью выполняется внутри инструкции `for`, поэтому и в теле цикла отпала необходимость. Особое внимание обратите на итерационное выражение:

```
sum += i++
```

Не стоит пугаться инструкций, подобных этой. Они весьма распространены в профессиональной среде и легко понимаются, если их разбить на части. Эта инструкция означает, что в переменную `sum` необходимо поместить результат сложения текущего значения переменной `sum` и значения переменной `i`, а затем инкрементировать

значение переменной i. Таким образом, предыдущая инструкция эквивалентна следующим:

```
sum = sum + i;
i++;
```

Объявление управляющей переменной в цикле for

Часто переменная, которая управляет циклом for, необходима только для этого цикла и больше никак не используется. В этом случае можно объявить ее в разделе инициализации цикла. Например, следующая программа вычисляет как сумму, так и факториал чисел от 1 до 5. Управляющая переменная i здесь объявляется в цикле for.

```
// Объявление управляющей переменной в цикле for.

using System;

class ForVar {
  public static void Main() {
    int sum = 0;
    int fact = 1;

    // Вычисляем сумму и факториал чисел от 1 до 5.
    for(int i = 1; i <= 5; i++) {
      sum += i;  // i известна только в пределах цикла.
      fact *= i;
    }

    // Но здесь переменная i неизвестна.

    Console.WriteLine("Сумма равна " + sum);
    Console.WriteLine("Факториал равен " + fact);
  }
}
```

При объявлении переменной внутри цикла for необходимо помнить следующее: ее область видимости завершается с завершением этого цикла. Другими словами, область видимости этой переменной ограничена циклом for. Вне цикла такая переменная прекращает свое существование. Таким образом, в предыдущем примере переменная i недоступна вне цикла for. Если нужно использовать управляющую переменную цикла еще где-то в программе, вы не должны объявлять ее внутри цикла for.

Прежде чем двигаться дальше, не помешало бы поэкспериментировать с собственными вариациями на тему цикла for.

Цикл while

Общая форма цикла while имеет такой вид:

while(*условие*)*инструкция*;

Здесь под элементом *инструкция* понимается либо одиночная инструкция, либо блок инструкций. Работой цикла управляет элемент *условие*, который представляет собой любое допустимое выражение типа bool. Элемент *инструкция* выполняется до тех пор, пока условное выражение возвращает значение ИСТИНА. Как только это *условие* становится ложным, управление передается инструкции, которая следует за этим циклом.

Перед вами простой пример, в котором цикл while используется для вычисления порядка заданного целого числа.

```
// Вычисление порядка целого числа.

using System;

class WhileDemo {
  public static void Main() {
    int num;
    int mag;

    num = 435679;
    mag = 0;

    Console.WriteLine("Число: " + num);

    while(num > 0) {
      mag++;
      num = num / 10;
    };

    Console.WriteLine("Порядок: " + mag);
  }
}
```

А вот результаты выполнения этой программы:

```
Число: 435679
Порядок: 6
```

Цикл while работает следующим образом. Проверяется значение переменной num. Если оно больше нуля, счетчик mag инкрементируется, а значение num делится на 10. Цикл повторяется до тех пор, пока num больше нуля. Когда num станет равным нулю, цикл завершится, а переменная mag будет содержать порядок исходного числа.

Подобно циклу for, условное выражение проверяется при входе в цикл while, а это значит, что тело цикла может не выполниться ни разу. Это свойство цикла (иллюстрируемое следующей программой) устраняет необходимость отдельного тестирования до начала цикла.

```
// Вычисление целых степеней числа 2.

using System;

class Power {
  public static void Main() {
    int e;
    int result;

    for(int i=0; i < 10; i++) {
      result = 1;
      e = i;

      while(e > 0) {
        result *= 2;
        e--;
      }

      Console.WriteLine("2 в степени " + i +
                        " равно " + result);
```

```
    }
  }
}
```

Результаты выполнения этой программы выглядят так:

```
2 в степени 0 равно 1
2 в степени 1 равно 2
2 в степени 2 равно 4
2 в степени 3 равно 8
2 в степени 4 равно 16
2 в степени 5 равно 32
2 в степени 6 равно 64
2 в степени 7 равно 128
2 в степени 8 равно 256
2 в степени 9 равно 512
```

Обратите внимание на то, что цикл `while` выполняется только в том случае, если значение переменной `e` больше нуля. Таким образом, когда `e` равно нулю, что имеет место в первой итерации цикла `for`, цикл `while` опускается.

Цикл do-while

Третьим циклом в C# является цикл `do-while`. В отличие от циклов `for` и `while`, в которых условие проверяется при входе, цикл `do-while` проверяет условие при выходе из цикла. Это значит, что цикл `do-while` всегда выполняется хотя бы один раз. Его общий формат имеет такой вид:

```
do {
   инструкции;
} while (условие);
```

Несмотря на то что фигурные скобки необязательны, если элемент *инструкции* состоит только из одной инструкции, они часто используются для улучшения читабельности конструкции `do-while`, не допуская тем самым путаницы с циклом `while`. Цикл `do-while` выполняется до тех пор, пока остается истинным элемент *условие*, который представляет собой условное выражение.

В следующей программе цикл `do-while` используется для отображения в обратном порядке цифр, составляющих заданное целое число.

```
// Отображение в обратном порядке цифр целого числа.

using System;

class DoWhileDemo {
  public static void Main() {
    int num;
    int nextdigit;

    num = 198;

    Console.WriteLine("Число: " + num);

    Console.Write("Число с обратным порядком цифр: ");

    do {
      nextdigit = num % 10;
      Console.Write(nextdigit);
```

```
      num = num / 10;
    } while(num > 0);

    Console.WriteLine();
  }
}
```

Результат выполнения этой программы выглядит так:

```
Число: 198
Число с обратным порядком цифр: 891
```

Вот как работает этот цикл. На каждой итерации крайняя справа цифра определяется как остаток от целочисленного деления заданного числа на 10. Полученная цифра тут же отображается на экране. Затем результат этого деления запоминается в той же переменной `num`. Поскольку деление целочисленное, его результат равносилен отбрасыванию крайней правой цифры. Этот процесс повторяется до тех пор, пока число `num` не станет равным нулю.

Цикл `foreach`

Цикл `foreach` предназначен для опроса элементов *коллекции*. Коллекция — это группа объектов. В C# определено несколько типов коллекций, среди которых можно выделить массив. Цикл `foreach` рассматривается в главе 7, посвященной массивам.

Использование инструкции `break` для выхода из цикла

С помощью инструкции `break` можно организовать немедленный выход из цикла, опустив выполнение кода, оставшегося в его теле, и проверку условного выражения. При обнаружении внутри цикла инструкции `break` цикл завершается, а управление передается инструкции, следующей после цикла. Рассмотрим простой пример.

```
// Использование инструкции break для выхода из цикла.

using System;

class BreakDemo {
  public static void Main() {

    // Используем break для выхода из цикла.
    for(int i=-10; i <= 10; i++) {
      if(i > 0) break; // Завершение цикла при i > 0.
      Console.Write(i + " ");
    }
    Console.WriteLine("Готово!");
  }
}
```

Эта программа генерирует следующие результаты:

```
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 Готово!
```

Как видите, несмотря на то, что этот цикл `for` спроектирован для перебора значений `i` в диапазоне от −10 до 10, инструкция `break` "досрочно" прекращает его выполнение, когда значение переменной `i` становится положительным.

Инструкцию `break` можно использовать с любым C#-циклом, включая "бесконечный". Например, предыдущая программа, переделанная для использования `do-while`-цикла, имеет следующий вид:

```
// Использование инструкции break для выхода
// из цикла do-while.

using System;

class BreakDemo2 {
  public static void Main() {
    int i;

    i = -10;
    do {
      if(i > 0) break;
      Console.Write(i + " ");
      i++;
    } while(i <= 10);

    Console.WriteLine("Готово!");
  }
}
```

Теперь рассмотрим более реальный пример. Следующая программа находит наименьший множитель заданного числа.

```
// Определение наименьшего множителя числа.

using System;

class FindSmallestFactor {
  public static void Main() {
    int factor = 1;
    int num = 1000;

    for(int i=2; i < num/2; i++) {
      if((num%i) == 0) {
        factor = i;
        break; // Цикл прекращается, когда найден множитель.
      }
    }
    Console.WriteLine(
              "Наименьший множитель равен " + factor);
  }
}
```

Результаты выполнения этой программы выглядят так:

```
Наименьший множитель равен 2
```

Здесь инструкция `break` останавливает выполнение цикла `for`, как только находит множитель числа. Тем самым предотвращается попытка опробовать любые другие значения — кандидаты на "звание" множителя.

При использовании внутри множества вложенных циклов инструкция `break` прерывает только самый внутренний цикл.

```
// Использование инструкции break с вложенными циклами.

using System;

class BreakNested {
  public static void Main() {

    for(int i=0; i<3; i++) {
      Console.WriteLine(
             "Подсчет итераций внешнего цикла: " + i);
      Console.Write(
             "    Подсчет итераций внутреннего цикла: ");

      int t = 0;
      while(t < 100) {
        if(t == 10) break; // Останов цикла, когда
                           // t равно 10.
        Console.Write(t + " ");
        t++;
      }
      Console.WriteLine();
    }
    Console.WriteLine("Циклы завершены.");
  }
}
```

Результаты работы этой программы выглядят следующим образом:

```
Подсчет итераций внешнего цикла: 0
    Подсчет итераций внутреннего цикла: 0 1 2 3 4 5 6 7 8 9
Подсчет итераций внешнего цикла: 1
    Подсчет итераций внутреннего цикла: 0 1 2 3 4 5 6 7 8 9
Подсчет итераций внешнего цикла: 2
    Подсчет итераций внутреннего цикла: 0 1 2 3 4 5 6 7 8 9
Циклы завершены.
```

Как видите, инструкция `break`, находящаяся во внутреннем цикле, прекращает выполнение только этого цикла, а на внешний не оказывает никакого воздействия.

Хотелось бы также обратить ваше внимание на то, что в одном цикле можно использовать не одну, а несколько инструкций `break`, однако слишком большое их количество способно нарушить структуру кода. И еще. Инструкция `break`, которая завершает выполнение инструкции `switch`, влияет только на инструкцию `switch`, а не на содержащий ее цикл.

Использование инструкции `continue`

Помимо средства "досрочного" выхода из цикла, существует средство "досрочного" выхода из текущей его итерации. Этим средством является инструкция `continue`. Она принудительно выполняет переход к следующей итерации, опуская выполнение оставшегося кода в текущей. Инструкцию `continue` можно расценивать как дополнение к более "радикальной" инструкции `break`. Например, в следующей программе используется инструкция `continue` для "ускоренного" поиска четных чисел в диапазоне от 0 до 100.

```
// Использование инструкции continue.

using System;
```

```
class ContDemo {
 public static void Main() {
   // Выводим четные числа между 0 и 100.
   for(int i = 0; i <= 100; i++) {
     if((i%2) != 0) continue; // Переход на следующую
                              // итерацию.
     Console.WriteLine(i);
   }
 }
}
```

Здесь выводятся только четные числа, поскольку при обнаружении нечетного числа происходит преждевременный переход к следующей итерации, а метод `WriteLine()` не вызывается.

В циклах `while` и `do-while` инструкция `continue` передает управление непосредственно инструкции, проверяющей условное выражение, после чего циклический процесс продолжает "идти своим чередом". А в цикле `for` после выполнения инструкции `continue` сначала вычисляется итерационное выражение, а затем — условное. И только после этого циклический процесс будет продолжен.

Инструкция `continue` используется программистами не слишком часто, хотя в некоторых случаях он оказывается весьма кстати.

Инструкция `return`

Инструкция `return` обеспечивает возврат из метода. Ее можно использовать для возвращения методом значения. Подробнее см. главу 6.

Инструкция `goto`

Инструкция `goto` — это C#-инструкция безусловного перехода. При ее выполнении управление программой передается инструкции, указанной с помощью метки. Долгие годы эта инструкция находилась в немилости у программистов, поскольку способствовала, с их точки зрения, созданию "спагетти-кода". Однако инструкция `goto` по-прежнему используется, и иногда даже очень эффективно. В этой книге не делается попытка "реабилитации" законных прав этой инструкции в качестве одной из форм управления программой. Более того, необходимо отметить, что в любой ситуации (в области программирования) можно обойтись без инструкции `goto`, поскольку она не является элементом, обеспечивающим полноту описания языка программирования. Вместе с тем в определенных ситуациях ее использование может быть очень полезным. В этой книге было решено ограничить использование инструкции `goto` рамками этого раздела, так как, по мнению большинства программистов, она вносит в программу лишь беспорядок и делает ее практически нечитабельной. Но поскольку использование инструкции `goto` в некоторых случаях может сделать намерение программиста яснее, ей стоит уделить некоторое внимание.

Инструкция `goto` требует наличие в программе метки. Метка — это действительный в C# идентификатор, за которым поставлено двоеточие. Метка должна находиться в одном методе с инструкцией `goto`, которая ссылается на эту метку. Например, с помощью `goto` можно организовать следующий цикл на 100 итераций:

```
x = 1;
loop1:
  x++;
  if(x < 100) goto loop1;
```

Инструкцию goto можно также использовать для перехода к case- или default-ветви внутри инструкции switch. Ведь, по сути, case- и default-инструкции являются метками. Следовательно, они могут принимать "эстафету" управления, передаваемую инструкцией goto. Но в этом случае инструкция goto должна обязательно находиться в "рамках" той же инструкции switch. Это значит, что с помощью какой-либо "внешней" инструкции goto нельзя попасть в инструкцию switch. Рассмотрим пример, который иллюстрирует использование инструкций goto и switch.

```
// Использование инструкций goto и switch.

using System;

class SwitchGoto {
  public static void Main() {

    for(int i=1; i < 5; i++) {
      switch(i) {
        case 1:
          Console.WriteLine("В ветви case 1");
          goto case 3;
        case 2:
          Console.WriteLine("В ветви case 2");
          goto case 1;
        case 3:
          Console.WriteLine("В ветви case 3");
          goto default;
        default:
          Console.WriteLine("В ветви default");
          break;
      }

      Console.WriteLine();
    }

//    goto case 1; // Ошибка! Нельзя впрыгнуть
                   // в инструкцию switch.
  }
}
```

Результаты выполнения этой программы выглядят так:

```
В ветви case 1
В ветви case 3
В ветви default

В ветви case 2
В ветви case 1
В ветви case 3
В ветви default

В ветви case 3
В ветви default

В ветви default
```

Обратите внимание на то, как используется инструкция goto для перехода к ветвям case и default инструкции switch. Обратите также внимание на то, что case-последовательность инструкций не завершается инструкцией break. Поскольку goto не позволяет case-последовательности "провалиться" в следующую case-последова-

тельность, то специальное средство от "провала" (break) не требуется. Как разъяснялось выше, инструкцию goto нельзя использовать для проникновения извне в switch-конструкцию. Если удалить символы комментария в начале строки

```
//    goto case 1; // Ошибка!...,
```

то программа не скомпилируется. Использование инструкции goto совместно с инструкцией switch может быть полезно в особых случаях, но не рекомендуется как общий стиль программирования.

Иногда инструкцию goto стоит использовать для выхода из глубоко вложенных инструкций. Вот простой пример:

```
// Демонстрация использования инструкции goto.

using System;

class Use_goto {
  public static void Main() {
    int i=0, j=0, k=0;

    for(i=0; i < 10; i++) {
      for(j=0; j < 10; j++ ) {
        for(k=0; k < 10; k++) {
          Console.WriteLine("i, j, k: " + i + " " +
                            j + " " + k);
          if(k == 3) goto stop;
        }
      }
    }

stop:
    Console.WriteLine("Все, хватит! i, j, k: " + i +
                      ", " + j + " " + k);

  }
}
```

Результаты выполнения этой программы выглядят так:

```
i, j, k: 0 0 0
i, j, k: 0 0 1
i, j, k: 0 0 2
i, j, k: 0 0 3
Все, хватит! i, j, k: 0, 0 3
```

Для того чтобы получить такие же результаты, но без инструкции goto, пришлось бы использовать три пары инструкций if и break. В данном случае инструкция goto существенно упрощает программный код. Несмотря на то что этот пример — совершенно искусственный, вы, вероятно, смогли представить ситуации, в которых применение инструкции goto может иметь преимущества перед другими вариантами.

Полный справочник по C#

Глава 6

Введение в классы, объекты и методы

В этой главе вы познакомитесь с классом. В нем вся суть C#, поскольку именно классом определяется природа объекта. Это — фундамент, на котором построен язык C#. Класс как таковой формирует основу для объектно-ориентированного программирования в C#. Внутри класса определяются данные и код действий, выполняемых над этими данными. Этот код сосредоточен в методах. Получив представление о классах, объектах и методах, вы сможете писать более сложные программы и лучше понимать ключевые элементы C#, описанные в следующих главах.

Введение в классы

Поскольку все C#-программы оформляются в виде класса, мы работаем с классами с самого начала этой книги. Конечно же, мы использовали самые простые классы, с помощью которых нельзя продемонстрировать все их достоинства. Как будет показано ниже, классы — это очень мощный инструмент, и вы еще сумеете оценить их богатые возможности.

Итак, начнем с азов. Класс — это шаблон, который определяет форму объекта. Он задает как данные, так и код, который оперирует этими данными. C# использует спецификацию класса для создания *объекта*. Объекты — это *экземпляры* класса. Таким образом, класс — это множество намерений (планов), определяющих, как должен быть построен объект. Важно четко понимать следующее: класс — это *логическая абстракция*. О ее реализации нет смысла говорить до тех пор, пока не создан объект класса, и в памяти не появилось физическое его представление.

И еще. Вспомните, что методы и переменные, составляющие класс, называются *членами* класса.

Общая форма определения класса

Определяя класс, вы определяете данные, которые он содержит, и код, манипулирующий этими данными. Несмотря на то что очень простые классы могут включать только код или только данные, большинство реальных классов содержат и то, и другое.

Данные содержатся в переменных экземпляров, определяемых классом, а код — в методах. Однако важно с самого начала отметить, что класс определяет также ряд специальных членов данных и методов-членов, например статические переменные, константы, конструкторы, деструкторы, индексаторы, события, операторы и свойства. Пока мы ограничимся рассмотрением переменных экземпляров и методов класса, а к концу главы познакомимся с конструкторами и деструкторами. Остальные типы членов класса описаны в последующих главах.

Класс создается с помощью ключевого слова `class`. Общая форма определения класса, который содержит только переменные экземпляров и методы, имеет следующий вид:

```
class имя_класса {
   // Объявление переменных экземпляров.
   доступ тип переменная1;
   доступ тип переменная2;
   //...
   доступ тип переменнаяN;

   // Объявление методов.
   доступ тип_возврата метод1(параметры) {
      // тело метода
   }
```

```
    доступ тип_возврата метод2(параметры) {
        // тело метода
    }

        //...
    доступ тип_возврата методN(параметры) {
        // тело метода
    }
}
```

Обратите внимание на то, что объявление каждой переменной и каждого метода предваряется элементом *доступ*. Здесь элемент *доступ* означает *спецификатор доступа* (например, `public`), который определяет, как к этому члену можно получить доступ. Как упоминалось в главе 2, члены класса могут быть закрытыми в рамках класса или более доступными. Спецификатор доступа определяет, какой именно разрешен тип доступа. Спецификатор доступа необязателен, и, если он не указан, подразумевается, что этот член закрыт (`private`). Члены с закрытым доступом (закрытые члены) могут использоваться только другими членами своего класса. В примерах этой главы все члены классов определяются как `public`-члены, а это значит, что их могут использовать все остальные составные части программного кода, даже те, которые определены вне класса. К рассмотрению спецификаторов доступа мы вернемся в главе 8.

Несмотря на отсутствие специального синтаксического правила определения класса (его качественного и количественного состава), все же считается, что класс должен определять только одну логическую сущность. Например, класс, в котором хранятся имена лиц и их телефонные номера, не должен (по общепринятым меркам) также содержать информацию о среднем количестве осадков, циклах возникновения пятен на Солнце и прочую не связанную с конкретными лицами информацию. Другими словами, правильно определенный класс должен содержать логически связанные данные. И наоборот, помещая в один класс логически несвязанные данные, вы рискуете деструктурировать свой код.

Классы, которые мы использовали в этой книге до сих пор, содержали только один метод — `Main()`. Вскоре мы узнаем, как создавать и другие методы. Однако заметьте, что в общей форме определения класса метод `Main()` не задан. Он нужен только в том случае, если определяемый класс является отправной точкой программы.

Определение класса

Для иллюстрации мы создадим класс, который инкапсулирует информацию о зданиях (домах, складских помещениях, офисах и пр.). В этом классе (назовем его `Building`) будут храниться три элемента информации о зданиях (количество этажей, общая площадь и количество жильцов).

Ниже представлена первая версия класса `Building`. В нем определены три переменные экземпляра: `floors`, `area` и `occupants`. Обратите внимание на то, что класс `Building` не содержит ни одного метода. Поэтому пока его можно считать классом данных. (В следующих разделах мы дополним его методами.)

```
class Building {
  public int floors;     // количество этажей
  public int area;       // общая площадь основания здания
  public int occupants; // количество жильцов
}
```

Переменные экземпляра, определенные в классе `Building`, иллюстрируют общий способ их объявления. Формат объявления переменной экземпляра такой:

```
доступ тип имя_переменной;
```

Здесь элемент *доступ* представляет спецификатор доступа, элемент *тип* — тип переменной экземпляра, а элемент *имя_переменной* — имя этой переменной. Таким образом, если не считать спецификатор доступа, то переменная экземпляра объявляется так же, как локальная переменная. В классе `Building` все переменные экземпляра объявлены с использованием модификатора доступа `public`, который, как упоминалось выше, позволяет получать к ним доступ со стороны кода, расположенного даже вне класса `Building`.

Определение `class` создает новый тип данных. В данном случае этот новый тип данных называется `Building`. Это имя можно использовать для объявления объектов типа `Building`. Помните, что объявление `class` — это лишь описание типа; оно не создает реальных объектов. Таким образом, предыдущий код не означает существования объектов типа `Building`.

Чтобы реально создать объект класса `Building`, используйте, например, такую инструкцию:

```
Building house = new Building(); // Создаем объект
                                 // типа Building.
```

После выполнения этой инструкции `house` станет экземпляром класса `Building`, т.е. обретет "физическую" реальность. Подробно эту инструкцию мы рассмотрим ниже.

При каждом создании экземпляра класса создается объект, который содержит собственную копию каждой переменной экземпляра, определенной этим классом. Таким образом, каждый объект класса `Building` будет содержать собственные копии переменных экземпляра `floors`, `area` и `occupants`. Для доступа к этим переменным используется *оператор "точка"* (`.`). Оператор "точка" связывает имя объекта с именем его члена. Общий формат этого оператора имеет такой вид:

объект.член

Как видите, объект указывается слева от оператора "точка", а его член — справа. Например, чтобы присвоить переменной `floors` значение 2, используйте следующую инструкцию.

```
house.floors = 2;
```

В общем случае оператор "точка" можно использовать для доступа как к переменным экземпляров, так и методам.

Рассмотрим полную программу, в которой используется класс `Building`.

```
// Программа, в которой используется класс Building.

using System;

class Building {
  public int floors;    // количество этажей
  public int area;      // общая площадь основания здания
  public int occupants; // количество жильцов
}

// Этот класс объявляет объект типа Building.
class BuildingDemo {
  public static void Main() {
    Building house = new Building(); // Создание объекта
                                     // типа Building.
    int areaPP; // Площадь, приходящаяся на одного жильца.

    // Присваиваем значения полям в объекте house.
    house.occupants = 4;
    house.area = 2500;
    house.floors = 2;
```

```
    // Вычисляем площадь, приходящуюся на одного жильца дома.
    areaPP = house.area / house.occupants;

    Console.WriteLine(
              "Дом имеет:\n  " +
              house.floors + " этажа\n  " +
              house.occupants + " жильца\n  " +
              house.area +
              " квадратных футов общей площади, из них\n  " +
              areaPP + " приходится на одного человека");
  }
}
```

Эта программа состоит из двух классов: `Building` и `BuildingDemo`. Внутри класса `BuildingDemo` метод `Main()` сначала создает экземпляр класса `Building` с именем `house`, а затем получает доступ к переменным этого экземпляра `house`, присваивая им конкретные значения и используя эти значения в вычислениях. Важно понимать, что `Building` и `BuildingDemo` — это два отдельных класса. Единственная связь между ними состоит в том, что один класс создает экземпляр другого. Хотя это отдельные классы, код класса `BuildingDemo` может получать доступ к членам класса `Building`, поскольку они объявлены открытыми, т.е. `public`-членами. Если бы в их объявлении не было спецификатора доступа `public`, доступ к ним ограничивался бы рамками класса `Building`, а класс `BuildingDemo` не имел бы возможности использовать их.

Если предыдущую программу назвать `UseBuilding.cs`, то в результате ее компиляции будет создан файл `UseBuilding.exe`. Классы `Building` и `BuildingDemo` автоматически становятся составными частями этого исполняемого файла. При его выполнении получим такие результаты:

```
Дом имеет:
  2 этажа
  4 жильца
  2500 квадратных футов общей площади, из них
  625 приходится на одного человека
```

В действительности совсем не обязательно классам `Building` и `BuildingDemo` находиться в одном исходном файле. Можно поместить каждый класс в отдельный файл и назвать эти файлы `Building.cs` и `BuildingDemo.cs`, соответственно. После этого необходимо дать компилятору команду скомпилировать оба файла и скомпоновать их. Для этого можно использовать следующую командную строку:

```
csc Building.cs BuildingDemo.cs
```

Если вы работаете в среде Visual Studio IDE, нужно поместить оба файла в проект и выполнить команду построения этого проекта.

Прежде чем идти дальше, имеет смысл вспомнить основной принцип программирования классов: каждый объект класса имеет собственные копии переменных экземпляра, определенных в этом классе. Таким образом, содержимое переменных в одном объекте может отличаться от содержимого аналогичных переменных в другом. Между двумя объектами нет связи, за исключением того, что они являются объектами одного и того же типа. Например, если у вас есть два объекта типа `Building` и каждый объект имеет свою копию переменных `floors`, `area` и `occupants`, то содержимое соответствующих (одноименных) переменных этих двух экземпляров может быть разным. Следующая программа демонстрирует это.

```
// Эта программа создает два объекта класса Building.


using System;
```

```
class Building {
  public int floors;     // количество этажей
  public int area;       // общая площадь основания здания
  public int occupants; // количество жильцов
}

// Этот класс объявляет два объекта типа Building.
class BuildingDemo {
  public static void Main() {
    Building house = new Building();
    Building office = new Building();

    int areaPP; // Площадь, приходящаяся на одного жильца.

    // Присваиваем значения полям в объекте house.
    house.occupants = 4;
    house.area = 2500;
    house.floors = 2;

    // Присваиваем значения полям в объекте office.
    office.occupants = 25;
    office.area = 4200;
    office.floors = 3;

    // Вычисляем площадь, приходящуюся на одного жильца.
    areaPP = house.area / house.occupants;

    Console.WriteLine("Дом имеет:\n  " +
         house.floors + " этажа\n  " +
         house.occupants + " жильца\n  " +
         house.area +
         " квадратных футов общей площади, из них\n  " +
         areaPP + " приходится на одного человека");

    Console.WriteLine();

    // Вычисляем площадь, приходящуюся на одного
    // работника офиса.
    areaPP = office.area / office.occupants;

    Console.WriteLine("Офис имеет:\n  " +
          office.floors + " этажа\n  " +
          office.occupants + " работников\n  " +
          office.area +
          " квадратных футов общей площади, из них\n  " +
          areaPP + " приходится на одного человека");
  }
}
```

Вот каков результат выполнения этой программы:

```
Дом имеет:
  2 этажа
  4 жильца
  2500 квадратных футов общей площади, из них
  625 приходится на одного человека

Офис имеет:
  3 этажа
  25 работников
  4200 квадратных футов общей площади, из них
  168 приходится на одного человека
```

Как видите, данные о доме (содержащиеся в объекте `house`) совершенно не связаны с данными об офисе (содержащимися в объекте `office`). Эта ситуация отображена на рис. 6.1.

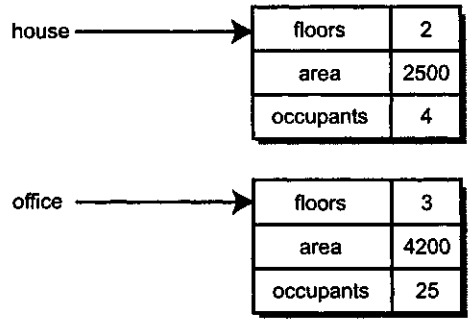


Рис. 6.1. Переменные экземпляров не связаны

Создание объектов

В предыдущих программах с помощью следующей строки был объявлен объект типа `Building`.

```
Building house = new Building();
```

Это объявление выполняет две функции. Во-первых, оно объявляет переменную с именем `house` классового типа `Building`. Но эта переменная не определяет объект, а может лишь *ссылаться* на него. Во-вторых, рассматриваемое объявление создает реальную физическую копию объекта и присваивает переменной `house` ссылку на этот объект. И все это — "дело рук" оператора `new`. Таким образом, после выполнения приведенной выше строки кода переменная `house` будет ссылаться на объект типа `Building`.

Оператор `new` динамически (т.е. во время выполнения программы) выделяет память для объекта и возвращает ссылку на него. Эта ссылка (сохраненная в конкретной переменной) служит адресом объекта в памяти, выделенной для него оператором `new`. Таким образом, в C# для всех объектов классов должна динамически выделяться память.

Предыдущую инструкцию, объединяющую в себе два действия, можно переписать в виде двух инструкций.

```
Building house;           // Объявление ссылки на объект.
house = new Building();   // Выделение памяти для объекта
                          // типа Building.
```

В первой строке объявляется переменная `house` как ссылка на объект типа `Building`. Поэтому `house` — это переменная, которая может ссылаться на объект, но не сам объект. В этот момент (после выполнения первой инструкции) переменная `house` содержит значение `null`, которое означает, что она не ссылается ни на какой объект. После выполнения второй инструкции будет создан новый объект класса `Building`, а ссылка на него будет присвоена переменной `house`. Вот теперь ссылка `house` связана с объектом.

Тот факт, что к объектам классов доступ осуществляется посредством ссылок, объясняет, почему классы называются *ссылочными типами*. Ключевое различие между ти-

пами значений и ссылочными типами состоит в значении, которое содержит переменная каждого типа. Переменная типа значения сама содержит значение. Например, после выполнения следующих инструкций

```
int x;
x = 10;
```

переменная x содержит значение 10, поскольку x — это переменная типа int, т.е. переменная типа значения. Но при выполнении инструкции

```
Building house = new Building();
```

сама переменная house будет содержать не объект, а ссылку на этот объект.

Переменные ссылочного типа и присвоение им значений

В операции присвоения ссылочные переменные действуют не так, как переменные типа значений (например, типа int). Присваивая одной переменной (типа значения) значение другой, мы имеем довольно простую ситуацию. Переменная слева (от оператора присваивания) получает *копию значения* переменной справа. При выполнении аналогичной (казалось бы) операции присваивания между двумя переменными ссылочного типа ситуация усложняется, поскольку мы изменяем объект, на который ссылается ссылочная переменная, что может привести к неожиданным результатам. Например, рассмотрим следующий фрагмент программы:

```
Building house1 = new Building();
Building house2 = house1;
```

На первый взгляд может показаться, что house1 и house2 ссылаются на различные объекты, но это не так. Обе переменные, house1 и house2, ссылаются на *один и тот же* объект. Присвоение значения переменной house1 переменной house2 просто заставляет переменную house2 ссылаться на тот же объект, на который ссылается и переменная house1. В результате на этот объект можно воздействовать, используя либо имя house1, либо имя house2. Например, присвоив

```
house1.area = 2600;
```

мы добьемся того, что обе инструкции

```
Console.WriteLine(house1.area);
Console.WriteLine(house2.area);
```

отобразят одно и то же значение — 2600.

Несмотря на то что обе переменные, house1 и house2, ссылаются на один и тот же объект, они никак не связаны между собой. Например, очередное присвоение переменной house2 просто заменяет объект, на который она ссылается. После выполнения последовательности инструкций

```
Building house1 = new Building();
Building house2 = house1;
Building house3 = new Building();

Building house2 = house3; // Теперь переменные house2 и
                          // house3 ссылаются на один и
                          // тот же объект.
```

переменная house2 будет ссылаться на тот же объект, на который ссылается переменная house3. Объект, на который ссылается переменная house1, не меняется.

Методы

Как упоминалось выше, переменные экземпляров и методы — две основные составляющие классов. Пока наш класс `Building` содержит только данные. Хотя такие классы (без методов) вполне допустимы, большинство классов имеют методы. *Методы* — это процедуры (подпрограммы), которые манипулируют данными, определенными в классе, и во многих случаях обеспечивают доступ к этим данным. Обычно различные части программы взаимодействуют с классом посредством его методов.

Любой метод содержит одну или несколько инструкций. В хорошей C#-программе один метод выполняет только одну задачу. Каждый метод имеет имя, и именно это имя используется для его вызова. В общем случае методу можно присвоить любое имя. Но помните, что имя `Main()` зарезервировано для метода, с которого начинается выполнение программы. Кроме того, в качестве имен методов нельзя использовать ключевые слова C#.

Имена методов в тексте этой книги сопровождаются парой круглых скобок. Например, если метод имеет имя `getval`, то в тексте будет написано `getval()`. Это помогает отличать имена переменных от имен методов.

Формат записи метода такой:

```
доступ тип_возврата имя(список_параметров) {
      // тело метода
}
```

Здесь элемент *доступ* означает *модификатор доступа*, который определяет, какие части программы могут получить доступ к методу. Как упоминалось выше, модификатор доступа необязателен, и, если он не указан, подразумевается, что метод закрыт (`private`) в рамках класса, где он определен. Пока мы будем объявлять все методы как `public`-члены, чтобы их могли вызывать все остальные составные части программного кода, даже те, которые определены вне класса.

С помощью элемента *тип_возврата* указывается тип значения, возвращаемого методом. Это может быть любой допустимый тип, включая типы классов, создаваемые программистом. Если метод не возвращает никакого значения, необходимо указать тип `void`. Имя метода, как нетрудно догадаться, задается элементом *имя*. В качестве имени метода можно использовать любой допустимый идентификатор, отличный от тех, которые уже использованы для других элементов программы в пределах текущей области видимости. Элемент *список_параметров* представляет собой последовательность пар (состоящих из типа данных и идентификатора), разделенных запятыми. Параметры — это переменные, которые получают значения аргументов, передаваемых методу при вызове. Если метод не имеет параметров, *список_параметров* остается пустым.

Добавление методов в класс `Building`

Как вам уже известно, методы класса, как правило, манипулируют данными, определенными в классе, и обеспечивают доступ к этим данным. Зная это, вспомним, что метод `Main()` в предыдущей программе вычислял площадь, приходящуюся на одного человека, путем деления общей площади здания на количество жильцов.

Несмотря на формальную корректность, эти вычисления выполнены не самым удачным образом. Ведь с вычислением площади, приходящейся на одного человека, вполне может справиться сам класс `Building`, поскольку эта величина зависит только от значений переменных `area` и `occupants`, которые инкапсулированы в классе `Building`. Как говорится, сам Бог велел классу `Building` выполнить это арифметиче-

ское действие. Более того, если оно таки будет "закреплено" за этим классом, то другой программе, которая его использует, не придется делать это действие "вручную". Здесь налицо не просто удобство для "других" программ, а предотвращение неоправданного дублирования кода. Наконец, внося в класс `Building` метод, который вычисляет площадь, приходящуюся на одного человека, вы улучшаете его объектно-ориентированную структуру, инкапсулируя внутри рассматриваемого класса величины, связанные непосредственно со зданием.

Чтобы добавить в класс `Building` метод, необходимо определить его внутри объявления класса. Например, следующая версия класса `Building` содержит метод с именем `areaPerPerson()`, который отображает значение площади конкретного здания, приходящейся на одного человека.

```
// Добавление метода в класс Building.

using System;

class Building {
  public int floors;    // количество этажей
  public int area;      // общая площадь здания
  public int occupants; // количество жильцов

  // Отображаем значение площади, приходящейся
  // на одного человека.
  public void areaPerPerson() {
    Console.WriteLine("  " + area / occupants +
                      " приходится на одного человека");
  }
}

// Используем метод areaPerPerson().
class BuildingDemo {
  public static void Main() {
    Building house = new Building();
    Building office = new Building();


    // Присваиваем значения полям в объекте house.
    house.occupants = 4;
    house.area = 2500;
    house.floors = 2;

    // Присваиваем значения полям в объекте office.
    office.occupants = 25;
    office.area = 4200;
    office.floors = 3;


    Console.WriteLine("Дом имеет:\n  " +
             house.floors + " этажа\n  " +
             house.occupants + " жильца\n  " +
             house.area +
             " квадратных футов общей площади, из них");
    house.areaPerPerson();

    Console.WriteLine();

    Console.WriteLine("Офис имеет:\n  " +
```

```
                office.floors + " этажа\n  " +
                office.occupants + " работников\n  " +
                office.area +
                " квадратных футов общей площади, из них");
      office.areaPerPerson();
    }
}
```

Эта программа генерирует результаты, которые совпадают с предыдущими:

```
Дом имеет:
  2 этажа
  4 жильца
  2500 квадратных футов общей площади, из них
  625 приходится на одного человека

Офис имеет:
  3 этажа
  25 работников
  4200 квадратных футов общей площади, из них
  168 приходится на одного человека
```

Теперь рассмотрим ключевые элементы этой программы, начиная с самого метода `areaPerPerson()`. Первая строка этого метода выглядит так:

```
public void areaPerPerson() {
```

В этой строке объявляется метод с именем `areaPerPerson()`, который не имеет параметров. Этот метод определен с использованием спецификатора доступа `public`, поэтому его могут использовать все остальные части программы. Метод `areaPerPerson()` возвращает значение типа `void`, т.е. не возвращает никакого значения. Эта строка завершается открывающей фигурной скобкой, за которой должно находиться тело метода.

Так и есть. Тело метода `areaPerPerson()` состоит из единственной инструкции:

```
Console.WriteLine("  " + area / occupants +
                  " приходится на одного человека");
```

Эта инструкция отображает площадь здания, которая приходится на одного человека, путем деления значения переменной `area` на значение переменной `occupants`. Поскольку каждый объект типа `Building` имеет собственную копию значений `area` и `occupants`, то при вызове метода `areaPerPerson()` в вычислении площади здания, которая приходится на одного человека, будут использоваться копии этих переменных, принадлежащие конкретному вызывающему объекту.

Метод `areaPerPerson()` завершается закрывающей фигурной скобкой, т.е. при обнаружении закрывающей фигурной скобки управление программой передается вызывающему объекту.

Теперь рассмотрим внимательнее строку кода из метода `Main()`:

```
house.areaPerPerson();
```

Эта инструкция вызывает метод `areaPerPerson()` для объекта `house`. Как видите, для этого используется имя объекта, за которым следует оператор "точка". При вызове метода управление выполнением программы передается телу метода, а после его завершения управление возвращается автору вызова, и выполнение программы возобновляется со строки кода, которая расположена сразу за вызовом метода.

В данном случае в результате вызова `house.areaPerPerson()` отображается значение площади, которая приходится на одного человека, для здания, определенного объектом `house`. Точно так же в результате вызова `office.areaPerPerson()` отображается значение площади, которая приходится на одного человека, для здания,

определенного объектом office. Другими словами, каждый раз, когда вызывается метод areaPerPerson(), отображается значение площади, которая приходится на одного человека, для здания, описываемого заданным объектом.

Обратите внимание вот на что. Переменные экземпляра area и occupants используются внутри метода areaPerPerson() без каких бы то ни было атрибутов, т.е. им не предшествует ни имя объекта, ни оператор "точка". Это очень важный момент: если метод задействует переменную экземпляра, которая определена в его классе, он делает это напрямую, без явной ссылки на объект и без оператора "точка". И это логично. Ведь метод всегда вызывается для некоторого объекта конкретного класса. И если уж вызов состоялся, объект, стало быть, известен. Таким образом, нет необходимости указывать внутри метода объект во второй раз. Это значит, что значения area и occupants внутри метода areaPerPerson() неявно указывают на копии этих переменных, принадлежащих объекту, который вызывает метод areaPerPerson().

Возвращение из метода

В общем случае существует два варианта условий для возвращения из метода. Первый связан с обнаружением закрывающей фигурной скобки, обозначающей конец тела метода (как продемонстрировано на примере метода areaPerPerson()). Второй вариант состоит в выполнении инструкции return. Возможны две формы использования инструкции return: одна предназначена для void-методов (которые не возвращают значений), а другая — для возврата значений. В этом разделе мы рассмотрим первую форму, а в следующем — вторую.

Немедленное завершение void-метода можно организовать с помощью следующей формы инструкции return:

```
return;
```

При выполнении этой инструкции управление программой передается автору вызова метода, а оставшийся код опускается. Рассмотрим, например, следующий метод:

```
public void myMeth() {
  int i;

  for(i=0; i<10; i++) {
    if(i == 5) return; // Прекращение выполнения
                       // метода при i = 5.
    Console.WriteLine();
  }
}
```

Здесь цикл for будет работать при значениях i в диапазоне только от 0 до 5, поскольку, как только значение i станет равным 5, будет выполнен возврат из метода myMeth().

Метод может иметь несколько инструкций return. Например, выход из метода

```
public void myMeth() {
  //...
  if(done) return;
  //...
  if(error) return;
}
```

произойдет либо в случае его корректного завершения, либо при возникновении ошибки. Однако наличие слишком большого количества точек выхода из метода может деструктурировать код. Поэтому, несмотря на допустимость их множественного применения, следует все же использовать эту возможность с большой осторожностью.

Итак, завяжем "узелок на память": выход из void-метода может быть осуществлен двумя способами: по достижении закрывающей фигурной скобки или при выполнении инструкции return.

Возврат значения

Хотя void-методы — не редкость, большинство методов все же возвращают значение. И в самом деле, способность возвращать значение — одно из самых полезных качеств метода. Мы уже рассматривали пример возврата значения методом в главе 3, когда использовали метод Math.Sqrt() для получения квадратного корня.

Значения, возвращаемые методами, используются в программировании поразному. В одних случаях (как в методе Math.Sqrt()) возвращаемое значение является результатом вычислений, в других — оно просто означает, успешно или нет выполнены действия, составляющие метод, а в третьих — оно может представлять собой код состояния. Однако независимо от цели применения, использование значений, возвращаемых методами, является неотъемлемой частью C#-программирования.

Методы возвращают значения вызывающим их процедурам, используя следующую форму инструкции return:

```
return значение;
```

Здесь элемент *значение* и представляет значение, возвращаемое методом.

Способность методов возвращать значения можно использовать для улучшения реализации метода areaPerPerson(). Вместо того чтобы отображать значение площади, которая приходится на одного человека, метод areaPerPerson() будет теперь возвращать это значение, которое можно использовать в других вычислениях. В следующем примере представлен модифицированный вариант метода areaPerPerson(), который возвращает значение площади, приходящейся на одного человека, а не отображает его (как в предыдущем варианте).

```
// Демонстрация возврата значения методом areaPerPerson().

using System;

class Building {
  public int floors;     // количество этажей
  public int area;       // общая площадь здания
  public int occupants; // количество жильцов

  // Возврат значения площади, которая
  // приходится на одного человека.
  public int areaPerPerson() {
    return area / occupants;
  }
}

// Использование значения от метода areaPerPerson().
class BuildingDemo {
  public static void Main() {
    Building house = new Building();
    Building office = new Building();
    int areaPP; // Площадь, которая приходится
                // на одного человека.
    // Присваиваем значения полям в объекте house.
    house.occupants = 4;
    house.area = 2500;
    house.floors = 2;
```

```
    // Присваиваем значения полям в объекте office.
    office.occupants = 25;
    office.area = 4200;
    office.floors = 3;

    // Получаем для объекта house площадь, которая
    // приходится на одного человека..
    areaPP = house.areaPerPerson();

    Console.WriteLine("Дом имеет:\n  " +
          house.floors + " этажа\n  " +
          house.occupants + " жильца\n  " +
          house.area +
          " квадратных футов общей площади, из них\n  " +
          areaPP + " приходится на одного человека");


    Console.WriteLine();

    // Получаем площадь для объекта office, которая
    // приходится на одного человека..

    areaPP = office.areaPerPerson();

    Console.WriteLine("Офис имеет:\n  " +
          office.floors + " этажа\n  " +
          office.occupants + " работников\n  " +
          office.area +
          " квадратных футов общей площади, из них\n  " +
          areaPP + " приходится на одного человека");
  }
}
```

Результаты выполнения этого варианта программы аналогичны предыдущим.

Обратите внимание на вызов метода `areaPerPerson()`: его имя находится справа от оператора присваивания. В левой части стоит переменная, которая и получает значение, возвращаемое методом `areaPerPerson()`. Таким образом, после выполнения инструкции

```
areaPP = house.areaPerPerson();
```

значение площади, приходящейся на одного человека для объекта `house`, будет сохранено в переменной `areaPP`.

Обратите также внимание на то, что метод `areaPerPerson()` имеет в этом примере другой тип возвращаемого значения, а именно тип `int`. Это означает, что метод возвращает автору вызова целое число. Тип значения, возвращаемого методом, — очень важная характеристика метода, поскольку тип данных, возвращаемых методом, должен быть совместимым с типом возвращаемого значения, указанного в заголовке определения метода. Следовательно, если вы хотите, чтобы метод возвращал данные типа `double`, при его определении в качестве типа возвращаемого значения следует указать `double`.

Несмотря на корректность предыдущей программы, ее эффективность оставляет желать лучшего. В частности, нет необходимости в использовании переменной `areaPP`. Вызов метода `areaPerPerson()` можно реализовать непосредственно в инструкции вызова метода `WriteLine()`:

```
Console.WriteLine("Дом имеет:\n  " +
                  house.floors + " этажа\n  " +
```

```
                house.occupants + " жильца\n  " +
                house.area + " общей площади, из них\n  " +
                house.areaPerPerson() +
                        " приходится на одного человека");
```

В этом случае при вызове метода `WriteLine()` автоматически вызывается метод `house.areaPerPerson()`, и возвращаемое им значение передается методу `WriteLine()`. Более того, теперь обращение к методу `house.areaPerPerson()` можно использовать везде, где необходимо значение площади для объекта класса `Building`, которая приходится на одного человека. Например, в следующей инструкции сравниваются такие значения площадей двух зданий.

```
If (b1.areaPerPerson() > b2.areaPerPerson())
  Console.WriteLine(
      "В здании b1 больше места для каждого человека.");
```

Использование параметров

При вызове методу можно передать одно или несколько значений. Как упоминалось выше, значение, передаваемое методу, называется *аргументом*. Переменная внутри метода, которая принимает значение аргумента, называется *параметром*. Параметры объявляются внутри круглых скобок, которые следуют за именем метода. Синтаксис объявления параметров аналогичен синтаксису, применяемому для переменных. Параметр находится в области видимости своего метода, и, помимо специальной задачи получения аргумента, действует подобно любой локальной переменной.

Перед вами простой пример использования метода с параметром. В классе `ChkNum` определен метод `isPrime`, который возвращает значение `true`, если переданное ему значение является простым, и значение `false` в противном случае. Следовательно, метод `isPrime` возвращает значение типа `bool`.

```
// Простой пример использования параметра.

using System;

class ChkNum {
  // Метод возвращает true, если x - простое число.
  public bool isPrime(int x) {
    for(int i=2; i < x/2 + 1; i++)
      if((x %i) == 0) return false;

    return true;
  }
}

class ParmDemo {
  public static void Main() {
    ChkNum ob = new ChkNum();

    for(int i=1; i < 10; i++)
      if(ob.isPrime(i)) Console.WriteLine(i +
                                " простое число.");
      else Console.WriteLine(i + " не простое число.");

  }
}
```

Эта программа генерирует следующие результаты:

```
1 простое число.
2 простое число.
3 простое число.
4 не простое число.
5 простое число.
6 не простое число.
7 простое число.
8 не простое число.
9 не простое число.
```

В этой программе метод `isPrime` вызывается девять раз, и каждый раз ему передается новое значение. Рассмотрим этот процесс более внимательно. Во-первых, обратите внимание на то, как происходит обращение к методу `isPrime`. Передаваемый аргумент указывается между круглыми скобками. При первом вызове метода `isPrime` ему передается значение 1. Перед началом выполнения этого метода параметр `x` получит значение 1. При втором вызове аргумент будет равен числу 2, а значит, и параметр получит значение 2 и т.д. Важно то, что значение, переданное как аргумент при вызове функции `isPrime`, представляет собой значение, получаемое параметром `x`.

Метод может иметь более одного параметра. В этом случае достаточно объявить каждый параметр, отделив его от следующего запятой. Расширим, например, уже знакомый нам по предыдущей программе класс `ChkNum`, добавив в него метод `lcd()`, который возвращает наименьший общий знаменатель (*l*east *c*ommon *d*enominator) для двух передаваемых ему значений.

```
// Добавляем метод, который принимает два аргумента.

using System;

class ChkNum {
  // Метод возвращает true, если x - простое число.
  public bool isPrime(int x) {
    for(int i=2; i < x/2 + 1; i++)
      if((x %i) == 0) return false;

    return true;
  }

  // Метод возвращает наименьший общий знаменатель.
  public int lcd(int a, int b) {
    int max;

    if(isPrime(a) | isPrime(b)) return 1;

    max = a < b ? a : b;

    for(int i=2; i < max/2 + 1; i++)
      if(((a%i) == 0) & ((b%i) == 0)) return i;

    return 1;
  }
}

class ParmDemo {
  public static void Main() {
    ChkNum ob = new ChkNum();
    int a, b;

    for(int i=1; i < 10; i++)
```

```
      if(ob.isPrime(i)) Console.WriteLine(i +
                                  " простое число.");
      else Console.WriteLine(i + " не простое число.");

    a = 7;
    b = 8;
    Console.WriteLine("Наименьший общий знаменатель для " +
                      a + " и " + b + " равен " +
                      ob.lcd(a, b));

    a = 100;
    b = 8;
    Console.WriteLine("Наименьший общий знаменатель для " +
                      a + " и " + b + " равен " +
                      ob.lcd(a, b));

    a = 100;
    b = 75;
    Console.WriteLine("Наименьший общий знаменатель для " +
                      a + " и " + b + " равен " +
                      ob.lcd(a, b));

  }
}
```

Обратите внимание на то, что при вызове метода `lcd()` аргументы также разделяются запятыми. Вот результаты выполнения этой программы:

```
1 простое число.
2 простое число.
3 простое число.
4 не простое число.
5 простое число.
6 не простое число.
7 простое число.
8 не простое число.
9 не простое число.
Наименьший общий знаменатель для 7 и 8 равен 1
Наименьший общий знаменатель для 100 и 8 равен 2
Наименьший общий знаменатель для 100 и 75 равен 5
```

При передаче методу нескольких параметров каждый из них должен сопровождаться указанием собственного типа, причем типы параметров могут быть различными. Например, следующая запись вполне допустима:

```
int myMeth(int a, double b, float c) {
  // ...
```

Добавление параметризованного метода в класс `Building`

Для добавления в класс `Building` нового средства (вычисления максимально допустимого количества обитателей здания) можно использовать параметризованный метод. При этом предполагается, что площадь, приходящаяся на каждого человека, не должна быть меньше определенного минимального значения. Назовем этот новый метод `maxOccupant()` и приведем его определение.

```
/* Метод возвращает максимальное количество человек,
   если на каждого должна приходиться заданная
   минимальная площадь. */
```

```
public int maxOccupant(int minArea) {
  return area / minArea;
}
```

При вызове метода `maxOccupant()` параметр `minArea` получает значение минимальной площади, необходимой для жизнедеятельности каждого человека. Результат, возвращаемый методом `maxOccupant()`, получается как частное от деления общей площади здания на это значение.

Приведем полное определение класса `Building`, включающее метод `maxOccupant()`.

```
/*
   Добавляем параметризованный метод, вычисляющий
   максимальное количество человек, которые могут
   занимать это здание в предположении, что на каждого
   должна приходиться заданная минимальная площадь.
*/

using System;

class Building {
  public int floors;    // количество этажей
  public int area;      // общая площадь здания
  public int occupants; // количество жильцов

  // Метод возвращает площадь, которая приходится
  // на одного человека.
  public int areaPerPerson() {
    return area / occupants;
  }

  /* Метод возвращает максимальное возможное количество
     человек в здании, если на каждого должна приходиться
     заданная минимальная площадь. */
  public int maxOccupant(int minArea) {
    return area / minArea;
  }
}

// Использование метода maxOccupant().
class BuildingDemo {
  public static void Main() {
    Building house = new Building();
    Building office = new Building();

    // Присваиваем значения полям в объекте house.
    house.occupants = 4;
    house.area = 2500;
    house.floors = 2;

    // Присваиваем значения полям в объекте office.
    office.occupants = 25;
    office.area = 4200;
    office.floors = 3;

    Console.WriteLine(
                "Максимальное число человек для дома, \n" +
                "если на каждого должно приходиться " +
                300 + " квадратных футов: " +
```

```
                house.maxOccupant(300));

    Console.WriteLine(
                "Максимальное число человек для офиса, \n" +
                "если на каждого должно приходиться " +
                300 + " квадратных футов: " +
                office.maxOccupant(300));
  }
}
```

Результаты выполнения этой программы выглядят так.

```
Максимальное число человек для дома,
если на каждого должно приходиться 300 квадратных футов: 8
Максимальное число человек для офиса,
если на каждого должно приходиться 300 квадратных футов: 14
```

Как избежать написания недостижимого кода

При создании методов старайтесь не попадать в ситуации, когда часть кода ни при каких обстоятельствах не может быть выполнена. Никогда не выполняемый код называется *недостижимым* и считается некорректным в C#. Компилятор при обнаружении такого кода выдаст предупреждающее сообщение. Вот пример:

```
public void m() {
  char a, b;

  // ...
  if(a==b) {
     Console.WriteLine("равны");
     return;
  } else {
     Console.WriteLine("не равны");
     return;
  }
 Console.WriteLine("Это недостижимый код.");
}
```

Здесь последняя инструкция вызова метода `WriteLine()` в методе `m()` никогда не будет выполнена, поскольку до нее при любых обстоятельствах будет совершен выход из метода `m()`. При попытке скомпилировать этот метод вы получите предупреждение. В общем случае недостижимый код свидетельствует об ошибке с вашей стороны, поэтому имеет смысл серьезно отнестись к предупреждению компилятора.

Конструкторы

В предыдущих примерах переменные каждого `Building`-объекта устанавливались "вручную" с помощью следующей последовательности инструкций:

```
house.occupants = 4;
house.area = 2500;
house.floors = 2;
```

Профессионал никогда бы не использовал подобный подход. И дело не столько в том, что таким образом можно попросту "забыть" об одном или нескольких данных, сколько в том, что существует гораздо более удобный способ это сделать. Этот способ — использование конструктора.

Конструктор инициализирует объект при его создании. Он имеет такое же имя, что и сам класс, а синтаксически подобен методу. Однако в определении конструкторов не указывается тип возвращаемого значения. Формат записи конструктора такой:

```
доступ имя_класса() {
      // тело конструктора
}
```

Обычно конструктор используется, чтобы придать переменным экземпляра, определенным в классе, начальные значения или выполнить исходные действия, необходимые для создания полностью сформированного объекта. Кроме того, обычно в качестве элемента *доступ* используется модификатор доступа `public`, поскольку конструкторы, как правило, вызываются вне их класса.

Все классы имеют конструкторы независимо от того, определите вы их или нет, поскольку C# автоматически предоставляет конструктор по умолчанию, который инициализирует все переменные-члены, имеющие тип значений, нулями, а переменные-члены ссылочного типа — `null`-значениями. Но если вы определите собственный конструктор, конструктор по умолчанию больше не используется.

Вот пример использования конструктора:

```
// Использование простого конструктора.

using System;

class MyClass {
  public int x;

  public MyClass() {
    x = 10;
  }
}

class ConsDemo {
  public static void Main() {
    MyClass t1 = new MyClass();
    MyClass t2 = new MyClass();

    Console.WriteLine(t1.x + " " + t2.x);
  }
}
```

В этом примере программы конструктор класса `MyClass` имеет следующий вид:

```
public MyClass() {
    x = 10;
  }
```

Обратите внимание на `public`-определение конструктора, которое позволяет вызывать его из кода, определенного вне класса `MyClass`. Этот конструктор присваивает переменной экземпляра `x` значение 10. Конструктор `MyClass()` вызывается оператором `new` при создании объекта класса `MyClass`. Например, при выполнении строки

```
MyClass t1 = new MyClass();
```

для объекта `t1` вызывается конструктор `MyClass()`, который присваивает переменной экземпляра `t1.x` значение 10. То же самое справедливо и в отношении объекта `t2`, т.е. в результате создания объекта `t2` значение переменной экземпляра `t2.x` также станет равным 10. Таким образом, после выполнения этой программы получаем следующий результат:

```
10 10
```

Параметризованные конструкторы

В предыдущем примере использовался конструктор без параметров. Но чаще приходится иметь дело с конструкторами, которые принимают один или несколько параметров. Параметры вносятся в конструктор точно так же, как в метод: для этого достаточно объявить их внутри круглых скобок после имени конструктора. Например, в следующей программе используется параметризованный конструктор.

```
// Использование параметризованного конструктора.

using System;

class MyClass {
  public int x;

  public MyClass(int i) {
    x = i;
  }
}

class ParmConsDemo {
  public static void Main() {
    MyClass t1 = new MyClass(10);
    MyClass t2 = new MyClass(88);

    Console.WriteLine(t1.x + " " + t2.x);
  }
}
```

Результат выполнения этой программы выглядит так:

```
10 88
```

В конструкторе `MyClass()` этой версии программы определен один параметр с именем `i`, который используется для инициализации переменной экземпляра `x`. Таким образом, при выполнении строки кода

```
MyClass t1 = new MyClass(10);
```

параметру `i` передается значение 10, которое затем присваивается переменной экземпляра `x`.

Добавление конструктора в класс `Building`

Мы можем улучшить класс `Building`, добавив в него конструктор, который при создании объекта автоматически инициализирует поля (т.е. переменные экземпляра) `floors`, `area` и `occupants`. Обратите особое внимание на то, *как* создаются объекты класса `Building`.

```
// Добавление конструктора в класс Building.

using System;

class Building {
  public int floors;     // количество этажей
  public int area;       // общая площадь основания здания
  public int occupants; // количество жильцов


  public Building(int f, int a, int o) {
```

```
    floors = f;
    area = a;
    occupants = o;
  }

  // Метод возвращает значение площади, которая
  // приходится на одного человека.
  public int areaPerPerson() {
    return area / occupants;
  }

  /* Метод возвращает максимальное возможное количество
     человек в здании, если на каждого должна приходиться
     заданная минимальная площадь. */
  public int maxOccupant(int minArea) {
    return area / minArea;
  }
}

// Используем параметризованный конструктор Building().
class BuildingDemo {
  public static void Main() {
    Building house = new Building(2, 2500, 4);
    Building office = new Building(3, 4200, 25);

    Console.WriteLine(
                "Максимальное число человек для дома, \n" +
                "если на каждого должно приходиться " +
                300 + " квадратных футов: " +
                house.maxOccupant(300));

    Console.WriteLine(
                "Максимальное число человек для офиса, \n" +
                "если на каждого должно приходиться " +
                300 + " квадратных футов: " +
                office.maxOccupant(300));
  }
}
```

Результаты выполнения этой программы совпадают с результатами выполнения предыдущей ее версии.

Оба объекта, `house` и `office`, в момент создания инициализируются в программе конструктором `Building()`. Каждый объект инициализируется в соответствии с тем, как заданы параметры, передаваемые конструктору. Например, при выполнении строки

```
Building house = new Building(2, 2500, 4);
```

конструктору `Building()` передаются значения 2, 2500 и 4 в момент, когда оператор `new` создает объект класса `Building`. В результате этого копии переменных `floors`, `area` и `occupants`, принадлежащие объекту `house`, будут содержать значения 2, 2500 и 4, соответственно.

Использование оператора `new`

Теперь, когда вы больше знаете о классах и их конструкторах, можно подробнее ознакомиться с оператором `new`. Формат его таков:

переменная_типа_класса = new *имя_класса*();

Здесь элемент *переменная_типа_класса* означает имя создаваемой переменной типа класса. Нетрудно догадаться, что под элементом *имя_класса* понимается имя реализуемого в объекте класса. Имя класса вместе со следующей за ним парой круглых скобок — это ни что иное, как конструктор реализуемого класса. Если в классе конструктор не определен явным образом, оператор `new` будет использовать конструктор по умолчанию, который предоставляется средствами языка C#. Таким образом, оператор `new` можно использовать для создания объекта любого "классового" типа.

Поскольку объем памяти компьютера ограничен, вероятна ситуация, когда оператор `new` не сможет выделить область, необходимую для создаваемого объекта, по причине ее отсутствия в достаточном количестве. В этом случае возникнет исключительная ситуация соответствующего типа. (Как обрабатывать эту и другие исключительные ситуации, вы узнаете в главе 13.) Что касается программ, приведенных в этой книге, об "утечке" памяти беспокоиться не стоит, но в собственных программах вы всегда должны учитывать эту возможность.

Применение оператора `new` к переменным типа значений

Вероятно, вас удивил этот заголовок, и вы, возможно, попробовали бы заменить его таким: "Почему не следует применять оператор `new` к таким переменным типа значений, как `int` или `float`". В C# переменная типа значения содержит собственное значение. Во время компиляции программы компилятор автоматически выделяет память для хранения этого значения. Следовательно, нет необходимости использовать оператор `new` для явного выделения памяти. И напротив, в переменных ссылочного типа хранится ссылка на объект, а память для хранения этого объекта выделяется динамически, т.е. во время выполнения программы.

Отсутствие преобразования значений таких фундаментальных типов, как `int` или `char`, в значения ссылочных типов существенно улучшает производительность программы. При использовании же ссылочных типов существует уровень косвенности, который несет с собой дополнительные затраты системных ресурсов на доступ к каждому объекту. Этих дополнительных затрат нет при использовании типов значений.

Тем не менее вполне допустимо использовать оператор `new` и с типами значений. Вот пример:

```
int i = new int();
```

В этом случае вызывается конструктор по умолчанию для типа `int`, который инициализирует переменную `i` нулем. Рассмотрим следующую программу:

```
// Использование оператора new с типами значений.

using System;

class newValue {
  public static void Main() {
    int i = new int(); // Инициализация i нулем.

    Console.WriteLine("Значение переменной i равно: " + i);
  }
}
```

При выполнении этой программы мы видим следующие результаты:

```
Значение переменной i равно: 0
```

Как подтверждают результаты, переменная `i` действительно была установлена равной нулю. Вспомните: без оператора `new` переменная `i` осталась бы неинициализированной, и попытка использовать ее в методе `WriteLine()` без явного присвоения ей конкретного значения привела бы к ошибке.

В общем случае вызов оператора new для любого нессылочного типа означает вызов конструктора по умолчанию для соответствующего типа. Но в этом случае динамического выделения памяти не происходит. Большинство программистов не используют оператор new с нессылочными типами.

Сбор "мусора" и использование деструкторов

Как упоминалось выше, при использовании оператора new объектам динамически выделяется память из пула свободной памяти. Безусловно, объем буфера динамически выделяемой памяти не бесконечен, и рано или поздно свободная память может исчерпаться. Следовательно, результат выполнения оператора new может быть неудачным из-за недостатка свободной памяти для создания желаемого объекта. Поэтому одним из ключевых компонентов схемы динамического выделения памяти является восстановление свободной памяти от неиспользуемых объектов, что позволяет делать ее доступной для создания последующих объектов. Во многих языках программирования освобождение ранее выделенной памяти выполняется вручную. Например, в C++ для этого служит оператор delete. Однако в C# эта проблема решается по-другому, а именно с использованием *системы сбора мусора*.

Система сбора мусора C# автоматически возвращает память для повторного использования, действуя незаметно и без вмешательства программиста. Ее работа заключается в следующем. Если не существует ни одной ссылки на объект, то предполагается, что этот объект больше не нужен, и занимаемая им память освобождается. Эту (восстановленную) память снова можно использовать для размещения других объектов.

Система сбора мусора действует только спорадически во время выполнения отдельной программы. Эта система может и бездействовать: она не "включается" лишь потому, что существует один или несколько объектов, которые больше не используются в программе. Поскольку на сбор мусора требуется определенное время, динамическая система C# активизирует этот процесс только по необходимости или в специальных случаях. Таким образом, вы даже не будете знать, когда происходит сбор мусора, а когда — нет.

Деструкторы

Средства языка C# позволяют определить метод, который должен вызываться непосредственно перед тем, как объект будет окончательно разрушен системой сбора мусора. Этот метод называется *деструктором*, и его можно использовать для обеспечения гарантии "чистоты" ликвидации объекта. Например, вы могли бы использовать деструктор для гарантированного закрытия файла, открытого некоторым объектом.

Формат записи деструктора такой:

```
~имя_класса() {
  // код деструктора
}
```

Очевидно, что элемент *имя_класса* здесь означает имя класса. Таким образом, деструктор объявляется подобно конструктору за исключением того, что его имени предшествует символ "тильда" (~). (Подобно конструктору, деструктор не возвращает значения.)

Чтобы добавить деструктор в класс, достаточно включить его как член. Он вызывается в момент, предшествующий процессу утилизации объекта. В теле деструктора вы указываете действия, которые, по вашему мнению, должны быть выполнены перед разрушением объекта.

Важно понимать, что деструктор вызывается только перед началом работы системы сбора мусора и не вызывается, например, когда объект выходит за пределы области видимости. (Этим C#-деструкторы отличаются от C++-деструкторов, которые как раз *вызываются*, когда объект выходит за пределы области видимости.) Это означает, что вы не можете точно знать, когда будет выполнен деструктор. Однако точно известно, что все деструкторы будут вызваны перед завершением программы.

Использование деструктора демонстрируется в следующей программе, которая создает и разрушает большое количество объектов. В определенный момент выполнения этого процесса будет активизирован сбор мусора, а значит, вызваны деструкторы разрушаемых объектов.

```
// Демонстрация использования деструктора.

using System;

class Destruct {
  public int x;

  public Destruct(int i) {
    x = i;
  }

  // Вызывается при утилизации объекта.
  ~Destruct() {
    Console.WriteLine("Деструктуризация  " + x);
  }

  // Метод создает объект, который немедленно
  // разрушается.
  public void generator(int i) {
    Destruct o = new Destruct(i);
  }

}

class DestructDemo {
  public static void Main() {
    int count;

    Destruct ob = new Destruct(0);

    /* Теперь сгенерируем большое число объектов.
       В какой-то момент начнется сбор мусора.
       Замечание: возможно, для активизации этого
       процесса вам придется увеличить количество
       генерируемых объектов. */

    for(count=1; count < 100000; count++)
      ob.generator(count);

    Console.WriteLine("Готово!");
  }
}
```

Вот как работает эта программа. Конструктор устанавливает переменную экземпляра `x` равной известному числу. В данном примере `x` используется как ID (идентификационный номер) объекта. Деструктор отображает значение переменной `x` при утилизации объекта. Рассмотрим метод `generator()`. Он создает объект класса `Destruct`, а затем разрушает его с уведомлением об этом. Класс `DestructDemo` создает исходный объект класса `Destruct` с именем `ob`. Затем, используя объект `ob`, он

создает еще 100 000 объектов, вызывая для него метод `generator()`. В различные моменты этого процесса будет активизироваться сбор мусора. Насколько часто и когда именно, — зависит от таких факторов, как исходный объем свободной памяти, операционная система и пр. Но в некоторый момент времени на экране появится сообщение, сгенерированное деструктором. Если вы не увидите его до завершения программы (т.е. до вывода сообщения "Готово!"), попробуйте увеличить количество генерируемых объектов в цикле `for`.

Из-за недетерминированных условий вызова деструкторы не следует использовать для выполнения действий, которые должны быть привязаны к определенной точке программы. И еще. Существует возможность принудительного выполнения сбора мусора. Об этом вы прочтете в части II при рассмотрении библиотеки C#-классов. Все же в большинстве случаев процесс сбора мусора инициировать вручную не рекомендуется, так как это может снизить неэффективность работы программы. Кроме того, даже если в явном виде активизировать сбор мусора, то из-за особенностей организации этого процесса все равно не удастся точно узнать, когда утилизирован указанный объект.

Ключевое слово `this`

В заключение стоит представить ключевое слово `this`. При вызове метода ему автоматически передается неявно заданный аргумент, который представляет собой ссылку на вызывающий объект (т.е. объект, для которого вызывается метод). Эта ссылка и называется ключевым словом `this`. Чтобы понять смысл ссылки `this`, рассмотрим сначала программу, создающую класс `Rect`, который инкапсулирует значения ширины и высоты прямоугольника и включает метод `area()`, вычисляющий площадь прямоугольника.

```
using System;

class Rect {
  public int width;
  public int height;

  public Rect(int w, int h) {
    width = w;
    height = h;
  }

  public int area() {
    return width * height;
  }
}

class UseRect {
  public static void Main() {
    Rect r1 = new Rect(4, 5);
    Rect r2 = new Rect(7, 9);

    Console.WriteLine(
                "Площадь прямоугольника r1: " + r1.area());

    Console.WriteLine(
                "Площадь прямоугольника r2: " + r2.area());

  }
}
```

Как вам уже известно, внутри метода можно получить прямой доступ к другим членам класса, т.е. без указания имени объекта или класса. Таким образом, внутри метода `area()` инструкция

```
return width * height;
```

означает, что будут перемножены копии переменных `width` и `height`, связанные с вызывающим объектом, и метод вернет их произведение. Но та же самая инструкция может быть переписана следующим образом:

```
return this.width * this.height;
```

Здесь слово `this` ссылается на объект, для которого вызван метод `area()`. Следовательно, выражение `this.width` ссылается на копию переменной `width` этого объекта, а выражение `this.height` — на копию переменной `height` того же объекта. Например, если бы метод `area()` был вызван для объекта с именем `x`, то ссылка `this` в предыдущей инструкции была бы ссылкой на объект `x`. Запись этой инструкции без использования слова `this` — это по сути ее сокращенный вариант.

Вот как выглядит полный класс `Rect`, написанный с использованием ссылки `this`:

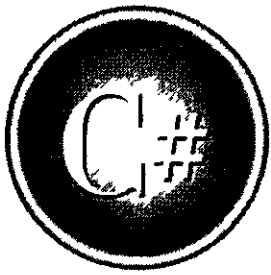
```
using System;

class Rect {
  public int width;
  public int height;

  public Rect(int w, int h) {
    this.width = w;
    this.height = h;
  }

  public int area() {
    return this.width * this.height;
  }
}

class UseRect {
  public static void Main() {
    Rect r1 = new Rect(4, 5);
    Rect r2 = new Rect(7, 9);

    Console.WriteLine(
                "Площадь прямоугольника r1: " + r1.area());

    Console.WriteLine(
                "Площадь прямоугольника r2: " + r2.area());

  }
}
```

В действительности ни один C#-программист не использует ссылку `this` так, как показано в этой программе, поскольку это не дает никакого выигрыша, да и стандартная форма выглядит проще. Однако из `this` можно иногда извлечь пользу. Например, синтаксис C# допускает, чтобы имя параметра или локальной переменной совпадало с именем переменной экземпляра. В этом случае локальное имя будет скрывать переменную экземпляра. И тогда доступ к скрытой переменной экземпляра можно получить с помощью ссылки `this`. Например, следующий фрагмент кода (хотя его стиль написания не рекомендуется к применению) представляет собой синтаксически допустимый способ определения конструктора `Rect()`.

```
public Rect(int width, int height) {
  this.width = width;
  this.height = height;
}
```

В этой версии конструктора имена параметров совпадают с именами переменных экземпляра, в результате чего за первыми скрываются вторые, а ключевое слово `this` как раз и используется для доступа к скрытым переменным экземпляра.

Полный справочник по



Глава 7

Массивы и строки

В этой главе мы возвращаемся к теме типов данных языка C# (познакомимся с типом данных `string`). Помимо массивов здесь будет уделено внимание использованию цикла `foreach`.

Массивы

Массив (array) — это коллекция переменных одинакового типа, обращение к которым происходит с использованием общего для всех имени. В C# массивы могут быть одномерными или многомерными, хотя в основном используются одномерные массивы. Массивы представляют собой удобное средство группирования связанных переменных. Например, массив можно использовать для хранения значений максимальных дневных температур за месяц, списка цен на акции или названий книг по программированию из домашней библиотеки.

Массив организует данные таким способом, который позволяет легко ими манипулировать. Например, если у вас есть массив, содержащий дивиденды, выплачиваемые по выбранной группе акций, то, построив цикл опроса всего массива, нетрудно вычислить средний доход от этих акций. Кроме того, организация данных в форме массива позволяет легко их сортировать в нужном направлении.

Хотя массивы в C# можно использовать по аналогии с тем, как они используются в других языках программирования, C#-массивы имеют один специальный атрибут, а именно: они реализованы как объекты. Этот факт и стал причиной того, что рассмотрение массивов в этой книге было отложено до введения понятия объекта Реализация массивов в виде объектов позволила получить ряд преимуществ, причем одно из них (и к тому же немаловажное) состоит в том, что неиспользуемые массивы могут автоматически утилизироваться системой сбора мусора.

Одномерные массивы

Одномерный массив — это список связанных переменных. Такие списки широко распространены в программировании Например, один одномерный массив можно использовать для хранения номеров счетов активных пользователей сети. В другом — можно хранить количество мячей, забитых в турнире бейсбольной командой.

Для объявления одномерного массива используется следующая форма записи.

```
тип[] имя_массива = new тип[размер];
```

Здесь с помощью элемента записи *тип* объявляется базовый тип массива. *Базовый тип* определяет тип данных каждого элемента, составляющего массив. Обратите внимание на *одну* пару квадратных скобок за элементом записи *тип*. Это означает, что определяется одномерный массив. Количество элементов, которые будут храниться в массиве, определяется элементом записи *размер*. Поскольку массивы реализуются как объекты, их создание представляет собой двухступенчатый процесс. Сначала объявляется ссылочная переменная на массив, а затем для него выделяется память, и переменной массива присваивается ссылка на эту область памяти. Таким образом, в C# массивы динамически размещаются в памяти с помощью оператора `new`.

На заметку

Если вы уже знакомы с языками C или C++, обратите внимание на объявление массивов в C# Квадратные скобки располагаются за именем типа, а не за именем массива

Рассмотрим пример. При выполнении приведенной ниже инструкции создается int-массив (состоящий из 10 элементов), который связывается со ссылочной переменной массива sample.

```
int[] sample = new int[10];
```

Это объявление работает подобно любому объявлению объекта. Переменная sample содержит ссылку на область памяти, выделенную оператором new.

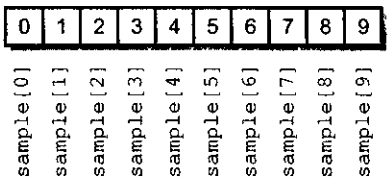
Доступ к отдельному элементу массива осуществляется посредством индекса. *Индекс* описывает позицию элемента внутри массива. В C# первый элемент массива имеет нулевой индекс. Поскольку массив sample содержит 10 элементов, его индексы изменяются от 0 до 9. Чтобы получить доступ к элементу массива по индексу, достаточно указать нужный номер элемента в квадратных скобках. Так, первым элементом массива sample является sample[0], а последним — sample[9]. Например, следующая программа помещает в массив sample числа от 0 до 9.

```
// Демонстрация использования одномерного массива.

using System;

class ArrayDemo {
  public static void Main() {
    int[] sample = new int[10];
    int i;

    for(i = 0; i < 10; i = i+1)
      sample[i] = i;

    for(i = 0; i < 10; i = i+1)
      Console.WriteLine("sample[" + i + "]: " +
                        sample[i]);
  }
}
```

Результаты выполнения этой программы имеют такой вид:

```
sample[0]: 0
sample[1]: 1
sample[2]: 2
sample[3]: 3
sample[4]: 4
sample[5]: 5
sample[6]: 6
sample[7]: 7
sample[8]: 8
sample[9]: 9
```

Схематично массив sample можно представить в таком виде:



Массивы широко применяются в программировании, поскольку позволяют легко обрабатывать большое количество связанных переменных. Например, следующая программа вычисляет среднее арифметическое от множества значений, хранимых в массиве nums, опрашивая в цикле for все его элементы.

```
// Вычисление среднего арифметического от
// множества значений.

using System;

class Average {
  public static void Main() {
    int[] nums = new int[10];
    int avg = 0;

    nums[0] = 99;
    nums[1] = 10;
    nums[2] = 100;
    nums[3] = 18;
    nums[4] = 78;
    nums[5] = 23;
    nums[6] = 63;
    nums[7] = 9;
    nums[8] = 87;
    nums[9] = 49;

    for(int i=0; i < 10; i++)
      avg = avg + nums[i];

    avg = avg / 10;

    Console.WriteLine("Среднее: " + avg);
  }
}
```

Вот результат выполнения программы:

```
Среднее: 53
```

Инициализация массива

В предыдущей программе значения массиву `nums` были присвоены вручную, т.е. с помощью десяти отдельных инструкций присваивания. Существует более простой путь достижения той же цели: массивы можно инициализировать в момент их создания. Формат инициализации одномерного массива имеет следующий вид:

тип`[]` *имя_массива* `=` `[`*val1*`,` *val2*`, ..., `*valN*`];`

Здесь начальные значения, присваиваемые элементам массива, задаются с помощью последовательности *val1*—*valN*. Значения присваиваются слева направо, в порядке возрастания индекса элементов массива. C# автоматически выделяет для массива область памяти достаточно большого размера, чтобы хранить заданные значения инициализации (инициализаторы). В этом случае нет необходимости использовать в явном виде оператор `new`. Теперь рассмотрим более удачный вариант программы `Average`.

```
// Вычисление среднего арифметического от
// множества значений.

using System;

class Average {
  public static void Main() {
    int[] nums = { 99, 10, 100, 18, 78, 23,
                   63, 9, 87, 49 };
```

```
    int avg = 0;;

    for(int i=0; i < 10; i++)
      avg = avg + nums[i];

    avg = avg / 10;

    Console.WriteLine("Среднее: " + avg);
  }
}
```

Хотя, как уже было отмечено выше, в этом нет необходимости, при инициализации массива все же можно использовать оператор new. Например, массив nums из предыдущей программы можно инициализировать и таким способом, хотя он и несет в себе некоторую избыточность.

```
int[] nums = new int[] { 99, 10, 100, 18, 78, 23,
                         63, 9, 87, 49 };
```

Несмотря на избыточность new-форма инициализации массива оказывается полезной в том случае, когда уже существующей ссылочной переменной массива присваивается новый массив. Например:

```
int[] nums;
nums = new int[] { 99, 10, 100, 18, 78, 23,
                   63, 9, 87, 49 };
```

Здесь массив nums объявляется в первой инструкции и инициализируется во второй.

И еще. При инициализации массива допустимо также явно указывать его размер, но размер в этом случае должен соответствовать количеству инициализаторов. Вот, например, еще один способ инициализации массива nums.

```
int[] nums = new int[10] { 99, 10, 100, 18, 78, 23,
                           63, 9, 87, 49 };
```

В этом объявлении размер массива nums явно задан равным 10.

Соблюдение "пограничного режима"

Границы массивов в C# строго "охраняются законом". Выход за границы расценивается как динамическая ошибка. Чтобы убедиться в этом, попытайтесь выполнить следующую программу, в которой намеренно делается попытка нарушения границ массива.

```
// Демонстрация выхода за границу массива.

using System;

class ArrayErr {
  public static void Main() {
    int[] sample = new int[10];
    int i;

    // Организуем нарушение границы массива.
    for(i = 0; i < 100; i = i+1)
      sample[i] = i;
  }
}
```

Как только i примет значение 10, будет сгенерирована исключительная ситуация типа IndexOutOfRangeException, и программа прекратит выполнение.

Многомерные массивы

Несмотря на то что в программировании чаще всего используются одномерные массивы, их многомерные "собратья" — также не редкость. *Многомерным* называется такой массив, который характеризуется двумя или более измерениями, а доступ к отдельному элементу осуществляется посредством двух или более индексов.

Двумерные массивы

Простейший многомерный массив — двумерный. В двумерном массиве позиция любого элемента определяется двумя индексами. Если представить двумерный массив в виде таблицы данных, то один индекс означает строку, а второй — столбец.

Чтобы объявить двумерный массив целочисленных значений размером 10×20 с именем `table`, достаточно записать следующее:

```
int[,] table = new int[10, 20];
```

Обратите особое внимание на то, что значения размерностей отделяются запятой. Синтаксис первой части этого объявления

```
[,]
```

означает, что создается ссылочная переменная двумерного массива. Для реального выделения памяти для этого массива с помощью оператора `new` используется более конкретный синтаксис:

```
int[10, 20]
```

Тем самым обеспечивается создание массива размером 10×20, причем значения размерностей также отделяются запятой.

Чтобы получить доступ к элементу двумерного массива, необходимо указать оба индекса, разделив их запятой. Например, чтобы присвоить число 10 элементу массива `table`, позиция которого определяется координатами 3 и 5, можно использовать следующую инструкцию:

```
table[3, 5] = 10;
```

Рассмотрим пример программы, которая заполняет двумерный массив числами от 1 до 12, а затем отображает содержимое этого массива.

```
// Демонстрация использования двумерного массива.

using System;

class TwoD {
  public static void Main() {
    int t, i;
    int[,] table = new int[3, 4];

    for(t=0; t < 3; ++t) {
      for(i=0; i < 4; ++i) {
        table[t,i] = (t*4)+i+1;
        Console.Write(table[t,i] + " ");
      }
      Console.WriteLine();
    }
  }
}
```

В этом примере элемент массива `table[0,0]` получит число 1, элемент `table[0,1]` — число 2, элемент `table[0,2]` — число 3 и т.д. Значение элемента `table[2,3]` будет равно 12. Схематически этот массив можно представить, как показано на рис. 7.1.

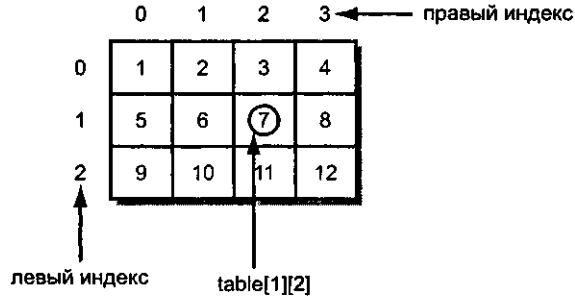


Рис. 7.1. Схематическое представление массива `table`, созданное программой `TwoD`

Если вы уже знакомы с языками C или C++, обратите внимание на объявление многомерных массивов в C# и доступ к их элементам. В языках C или C++ значения размерностей массивов и индексы указываются в отдельных парах квадратных скобок. В C# значения размерностей отделяются запятыми.

Массивы трех и более измерений

В C# можно определять массивы трех и более измерений. Вот как объявляется многомерный массив:

```
тип[, .. ,] имя = new тип[размер1, ..., размерN];
```

Например, с помощью следующего объявления создается трехмерный целочисленный массив размером 4×10×3:

```
int[, ,] multidim = new int[4, 10, 3];
```

Чтобы присвоить число 100 элементу массива `multidim`, занимающему позицию с координатами 2,4,1, используйте такую инструкцию:

```
multidim[2, 4, 1] = 100;
```

Рассмотрим программу, в которой используется трехмерный массив, содержащий 3×3×3-матрицу значений.

```
// Программа суммирует значения, расположенные
// на диагонали 3x3x3-матрицы.

using System;

class ThreeDMatrix {
  public static void Main() {
    int[,,] m = new int[3, 3, 3];
    int sum = 0;
    int n = 1;

    for(int x=0; x < 3; x++)
      for(int y=0; y < 3; y++)
        for(int z=0; z < 3; z++)
          m[x, y, z] = n++;
```

```
    sum = m[0,0,0] + m[1,1,1] + m[2, 2, 2];

    Console.WriteLine("Сумма первой диагонали: " + sum);
  }
}
```

Вот результаты выполнения этой программы:

```
Сумма первой диагонали: 42
```

Инициализация многомерных массивов

Многомерный массив можно инициализировать, заключив список инициализаторов каждой размерности в собственный набор фигурных скобок. Например, вот каков формат инициализации двумерного массива:

```
тип[,] имя_массива = {
  {val, val, val, ..., val}
  {val, val, val, ..., val}
.
.
.
  {val, val, val, ..., val}
};
```

Здесь элемент *val* — значение инициализации. Каждый внутренний блок означает строку. В каждой строке первое значение будет сохранено в первой позиции массива, второе значение — во второй и т.д. Обратите внимание на то, что блоки инициализаторов отделяются запятыми, а точка с запятой становится только после закрывающей фигурной скобки.

Например, следующая программа инициализирует массив `sqrs` числами от 1 до 10 и квадратами этих чисел.

```
// Инициализация двумерного массива.

using System;

class Squares {
  public static void Main() {
    int[,] sqrs = {
      { 1, 1 },
      { 2, 4 },
      { 3, 9 },
      { 4, 16 },
      { 5, 25 },
      { 6, 36 },
      { 7, 49 },
      { 8, 64 },
      { 9, 81 },
      { 10, 100 }
    };
    int i, j;

    for(i=0; i < 10; i++) {
      for(j=0; j < 2; j++)
        Console.Write(sqrs[i,j] + " ");
      Console.WriteLine();
    }
  }
}
```

Результаты выполнения этой программы:

```
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
```

Рваные массивы

В предыдущем примере мы создавали двумерный массив, который в C# называется *прямоугольным*. Если двумерный массив можно представить в виде таблицы, то прямоугольный массив можно определить как массив, строки которого имеют одинаковую длину. Однако C# позволяет создавать двумерный массив специального типа, именуемый *рваным*, или *с рваными краями*. У такого массива строки могут иметь различную длину. Следовательно, рваный массив можно использовать для создания таблицы со строками разной длины.

Рваные массивы объявляются с помощью наборов квадратных скобок, обозначающих размерности массива. Например, чтобы объявить двумерный рваный массив, используется следующий формат записи:

```
тип[][] имя = new тип[размер][];
```

Здесь элемент *размер* означает количество строк в массиве. Для самих строк память выделяется индивидуально, что позволяет строкам иметь разную длину. Например, следующий фрагмент программы при объявлении массива `jagged` выделяет память для его первой размерности, а память для его второй размерности выделяется "вручную".

```
int[][] jagged = new int[3][];
jagged[0] = new int[4];
jagged[1] = new int[3];
jagged[2] = new int[5];
```

После выполнения этого фрагмента кода массив `jagged` выглядит так:

jagged[0][0]	jagged[0][1]	jagged[0][2]	jagged[0][3]	
jagged[1][0]	jagged[1][1]	jagged[1][2]		
jagged[2][0]	jagged[2][1]	jagged[2][2]	jagged[2][3]	jagged[2][4]

Теперь вам, вероятно, понятно, откуда у рваных массивов такое название.

После создания рваного массива доступ к элементу осуществляется посредством задания индекса внутри собственного набора квадратных скобок. Например, чтобы присвоить число 10 элементу массива `jagged` с координатами 2 и 1, используйте такую инструкцию:

```
jagged[2][1] = 10;
```

Обратите внимание на то, что этот синтаксис отличается от того, который используется для доступа к элементам прямоугольного массива.

Следующая программа демонстрирует создание рваного двумерного массива.

```
// Демонстрация рваных массивов.

using System;

class Jagged {
  public static void Main() {
    int[][] jagged = new int[3][];
    jagged[0] = new int[4];
    jagged[1] = new int[3];
    jagged[2] = new int[5];

    int i;

    // Сохраняем значения в первом массиве.
    for(i=0; i < 4; i++)
      jagged[0][i] = i;

    // Сохраняем значения во втором массиве.
    for(i=0; i < 3; i++)
      jagged[1][i] = i;

    // Сохраняем значения в третьем массиве.
    for(i=0; i < 5; i++)
      jagged[2][i] = i;


    // Отображаем значения первого массива.
    for(i=0; i < 4; i++)
      Console.Write(jagged[0][i] + " ");

    Console.WriteLine();

    // Отображаем значения второго массива.
    for(i=0; i < 3; i++)
      Console.Write(jagged[1][i] + " ");

    Console.WriteLine();

    // Отображаем значения третьего массива.
    for(i=0; i < 5; i++)
      Console.Write(jagged[2][i] + " ");

    Console.WriteLine();
  }
}
```

Вот как выглядят результаты выполнения этой программы:

```
0 1 2 3
0 1 2
0 1 2 3 4
```

Рваные массивы используются нечасто, но в некоторых ситуациях они могут оказаться весьма эффективными. Например, если вам нужен очень большой двумерный массив с неполным заполнением (т.е. массив, в котором используются не все его элементы), то идеальным решением может оказаться массив именно такой, неправильной формы.

И еще. Поскольку рваные массивы — это по сути массивы массивов, то совсем не обязательно, чтобы "внутренние" массивы имели одинаковый тип. Например, эта инструкция создает массив двумерных массивов:

```
int[][,] jagged = new int[3][,];
```

А эта инструкция присваивает элементу `jagged[0]` ссылку на массив размером 4×2:

```
jagged[0] = new int[4][2];
```

Следующая инструкция присваивает значение переменной `i` элементу `jagged[0][1,0]`.

```
jagged[0][1,0] = i;
```

Присвоение значений ссылочным переменным массивов

Как и в случае других объектов, присваивая одной ссылочной переменной массива другую, вы просто изменяете объект, на который ссылается эта переменная. При этом не делается копия массива и не копируется содержимое одного массива в другой. Рассмотрим, например, следующую программу:

```
// Присвоение значений ссылочным переменным массивов.

using System;

class AssignARef {
  public static void Main() {
    int i;

    int[] nums1 = new int[10];
    int[] nums2 = new int[10];

    for(i=0; i < 10; i++) nums1[i] = i;

    for(i=0; i < 10; i++) nums2[i] = -i;

    Console.Write("Содержимое массива nums1: ");
    for(i=0; i < 10; i++)
      Console.Write(nums1[i] + " ");
    Console.WriteLine();

    Console.Write("Содержимое массива nums2: ");
    for(i=0; i < 10; i++)
      Console.Write(nums2[i] + " ");
    Console.WriteLine();

    nums2 = nums1; // Теперь nums2 ссылается на nums1.

    Console.Write(
       "Содержимое массива nums2 после \nприсваивания: ");
    for(i=0; i < 10; i++)
      Console.Write(nums2[i] + " ");
    Console.WriteLine();

   // Теперь воздействуем на массив nums1
```

```
    // посредством переменной nums2.
    nums2[3] = 99;

    Console.Write("Содержимое массива nums1 после\n" +
                  "его изменения посредством nums2: ");
    for(i=0; i < 10; i++)
      Console.Write(nums1[i] + " ");
    Console.WriteLine();
  }
}
```

Вот результаты выполнения этой программы:

```
Содержимое массива nums1: 0 1 2 3 4 5 6 7 8 9
Содержимое массива nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
Содержимое массива nums2 после
присваивания: 0 1 2 3 4 5 6 7 8 9
Содержимое массива nums1 после
его изменения посредством nums2: 0 1 2 99 4 5 6 7 8 9
```

Как видно по результатам выполнения этой программы, после присвоения содержимого переменной `nums1` переменной `nums2` обе они ссылаются на один и тот же объект.

Использование свойства `Length`

Из того факта, что в C# массивы реализованы как объекты, программисты могут извлечь много пользы. Например, с каждым массивом связано свойство `Length`, содержащее количество элементов, которое может хранить массив. Использование этого свойства демонстрируется в следующей программе.

```
// Использование свойства Length.

using System;

class LengthDemo {
  public static void Main() {
    int[] nums = new int[10];

    Console.WriteLine("Длина массива nums равна " +
                      nums.Length);

    // Используем Length для инициализации массива nums.
    for(int i=0; i < nums.Length; i++)
      nums[i] = i * i;

    // Теперь используем Length для отображения nums.
    Console.Write("Содержимое массива nums: ");
    for(int i=0; i < nums.Length; i++)
      Console.Write(nums[i] + " ");

    Console.WriteLine();
  }
}
```

При выполнении этой программы получаются следующие результаты:

```
Длина массива nums равна 10
Содержимое массива nums: 0 1 4 9 16 25 36 49 64 81
```

Обратите внимание на то, как в классе LengthDemo цикл for использует свойство nums.Length для управления количеством итераций. Поскольку каждый массив сопровождается информацией о собственной длине, можно использовать эту информацию, а не вручную отслеживать размер массива. При этом следует иметь в виду, что свойство Length никак не связано с реально используемым количеством элементов массива. Оно содержит количество элементов, которое массив способен хранить.

При получении длины многомерного массива возвращается общее количество элементов, обусловленное "врожденной" способностью массива хранить данные. Например:

```
// Использование свойства Length для 3-х-мерного массива.

using System;

class LengthDemo3D {
  public static void Main() {
    int[,,] nums = new int[10, 5, 6];

    Console.WriteLine("Длина массива равна " + nums.Length);

  }
}
```

Вот результаты выполнения этой программы:

```
Длина массива равна 300
```

Как подтверждает результат запуска предыдущей программы, свойство Length содержит количество элементов, которое может хранить массив nums и которое в данном случае равно 300 (10 × 5 × 6). При этом у нас нет возможности получить длину массива по конкретному измерению (координате).

Использование свойства Length упрощает многие алгоритмы за счет выполнения более простых операций над массивами. Например, следующая программа использует свойство Length для реверсирования содержимого массива посредством его копирования в другой массив в направлении от конца к началу.

```
// Реверсирование массива.

using System;

class RevCopy {
  public static void Main() {
    int i,j;
    int[] nums1 = new int[10];
    int[] nums2 = new int[10];

    for(i=0; i < nums1.Length; i++) nums1[i] = i;

    Console.Write("Исходное содержимое массива: ");
    for(i=0; i < nums2.Length; i++)
      Console.Write(nums1[i] + " ");

    Console.WriteLine();

    // Копируем массив nums1 в массив nums2 с
    // изменением порядка следования элементов.
    if(nums2.Length >= nums1.Length) // Необходимо
                        // убедиться, что массив nums2
                        // достаточно велик.
```

```
      for(i=0, j=nums1.Length-1; i < nums1.Length; i++, j--)
        nums2[j] = nums1[i];

    Console.Write("Содержимое массива в обратном порядке: ");
    for(i=0; i < nums2.Length; i++)
      Console.Write(nums2[i] + " ");

    Console.WriteLine();
  }
}
```

Вот как выглядят результаты выполнения этой программы:

```
Исходное содержимое массива: 0 1 2 3 4 5 6 7 8 9
Содержимое массива в обратном порядке: 9 8 7 6 5 4 3 2 1 0
```

Здесь свойство `Length` выполняет две важных функции. Во-первых, оно подтверждает, что массив-приемник имеет размер, достаточный для хранения содержимого исходного массива. Во-вторых, оно обеспечивает условие завершения цикла `for`, который выполняет реверсное копирование. Безусловно, это очень простой пример, и размер массива легко узнать и без свойства `Length`, но аналогичный подход можно применить к широкому диапазону более сложных ситуаций.

Использование свойства `Length` при работе с рваными массивами

При работе с рваными массивами использование свойства `Length` приобретает особое значение, поскольку позволяет получить длину каждого отдельного ("строчного") массива. Рассмотрим, например, следующую программу, которая имитирует работу центрального процессора (ЦП) в сети с четырьмя узлами.

```
// Демонстрация использования свойства Length при
// работе с рваными массивами.

using System;

class Jagged {
  public static void Main() {
    int[][] network_nodes = new int[4][];
    network_nodes[0] = new int[3];
    network_nodes[1] = new int[7];
    network_nodes[2] = new int[2];
    network_nodes[3] = new int[5];


    int i, j;

    // Создаем фиктивные данные по
    // использованию ЦП.
    for(i=0; i < network_nodes.Length; i++)
      for(j=0; j < network_nodes[i].Length; j++)
        network_nodes[i][j] = i * j + 70;


    Console.WriteLine("Общее количество сетевых узлов: " +
                      network_nodes.Length + "\n");

    for(i=0; i < network_nodes.Length; i++) {
      for(j=0; j < network_nodes[i].Length; j++) {
```

```
            Console.Write("Использование ЦП на узле " + i +
                          " для ЦП " + j + ": ");
            Console.Write(network_nodes[i][j] + "% ");
            Console.WriteLine();
          }
          Console.WriteLine();
        }

      }
    }
```

Результаты выполнения этой программы выглядят так:

```
Общее количество сетевых узлов: 4

Использование ЦП на узле 0 для ЦП 0: 70%
Использование ЦП на узле 0 для ЦП 1: 70%
Использование ЦП на узле 0 для ЦП 2: 70%

Использование ЦП на узле 1 для ЦП 0: 70%
Использование ЦП на узле 1 для ЦП 1: 71%
Использование ЦП на узле 1 для ЦП 2: 72%
Использование ЦП на узле 1 для ЦП 3: 73%
Использование ЦП на узле 1 для ЦП 4: 74%
Использование ЦП на узле 1 для ЦП 5: 75%
Использование ЦП на узле 1 для ЦП 6: 76%

Использование ЦП на узле 2 для ЦП 0: 70%
Использование ЦП на узле 2 для ЦП 1: 72%

Использование ЦП на узле 3 для ЦП 0: 70%
Использование ЦП на узле 3 для ЦП 1: 73%
Использование ЦП на узле 3 для ЦП 2: 76%
Использование ЦП на узле 3 для ЦП 3: 79%
Использование ЦП на узле 3 для ЦП 4: 82%
```

Обратите внимание на то, как свойство `Length` используется для рваного массива `network_nodes`. Вспомните, что двумерный рваный массив — это массив массивов. Поэтому выражение

```
network_nodes.Length
```

возвращает количество *массивов*, хранимых в массиве `network_nodes` (в данном случае это значение равно 4). Чтобы получить длину отдельного массива во "внешнем" рваном массиве, используйте, например, такое выражение:

```
network_nodes[0].Length
```

Это выражение возвращает длину первого массива.

Цикл `foreach`

В главе 5 упоминалось о том, что в языке C# определен цикл `foreach`, но подробное его рассмотрение было отложено на "потом". По заголовку этого раздела нетрудно догадаться, что время для этого наступило.

Цикл `foreach` используется для опроса элементов *коллекции*. Коллекция — это группа объектов. C# определяет несколько типов коллекций, и одним из них является массив. Формат записи цикла `foreach` имеет такой вид:

`foreach(`*тип имя_переменной* `in` *коллекция*`)` *инструкция*`;`

Здесь элементы *тип* и *имя_переменной* задают тип и имя итерационной переменной, которая при функционировании цикла `foreach` будет получать значения элементов из коллекции. Элемент *коллекция* служит для указания опрашиваемой коллекции (в данном случае в качестве коллекции мы рассматриваем массив). Таким образом, элемент *тип* должен совпадать (или быть совместимым) с базовым типом массива. Здесь важно запомнить, что итерационную переменную применительно к массиву можно использовать только для чтения. Следовательно, невозможно изменить содержимое массива, присвоив итерационной переменной новое значение.

Рассмотрим простой пример использования цикла `foreach`. Приведенная ниже программа создает массив для хранения целых чисел и присваивает его элементам начальные значения. Затем она отображает элементы массива, попутно вычисляя их сумму.

```
// Использование цикла foreach.

using System;

class ForeachDemo {
  public static void Main() {
    int sum = 0;
    int[] nums = new int[10];

    // Присваиваем элементам массива nums значения.
    for(int i = 0; i < 10; i++)
      nums[i] = i;

    // Используем цикл foreach для отображения значений
    // элементов массива и их суммирования.
    foreach(int x in nums) {
      Console.WriteLine("Значение элемента равно: " + x);
      sum += x;
    }
    Console.WriteLine("Сумма равна: " + sum);
  }
}
```

При выполнении этой программы получим следующие результаты:

```
Значение элемента равно: 0
Значение элемента равно: 1
Значение элемента равно: 2
Значение элемента равно: 3
Значение элемента равно: 4
Значение элемента равно: 5
Значение элемента равно: 6
Значение элемента равно: 7
Значение элемента равно: 8
Значение элемента равно: 9
Сумма равна: 45
```

Как видно из приведенных выше результатов, цикл `foreach` последовательно опрашивает элементы массива в направлении от наименьшего индекса к наибольшему.

Несмотря на то что цикл `foreach` работает до тех пор, пока не будут опрошены все элементы массива, существует возможность досрочного его останова с помощью инструкции `break`. Например, следующая программа суммирует только пять первых элементов массива `nums`.

```
// Использование инструкции break в цикле foreach.

using System;
```

```
class ForeachDemo {
 public static void Main() {
   int sum = 0;
   int[] nums = new int[10];

   // Присваиваем элементам массива nums значения.
   for(int i = 0; i < 10; i++)
     nums[i] = i;

   // Используем цикл foreach для отображения значений
   // элементов массива и их суммирования.
   foreach(int x in nums) {
     Console.WriteLine("Значение элемента равно: " + x);
     sum += x;
     if(x == 4) break; // Останов цикла, когда x равен 4.
   }
   Console.WriteLine("Сумма первых 5 элементов: " + sum);
 }
}
```

Вот как выглядят результаты выполнения этой программы:

```
Значение элемента равно: 0
Значение элемента равно: 1
Значение элемента равно: 2
Значение элемента равно: 3
Значение элемента равно: 4
Сумма первых 5 элементов: 10
```

Очевидно, что цикл `foreach` останавливается после получения пятого элемента массива.

Цикл `foreach` работает и с многомерными массивами. В этом случае он возвращает элементы в порядке следования строк: от первой до последней.

```
// Использование цикла foreach с двумерным массивом.

using System;

class ForeachDemo2 {
  public static void Main() {
    int sum = 0;
    int[,] nums = new int[3,5];

    // Присваиваем элементам массива nums значения.
    for(int i = 0; i < 3; i++)
      for(int j=0; j < 5; j++)
        nums[i,j] = (i+1)*(j+1);

    // Используем цикл foreach для отображения значений
    // элементов массива и их суммирования.
    foreach(int x in nums) {
      Console.WriteLine("Значение элемента равно: " + x);
      sum += x;
    }
    Console.WriteLine("Сумма равна: " + sum);
  }
}
```

Вот результаты выполнения этой программы:

```
Значение элемента равно: 1
Значение элемента равно: 2
Значение элемента равно: 3
Значение элемента равно: 4
Значение элемента равно: 5
Значение элемента равно: 2
Значение элемента равно: 4
Значение элемента равно: 6
Значение элемента равно: 8
Значение элемента равно: 10
Значение элемента равно: 3
Значение элемента равно: 6
Значение элемента равно: 9
Значение элемента равно: 12
Значение элемента равно: 15
Сумма равна: 90
```

Поскольку цикл `foreach` может опрашивать массив последовательно (от начала к концу), может сложиться впечатление, что его использование носит весьма ограниченный характер. Однако это не так. Для функционирования широкого круга алгоритмов требуется именно такой механизм. Одним из них является алгоритм поиска. Например, следующая программа использует цикл `foreach` для поиска в массиве заданного значения. Когда значение найдено, цикл останавливается.

```
// Поиск значения в массиве с помощью цикла foreach.

using System;

class Search {
  public static void Main() {
    int[] nums = new int[10];
    int val;
    bool found = false;

    // Присваиваем элементам массива nums значения.
    for(int i = 0; i < 10; i++)
      nums[i] = i;

    val = 5;

    // Используем цикл foreach для поиска в массиве nums
    // заданного значения.
    foreach(int x in nums) {
      if(x == val) {
        found = true;
        break;
      }
    }

    if(found)
      Console.WriteLine("Значение найдено!");
  }
}
```

Цикл `foreach` также используется для вычисления среднего значения, определения минимального или максимально числа в наборе чисел, поиска дубликатов и т.д. Как будет показано далее, цикл `foreach` особенно полезен при работе с другими типами коллекций.

Строки

С точки зрения ежедневного программирования одним из самых важных типов данных C# является тип `string`. Он предназначен для определения и поддержки символьных строк. Во многих других языках программирования строка представляет собой массив символов. В C# дело обстоит иначе: здесь строки являются объектами. Таким образом, `string` — это ссылочный тип. Несмотря на то что `string` — встроенный тип данных, для его рассмотрения необходимо иметь представление о классах и объектах.

На самом деле мы негласно используем класс `string`, начиная с главы 2, но вы попросту об этом не знали. При создании строкового литерала в действительности создается объект класса `string`. Например, в инструкции

```
Console.WriteLine("В C# строки являются объектами.");
```

строка "В C# строки являются объектами." средствами языка C# автоматически превращена в объект класса `string`. Таким образом, в предыдущих программах мы подспудно использовали класс `string`. В этом разделе мы научимся работать с ними в явном виде.

Создание строк

Самый простой способ создать объект типа `string` — использовать строковый литерал. Например, после выполнения приведенной ниже инструкции `str` будет объявлена ссылочной переменной типа `string`, которой присваивается ссылка на строковый литерал.

```
string str = "C#-строки - это мощная сила.";
```

В данном случае переменная `str` инициализируется последовательностью символов "C#-строки - это мощная сила.".

Можно также создать `string`-объект из массива типа `char`. Вот пример:

```
char[] charray = {'t', 'e', 's', 't'};
string str = new string(charray);
```

После создания `string`-объект можно использовать везде, где разрешается использование строки символов, заключенной в кавычки. Например, `string`-объект можно использовать в качестве аргумента функции `WriteLine()`, как показано в следующем примере.

```
// Знакомство со строками.

using System;

class StringDemo {
  public static void Main() {

    char[] charray = {'A', ' ', 's', 't', 'r', 'i', 'n', 'g', '.' };
    string str1 = new string(charray);
    string str2 = "Еще один string-объект.";

    Console.WriteLine(str1);
    Console.WriteLine(str2);
  }
}
```

Результаты выполнения этой программы таковы:

```
A string.
Еще один string-объект.
```

Работа со строками

Класс `string` содержит ряд методов, которые предназначены для обработки строк (некоторые из них показаны в табл. 7.1). Тип `string` также включает свойство `Length`, которое содержит длину строки.

Чтобы получить значение отдельного символа строки, достаточно использовать индекс. Например:

```
string str = "test";
Console.WriteLine(string[0]);
```

При выполнении этого фрагмента программы на экран будет выведен символ `t` (первый символ слова "test"). Как и у массивов, индексация строк начинается с нуля. Однако здесь важно отметить, что с помощью индекса нельзя присвоить символу внутри строки новое значение. Индекс можно использовать только для получения символа.

Таблица 7.1. Наиболее часто используемые методы обработки строк

`static string Copy(string str)`	Возвращает копию строки *str*
`int CompareTo(string str)`	Возвращает отрицательное значение, если вызывающая строка меньше строки *str*, положительное значение, если вызывающая строка больше строки *str*, и нуль, если сравниваемые строки равны
`int IndexOf(string str)`	Выполняет в вызывающей строке поиск подстроки, заданной параметром *str*. Возвращает индекс первого вхождения искомой подстроки или -1, если она не будет обнаружена
`int LastIndexOf(string str)`	Выполняет в вызывающей строке поиск подстроки, заданной параметром *str*. Возвращает индекс последнего вхождения искомой подстроки или -1, если она не будет обнаружена
`string ToLower()`	Возвращает строчную версию вызывающей строки
`string ToUpper()`	Возвращает прописную версию вызывающей строки

Чтобы узнать, равны ли две строки, необходимо использовать оператор "==". Обычно, когда оператор "==" применяется к ссылочным объектам, он определяет, относятся ли обе ссылки к одному и тому же объекту. Но применительно к объектам типа `string` дело обстоит иначе. В этом случае проверяется равенство содержимого двух строк. То же справедливо и в отношении оператора "`!=`". Что касается остальных операторов отношения (например, ">" или ">="), то они сравнивают ссылки так же, как и объекты других типов.

Рассмотрим программу, которая демонстрирует выполнение ряда операций над строками.

```
// Демонстрация выполнения некоторых операций над строками.

using System;

class StrOps {
  public static void Main() {
    string str1 =
      "В .NET-программировании без C# не обойтись.";
    string str2 = string.Copy(str1);
```

```
    string str3 = "C#-строки -- могучая сила.";
    string strUp, strLow;
    int result, idx;

    Console.WriteLine("str1: " + str1);

    Console.WriteLine("Длина строки str1: " +
                      str1.Length);

    // Создаем прописную и строчную версии строки str1.
    strLow = str1.ToLower();
    strUp =  str1.ToUpper();
    Console.WriteLine("Строчная версия строки str1:\n    " +
                      strLow);
    Console.WriteLine("Прописная версия строки str1:\n    " +
                      strUp);

    Console.WriteLine();

    // Отображаем str1 в символьном режиме.
    Console.WriteLine("Отображаем str1 посимвольно.");
    for(int i=0; i < str1.Length; i++)
      Console.Write(str1[i]);
    Console.WriteLine("\n");

    // Сравниваем строки.
    if(str1 == str2)
      Console.WriteLine("str1 == str2");
    else
      Console.WriteLine("str1 != str2");

    if(str1 == str3)
      Console.WriteLine("str1 == str3");
    else
      Console.WriteLine("str1 != str3");

    result = str1.CompareTo(str3);
    if(result == 0)
      Console.WriteLine("str1 и str3 равны.");
    else if(result < 0)
      Console.WriteLine("str1 меньше, чем str3");
    else
      Console.WriteLine("str1 больше, чем str3");

    Console.WriteLine();

    // Присваиваем str2 новую строку.
    str2 = "Один Два Три Один";

    // Поиск строк.
    idx = str2.IndexOf("Один");
    Console.WriteLine(
       "Индекс первого вхождения подстроки Один: " + idx);
    idx = str2.LastIndexOf("Один");
    Console.WriteLine(
       "Индекс последнего вхождения подстроки Один: " + idx);

  }
}
```

При выполнении этой программы получаем следующие результаты:

```
str1: В .NET-программировании без C# не обойтись.
Длина строки str1: 43
Строчная версия строки str1:
    в .net-программировании без c# не обойтись.
Прописная версия строки str1:
    В .NET-ПРОГРАММИРОВАНИИ БЕЗ C# НЕ ОБОЙТИСЬ.

Отображаем str1 посимвольно.
В .NET-программировании без C# не обойтись.

str1 == str2
str1 != str3
str1 больше, чем str3

Индекс первого вхождения подстроки Один: 0
Индекс последнего вхождения подстроки Один: 13
```

С помощью оператора "+" можно конкатенировать (объединить) несколько строк. Например, при выполнении этого фрагмента кода

```
string str1 = "Один";
string str2 = "Два";
string str3 = "Три";
string str4 = str1 + str2 + str3;
```

переменная `str4` инициализируется строкой `"ОдинДваТри"`.

И еще. Ключевое слово `string` представляет собой псевдоним для класса `System.String`, определенного библиотекой классов среды .NET Framework. Таким образом, поля и методы, определяемые типом `string`, по сути являются полями и методами класса `System.String` (здесь были представлены только некоторые из них). Подробно класс `System.String` рассматривается в части II.

Массивы строк

Подобно другим типам данных, строки могут быть собраны в массивы. Рассмотрим пример.

```
// Демонстрация использования массивов строк.

using System;

class StringArrays {
  public static void Main() {
    string[] str = { "Это", "очень", "простой", "тест." };

    Console.WriteLine("Исходный массив: ");
    for(int i=0; i < str.Length; i++)
      Console.Write(str[i] + " ");
    Console.WriteLine("\n");

    // Изменяем строку.
    str[1] = "тоже";
    str[3] = "тест, не правда ли?";

    Console.WriteLine("Модифицированный массив: ");
    for(int i=0; i < str.Length; i++)
      Console.Write(str[i] + " ");
  }
}
```

После выполнения этой программы получаем такие результаты:

```
Исходный массив:
Это очень простой тест.

Модифицированный массив:
Это тоже простой тест, не правда ли?
```

А вот пример поинтереснее. Следующая программа отображает целочисленное значение с помощью слов. Например, значение 19 будет отображено как словосочетание "один девять".

```
// Отображение цифр целого числа с помощью слов.

using System;

class ConvertDigitsToWords {
  public static void Main() {
    int num;
    int nextdigit;
    int numdigits;
    int[] n = new int[20];

    string[] digits = { "нуль", "один", "два",
                        "три", "четыре", "пять",
                        "шесть", "семь", "восемь",
                        "девять" };

    num = 1908;

    Console.WriteLine("Число: " + num);

    Console.Write("Число в словах: ");

    nextdigit = 0;
    numdigits = 0;

    /* Получаем отдельные цифры и сохраняем их в массиве n.
       Эти цифры хранятся в обратном порядке. */
    do {
      nextdigit = num % 10;
      n[numdigits] = nextdigit;
      numdigits++;
      num = num / 10;
    } while(num > 0);
    numdigits--;

    // Отображаем слова.
    for( ; numdigits >= 0; numdigits--)
      Console.Write(digits[n[numdigits]] + " ");

    Console.WriteLine();
  }
}
```

Вот результаты выполнения этой программы:

```
Число: 1908
Число в словах: один девять нуль восемь
```

В этой программе string-массив digits хранит в порядке возрастания словесные эквиваленты цифр от нуля до девяти. Чтобы заданное целочисленное значение преобразовать в слова, сначала выделяется каждая цифра этого значения, начиная с крайней правой, и полученные цифры запоминаются в обратном порядке в int-массиве с именем n. Затем этот массив опрашивается от конца к началу, и каждое целое значение массива n используется в качестве индекса для доступа к элементам массива digits, чтобы отобразить на экране соответствующую строку.

Постоянство строк

Следующее утверждение, вероятно, вас удивит: содержимое string-объектов неизменно. Другими словами, последовательность символов, составляющих строку, изменить нельзя. Хотя это кажется серьезным недостатком, на самом деле это не так. Это ограничение позволяет C# эффективно использовать строки. Если вам понадобится строка, которая должна представлять собой "вариацию на тему" уже существующей строки, создайте новую строку, которая содержит желаемые изменения. Поскольку неиспользуемые строковые объекты автоматически утилизируются системой сбора мусора, даже не нужно беспокоиться о "брошенных" строках.

При этом необходимо понимать, что ссылочные переменные типа string могут менять объекты, на которые они ссылаются. А содержимое созданного string-объекта изменить уже невозможно.

Чтобы до конца понять, почему неизменяемые строки не являются препятствием для программиста, воспользуемся еще одним методом класса string: Substring(). Этот метод возвращает новую строку, которая содержит заданную часть вызывающей строки. Поскольку создается новый string-объект, содержащий подстроку, исходная строка остается неизменной, и правило постоянства строк не нарушается. Вот формат вызова метода Substring():

```
string Substring(int start, int len)
```

Здесь параметр *start* означает индекс начала, а параметр *len* задает длину подстроки.

Рассмотрим программу, которая демонстрирует метод Substring() и принцип постоянства строк.

```
// Использование метода Substring().

using System;

class SubStr {
  public static void Main() {
    string orgstr = "C# упрощает работу со строками.";

    // Создание подстроки.
    string substr = orgstr.Substring(4, 14);

    Console.WriteLine("orgstr: " + orgstr);
    Console.WriteLine("substr: " + substr);

  }
}
```

А вот как выглядят результаты работы этой программы:

```
orgstr: C# упрощает работу со строками.
substr: прощает работу
```

Как видите, исходная строка orgstr не изменена, а строка substr содержит нужную подстроку.

И хотя постоянство string-объектов обычно не является ограничением, не исключены ситуации, когда возможность модифицировать строки могла бы оказаться весьма кстати. Для таких случаев в C# предусмотрен класс StringBuilder, который определен в пространстве имен System.Text. Этот класс создает объекты, которые можно изменять. Однако в большинстве случаев все же используются string-объекты, а не объекты класса StringBuilder.

Использование строк в switch-инструкциях

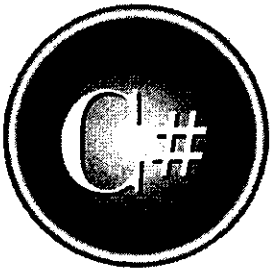
Для управления switch-инструкциями можно использовать string-объекты, причем это единственный тип, который там допускается, помимо типа int. Эта возможность в некоторых случаях облегчает обработку. Например, следующая программа отображает цифровой эквивалент слов "один", "два" и "три".

```
// Демонстрация возможности строкового управления
// инструкцией switch.

using System;

class StringSwitch {
  public static void Main() {
    string[] strs = { "один", "два", "три", "два", "один" };

    foreach(string s in strs) {
      switch(s) {
        case "один":
          Console.Write(1);
          break;
        case "два":
          Console.Write(2);
          break;
        case "три":
          Console.Write(3);
          break;
      }
    }
    Console.WriteLine();

  }
}
```

Вот результат выполнения этой программы:

```
12321
```

Полный справочник по



Глава 8

Подробнее о методах и классах

В этой главе мы снова обращаемся к рассмотрению методов и классов. Начнем с управления доступом к членам класса. Затем обсудим возможность передачи методам объектов и их возврата, после чего рассмотрим перегрузку методов, различные формы метода `Main()`, рекурсию и использование ключевого слова `static`.

Управление доступом к членам класса

Поддерживая инкапсуляцию, класс обеспечивает два положительных момента. Во-первых, он связывает данные с кодом (мы используем это преимущество начиная с главы 6). Во-вторых, он предоставляет средства управления доступом к членам класса. На этом мы сейчас и остановимся.

Хотя в действительности дело обстоит несколько сложнее, но по сути существуют два базовых типа членов класса: открытые и закрытые. К открытому члену класса может свободно получить доступ код, определенный вне этого класса. До сих пор мы как раз и использовали члены такого типа. К закрытому же члену класса доступ могут получить методы, определенные только в этом классе. Благодаря использованию закрытых членов класса мы и имеем возможность управлять доступом.

Ограничение доступа к членам класса — это фундаментальная часть объектно-ориентированного программирования, поскольку она предотвращает неверное использование объекта. Разрешая доступ к закрытым данным только посредством строго определенного набора методов, вы имеете возможность не допустить присвоение этим данным неподходящих значений, выполнив, например, проверку диапазона. Код, не принадлежащий классу, не может устанавливать закрытые члены напрямую. И именно программист управляет тем, как и когда будут использоваться данные объекта. Таким образом, при корректной реализации класс создает "черный ящик", с которым можно работать, но внутреннее функционирование которого закрыто для вмешательства извне.

Спецификаторы доступа C#

Управление доступом к членам класса достигается за счет использования четырех *спецификаторов доступа*: `public`, `private`, `protected` и `internal`. В этой главе мы ограничимся рассмотрением спецификаторов `public` и `private`. Модификатор `protected` применяется только при включении интерфейсов и описан в главе 9. Модификатор `internal` применяется в основном при использовании *компоновочных файлов* (assembly) и описан в главе 16.

Спецификатор `public` разрешает доступ к соответствующему члену класса со стороны другого кода программы, включая методы, определенные внутри других классов.

Спецификатор `private` разрешает доступ к соответствующему члену класса только для методов, определенных внутри того же класса. Таким образом, методы других классов не могут получить доступ к `private`-члену не их класса. Как разъяснялось в главе 6, при отсутствии спецификатора доступа член класса является закрытым (`private`) по умолчанию. Следовательно, при создании закрытых членов класса спецификатор `private` необязателен.

Спецификатор доступа должен стоять первым в списке спецификаторов типа любого члена класса. Вот несколько примеров:

```
public string errMsg;
private double bal;
private bool isError(byte status) { // ...
```

Чтобы лучше понять разницу между спецификаторами public и private, рассмотрим следующую программу:

```
// Сравнение доступа к открытым и закрытым членам класса.

using System;

class MyClass {
  private int alpha; // Явно задан спецификатор private.
  int beta;          // Спецификатор private по умолчанию.
  public int gamma;  // Явно задан спецификатор public.

  /* Методы для получения доступа к членам alpha и beta.
     Другие члены класса беспрепятственно получают доступ
     к private-члену того же класса.
  */
  public void setAlpha(int a) {
    alpha = a;
  }

  public int getAlpha() {
    return alpha;
  }

  public void setBeta(int a) {
    beta = a;
  }

  public int getBeta() {
    return beta;
  }
}

class AccessDemo {
  public static void Main() {
    MyClass ob = new MyClass();

    /* Доступ к private-членам alpha и beta разрешен
       только посредством соответствующих методов. */
    ob.setAlpha(-99);
    ob.setBeta(19);
    Console.WriteLine("Член ob.alpha равен " +
                      ob.getAlpha());
    Console.WriteLine("Член ob.beta равен " +
                      ob.getBeta());

    // К private-членам alpha или beta нельзя получить
    // доступ таким образом:
//  ob.alpha = 10; // Неверно! alpha -- закрытый член!
//  ob.beta = 9;   // Неверно! beta -- закрытый член!

    // Можно получить прямой доступ
    // к члену gamma, поскольку он открытый член.
    ob.gamma = 99;
  }
}
```

Как видите, внутри класса MyClass член alpha явно определен как private-член, beta — private-член по умолчанию, а gamma определен как public-член. Поскольку

alpha и beta — закрытые члены, к ним нельзя получить доступ не из их "родного" класса. Следовательно, внутри класса AccessDemo к этим членам нельзя обратиться напрямую. К каждому из них необходимо обращаться только через открытые (public-) методы, например setAlpha() или getAlpha(). Если удалить символ комментария в начале строки

```
// ob.alpha = 10; // Неверно! alpha -- закрытый член!,
```

то вы бы не смогли скомпилировать эту программу по причине нарушения доступа к закрытому члену класса. Несмотря на то что доступ к члену alpha вне класса MyClass не разрешен, методы, определенные в классе MyClass (setAlpha() и getAlpha()), могут к нему обращаться. Это справедливо и для члена beta.

Итак, к закрытым членам могут свободно обращаться другие члены того же класса, но не методы, определенные вне этого класса.

Применение спецификаторов доступа public и private

Надлежащее использование спецификаторов доступа public и private — залог успеха объектно-ориентированного программирования. Хотя на этот счет не существует жестких правил, все же программисты выработали общие принципы, которыми следует руководствоваться при программировании классов.

1. Члены, которые используются только внутри класса, следует определить как закрытые.
2. Данные экземпляров, которые должны находиться в пределах заданного диапазона, следует определить как закрытые, а доступ к ним обеспечить через открытые методы, выполняющие проверку вхождения в диапазон.
3. Если изменение члена может вызвать эффект, распространяющийся за пределы самого члена (т.е. действует на другие аспекты объекта), этот член следует определить как закрытый и обеспечить к нему контролируемый доступ.
4. Члены, при некорректном использовании которых на объект может быть оказано негативное воздействие, следует определить как закрытые, а доступ к ним обеспечить через открытые методы, предохраняющие эти члены от некорректного использования.
5. Методы, которые получают или устанавливают значения закрытых данных, должны быть открытыми.
6. Объявление переменных экземпляров открытыми допустимо, если нет причин делать их закрытыми.

Безусловно, существует множество нюансов, не охваченных перечисленными выше принципами. Кроме того, в некоторых случаях одно или несколько правил приходиться нарушать, но чаще всего соблюдение этих принципов позволяет создать объекты с высоким "иммунитетом" к некорректному использованию.

Управление доступом: учебный проект

Учебный проект поможет вам глубже понять управление доступом к членам класса. Один из распространенных примеров объектно-ориентированного программирования — класс, реализуемый в стеке. (Стек — это структура данных, которая реализует список элементов по принципу: первым вошел — последним вышел. В качестве бытового примера стека можно привести стопку тарелок, из которых первая поставленная на стол тарелка, будет использована последней.)

Стек — это классический пример объектно-ориентированного программирования, в котором сочетаются как средства хранения информации, так и методы получения доступа к этой информации. Для реализации этого наилучшим образом подходит класс, в котором члены, обеспечивающие хранение данных стека, являются закрытыми, а доступ к ним осуществляется посредством открытых методов.

В стеке необходимо выполнить две операции: *поместить* данные в стек и *извлечь* их оттуда. Каждое значение помещается в вершину стека и извлекается также из его вершины. Извлеченное из стека значение удаляется и не может быть извлечено снова.

В приведенном ниже примере создается класс `Stack`, который реализует работу стека. Хранение данных стека обеспечивается на основе закрытого массива. Операции помещения данных в стек и извлечения их из него доступны через открытые методы класса `Stack`. Таким образом, механизм "первым вошел — последним вышел" обеспечивается открытыми методами. В нашем примере класс `Stack` предназначен для хранения символов, но аналогичный механизм можно использовать для хранения данных любого другого типа.

```
// Класс стека для хранения символов.

using System;

class Stack {
  // Эти члены закрытые.
  char[] stck; // Массив для хранения данных стека.
  int tos;     // Индекс вершины стека.

  // Создаем пустой класс Stack заданного размера.
  public Stack(int size) {
    stck = new char[size]; // Выделяем память для стека.
    tos = 0;
  }

  // Помещаем символы в стек.
  public void push(char ch) {
    if(tos==stck.Length) {
      Console.WriteLine(" - Стек заполнен.");
      return;
    }

    stck[tos] = ch;
    tos++;
  }

  // Извлекаем символ из стека.
  public char pop() {
    if(tos==0) {
      Console.WriteLine(" - Стек пуст.");
      return (char) 0;
    }

    tos--;
    return stck[tos];
  }

  // Метод возвращает значение true, если стек полон.
  public bool full() {
    return tos==stck.Length;
  }
```

```
  // Метод возвращает значение true, если стек пуст.
  public bool empty() {
    return tos==0;
  }

  // Возвращает общий объем стека.
  public int capacity() {
    return stck.Length;
  }

  // Возвращает текущее количество объектов в стеке.
  public int getNum() {
    return tos;
  }
}
```

Рассмотрим класс `Stack` подробнее. Его объявление начинается с объявления двух переменных экземпляров:

```
char[] stck; // Массив для хранения данных стека.
int tos;     // Индекс вершины стека.
```

Массив `stck` обеспечивает хранение данных стека, которыми в нашем случае являются символы. Обратите внимание на то, что память для массива здесь не выделяется. Это делается в конструкторе класса `Stack`. Член `tos` содержит индекс вершины стека.

Оба члена `stck` и `tos` по умолчанию объявлены закрытыми, и именно этот факт позволяет обеспечить функционирование механизма "первым вошел — последним вышел". Если бы к массиву `stck` был разрешен открытый доступ, то к элементам стека можно было бы обращаться совершенно беспорядочно. Кроме того, поскольку член `tos` содержит индекс "верхнего" элемента стека, чтобы избежать искажения стека, необходимо предотвратить манипуляции над этим членом вне класса `Stack`. Доступ пользователя к членам `stck` и `tos` должен быть организован косвенным образом, посредством специальных открытых методов.

Вот как выглядит конструктор стека:

```
// Создаем пустой класс Stack заданного размера.
public Stack(int size) {
    stck = new char[size]; // Выделяем память для стека.
    tos = 0;
}
```

Этому конструктору передается необходимый размер стека. Поэтому он выделяет соответствующую область памяти для массива и устанавливает переменную экземпляра `tos` равной нулю. Таким образом, нулевое значение переменной `tos` служит признаком того, что стек пуст.

Открытый метод `push()` помещает в стек один элемент. Вот определение этого метода:

```
// Помещаем символы в стек.
  public void push(char ch) {
    if(tos==stck.Length) {
      Console.WriteLine(" - Стек заполнен.");
      return;
    }

    stck[tos] = ch;
    tos++;
  }
```

Элемент, помещаемый в стек, передается в качестве параметра ch. Прежде чем элемент будет добавлен в стек, выполняется проверка, хватит ли в массиве места, чтобы принять очередной элемент. Для выполнения этой проверки достаточно убедиться в том, что значение переменной tos не превышает длину массива stck. Если еще есть свободное место, элемент сохраняется в массиве stck по индексу, заданному значением переменной tos, после чего значение tos инкрементируется. Таким образом, переменная tos всегда содержит индекс следующего свободного элемента в массиве stck.

Чтобы удалить элемент из стека, необходимо вызвать метод pop(). Вот его определение:

```
// Извлекаем символ из стека.
  public char pop() {
    if(tos==0) {
      Console.WriteLine(" - Стек пуст.");
      return (char) 0;
    }

    tos--;
    return stck[tos];
  }
```

И здесь проверяется значение переменной tos. Если оно равно нулю, значит, стек пуст. В противном случае значение tos декрементируется, и по полученному индексу возвращается элемент стека.

Несмотря на то что push() и pop() — единственные, жизненно необходимые для реализации стека методы, существуют и другие действия, которые были бы полезны для его функционирования, поэтому в классе Stack реализовано еще четыре метода (full(), empty(), capacity() и getNum()). Эти методы предоставляют информацию о состоянии стека. Приведем их определения.

```
  // Метод возвращает значение true, если стек полон.
  public bool full() {
    return tos==stck.Length;
  }

  // Метод возвращает значение true, если стек пуст.
  public bool empty() {
    return tos==0;
  }

  // Возвращает общий объем стека.
  public int capacity() {
    return stck.Length;
  }

  // Возвращает текущее количество объектов в стеке.
  public int getNum() {
    return tos;
  }
```

Метод full() возвращает значение true, если стек полон, и значение false в противном случае. Метод empty() возвращает значение true, если стек пуст, и значение false в противном случае. Чтобы получить общий объем стека (т.е. количество элементов, которое он может содержать), достаточно вызвать метод capacity(). А чтобы узнать, сколько элементов хранится в стеке в данный момент, вызовите метод getNum(). Эти методы удобно использовать, поскольку для получения информации,

которую они предоставляют, требуется доступ к члену tos, который закрыт в рамках класса Stack.

Следующая программа демонстрирует работу стека.

```
// Демонстрация использование класса Stack.

using System;

class StackDemo {
  public static void Main() {
    Stack stk1 = new Stack(10);
    Stack stk2 = new Stack(10);
    Stack stk3 = new Stack(10);
    char ch;
    int i;

    // Помещаем ряд символов в стек stk1.
    Console.WriteLine(
          "Помещаем символы от A до Z в стек stk1.");
    for(i=0; !stk1.full(); i++)
      stk1.push((char) ('A' + i));

    if(stk1.full()) Console.WriteLine("Стек stk1 полон.");

    // Отображаем содержимое стека stk1.
    Console.Write("Содержимое стека stk1: ");
    while( !stk1.empty() ) {
      ch = stk1.pop();
      Console.Write(ch);
    }

    Console.WriteLine();

    if(stk1.empty()) Console.WriteLine("Стек stk1 пуст.\n");

    // Помещаем еще символы в стек stk1.
    Console.WriteLine(
       "Снова помещаем символы от A до Z в стек stk1.");
    for(i=0; !stk1.full(); i++)
      stk1.push((char) ('A' + i));

    /* Теперь извлекаем элементы из стека stk1 и помещаем их
       в стек stk2.
       В результате элементы стека stk2 должны быть
       расположены в обратном порядке. */
    Console.WriteLine(
              "Теперь извлекаем элементы из стека stk1 и\n" +
              " помещаем их в стек stk2.");
    while( !stk1.empty() ) {
      ch = stk1.pop();
      stk2.push(ch);
    }

    Console.Write("Содержимое стека stk2: ");
    while( !stk2.empty() ) {
      ch = stk2.pop();
      Console.Write(ch);
    }
```

```
    Console.WriteLine("\n");

    // Помещаем 5 символов в стек stk3.
    Console.WriteLine("Помещаем 5 символов в стек stk3.");
    for(i=0; i < 5; i++)
      stk3.push((char) ('A' + i));

    Console.WriteLine(
           "Объем стека stk3: " + stk3.capacity());
    Console.WriteLine(
           "Количество объектов в стеке stk3: " +
           stk3.getNum());

  }
}
```

При выполнении этой программы получаем следующие результаты:

```
Помещаем символы от A до Z в стек stk1.
Стек stk1 полон.
Содержимое стека stk1: JIHGFEDCBA
Стек stk1 пуст.

Снова помещаем символы от A до Z в стек stk1.
Теперь извлекаем элементы из стека stk1 и
 помещаем их в стек stk2.
Содержимое стека stk2: ABCDEFGHIJ

Помещаем 5 символов в стек stk3.
Объем стека stk3: 10
Количество объектов в стеке stk3: 5
```

Передача объектов методам

До сих пор в качестве параметров методов мы использовали значения типа `int` или `double`. Наряду с параметрами в виде значений методам можно передавать объекты. Рассмотрим, например, следующую программу:

```
// Демонстрация возможности передачи методам объектов.

using System;

class MyClass {
  int alpha, beta;

  public MyClass(int i, int j) {
    alpha = i;
    beta = j;
  }

  /* Метод возвращает true, если параметр ob содержит
     те же значения, что и вызывающий объект. */
  public bool sameAs(MyClass ob) {
    if((ob.alpha == alpha) & (ob.beta == beta))
       return true;
    else return false;
  }
```

```
  // Создаем копию объекта ob.
  public void copy(MyClass ob) {
    alpha = ob.alpha;
    beta  = ob.beta;
  }

  public void show() {
    Console.WriteLine("alpha: {0}, beta: {1}",
                      alpha, beta);
  }
}

class PassOb {
  public static void Main() {
    MyClass ob1 = new MyClass(4, 5);
    MyClass ob2 = new MyClass(6, 7);

    Console.Write("ob1: ");
    ob1.show();

    Console.Write("ob2: ");
    ob2.show();

    if(ob1.sameAs(ob2))
      Console.WriteLine(
                "ob1 и ob2 имеют одинаковые значения.");
    else
      Console.WriteLine(
                "ob1 и ob2 имеют разные значения.");

    Console.WriteLine();

    // Теперь делаем объект ob1 копией объекта ob2.
    ob1.copy(ob2);

    Console.Write("ob1 после копирования: ");
    ob1.show();

    if(ob1.sameAs(ob2))
      Console.WriteLine(
                "ob1 и ob2 имеют одинаковые значения.");
    else
      Console.WriteLine(
                "ob1 и ob2 имеют разные значения.");

  }
}
```

Выполнив эту программу, получаем такие результаты:

```
ob1: alpha: 4, beta: 5
ob2: alpha: 6, beta: 7
ob1 и ob2 имеют разные значения.

ob1 после копирования: alpha: 6, beta: 7
ob1 и ob2 имеют одинаковые значения.
```

Каждый из методов — `sameAs()` и `copy()` — принимает в качестве аргумента объект. Метод `sameAs()` сравнивает значения `alpha` и `beta` вызывающего объекта со значениями `alpha` и `beta` объекта, переданного в качестве аргумента `ob`. Этот метод

возвращает значение `true` только в том случае, если сравниваемые объекты содержат одинаковые значения в соответствующих переменных экземпляров. Метод `copy()` присваивает значения `alpha` и `beta` объекта, переданного в качестве аргумента `ob`, переменным экземпляра `alpha` и `beta` вызывающего объекта. Обратите внимание на то, что в обоих случаях в качестве типа параметра указан класс `MyClass`. Как видно из этой программы, объекты (имеющие тип класса) передаются методам точно так же, как и значения встроенных типов.

Как происходит передача аргументов

В предыдущем примере передача аргументов методу представляла собой простую задачу. Однако существуют некоторые нюансы, которые там не были показаны. В определенных случаях результаты передачи объекта будут отличаться от результатов передачи необъектных аргументов. Чтобы понять причину, необходимо рассмотреть два возможных способа передачи аргументов.

Первый способ называется *вызовом по значению* (call-by-value). В этом случае *значение* аргумента копируется в формальный параметр метода. Следовательно, изменения, внесенные в параметр метода, не влияют на аргумент, используемый при вызове. Второй способ передачи аргумента называется *вызовом по ссылке* (call-by-reference). Здесь для получения доступа к реальному аргументу, заданному при вызове, используется ссылка на аргумент. Это значит, что изменения, внесенные в параметр, окажут воздействие на аргумент, использованный при вызове метода.

При передаче методу значения нессылочного типа (например, `int` или `double`) имеет место вызов по значению. Таким образом, то, что происходит с параметром, который получает аргумент, никак не влияет на данные вне метода. Рассмотрим следующую программу:

```
// Демонстрация передачи простых типов по значению.

using System;

class Test {
  /* Этот метод не оказывает влияния на аргументы,
     используемые в его вызове. */
  public void noChange(int i, int j) {
    i = i + j;
    j = -j;
  }
}

class CallByValue {
  public static void Main() {
    Test ob = new Test();

    int a = 15, b = 20;

    Console.WriteLine("a и b перед вызовом: " +
                      a + " " + b);

    ob.noChange(a, b);

    Console.WriteLine("a и b после вызова: " +
                      a + " " + b);
  }
}
```

Результаты выполнения этой программы выглядят так:

```
a и b перед вызовом: 15 20
a и b после вызова: 15 20
```

Как видите, операции, которые выполняются внутри метода `noChange()`, не влияют на значения `a` и `b`, используемые при вызове метода.

При передаче методу ссылки на объект ситуация несколько усложняется. Строго говоря, сама ссылка передается по значению. Таким образом, здесь выполняется копирование ссылки, после чего, как мы уже знаем, изменения, вносимые в параметр, не окажут влияния на аргумент. (Например, если заставить параметр ссылаться на новый объект, это не коснется объекта, на который ссылается аргумент.) Однако (внимание, это очень важно) изменения, вносимые в объект, на который ссылается параметр, повлияют самым прямым образом на объект, на который ссылается аргумент. Давайте разберемся, почему так происходит.

Вспомните, что при создании переменной типа класса вы создаете лишь ссылку на объект. Следовательно, при передаче этой ссылки методу параметр, который получает ее, будет ссылаться на тот же объект, на который ссылается и аргумент. Это как раз означает, что объекты передаются методу посредством вызова по ссылке. Как следствие, изменения в объекте внутри метода влияют на объект, используемый в качестве аргумента. Рассмотрим, например, следующую программу:

```
// Демонстрация передачи объектов по ссылке.

using System;

class Test {
  public int a, b;

  public Test(int i, int j) {
    a = i;
    b = j;
  }
  /* Передаем объект. Теперь ob.a и ob.b в объекте,
     используемом при вызове, будут изменены. */
  public void change(Test ob) {
    ob.a = ob.a + ob.b;
    ob.b = -ob.b;
  }
}

class CallByRef {
  public static void Main() {
    Test ob = new Test(15, 20);

    Console.WriteLine("ob.a и ob.b перед вызовом: " +
                      ob.a + " " + ob.b);

    ob.change(ob);

    Console.WriteLine("ob.a и ob.b после вызова: " +
                      ob.a + " " + ob.b);
  }
}
```

Эта программа генерирует такие результаты:

```
ob.a и ob.b перед вызовом: 15 20
ob.a и ob.b после вызова: 35 -20
```

Как видите, в этом случае действия внутри метода `change()` влияют на объект, используемый в качестве аргумента.

Итак, при передаче методу ссылки на объект сама ссылка передается посредством вызова по значению. И поэтому делается копия этой ссылки. Но поскольку передаваемое значение ссылается на некоторый объект, копия этого значения ссылается на тот же объект.

Использование `ref`- и `out`-параметров

Выше мы рассмотрели пример, в котором значения нессылочного типа (например, `int` или `char`) передавались методу по значению. И мы убедились в том, что изменения, вносимые в параметр, который получает значение нессылочного типа, не влияет на реальный аргумент, используемый при вызове метода. Однако такое поведение можно изменить. Используя ключевые слова `ref` и `out`, можно передать значение любого нессылочного типа по ссылке. Тем самым мы позволим методу изменить аргумент, используемый при вызове.

Прежде чем вникнуть в механизм использования ключевых слов `ref` и `out`, стоит понять, когда может потребоваться передача нессылочного типа по ссылке. В общем случае существует две причины: позволить методу менять содержимое его аргументов или возвращать более одного значения. Рассмотрим подробно каждую из причин.

Часто программисту нужен метод, способный оперировать реальными аргументами, передаваемыми ему при вызове. Классическим примером служит метод `swap()`, который меняет местами значения двух аргументов. При передаче значений нессылочного типа по значению невозможно написать метод обмена значениями двух аргументов, например типа `int`, используя действующий по умолчанию C#-механизм передачи параметров по значению. Эта проблема решается с помощью модификатора `ref`.

Как вы знаете, инструкция `return` позволяет методу возвратить значение тому, кто сделал вызов. Однако метод может вернуть в результате одного вызова *только одно* значение. А как быть, если нужно вернуть два или больше значений? Например, нужен метод, который разбивает вещественное число на целую и дробную части. Ведь в этом случае метод должен возвратить два значения: целую часть и дробную составляющую. Такой метод невозможно написать, используя только одно возвращаемое значение. Вот с этой проблемой и помогает справиться модификатор `out`.

Использование модификатора `ref`

Модификатор параметра `ref` заставляет C# организовать вместо вызова по значению вызов по ссылке. Модификатор `ref` используется при объявлении метода и его вызове. Рассмотрим простой пример. Следующая программа создает метод `sqr()`, который возвращает квадрат целочисленного аргумента. Обратите внимание на использование и расположение модификатора `ref`.

```
// Использование модификатора ref для передачи
// значения нессылочного типа по ссылке.

using System;

class RefTest {
  /* Этот метод изменяет свои аргументы.
     Обратите внимание на использование модификатора ref. */
  public void sqr(ref int i) {
    i = i * i;
```

```
  }
}

class RefDemo {
  public static void Main() {
    RefTest ob = new RefTest();

    int a = 10;

    Console.WriteLine("a перед вызовом: " + a);

    ob.sqr(ref a); // Обратите внимание
                   // на использование модификатора ref.

    Console.WriteLine("a после вызова: " + a);
  }
}
```

Обратите внимание на то, что модификатор стоит в начале объявления параметра в методе и предшествует имени аргумента при вызове метода. Приведенные ниже результаты выполнения этой программы подтверждают, что значение аргумента `a` действительно было модифицировано методом `sqr()`.

```
a перед вызовом: 10
a после вызова: 100
```

Используя модификатор `ref`, можно написать метод, который меняет значения двух аргументов нессылочного типа. Например, рассмотрим программу, которая содержит метод `swap()`, меняющий значения двух целочисленных аргументов, передаваемых ему при вызове.

```
// Обмен значениями двух аргументов.

using System;

class Swap {
  // Этот метод меняет местами значения своих аргументов.
  public void swap(ref int a, ref int b) {
    int t;

    t = a;
    a = b;
    b = t;
  }
}

class SwapDemo {
  public static void Main() {
    Swap ob = new Swap();

    int x = 10, y = 20;

    Console.WriteLine("x и y перед вызовом: " +
                      x + " " + y);

    ob.swap(ref x, ref y);

    Console.WriteLine("x и y после вызова: " +
                      x + " " + y);
  }
}
```

Вот результаты выполнения этой программы:

```
x и y перед вызовом: 10 20
x и y после вызова: 20 10
```

И еще одно немаловажное замечание. Аргументу, передаваемому методу "в сопровождении" модификатора `ref`, должно быть присвоено значение до вызова метода. Дело в том, что, если метод получает такой аргумент, значит, параметр ссылается на действительное значение. Поэтому, используя модификатор `ref`, нельзя использовать метод, присваивая его аргументу начальное значение.

Использование модификатора `out`

Иногда приходится использовать ссылочный параметр не для передачи значения методу, а для его получения из метода. Например, может понадобиться метод, который выполняет некоторую функцию, например открывает сетевой сокет, и возвращает код в параметре ссылочного типа, означающем удачное или неудачное выполнение этой операции. В этом случае методу не нужно передавать какую бы то ни было информацию, но от метода необходимо получить определенный результат. Если воспользоваться модификатором `ref`, то мы должны инициализировать `ref`-параметр некоторым значением до вызова метода. Таким образом, использование `ref`-параметра потребовало бы присвоения его аргументу фиктивного значения только для того, чтобы удовлетворить это требование. К счастью, в C# предусмотрена альтернатива получше, а именно использование модификатора `out`.

Модификатор `out` подобен модификатору `ref` за одним исключением: его можно использовать только для передачи значения из метода. Совсем не обязательно (и даже не нужно) присваивать переменной, используемой в качестве `out`-параметра, начальное значение до вызова метода. Более того, предполагается, что `out`-параметр всегда "поступает" в метод без начального значения, но метод (до своего завершения) обязательно *должен* присвоить этому параметру значение. Таким образом, после обращения к методу `out`-параметр будет содержать определенное значение.

Перед вами пример использования `out`-параметра. В классе `Decompose` метод `parts()` разбивает вещественное число на целую и дробную части. Обратите внимание на то, как возвращается автору вызова этого метода каждый компонент.

```
// Использование модификатора out.

using System;

class Decompose {

  /* Метод разбивает число с плавающей точкой на
     целую и дробную части. */
  public int parts(double n, out double frac) {
    int whole;

    whole = (int) n;
    frac = n - whole; // Передаем дробную часть
                      // посредством параметра frac.
    return whole; // Возвращаем целую часть числа.
  }
}

class UseOut {
  public static void Main() {
    Decompose ob = new Decompose();
```

```
    int i;
    double f;

    i = ob.parts(10.125, out f);

    Console.WriteLine("Целая часть числа равна " + i);
    Console.WriteLine("Дробная часть числа равна " + f);
  }
}
```

При выполнении этой программы получаем такие результаты:

```
Целая часть числа равна 10
Дробная часть числа равна 0.125
```

Метод `parts()` возвращает два значения. Целая часть числа `n` возвращается с помощью инструкции `return`. Дробная часть числа `n` передается автору вызова посредством `out`-параметра `frac`. Как показывает этот пример, используя `out`-параметр, можно добиться того, чтобы метод возвращал не одно, а два значения.

Более того, синтаксис языка C# не ограничивает вас использованием только одного `out`-параметра. Метод может возвращать посредством `out`-параметров столько значений, сколько вам нужно. Ниже приводится пример использования двух `out`-параметров. Метод `isComDenom()` выполняет две функции. Во-первых, он определяет, существует ли у двух заданных целых чисел общий множитель. При этом метод возвращает значение `true`, если такой множитель существует, и значение `false` в противном случае. Во-вторых, если эти числа таки имеют общий множитель, метод `isComDenom()` с помощью `out`-параметров возвращает наименьший и наибольший общие множители.

```
// Демонстрация использования двух out-параметров.

using System;

class Num {
  /* Метод определяет, имеют ли x и v общий множитель.
     Если да, метод возвращает наименьший и наибольший
     общие множители в out-параметрах. */
  public bool isComDenom(int x, int y,
                         out int least,
                         out int greatest) {
    int i;
    int max = x < y ? x : y;
    bool first = true;

    least = 1;
    greatest = 1;

    // Находим наименьший и наибольший общие множители.
    for(i=2; i <= max/2 + 1; i++) {
      if( ((y%i)==0) & ((x%i)==0) ) {
        if(first) {
          least = i;
          first = false;
        }
        greatest = i;
      }
    }

    if(least != 1) return true;
```

```
    else return false;
  }
}

class DemoOut {
  public static void Main() {
    Num ob = new Num();
    int lcd, gcd;

    if(ob.isComDenom(231, 105, out lcd, out gcd)) {
      Console.WriteLine("Lcd для чисел 231 и 105 равен " +
                        lcd);
      Console.WriteLine("Gcd для чисел 231 и 105 равен " +
                        gcd);
    }
    else
      Console.WriteLine(
              "Для чисел 35 и 49 общего множителя нет.");

    if(ob.isComDenom(35, 51, out lcd, out gcd)) {
      Console.WriteLine("Lcd для чисел 35 и 51 равен " +
                        lcd);
      Console.WriteLine("Gcd для чисел 35 и 51 равен " +
                        gcd);
    }
    else
      Console.WriteLine(
              "Для чисел 35 и 51 общего множителя нет.");

  }
}
```

Обратите внимание на то, что в функции `Main()` переменным `lcd` и `gcd` не присваиваются значения до вызова функции `isComDenom()`. Это было бы ошибкой, если бы эти переменные были не `out`-, а `ref`-параметрами. Этот метод в зависимости от существования общих множителей у заданных двух чисел возвращает либо `true`, либо `false`. Причем, если общие множители существуют, то наименьший и наибольший из них возвращаются в `out`-параметрах `lcd` и `gcd`, соответственно. (`Lcd` — это аббревиатура от *least common denominator*, т.е. наименьший общий множитель, а `gcd` — это аббревиатура от *greatest common denominator*, т.е. наибольший общий множитель.) Вот каковы результаты выполнения этой программы:

```
Lcd для чисел 231 и 105 равен 3
Gcd для чисел 231 и 105 равен 21
Для чисел 35 и 51 общего множителя нет.
```

Использование модификаторов `ref` и `out` для ссылочных параметров

Использование модификаторов `ref` и `out` не ограничивается параметрами типа значений. Их также можно применить к ссылочным параметрам, т.е. параметрам, обеспечивающим передачу объектов. Если параметр ссылочного типа модифицируется одним из модификаторов `ref` и `out`, то по сути реализуется передача ссылки по ссылке. Это позволяет методу изменять объект, на который указывает ссылка-параметр. Рассмотрим программу, в которой используются ссылочные `ref`-параметры для обмена объектами, на которые указывают две ссылки.

```
// Обмен двух ссылок.

using System;

class RefSwap {
  int a, b;

  public RefSwap(int i, int j) {
    a = i;
    b = j;
  }

  public void show() {
    Console.WriteLine("a: {0}, b: {1}", a, b);
  }

  // Этот метод теперь изменяет свои аргументы.
  public void swap(ref RefSwap ob1, ref RefSwap ob2) {
    RefSwap t;

    t = ob1;
    ob1 = ob2;
    ob2 = t;
  }
}

class RefSwapDemo {
  public static void Main() {
    RefSwap x = new RefSwap(1, 2);
    RefSwap y = new RefSwap(3, 4);

    Console.Write("x перед вызовом: ");
    x.show();

    Console.Write("y перед вызовом: ");
    y.show();

    Console.WriteLine();

    // Обмениваем объекты, на которые ссылаются x и y.
    x.swap(ref x, ref y);

    Console.Write("x после вызова: ");
    x.show();

    Console.Write("y после вызова: ");
    y.show();

  }
}
```

При выполнении этой программы получаем такие результаты:

```
x перед вызовом: a: 1, b: 2
y перед вызовом: a: 3, b: 4

x после вызова: a: 3, b: 4
y после вызова: a: 1, b: 2
```

В этом примере метод `swap()` меняет местами объекты, на которые ссылаются два его аргумента. До вызова метода `swap()` переменная `x` ссылается на объект, который содержит значения 1 и 2, а переменная `y` ссылается на объект, который содержит значения 3 и 4. После обращения к методу `swap()` переменная `x` ссылается на объект, который содержит значения 3 и 4, а переменная `y` ссылается на объект, который содержит значения 1 и 2. Если бы здесь не были использованы `ref`-параметры, то обмен внутри метода `swap()` никак не повлиял бы на "среду" за его пределами. Это можно доказать, убрав модификаторы из метода `swap()`.

Использование переменного количества аргументов

При создании метода обычно заранее известно количество аргументов, которые будут ему передаваться. Но иногда необходимо, чтобы метод принимал произвольное число аргументов. Рассмотрим, например, метод, который находит минимальное значение в наборе чисел. Такому методу может быть передано два, три или четыре значения. В любом случае метод должен возвращать наименьшее значение. Такой метод невозможно создать при использовании обычных параметров. Здесь необходимо применить специальный тип параметра, который заменяет собой произвольное количество параметров. Это реализуется с помощью модификатора `params`.

Модификатор `params` используется для объявления параметра-массива, который сможет получить некоторое количество аргументов (в том числе и нулевое). Количество элементов в массиве будет равно числу аргументов, переданных методу.

Рассмотрим пример, в котором модификатор `params` используется для создания метода `minVal()`, возвращающего минимальное значение из набора.

```
// Демонстрация использования модификатора params.

using System;

class Min {
  public int minVal(params int[] nums) {
    int m;

    if(nums.Length == 0) {
      Console.WriteLine("Ошибка: нет аргументов.");
      return 0;
    }

    m = nums[0];
    for(int i=1; i < nums.Length; i++)
      if(nums[i] < m) m = nums[i];

    return m;
  }
}

class ParamsDemo {
  public static void Main() {
    Min ob = new Min();
    int min;
    int a = 10, b = 20;
```

```
    // Вызываем метод с двумя значениями.
    min = ob.minVal(a, b);
    Console.WriteLine("Минимум равен " + min);

    // call with 3 values
    min = ob.minVal(a, b, -1);
    Console.WriteLine("Минимум равен " + min);

    // Вызываем метод с пятью значениями.
    min = ob.minVal(18, 23, 3, 14, 25);
    Console.WriteLine("Минимум равен " + min);

    // Этот метод можно также вызвать с int-массивом.
    int[] args = { 45, 67, 34, 9, 112, 8 };
    min = ob.minVal(args);
    Console.WriteLine("Минимум равен " + min);
  }
}
```

Вот результаты выполнения этой программы:

```
Минимум равен 10
Минимум равен -1
Минимум равен 3
Минимум равен 8
```

При каждом вызове метода `minVal()` аргументы передаются ему через массив `nums`. Длина этого массива равна количеству элементов. Поэтому метод `minVal()` можно использовать для определения минимального из любого числа элементов.

Несмотря на то что `params`-параметру можно передать любое количество аргументов, все они должны иметь тип, совместимый с типом массива, заданным этим параметром. Например, такой вызов метода `minVal()`

```
min = ob.minVal(1, 2.2);
```

неверен, поскольку автоматического преобразования значения типа `double(2.2)` в значение типа `int` (тип `int` имеет массив `nums` в методе `minVal()`) не существует.

При использовании модификатора `params` необходимо внимательно отнестись к граничным ситуациям задания аргументов, поскольку `params`-параметр может принять любое количество аргументов, *даже нулевое!* Например, синтаксически вполне допустимо вызвать метод `minVal()` следующим образом:

```
min = ob.minVal();   // аргументы отсутствуют
min = ob.minVal(3);  // один аргумент
```

Вот поэтому в методе `minVal()` до попытки доступа к элементу массива `nums` предусмотрена проверка существования хотя бы одного элемента в массиве. Если бы такая проверка отсутствовала, то при вызове метода `minVal()` без аргументов имела бы место исключительная ситуация. (Ниже в этой книге при рассмотрении исключительных ситуаций будет показан более удачный способ обработки таких типов ошибок.) Более того, код метода `minVal()` был написан так специально, чтобы разрешить его вызов с одним аргументом. В этом случае метод возвращает этот (единственный) аргумент.

Наряду с обычными параметрами методы могут иметь и параметр переменной длины. Например, в следующей программе метод `showArgs()` принимает один параметр типа `string` и `params`-массив целочисленного типа.

```
// Использование обычного параметра вместе
// с params-параметром.
```

```
using System;

class MyClass {
  public void showArgs(string msg, params int[] nums) {
    Console.Write(msg + ": ");

    foreach(int i in nums)
      Console.Write(i + " ");

    Console.WriteLine();
  }
}

class ParamsDemo2 {
  public static void Main() {
    MyClass ob = new MyClass();

    ob.showArgs("Вот несколько целых чисел",
                1, 2, 3, 4, 5);

    ob.showArgs("А вот еще два числа",
                17, 20);

  }
}
```

Программа генерирует следующие результаты:

```
Вот несколько целых чисел: 1 2 3 4 5
А вот еще два числа: 17 20
```

Когда метод принимает обычные параметры и `params`-параметр, `params`-параметр должен стоять в списке параметров последним и быть единственным в своем роде.

Возвращение методами объектов

Метод может возвращать данные любого типа, в том числе классового. Например, следующая версия класса `Rect` содержит метод `enlarge()`, который создает объект прямоугольника как результат пропорционального увеличения (при заданном коэффициенте увеличения) сторон вызывающего (этот метод) объекта прямоугольника.

```
// Демонстрация возвращения методом объекта.

using System;

class Rect {
  int width;
  int height;

  public Rect(int w, int h) {
    width = w;
    height = h;
  }

  public int area() {
    return width * height;
  }
```

```
  public void show() {
    Console.WriteLine(width + " " + height);
  }

  /* Метод возвращает прямоугольник, который увеличен по
     сравнению с вызывающим объектом прямоугольника с
     использованием заданного коэффициента увеличения. */
  public Rect enlarge(int factor) {
    return new Rect(width * factor, height * factor);
  }
}

class RetObj {
  public static void Main() {
    Rect r1 = new Rect(4, 5);

    Console.Write("Размеры прямоугольника r1: ");
    r1.show();
    Console.WriteLine("Площадь прямоугольника r1: " +
                      r1.area());

    Console.WriteLine();

    // Создаем прямоугольник, который вдвое больше
    // прямоугольника r1 .
    Rect r2 = r1.enlarge(2);

    Console.Write("Размеры прямоугольника r2: ");
    r2.show();
    Console.WriteLine("Площадь прямоугольника r2: " +
                      r2.area());
  }
}
```

Вот результаты выполнения этой программы:

```
Размеры прямоугольника r1: 4 5
Площадь прямоугольника r1: 20

Размеры прямоугольника r2: 8 10
Площадь прямоугольника r2: 80
```

В тех случаях, когда метод возвращает объект, существование этого объекта продолжается до тех пор, пока на него есть хотя бы одна ссылка. И только когда на объект больше нет ни одной ссылки, он подвергается утилизации, т.е. попадает в поле действия процесса сбора мусора. Таким образом, объект не будет разрушен только по причине завершения метода, который его создал.

Одним из применений классовых типов значений, возвращаемых методами, является генератор объектов класса, или *фабрика класса* (class factory). "Фабрика" класса — это метод, который используется для построения объектов заданного класса. В определенных случаях пользователям некоторого класса нежелательно предоставлять доступ к конструктору этого класса из соображений безопасности или по причине того, что создание объектов зависит от неких внешних факторов. В таких случаях для построения объектов и используется "фабрика" класса. Рассмотрим простой пример:

```
// Использование "фабрики" класса.

using System;
```

```
class MyClass {
  int a, b; // закрытые члены

  // Создаем "фабрику" класса для класса MyClass.
  public MyClass factory(int i, int j) {
    MyClass t = new MyClass();

    t.a = i;
    t.b = j;

    return t; // Метод возвращает объект.
  }

  public void show() {
    Console.WriteLine("a и b: " + a + " " + b);
  }

}

class MakeObjects {
  public static void Main() {
    MyClass ob = new MyClass();
    int i, j;

    // Генерируем объекты с помощью "фабрики" класса.
    for(i=0, j=10; i < 10; i++, j--) {
      MyClass anotherOb = ob.factory(i, j); // Создаем
                                            // объект.
      anotherOb.show();
    }

    Console.WriteLine();
  }
}
```

Вот результаты выполнения этой программы:

```
a и b: 0 10
a и b: 1 9
a и b: 2 8
a и b: 3 7
a и b: 4 6
a и b: 5 5
a и b: 6 4
a и b: 7 3
a и b: 8 2
a и b: 9 1
```

Рассмотрим этот пример более внимательно. В классе `MyClass` конструктор не определен, поэтому доступен только конструктор, создаваемый средствами C# по умолчанию. Следовательно, установить значения членов класса `a` и `b` с помощью конструктора невозможно. Однако создавать объекты с заданными значениями членов `a` и `b` способна "фабрика" класса, реализованная в виде метода `factory()`. Более того, поскольку члены `a` и `b` закрыты, использование метода `factory()` — единственный способ установки этих значений.

В функции `Main()` создается объект `ob` класса `MyClass`, а затем в цикле `for` создается еще десять объектов. Приведем здесь строку кода, которая представляет собой основной "конвейер" объектов.

```
MyClass anotherOb = ob.factory(i, j); // создаем объект
```

На каждой итерации цикла создается ссылочная переменная anotherOb, которой присваивается ссылка на объект, сгенерированный "фабрикой" объектов. В конце каждой итерации цикла ссылочная переменная anotherOb выходит из области видимости, и объект, на который она ссылалась, утилизируется.

Возвращение методами массивов

Поскольку в C# массивы реализованы как объекты, метод может возвратить массив. (В этом еще одно отличие C# от языка C++, в котором не допускается, чтобы метод, или функция, возвращал массив.) Например, в следующей программе метод findfactors() возвращает массив, который содержит множители аргумента, переданного этому методу.

```
// Демонстрация возврата методом массива.

using System;

class Factor {
  /* Метод возвращает массив, содержащий множители
     параметра num. После выполнения метода
     out-параметр numfactors будет содержать количество
     найденных множителей. */
  public int[] findfactors(int num, out int numfactors) {
    int[] facts = new int[80]; // Размер 80 взят произвольно.
    int i, j;

    // Находим множители и помещаем их в массив facts.
    for(i=2, j=0; i < num/2 + 1; i++)
      if( (num%i)==0 ) {
        facts[j] = i;
        j++;
      }

    numfactors = j;
    return facts;
  }
}

class FindFactors {
  public static void Main() {
    Factor f = new Factor();
    int numfactors;
    int[] factors;

    factors = f.findfactors(1000, out numfactors);

    Console.WriteLine("Множители числа 1000: ");
    for(int i=0; i < numfactors; i++)
      Console.Write(factors[i] + " ");

    Console.WriteLine();
  }
}
```

Вот результаты выполнения этой программы:

```
Множители числа 1000:
2 4 5 8 10 20 25 40 50 100 125 200 250 500
```

В классе `Factor` метод `findfactors()` объявляется следующим образом:

```
public int[] findfactors(int num, out int numfactors) {
```

Обратите внимание на то, как задан тип возвращаемого массива `int`. Этот синтаксис можно обобщить. Если вам нужно, чтобы метод возвращал массив, объявите его (метод) подобным образом, изменив при необходимости тип массива и размерность. Например, эта инструкция объявляет метод с именем `someMeth()`, который возвращает двумерный массив `double`-значений.

```
public double[,] someMeth() { // ...
```

Перегрузка методов

В этом разделе мы узнаем об одной из самых удивительных возможностей языка C# — перегрузке методов. В C# два или больше методов внутри одного класса могут иметь одинаковое имя, но при условии, что их параметры будут различными. Такую ситуацию называют *перегрузкой методов* (method overloading), а методы, которые в ней задействованы, — *перегруженными* (overloaded). Перегрузка методов — один из способов реализации полиморфизма в C#.

В общем случае для создания перегрузки некоторого метода достаточно объявить еще одну его версию. Об остальном позаботится компилятор. Но здесь необходимо отметить одно важное условие: все перегруженные методы должны иметь списки параметров, которые отличаются по типу и/или количеству. Методам для перегрузки *недостаточно* отличаться лишь типами возвращаемых значений. Они должны отличаться типами или числом параметров. (Другими словами, тип возвращаемого значения не обеспечивает достаточную информацию для C#, чтобы можно решить, какой именно метод должен быть вызван.) Конечно, перегруженные методы *могут* отличаться и типами возвращаемых значений. При вызове перегруженного метода выполняется та его версия, параметры которой совпадают (по типу и количеству) с заданными аргументами.

Вот простой пример, иллюстрирующий перегрузку методов:

```
// Демонстрация перегрузки методов.

using System;

class Overload {
  public void ovlDemo() {
    Console.WriteLine("Без параметров");
  }

  // Перегружаем метод ovlDemo() для одного
  // целочисленного параметра.
  public void ovlDemo(int a) {
    Console.WriteLine("Один параметр: " + a);
  }

  // Перегружаем метод ovlDemo() для двух
  // целочисленных параметров.
  public int ovlDemo(int a, int b) {
    Console.WriteLine("Два int-параметра: " + a + " " + b);
    return a + b;
  }

  // Перегружаем метод ovlDemo() для двух
```

```
  // double-параметров.
  public double ovlDemo(double a, double b) {
    Console.WriteLine("Два double-параметра: " +
                      a + " "+ b);
    return a + b;
  }
}

class OverloadDemo {
  public static void Main() {
    Overload ob = new Overload();
    int resI;
    double resD;

    // Вызываем все версии метода ovlDemo().
    ob.ovlDemo();
    Console.WriteLine();

    ob.ovlDemo(2);
    Console.WriteLine();

    resI = ob.ovlDemo(4, 6);
    Console.WriteLine("Результат вызова ob.ovlDemo(4, 6): "
                      + resI);
    Console.WriteLine();


    resD = ob.ovlDemo(1.1, 2.32);
    Console.WriteLine(
             "Результат вызова ob.ovlDemo(1.1, 2.2): " +
             resD);
  }
}
```

Программа генерирует следующие результаты:

```
Без параметров

Один параметр: 2

Два int-параметра: 4 6
Результат вызова ob.ovlDemo(4, 6): 10

Два double-параметра: 1.1 2.32
Результат вызова ob.ovlDemo(1.1, 2.2): 3.42
```

Как видите, метод `ovlDemo()` перегружается четыре раза. Первая версия вообще не принимает параметров, вторая принимает один целочисленный параметр, третья — два целочисленных параметра, а четвертая — два `double`-параметра. Обратите внимание на то, что первые две версии метода `ovlDemo()` возвращают тип `void`, т.е. не возвращают никакого значения, а вторые две возвращают значения соответствующих типов. Это вполне допустимо, но, как уже разъяснялось, перегрузка методов не достигается различием только в типе возвращаемого значения. Поэтому попытка использовать следующие две версии метода `ovlDemo()` приведет к ошибке:

```
// Одно объявление метода ovlDemo(int) вполне допустимо.
public void ovlDemo(int a) {
    Console.WriteLine("Один параметр: " + a);
}
```

```
// Ошибка! Два объявления метода ovlDemo(int) неприемлемы,
// несмотря на то, что типы возвращаемых ими значений
// разные.
public int ovlDemo(int a) {
    Console.WriteLine("Один параметр: " + a);
    return a * a;
}
```

Как отмечено в комментариях, различие в типах значений, возвращаемых методами, является недостаточным фактором для обеспечения их перегрузки.

Как указывалось в главе 3, в определенных пределах C# обеспечивает автоматическое преобразование типов. Эта возможность преобразования типов применяется и к параметрам перегруженных методов. Рассмотрим, например, следующую программу.

```
/* Возможность автоматического преобразования типов может
   повлиять на решение о перегрузке методов. */

using System;

class Overload2 {
  public void f(int x) {
    Console.WriteLine("Внутри метода f(int): " + x);
  }

  public void f(double x) {
    Console.WriteLine("Внутри метода f(double): " + x);
  }
}

class TypeConv {
  public static void Main() {
    Overload2 ob = new Overload2();

    int i = 10;
    double d = 10.1;

    byte b = 99;
    short s = 10;
    float f = 11.5F;


    ob.f(i); // Вызов метода ob.f(int).
    ob.f(d); // Вызов метода ob.f(double).

    ob.f(b); // Вызов метода ob.f(int) -- выполняется
             // преобразование типов.
    ob.f(s); // Вызов метода ob.f(int) -- выполняется
             // преобразование типов.
    ob.f(f); // Вызов метода ob.f(double) -- выполняется
             // преобразование типов.
  }
}
```

Вот результаты выполнения этой программы:

```
Внутри метода f(int): 10
Внутри метода f(double): 10.1
Внутри метода f(int): 99
Внутри метода f(int): 10
Внутри метода f(double): 11.5
```

В этом примере определены только две версии метода f(): одна с int-, а другая с double-параметром. Тем не менее методу f() можно передать помимо значений типа int и double также значения типа byte, short или float. В случае передачи byte- или short-параметров C# автоматически преобразует их в значения типа int (т.е. будет вызвана версия f(int)). В случае передачи float-параметра его значение будет преобразовано в значение типа double и будет вызвана версия f(double).

Здесь важно понимать, что автоматическое преобразование применяется только в том случае, когда не существует прямого соответствия параметра и аргумента. Дополним предыдущую программу версией метода f(), в которой определен параметр типа byte.

```
// Добавление к предыдущей программе версии f(byte).

using System;

class Overload2 {
  public void f(byte x) {
    Console.WriteLine("Внутри метода f(byte): " + x);
  }

  public void f(int x) {
    Console.WriteLine("Внутри метода f(int): " + x);
  }

  public void f(double x) {
    Console.WriteLine("Внутри метода f(double): " + x);
  }
}

class TypeConv {
  public static void Main() {
    Overload2 ob = new Overload2();

    int i = 10;
    double d = 10.1;

    byte b = 99;
    short s = 10;
    float f = 11.5F;


    ob.f(i); // Вызов метода ob.f(int).
    ob.f(d); // Вызов метода ob.f(double).

    ob.f(b); // Вызов метода ob.f(byte) - теперь без
             // преобразования типов.

    ob.f(s); // Вызов метода ob.f(int) -- выполняется
             // преобразование типов.
    ob.f(f); // Вызов метода ob.f(double) -- выполняется
             // преобразование типов.

  }
}
```

Этот вариант программы генерирует такие результаты:

```
Внутри метода f(int): 10
Внутри метода f(double): 10.1
```

```
Внутри метода f(byte): 99
Внутри метода f(int): 10
Внутри метода f(double): 11.5
```

В этом варианте, поскольку существует версия метода `f()`, которая предназначена для приема аргумента типа `byte`, при вызове метода `f()` с `byte`-аргументом будет вызвана версия `f(byte)`, и автоматического преобразования `byte`-аргумента в значение типа `int` не произойдет.

Наличие как `ref`-, так и `out`-модификатора играет роль в "зачете" перегруженных функций. Например, в следующем фрагменте кода определяются два различных метода.

```
public void f(int x) {
    Console.WriteLine("Внутри метода f(int): " + x);
  }

public void f(ref int x) {
    Console.WriteLine("Внутри метода f(ref int): " + x);
  }
```

Таким образом, при выполнении инструкции

```
ob.f(i);
```

вызывается метод `f(int x)`, но при выполнении инструкции

```
ob.f(ref i);
```

вызывается метод `f(ref int x)`.

Посредством перегрузки методов в C# поддерживается полиморфизм, поскольку это единственный способ реализации в C# парадигмы "один интерфейс — множество методов". Чтобы понять, как это происходит, рассмотрим следующее. В языке, который не поддерживает перегрузку методов, каждый метод должен иметь уникальное имя. Однако часто нужно реализовать один и тот же метод для различных типов данных. Возьмем, например, функцию, возвращающую абсолютное значение. В языках, которые не поддерживают перегрузку методов, обычно существует три или даже больше версий этой функции, причем их имена незначительно отличаются. Например, в языке C функция `abs()` возвращает абсолютное значение (модуль) целого числа, функция `labs()` возвращает модуль длинного целочисленного значения, а `fabs()` — модуль значения с плавающей точкой. Поскольку язык C не поддерживает перегрузку методов, каждая функция должна иметь собственное имя, несмотря на то, что все три функции выполняют по сути одно и то же действие. Это делает ситуацию сложнее, чем она есть на самом деле. Другими словами, при одних и тех же действиях программисту необходимо помнить имена всех трех (в данном случае) функций. Язык C# избавлен от ненужного "размножения" имен, поскольку все методы получения абсолютного значения могут использовать одно и то же имя. И в самом деле, библиотека стандартных классов C# включает метод получения абсолютного значения с именем `Abs()`. Этот метод перегружается C#-классом `System.Math`, что позволяет обрабатывать значения всех числовых типов, используя одно имя метода. Определение того, какая именно версия метода должна быть вызвана, основано на типе передаваемого аргумента.

Принципиальная значимость перегрузки состоит в том, что она позволяет обращаться к связанным методам посредством одного, общего для всех имени. Следовательно, имя `Abs()` представляет *общее действие*, которое выполняется во всех случаях. Компилятору остается правильно выбрать *конкретную* версию при конкретных обстоятельствах. А программисту нужно помнить лишь общую операцию, которая связана с именем того или иного метода. Благодаря полиморфизму применение нескольких имен сводится к одному. Несмотря на простоту приведенного примера, он все же позволяет понять, что перегрузка способна упростить процесс программирования.

Необходимо подчеркнуть, что каждая версия перегруженного метода может выполнять определенные действия. Не существует правила, которое бы обязывало программиста связывать перегруженные методы общими действиями. Однако с точки зрения стилистики перегрузка методов все-таки подразумевает определенное "родство" его версий. Таким образом, несмотря на то, что вы можете использовать одно и то же имя для перегрузки не связанных общими действиями методов, этого делать не стоит. Например, в принципе можно использовать имя `sqr` для создания метода, который возвращает квадрат целого числа, и метода, который возвращает значение квадратного корня из вещественного числа. Но поскольку эти операции фундаментально различны, применение механизма перегрузки методов в этом случае сводит на нет его первоначальную цель. Хороший стиль программирования состоит в организации перегрузки тесно связанных операций.

В C# используется термин *сигнатура* (signature), который представляет собой имя метода со списком его параметров. Таким образом, в целях обеспечения перегрузки никакие два метода внутри одного и того же класса не должны иметь одинаковую сигнатуру. Обратите внимание на то, что сигнатура не включает тип значения, возвращаемого методом, поскольку этот фактор не используется в C# для принятия решения о выполнении требуемого перегруженного метода. Сигнатура также не включает `params`-параметр, если таковой существует. Другими словами, модификатор `params` не является определяющим фактором отличия одного перегруженного метода от другого.

Перегрузка конструкторов

Подобно другим методам, конструкторы также можно перегружать. Это позволяет создавать объекты различными способами. Рассмотрим следующую программу:

```
// Демонстрация перегруженных конструкторов.

using System;

class MyClass {
  public int x;

  public MyClass() {
    Console.WriteLine("Внутри конструктора MyClass().");
    x = 0;
  }

  public MyClass(int i) {
    Console.WriteLine("Внутри конструктора MyClass(int).");
    x = i;
  }

  public MyClass(double d) {
    Console.WriteLine(
              "Внутри конструктора MyClass(double).");
    x = (int) d;
  }

  public MyClass(int i, int j) {
    Console.WriteLine(
              "Внутри конструктора MyClass(int, int).");
    x = i * j;
```

```
  }
}

class OverloadConsDemo {
  public static void Main() {
    MyClass t1 = new MyClass();
    MyClass t2 = new MyClass(88);
    MyClass t3 = new MyClass(17.23);
    MyClass t4 = new MyClass(2, 4);

    Console.WriteLine("t1.x: " + t1.x);
    Console.WriteLine("t2.x: " + t2.x);
    Console.WriteLine("t3.x: " + t3.x);
    Console.WriteLine("t4.x: " + t4.x);
  }
}
```

При выполнении этой программы получаем следующие результаты:

```
Внутри конструктора MyClass().
Внутри конструктора MyClass(int).
Внутри конструктора MyClass(double).
Внутри конструктора MyClass(int, int).
t1.x: 0
t2.x: 88
t3.x: 17
t4.x: 8
```

Конструктор `MyClass()` перегружен четырежды, и все конструкторы создают объекты по-разному. В зависимости от того, какие параметры заданы при выполнении оператора `new`, вызывается соответствующий конструктор. Перегружая конструктор класса, вы тем самым предоставляете пользователю этого класса определенную гибкость в выборе способа создания объектов.

Одна из самых распространенных причин перегрузки конструкторов — возможность инициализации одного объекта с помощью другого. Например, вот как выглядит усовершенствованная версия представленного выше класса `Stack`, которая позволяет создать один стек на основе другого:

```
// Класс стека для хранения символов.

using System;

class Stack {
  // Эти члены закрыты.
  char[] stck; // Этот массив содержит стек.
  int tos;     // Индекс вершины стека.

  // Создаем пустой объект класса Stack заданного размера.
  public Stack(int size) {
    stck = new char[size]; // Выделяем память для стека.
    tos = 0;
  }

  // Создаем Stack-объект на основе существующего стека.
  public Stack(Stack ob) {
    // Выделяем память для стека.
    stck = new char[ob.stck.Length];

    // Копируем элементы в новый стек.
    for(int i=0; i < ob.tos; i++)
```

```
      stck[i] = ob.stck[i];

    // Устанавливаем переменную tos для нового стека.
    tos = ob.tos;
  }

  // Помещаем символ в стек.
  public void push(char ch) {
    if(tos==stck.Length) {
      Console.WriteLine(" -- Стек заполнен.");
      return;
    }

    stck[tos] = ch;
    tos++;
  }

  // Извлекаем символ из стека.
  public char pop() {
    if(tos==0) {
      Console.WriteLine(" -- Стек пуст.");
      return (char) 0;
    }

    tos--;
    return stck[tos];
  }

  // Метод возвращает значение true, если стек заполнен.
  public bool full() {
    return tos==stck.Length;
  }

  // Метод возвращает значение true, если стек пуст.
  public bool empty() {
    return tos==0;
  }

  // Возвращает общий объем стека.
  public int capacity() {
    return stck.Length;
  }

  // Возвращает текущее количество объектов в стеке.
  public int getNum() {
    return tos;
  }
}

// Демонстрация использования класса Stack.
class StackDemo {
  public static void Main() {
    Stack stk1 = new Stack(10);
    char ch;
    int i;

    // Помещаем символы в стек stk1.
    Console.WriteLine(
```

```
        "Помещаем символы от A до Z в стек stk1.");
    for(i=0; !stk1.full(); i++)
      stk1.push((char) ('A' + i));

    // Создаем копию стека stck1.
    Stack stk2 = new Stack(stk1);

    // Отображаем содержимое стека stk1.
    Console.Write("Содержимое стека stk1: ");
    while( !stk1.empty() ) {
      ch = stk1.pop();
      Console.Write(ch);
    }

    Console.WriteLine();

    Console.Write("Содержимое стека stk2: ");
    while ( !stk2.empty() ) {
      ch = stk2.pop();
      Console.Write(ch);
    }

    Console.WriteLine("\n");

  }
}
```

Результаты выполнения этой программы:

```
Помещаем символы от A до Z в стек stk1.
Содержимое стека stk1: JIHGFEDCBA
Содержимое стека stk2: JIHGFEDCBA
```

В классе `StackDemo` создается пустым первый стек `stk1`, который заполняется символами. Этот стек затем используется для создания второго стека `stk2`, и в этом случае вызывается следующий конструктор класса `Stack`.

```
// Создаем Stack-объект из существующего стека.
  public Stack(Stack ob) {
    // Выделяем память для стека.
    stck = new char[ob.stck.Length];

    // Копируем элементы в новый стек.
    for(int i=0; i < ob.tos; i++)
      stck[i] = ob.stck[i];

    // Устанавливаем переменную tos для нового стека.
    tos = ob.tos;
  }
```

При выполнении кода этого конструктора для массива `stck` выделяется область памяти, причем ее размер позволяет поместить в этот массив все элементы, содержащиеся в стеке, заданном в качестве аргумента `ob`. Затем содержимое базового массива, на котором основан стек `ob`, копируется в новый массив, и соответствующим образом устанавливается переменная индекса `tos`. По завершении работы этого конструктора новый и исходный стеки являются отдельными объектами, но идентичны по своему содержимому.

Вызов перегруженного конструктора с помощью ссылки this

При работе с перегруженными конструкторами иногда необходимо обеспечить вызов одного конструктора из другого. В C# это реализуется с помощью еще одной формы ключевого слова this. Общий формат записи такого вызова:

```
имя_конструктора(список_параметров1) :
                        this(список_параметров2) {
  // ... Тело конструктора,
  // которое может быть пустым.
}
```

При выполнении перегруженного конструктора сначала вызывается та его версия, список параметров которой совпадает с элементом *список_параметров2*. При этом будут выполнены любые инструкции, содержащиеся внутри исходного конструктора. Например:

```
// Демонстрация вызова конструктора с помощью ссылки this.

using System;

class XYCoord {
  public int x, y;

  public XYCoord() : this(0, 0) {
    Console.WriteLine("Внутри конструктора XYCoord()");
  }

  public XYCoord(XYCoord obj) : this(obj.x, obj.y) {
    Console.WriteLine("Внутри конструктора XYCoord(obj)");
  }

  public XYCoord(int i, int j) {
    Console.WriteLine("Внутри конструктора XYCoord(int, int)");
    x = i;
    y = j;
  }
}

class OverloadConsDemo {
  public static void Main() {
    XYCoord t1 = new XYCoord();
    XYCoord t2 = new XYCoord(8, 9);
    XYCoord t3 = new XYCoord(t2);

    Console.WriteLine("t1.x, t1.y: " + t1.x + ", " + t1.y);
    Console.WriteLine("t2.x, t2.y: " + t2.x + ", " + t2.y);
    Console.WriteLine("t3.x, t3.y: " + t3.x + ", " + t3.y);
  }
}
```

Эта программа генерирует следующие результаты:

```
Внутри конструктора XYCoord(int, int)
Внутри конструктора XYCoord()
Внутри конструктора XYCoord(int, int)
Внутри конструктора XYCoord(int, int)
Внутри конструктора XYCoord(obj)
t1.x, t1.y: 0, 0
t2.x, t2.y: 8, 9
t3.x, t3.y: 8, 9
```

Вот как работает эта программа. В классе `XYCoord` единственным конструктором, который реально инициализирует члены `x` и `y`, является `XYCoord(int, int)`. Остальные два конструктора просто вызывают конструктор `XYCoord(int, int)`, используя ключевое слово `this`. Например, при создании объекта `t1` вызывается конструктор `XYCoord()`, выполняющий вызов `this(0, 0)`, который преобразуется в вызов конструктора `XYCoord(0, 0)`. Создание объекта `t2` происходит аналогично.

Преимущество использования ключевого слова `this` для вызова перегруженных конструкторов состоит в том, что можно избежать ненужного дублирования кода. В предыдущем примере применение слова `this` позволило избежать дублирования всеми тремя конструкторами одного и того же кода инициализации членов. Еще одно достоинство этого средства — возможность создавать конструкторы с заданием действующих "по умолчанию" аргументов, которые используются в том случае, когда аргументы конструктора не заданы явным образом. Например, вы могли бы создать еще один конструктор класса `XYCoord` следующим образом:

```
public XYCoord(int x) : this(x, x) { }
```

Этот конструктор автоматически устанавливает координату `y` равной значению координаты `x`. Конечно, использовать такие действующие "по умолчанию" аргументы нужно очень аккуратно, поскольку их неправильное использование может ввести пользователей в заблуждение.

Метод `Main()`

До сих пор мы использовали только одну форму метода `Main()`. Однако существует несколько перегруженных форм этого метода. Одни возвращают значение, а другие принимают аргументы. Рассмотрением этих форм мы и займемся в следующих разделах.

Возвращение значений из метода `Main()`

По завершении программы можно возвратить значение вызывающему процессу (часто в его роли выступает операционная система). Для этого используется следующая форма метода `Main()`:

```
public static int Main()
```

Обратите внимание на то, что вместо типа `void`, эта версия метода `Main()` имеет в качестве типа возвращаемого значения `int`.

Обычно значение, возвращаемое методом `Main()`, служит индикатором того, как была завершена программа (нормально или аварийно). По соглашению нулевое значение, как правило, подразумевает нормальное завершение. Все же другие значения соответствуют определенным типам ошибок.

Передача аргументов методу `Main()`

Многие программы принимают *аргументы командной строки*. Аргумент командной строки — это информация, которая указывается при запуске программы сразу после ее имени в командной строке. Эти аргументы затем передаются методу `Main()`. Для работы с аргументами командной строки необходимо использовать одну из следующих форм метода `Main()`:

```
public static void Main(string[] args)
public static int Main(string[] args)
```

Первая форма возвращает значение типа void, а вторую можно использовать для возврата целочисленного значения, как описано в предыдущем разделе. В обоих случаях аргументы командной строки хранятся как строки в string-массиве, передаваемом методу Main().

Следующая программа отображает все аргументы командной строки, с которыми она была вызвана.

```
// Отображение всей информации из командной строки.

using System;

class CLDemo {
  public static void Main(string[] args) {
    Console.WriteLine("Командная строка содержит " +
                      args.Length +
                      " аргументов.");

    Console.WriteLine("Вот они: ");
    for(int i=0; i<args.Length; i++)
      Console.WriteLine(args[i]);
  }
}
```

Предположим, мы запустили на выполнение программу CLDemo следующим образом:

```
CLDemo один два три четыре пять
```

В этом случае мы увидим такие результаты:

```
Командная строка содержит 5 аргументов.
Вот они:
один
два
три
четыре
пять
```

Чтобы "попробовать на вкус" возможности использования аргументов командной строки, рассмотрим следующую программу. Она кодирует и декодирует сообщения. Сообщение, предназначенное для кодирования или декодирования, указывается в командной строке. Метод шифрования очень прост: чтобы закодировать слово, код каждой его буквы инкрементируется на 1. В результате буква "А" превращается в букву "Б" и т.д. Чтобы декодировать слово, достаточно код каждой его буквы декрементировать на 1.

```
// Кодирование и декодирование сообщений.

using System;

class Cipher {
  public static int Main(string[] args) {

    // Проверка наличия аргументов.
    if(args.Length < 2) {
      Console.WriteLine(
             "ИСПОЛЬЗОВАНИЕ: " +
             "слово1: <<закодировать>>/<<раскодировать>> " +
             "[слово2... словоN]");
      return 1; // Возврат признака неверного выполнения.
    }

    // Если аргументы присутствуют, то первым аргументом
```

```
    // должно быть слово "закодировать" или "раскодировать".
    if(args[0] != "закодировать" & args[0] != "раскодировать") {
      Console.WriteLine(
              "Первым аргументом должно быть слово " +
              "\"закодировать\" или \"раскодировать\".");
      return 1; // Возврат признака неверного выполнения.
    }

    // Кодируем или декодируем сообщение.
    for(int n=1; n < args.Length; n++) {
      for(int i=0; i < args[n].Length; i++) {
        if(args[0]=="закодировать")
          Console.Write((char) (args[n][i] + 1) );
        else
          Console.Write((char) (args[n][i] - 1) );
      }
      Console.Write(" ");
    }

    Console.WriteLine();

    return 0;
  }
}
```

Чтобы использовать эту программу, укажите после ее имени командное слово "закодировать" или "раскодировать", а затем фразу, подлежащую соответствующей операции. В предположении, что эта программа называется `Cipher`, приводим два примера ее выполнения.

```
D:\Cipher закодировать один два
пейо ерб

D:\Cipher раскодировать пейо ерб
один два
```

В этой программе есть два интересных момента. Во-первых, обратите внимание на то, как проверяется наличие аргументов командной строки. Это очень важный момент, который можно обобщить. Если работа программы опирается на один или несколько аргументов командной строки, всегда необходимо удостовериться в том, что эти аргументы действительно переданы программе. Отсутствие такой проверки может привести к сбою программы. Кроме того, поскольку первым аргументом командной строки должно быть слово "закодировать" или "раскодировать", то, прежде чем выполнять кодирование или раскодирование текста, необходимо убедиться в наличии этого ключевого слова.

Во-вторых, обратите внимание на то, как программа возвращает код своего завершения. Если командная строка не записана должным образом, возвращается значение 1, которое свидетельствует о нештатной ситуации и аварийном завершении программы. Возвращаемое значение, равное 0, — признак нормальной работы программы и благополучного ее завершения.

Рекурсия

В C# метод может вызвать сам себя. Этот процесс называется *рекурсией*, а метод, который вызывает себя, называют *рекурсивным*. В общем случае рекурсия — это процесс определения чего-либо с использованием самого себя. Ключевым компонентом

рекурсивного метода является обязательное включение им инструкции обращения к самоме себе. Рекурсия — это мощный механизм управления.

Классическим примером рекурсии является вычисление факториала числа. Факториал числа *N* представляет собой произведение целых чисел от 1 до *N*. Например, факториал числа 3 равен 1×2×3, или 6. Рекурсивный способ вычисления факториала числа демонстрируется в следующей программе. Для сравнения сюда же включен и его нерекурсивный эквивалент.

```
// Простой пример рекурсии.

using System;

class Factorial {
  // Это рекурсивный метод.
  public int factR(int n) {
    int result;

    if(n==1) return 1;
    result = factR(n-1) * n;
    return result;
  }

  // А это его итеративный эквивалент.
  public int factI(int n) {
    int t, result;

    result = 1;
    for(t=1; t <= n; t++) result *= t;
    return result;
  }
}

class Recursion {
  public static void Main() {
    Factorial f = new Factorial();

    Console.WriteLine(
                "Факториалы, вычисленные с " +
                "использованием рекурсивного метода.");
    Console.WriteLine("Факториал числа 3 равен " +
                      f.factR(3));
    Console.WriteLine("Факториал числа 4 равен " +
                      f.factR(4));
    Console.WriteLine("Факториал числа 5 равен " +
                      f.factR(5));
    Console.WriteLine();

    Console.WriteLine(
                "Факториалы, вычисленные с " +
                "использованием итеративного метода.");
    Console.WriteLine("Факториал числа 3 равен " +
                      f.factI(3));
    Console.WriteLine("Факториал числа 4 равен " +
                      f.factI(4));
    Console.WriteLine("Факториал числа 5 равен " +
                      f.factI(5));
  }
}
```

Вот результаты выполнения этой программы:

```
Факториалы, вычисленные с использованием рекурсивного метода.
Факториал числа 3 равен 6
Факториал числа 4 равен 24
Факториал числа 5 равен 120

Факториалы, вычисленные с использованием итеративного метода.
Факториал числа 3 равен 6
Факториал числа 4 равен 24
Факториал числа 5 равен 120
```

Нерекурсивный метод `factI()` довольно прост. В нем используется цикл, в котором организовано перемножение последовательных чисел, начиная с 1 (поэтому начальное значение управляющей переменной равно 1) и заканчивая числом, заданным в качестве параметра метода.

Рекурсивный метод `factR()` несколько сложнее. Если он вызывается с аргументом, равным 1, то сразу возвращает значение 1. В противном случае он возвращает произведение `factR(n-1) * n`. Для вычисления этого выражения вызывается метод `factR()` с аргументом `n-1`. Этот процесс повторяется до тех пор, пока аргумент не станет равным 1, после чего вызванные ранее методы начнут возвращать значения. Например, при вычислении факториала числа 2 первое обращение к методу `factR()` приведет ко второму обращению к тому же методу, но с аргументом, равным 1. Второй вызов метода `factR()` возвратит значение 1, которое будет умножено на 2 (исходное значение параметра `n`). Возможно, вам будет интересно вставить в метод `factR()` инструкции с вызовом метода `WriteLine()`, чтобы показать уровень каждого вызова и промежуточные результаты.

Когда метод вызывает сам себя, в системном стеке выделяется память для новых локальных переменных и параметров, и код метода с самого начала выполняется с новыми переменными. Рекурсивный вызов не создает новой копии метода. Новыми являются только аргументы. При возвращении каждого рекурсивного вызова из стека извлекаются старые локальные переменные и параметры, и выполнение метода возобновляется с "внутренней" точки вызова этого метода.

Рассмотрим еще один пример рекурсии. Метод `displayRev()` использует рекурсию для отображения его строкового аргумента в обратном порядке.

```
// Отображение строки в обратном порядке с помощью рекурсии.

using System;

class RevStr {

  // Отображение строки в обратном порядке.
  public void displayRev(string str) {
    if(str.Length > 0)
      displayRev(str.Substring(1, str.Length-1));
    else
      return;

    Console.Write(str[0]);
  }
}

class RevStrDemo {
  public static void Main() {
    string s = "Этот тест";
    RevStr rsOb = new RevStr();
```

```
    Console.WriteLine("Исходная строка: " + s);

    Console.Write("Перевернутая строка: ");
    rsOb.displayRev(s);

    Console.WriteLine();
  }
}
```

Вот результаты выполнения этой программы:

```
Исходная строка: Этот тест
Перевернутая строка: тсет тотЭ
```

Если при каждом вызове метода `displayRev()` проверка показывает, что длина строки `str` больше нуля, то выполняется рекурсивный вызов `displayRev()` с новым аргументом-строкой, которая состоит из предыдущей строки `str` без ее первого символа. Этот процесс повторяется до тех пор, пока тому же методу не будет передана строка нулевой длины. После этого вызванные ранее методы начнут возвращать значения, и каждый возврат будет сопровождаться довыполнением метода, т.е. отображением первого символа строки `str`. В результате исходная строка посимвольно отобразится в обратном порядке.

Рекурсивные версии многих процедур выполняются медленнее, чем их итеративные эквиваленты, из-за дополнительных затрат системных ресурсов, связанных с многократными вызовами методов. Слишком большое количество рекурсивных обращений к методу может вызвать переполнение стека. Поскольку локальные переменные и параметры сохраняются в системном стеке и каждый новый вызов создает новую копию переменных, может настать момент, когда память стека будет исчерпана. В этом случае C#-системой будет сгенерировано соответствующее исключение. Но если рекурсия построена корректно, об этом вряд ли стоит волноваться.

Основное достоинство рекурсии состоит в том, что некоторые типы алгоритмов рекурсивно реализуются проще, чем их итеративные эквиваленты. Например, алгоритм сортировки QuickSort довольно трудно реализовать итеративным способом. Кроме того, некоторые задачи просто созданы для рекурсивных решений.

При написании рекурсивных методов необходимо включить в них инструкцию проверки условия (например, `if`-инструкцию), которая бы заставила вернуться из метода без выполнения рекурсивного вызова. Если этого не будет сделано, то, вызвав однажды метод, из него уже нельзя будет вернуться. При работе с рекурсией это самый распространенный тип ошибки. Поэтому при ее разработке не стоит скупиться на инструкции вызова метода `WriteLine()`, чтобы быть в курсе того, что происходит в методе, и прервать его работу в случае обнаружения ошибки.

Использование модификатора типа `static`

Иногда требуется определить член класса, который должен использоваться независимо от объекта этого класса. Обычно к члену класса доступ предоставляется через объект этого класса. Однако можно создать член, который заведомо разрешено использовать сам по себе, т.е. без ссылки на конкретный экземпляр. Чтобы создать такой член, предварите его объявление ключевым словом `static`. Если член объявлен как `static`, к нему можно получить доступ до создания объектов этого класса и без ссылки на объект. С использованием ключевого слова `static` можно объявлять как методы, так и переменные. В качестве первого примера `static`-члена приведем метод `Main()`, который должен быть вызван операционной системой в начале работы программы.

При использовании static-члена вне класса необходимо указать имя класса и следующий за ним оператор "точка". Объект при этом не нужно создавать. К static-члену получают доступ не через экземпляр класса, а с помощью имени класса. Например, чтобы присвоить число 10 static-переменной с именем count, которая является членом класса Timer, используйте следующую строку кода:

```
Timer.count = 10;
```

Этот формат подобен тому, что используется для доступа к обычной переменной экземпляра через объект, но здесь вместо имени объекта необходимо указать имя класса. Аналогично можно вызвать и static-метод, т.е. с помощью оператора "точка" после имени класса.

Переменные, объявленные как static-члены, являются по сути глобальными переменными. При объявлении объектов класса копии static-переменной не создаются, причем все экземпляры класса совместно используют одну и ту же static-переменную. Инициализация static-переменной происходит при загрузке класса. Если инициализатор явно не указан, static-переменная, предназначенная для хранения числовых значений, инициализируется нулем; объектные ссылки — null-значениями, а переменные типа bool — значением false. Таким образом, static-переменная всегда имеет значение.

Различие между static- и обычным методом состоит в том, что static-метод можно вызвать посредством имени класса, без необходимости создания объекта этого класса. Выше вы уже имели возможность рассмотреть пример такого вызова, когда обращались к static-методу Sqrt(), принадлежащему классу System.Math.

Теперь рассмотрим пример создания static-переменной и static-метода.

```
// Использование модификатора типа static.

using System;

class StaticDemo {
  // Объявление статической переменной.
  public static int val = 100;

  // Объявление статического метода.
  public static int valDiv2() {
    return val/2;
  }
}

class SDemo {
  public static void Main() {

    Console.WriteLine(
      "Начальное значение переменной StaticDemo.val равно "
      + StaticDemo.val);

    StaticDemo.val = 8;
    Console.WriteLine(
            "Значение переменной StaticDemo.val равно " +
            StaticDemo.val);
    Console.WriteLine("StaticDemo.valDiv2(): " +
                      StaticDemo.valDiv2());
  }
}
```

При выполнении эта программа генерирует следующие результаты:

```
Начальное значение переменной StaticDemo.val равно 100
Значение переменной StaticDemo.val равно 8
StaticDemo.valDiv2(): 4
```

Как видно по результатам выполнения программы, static-переменная инициализируется в начале ее работы, т.е. еще до создания объекта класса, в котором она определяется.

На static-методы накладывается ряд ограничений.

1. static-метод не имеет ссылки this.
2. static-метод может напрямую вызывать только другие static-методы. Он не может напрямую вызывать метод экземпляра своего класса. Дело в том, что методы экземпляров работают с конкретными экземплярами класса, чего не скажешь о static-методах.
3. static-метод должен получать прямой доступ только к static-данным. Он не может напрямую использовать переменные экземпляров, поскольку не работает с экземплярами класса.

Например, в следующем классе static-метод valDivDenom() недопустим:

```
class StaticError {
  int denom = 3; // обычная переменная экземпляра
  static int val = 1024; // статическая переменная

  /* Ошибка! Внутри статического метода прямой доступ
     к нестатической переменной недопустим. */
  static int valDivDenom() {
    return val/denom; // Инструкция не скомпилируется!
  }
}
```

Здесь denom — обычная переменная экземпляра, к которой невозможно получить доступ внутри статического метода. Однако с использованием переменной val проблем нет, поскольку это static-переменная.

Аналогичная проблема возникает при попытке вызвать нестатический метод из static-метода того же класса. Вот пример:

```
using System;

class AnotherStaticError {
  // Нестатический метод.
  void nonStaticMeth() {
     Console.WriteLine("Внутри метода nonStaticMeth().");
  }

  /* Ошибка! Внутри статического метода нельзя напрямую
     вызвать нестатический метод. */
  static void staticMeth() {
    nonStaticMeth(); // Инструкция не скомпилируется!
  }
}
```

В этом случае попытка вызвать нестатический метод (т.е. метод экземпляра) из статического метода приведет к ошибке компиляции.

Важно понимать, что static-метод *может* вызывать методы экземпляров и получать доступ к переменным экземпляров своего класса, но должен делать это через объект класса. Другими словами, он не может использовать обычные члены класса без указания конкретного объекта. Например, этот фрагмент программы совершенно корректен:

```
class MyClass {
  // Нестатический метод.
  void nonStaticMeth() {
     Console.WriteLine("Внутри метода nonStaticMeth().");
  }
  /* Внутри статического метода можно вызвать
     нестатический метод, использовав ссылку на объект. */
  public static void staticMeth(MyClass ob) {
    ob.nonStaticMeth(); // Здесь все в порядке.
  }
}
```

Поскольку static-поля не зависят от конкретного объекта, они используются при обработке информации, применимой ко всему классу. Рассмотрим пример такой ситуации. В следующей программе используется static-поле для обработки счетчика числа существующих объектов.

```
// Использование static-поля для подсчта экземпляров класса.

using System;

class CountInst {
  static int count = 0;

  // Инкрементируем счетчик при создании объекта.
  public CountInst() {
    count++;
  }

  // Декрементируем счетчик при разрушении объекта.
  ~CountInst() {
    count--;
  }

  public static int getcount() {
    return count;
  }
}

class CountDemo {
  public static void Main() {
    CountInst ob;


    for(int i=0; i < 10; i++) {
      ob = new CountInst();
      Console.WriteLine("Текущее содержимое счетчика: "
                        + CountInst.getcount());
    }

  }
}
```

Результаты выполнения этой программы выглядят так:

```
Текущее содержимое счетчика: 1
Текущее содержимое счетчика: 2
Текущее содержимое счетчика: 3
Текущее содержимое счетчика: 4
Текущее содержимое счетчика: 5
```

```
Текущее содержимое счетчика: 6
Текущее содержимое счетчика: 7
Текущее содержимое счетчика: 8
Текущее содержимое счетчика: 9
Текущее содержимое счетчика: 10
```

Каждый раз, когда создается объект типа `CountInst`, `static`-поле `count` инкрементируется. И каждый раз, когда объект типа `CountInst` разрушается, `static`-поле `count` декрементируется. Таким образом, статическая переменная `count` всегда содержит количество объектов, существующих в данный момент. Это возможно только благодаря использованию статического поля. Переменная экземпляра не в состоянии справиться с такой задачей, поскольку подсчет экземпляров класса связан с классом в целом, а не с конкретным его экземпляром.

А вот еще один пример использования `static`-членов класса. Выше в этой главе было показано, как использовать "фабрику" класса для создания объектов. В том примере в качестве генератора объектов класса выступал нестатический метод, а это значит, что его можно вызывать только через объектную ссылку, т.е. требовалось создать объект класса лишь для того, чтобы получить возможность вызвать метод генератора объектов. Поэтому лучше реализовать "фабрику" класса, используя `static`-метод, который позволяет обращаться к нему, не создавая ненужного объекта. Ниже приводится пример реализации "фабрики" класса, переписанный с учетом этого усовершенствования.

```
// Создание статической "фабрики" класса.

using System;

class MyClass {
  int a, b;

  // Создаем "фабрику" для класса MyClass.
  static public MyClass factory(int i, int j) {
    MyClass t = new MyClass();

    t.a = i;
    t.b = j;

    return t; // Метод возвращает объект.
  }

  public void show() {
    Console.WriteLine("a и b: " + a + " " + b);
  }

}

class MakeObjects {
  public static void Main() {
    int i, j;

    // Генерируем объекты с помощью "фабрики" класса.
    for(i=0, j=10; i < 10; i++, j--) {
      MyClass ob = MyClass.factory(i, j); // Получение
                                          // объекта.
      ob.show();
    }
```

```
    Console.WriteLine();
  }
}
```

В этой версии программы метод `factory()` вызывается посредством указания имени класса:

```
MyClass ob = MyClass.factory(i, j); // Получение объекта.
```

Этот пример показывает, что нет необходимости создавать объект класса `MyClass` до использования "фабрики" класса.

Статические конструкторы

Конструктор класса также можно объявить статическим. Статический конструктор обычно используется для инициализации атрибутов, которые применяются к классу в целом, а не к конкретному его экземпляру. Таким образом, статический конструктор служит для инициализации аспектов класса до создания объектов этого класса. Рассмотрим простой пример.

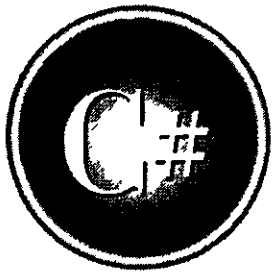
```
// Использование статического конструктора.

using System;

class Cons {
  public static int alpha;
  public int beta;

  // Статический конструктор.
  static Cons() {
    alpha = 99;
    Console.WriteLine("Внутри статического конструктора.");
  }

  // Конструктор экземпляра.
  public Cons() {
    beta = 100;
    Console.WriteLine("Внутри конструктора экземпляра.");
  }
}

class ConsDemo {
  public static void Main() {
    Cons ob = new Cons();

    Console.WriteLine("Cons.alpha: " + Cons.alpha);
    Console.WriteLine("ob.beta: " + ob.beta);

  }
}
```

Вот результаты выполнения этой программы:

```
Внутри статического конструктора.
Внутри конструктора экземпляра.
Cons.alpha: 99
ob.beta: 100
```

Обратите внимание на то, что статический конструктор вызывается автоматически, причем до вызова конструктора экземпляра. В общем случае `static`-конструктор будет выполнен до любого конструктора экземпляра. Кроме того, `static`-конструкторы должны быть закрытыми, и их не может вызвать ваша программа.

Полный справочник по



Глава 9

Перегрузка операторов

Язык C# позволяет определить значение оператора относительно создаваемого класса. Этот процесс называется *перегрузкой операторов*. Перегружая оператор, вы расширяете его использование для класса. Результат действия оператора полностью находится в ваших руках, и может быть разным при переходе от класса к классу. Например, класс, который определяет связный список, может использовать оператор "+" для добавления объектов в список. Класс, который реализует стек, может использовать оператор "+" для занесения объекта в стек. А какой-то другой класс может использовать этот оператор иным способом.

При перегрузке оператора ни одно из его исходных значений не теряется. Перегрузку оператора можно расценивать как введение новой операции для класса. Следовательно, перегрузка оператора "+", например, для обработки связного списка (в качестве оператора сложения) не изменяет его значение применительно к целым числам.

Главное достоинство перегрузки операторов состоит в том, что она позволяет бесшовно интегрировать новый тип класса со средой программирования. Эта *расширяемость типов* — важная составляющая мощи таких объектно-ориентированных языков программирования, как C#. Если для класса определены некоторые операторы, вы можете оперировать объектами этого класса, используя обычный C#-синтаксис выражений. Более того, вы можете использовать в выражениях объект, включающий другие типы данных. Перегрузка операторов — одно из самых мощных средств языка C#.

Основы перегрузки операторов

Перегрузка операторов тесно связана с перегрузкой методов. Для перегрузки операторов используется ключевое слово `operator`, позволяющее создать *операторный метод*, который определяет действие оператора, связанное с его классом.

Существует две формы методов `operator`: одна используется для унарных операторов, а другая — для бинарных. Общий же формат (для обоих случаев) таков:

```
// Общий формат перегрузки для унарного оператора.
public static тип_возврата operator op(
                              тип_параметра операнд)
{
  // операции
}

// Общий формат перегрузки для бинарного оператора.
public static тип_возврата operator op(
                              тип_параметра1 операнд1,
                              тип_параметра2 операнд2)
{
  // операции
}
```

Здесь элемент *op* — это оператор (например "+" или "/"), который перегружается. Элемент *тип_возврата* — это тип значения, возвращаемого при выполнении заданной операции. Несмотря на то что можно выбрать любой тип, тип возвращаемого значения чаще всего будет совпадать с типом класса, для которого этот оператор перегружается. Такая корреляция облегчает использование перегруженного оператора в выражениях. Для унарных операторов операнд передается в элементе *операнд*, а для бинарных — в элементах *операнд1* и *операнд2*.

Для унарных операторов тип операнда должен совпадать с классом, для которого определен оператор. Что касается бинарных операторов, то тип хотя бы одного опе-

ранда должен совпадать с соответствующим классом. Таким образом, C#-операторы нельзя перегружать для классов, не созданных вами. Например, вы не можете перегрузить оператор "+" для типов int или string.

И последнее: параметры операторов не должны использовать модификатор ref или out.

Перегрузка бинарных операторов

Чтобы разобраться, как работает перегрузка операторов, начнем с примера, в котором перегружаются два бинарных оператора — "+" и "-". В следующей программе создается класс ThreeD поддержки координат объекта в трехмерном пространстве. Перегруженный оператор "+" выполняет сложение отдельных координат двух ThreeD-объектов, а перегруженный оператор "-" вычитает координаты одного ThreeD-объекта из координат другого.

```
// Пример перегрузки операторов.

using System;

// Класс трехмерных координат.
class ThreeD {
  int x, y, z; // 3-х-мерные координаты.

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) {
                                x = i; y = j; z = k; }

  // Перегрузка бинарного оператора "+".
  public static ThreeD operator +(ThreeD op1,
                                  ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Суммирование координат двух точек
       и возврат результата. */
    result.x = op1.x + op2.x; // Эти операторы выполняют
    result.y = op1.y + op2.y; // целочисленное сложение.
    result.z = op1.z + op2.z;

    return result;
  }

  // Перегрузка бинарного оператора "-".
  public static ThreeD operator -(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Обратите внимание на порядок операндов.
       op1 - левый операнд, op2 - правый. */
    result.x = op1.x - op2.x; // Эти операторы выполняют
    result.y = op1.y - op2.y; // целочисленное вычитание.
    result.z = op1.z - op2.z;

    return result;
  }

  // Отображаем координаты X, Y, Z.
```

```
  public void show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  public static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();

    Console.Write("Координаты точки a: ");
    a.show();
    Console.WriteLine();
    Console.Write("Координаты точки b: ");
    b.show();
    Console.WriteLine();

    c = a + b; // Складываем a и b.
    Console.Write("Результат сложения a + b: ");
    c.show();
    Console.WriteLine();

    c = a + b + c; // Складываем a, b и c.
    Console.Write("Результат сложения a + b + c: ");
    c.show();
    Console.WriteLine();

    c = c - a; // Вычитаем a из c.
    Console.Write("Результат вычитания c - a: ");
    c.show();
    Console.WriteLine();

    c = c - b; // Вычитаем b из c.
    Console.Write("Результат вычитания c - b: ");
    c.show();
    Console.WriteLine();
  }
}
```

При выполнении эта программа генерирует следующие результаты:

```
Координаты точки a: 1, 2, 3

Координаты точки b: 10, 10, 10

Результат сложения a + b: 11, 12, 13

Результат сложения a + b + c: 22, 24, 26

Результат вычитания c - a: 21, 22, 23

Результат вычитания c - b: 11, 12, 13
```

Эту программу стоит рассмотреть подробнее. Начнем с перегруженного оператора "+". При воздействии оператора "+" на два объекта типа `ThreeD` величины соответствующих координат суммируются, как показано в методе `operator+()`. Однако заметьте, что этот метод не модифицирует значения ни одного из операндов. Этот ме-

тод возвращает новый объект типа ThreeD, который содержит результат выполнения рассматриваемой операции. Это происходит и в случае стандартного арифметического оператора сложения "+", примененного, например, к числам 10 и 12. Результат операции 10+12 равен 22, но при его получении ни 10, ни 12 не были изменены. Хотя не существует правила, которое бы не позволяло перегруженному оператору изменять значение одного из его операндов, все же лучше, чтобы он не противоречил общепринятым нормам.

Обратите внимание на то, что метод operator+() возвращает объект типа ThreeD. Несмотря на то что он мог бы возвращать значение любого допустимого в C# типа, тот факт, что он возвращает объект типа ThreeD, позволяет использовать оператор "+" в таких составных выражениях, как a+b+c. Здесь часть этого выражения, a+b, генерирует результат типа ThreeD, который затем суммируется с объектом c. И если бы выражение генерировало значение иного типа (а не типа ThreeD), такое составное выражение попросту не работало бы.

И еще один важный момент. При сложении координат внутри метода operator+() выполняется целочисленное сложение, поскольку отдельные координаты представляют собой целочисленные величины. Факт перегрузки оператора "+" для объектов типа ThreeD не влияет на оператор "+", применяемый к целым числам.

Теперь рассмотрим операторный метод operator-(). Оператор "-" работает подобно оператору "+" за исключением того, что здесь важен порядок следования операндов. Вспомните, что сложение коммутативно, а вычитание — нет (т.е. A–B не то же самое, что B–A). Для всех бинарных операторов первый параметр операторного метода будет содержать левый операнд, а второй параметр — правый. При реализации перегруженных версий некоммутативных операторов необходимо помнить, какой операнд является левым, а какой — правым.

Перегрузка унарных операторов

Унарные операторы перегружаются точно так же, как и унарные. Главное отличие, конечно же, состоит в том, что в этом случае существует только один операнд. Рассмотрим, например, метод, который перегружает унарный "минус" для класса ThreeD.

```
// Перегрузка унарного оператора "-".
  public static ThreeD operator -(ThreeD op)
  {
    ThreeD result = new ThreeD();

    result.x = -op.x;
    result.y = -op.y;
    result.z = -op.z;

    return result;
  }
```

Здесь создается новый объект, который содержит поля операнда, но со знаком "минус". Созданный таким образом объект и возвращается операторным методом operator-(). Обратите внимание на то, что сам операнд остается немодифицированным. Такое поведение соответствует обычному действию унарного "минуса". Например, в выражении

```
a = -b
```

a получает значение b, взятое с противоположным знаком, но само b при этом не меняется.

Однако в двух случаях операторный метод изменяет содержимое операнда. Речь идет об операторах инкремента (++) и декремента (--). Поскольку обычно ("в миру") эти операторы выполняют функции инкрементирования и декрементирования значений, соответственно, то перегруженные операторы "+" и "-", как правило, инкрементируют свой операнд. Таким образом, при перегрузке этих операторов операнд обычно модифицируется. Например, рассмотрим метод `operator++()` для класса `ThreeD`.

```
// Перегрузка унарного оператора "++".
  public static ThreeD operator ++(ThreeD op)
  {
    // Оператор "++" модифицирует аргумент.
    op.x++;
    op.y++;
    op.z++;

    return op;
  }
```

Обратите внимание: в результате выполнения этого операторного метода объект, на который ссылается операнд `op`, модифицируется. Итак, операнд, подвергнутый операции "++", инкрементируется. Более того, модифицированный объект возвращается этим методом, благодаря чему оператор "++" можно использовать в более сложных выражениях.

Ниже приведена расширенная версия предыдущего примера программы, которая, помимо прочего, демонстрирует определение и использование унарных операторов "-" и "++".

```
// Перегрузка большего числа операторов.

using System;

// Класс трехмерных координат.
class ThreeD {
  int x, y, z; // 3-х-мерные координаты.

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Перегрузка бинарного оператора "+".
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Суммирование координат двух точек
       и возврат результата. */
    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
  }

  // Перегрузка бинарного оператора "-".
  public static ThreeD operator -(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Обратите внимание на порядок операндов.
```

```
      op1 - левый операнд, op2 - правый. */
    result.x = op1.x - op2.x;
    result.y = op1.y - op2.y;
    result.z = op1.z - op2.z;

    return result;
  }

  // Перегрузка унарного оператора "-".
  public static ThreeD operator -(ThreeD op)
  {
    ThreeD result = new ThreeD();

    result.x = -op.x;
    result.y = -op.y;
    result.z = -op.z;

    return result;
  }

  // Перегрузка унарного оператора "++".
  public static ThreeD operator ++(ThreeD op)
  {
    // Оператор "++" модифицирует аргумент.
    op.x++;
    op.y++;
    op.z++;

    return op;
  }

  // Отображаем координаты X, Y, Z.
  public void show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  public static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();

    Console.Write("Координаты точки a: ");
    a.show();
    Console.WriteLine();
    Console.Write("Координаты точки b: ");
    b.show();
    Console.WriteLine();

    c = a + b; // Сложение a и b.
    Console.Write("Результат сложения a + b: ");
    c.show();
    Console.WriteLine();

    c = a + b + c; // Сложение a, b и c.
    Console.Write("Результат сложения a + b + c: ");
```

```
    c.show();
    Console.WriteLine();

    c = c - a; // Вычитание a из c.
    Console.Write("Результат вычитания c - a: ");
    c.show();
    Console.WriteLine();

    c = c - b; // Вычитание b из c.
    Console.Write("Результат вычитания c - b: ");
    c.show();
    Console.WriteLine();

    c = -a; // Присваивание -a объекту c.
    Console.Write("Результат присваивания -a: ");
    c.show();
    Console.WriteLine();

    a++; // Инкрементирование a.
    Console.Write("Результат инкрементирования a++: ");
    a.show();
  }
}
```

Результаты выполнения этой программы выглядят так:

```
Координаты точки a: 1, 2, 3

Координаты точки b: 10, 10, 10

Результат сложения a + b: 11, 12, 13

Результат сложения a + b + c: 22, 24, 26

Результат вычитания c - a: 21, 22, 23

Результат вычитания c - b: 11, 12, 13

Результат присваивания -a: -1, -2, -3

Результат инкрементирования a++: 2, 3, 4
```

Как вы уже знаете, операторы "++" и "--" имеют как префиксную, так и постфиксную форму. Например, инструкции

```
++a;
```

и

```
a++;
```

представляют собой допустимое использование оператора инкремента. Однако при перегрузке оператора "++" обе формы вызывают один и тот же метод. Следовательно, в этом случае невозможно отличить префиксную форму оператора "++" от постфиксной. Это касается и перегрузки оператора "--".

Выполнение операций над значениями встроенных C#-типов

Для любого заданного класса и оператора любой операторный метод сам может перегружаться. Одна из самых распространенных причин этого — разрешить операции между объектами этого класса и другими (встроенными) типами данных. В качестве примера давайте снова возьмем класс `ThreeD`. Вы видели, как перегрузить оператор "+", чтобы он суммировал координаты одного `ThreeD`-объекта с координатами другого. Однако это не единственный способ определения операции сложения для класса `ThreeD`. Например, может потребоваться суммирование какого-либо целого числа с каждой координатой `ThreeD`-объекта. Ведь тогда эту операцию можно использовать для смещения осей. Для ее реализации необходимо перегрузить оператор "+" еще раз, например, так:

```
// Перегружаем бинарный оператор "+" для суммирования
// объекта и int-значения.
  public static ThreeD operator +(ThreeD op1, int op2)
  {
    ThreeD result = new ThreeD();

    result.x = op1.x + op2;
    result.y = op1.y + op2;
    result.z = op1.z + op2;

    return result;
  }
```

Обратите внимание на то, что второй параметр имеет тип `int`. Таким образом, этот метод позволяет сложить `int`-значение с каждым полем `ThreeD`-объекта. Это вполне допустимо, поскольку, как разъяснялось выше, при перегрузке бинарного оператора тип только одного из его операндов должен совпадать с типом класса, для которого перегружается этот оператор. Другой операнд может иметь любой тип.

Ниже приведена версия класса `ThreeD`, которая имеет два перегруженных метода `operator+()`.

```
/* Перегрузка оператора сложения для вариантов:
   объект + объект и объект + int-значение. */

using System;

// Класс трехмерных координат.
class ThreeD {
  int x, y, z; // 3-х-мерные координаты.

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) {
                                x = i; y = j; z = k; }

  // Перегружаем бинарный оператор "+" для варианта
  // "объект + объект".
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Суммирование координат двух точек
       и возврат результата. */
```

```
    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
  }

  // Перегружаем бинарный оператор "+" для варианта
  // "объект + int-значение".
  public static ThreeD operator +(ThreeD op1, int op2)
  {
    ThreeD result = new ThreeD();

    result.x = op1.x + op2;
    result.y = op1.y + op2;
    result.z = op1.z + op2;

    return result;
  }

  // Отображаем координаты X, Y, Z.
  public void show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  public static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();

    Console.Write("Координаты точки a: ");
    a.show();
    Console.WriteLine();
    Console.Write("Координаты точки b: ");
    b.show();
    Console.WriteLine();

    c = a + b; // объект + объект
    Console.Write("Результат сложения a + b: ");
    c.show();
    Console.WriteLine();

    c = b + 10; // объект + int-значение
    Console.Write("Результат сложения b + 10: ");
    c.show();
  }
}
```

При выполнении программа генерирует следующие результаты:

```
Координаты точки a: 1, 2, 3

Координаты точки b: 10, 10, 10

Результат сложения a + b: 11, 12, 13

Результат сложения b + 10: 20, 20, 20
```

Как подтверждают результаты выполнения этой программы, если оператор "+" применяется к двум объектам, их соответствующие координаты суммируются. А если оператор "+" применяется к объекту и целому числу, то значения координат объекта увеличиваются на это целое число.

Несмотря на то что приведенный выше способ перегрузки оператора "+" существенным образом расширяет возможности класса `ThreeD`, работа на этом еще не окончена. И вот почему. Метод `operator+(ThreeD, int)` позволяет выполнять инструкции, подобные следующей.

```
ob1 = ob2 + 10;
```

Но, к сожалению, он не позволяет выполнять инструкции такого рода:

```
ob1 = 10 + ob2;
```

Дело в том, что целочисленное значение принимается в качестве второго аргумента, которым является правый операнд. А в предыдущей инструкции целочисленный аргумент находится слева. Чтобы сделать допустимыми две формы инструкций, необходимо перегрузить оператор "+" еще раз. Новая версия должна будет в качестве первого параметра принимать значение типа `int`, а в качестве второго — объект типа `ThreeD`. И тогда старая версия метода `operator+()` будет обрабатывать вариант "*объект* + int-*значение*", а новая — вариант "int-*значение* + *объект*". Перегрузка оператора "+" (или любого другого бинарного оператора), выполненная подобным образом, позволит значению встроенного типа находиться слева *или* справа от оператора. Ниже приводится версия класса `ThreeD`, которая перегружает оператор "+" с учетом описанных выше вариантов приема аргументов.

```
/* Перегрузка оператора "+" для следующих вариантов:
   объект + объект,
   объект + int-значение и
   int-значение + объект. */

using System;

// Класс трехмерных координат.
class ThreeD {
  int x, y, z; // 3-х-мерные координаты.

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) {
                                  x = i; y = j; z = k; }

  // Перегружаем бинарный оператор "+" для варианта
  // "объект + объект".
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Суммирование координат двух точек
       и возврат результата. */
    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
  }

  // Перегружаем бинарный оператор "+" для варианта
  // "объект + int-значение".
```

```
  public static ThreeD operator +(ThreeD op1, int op2)
  {
    ThreeD result = new ThreeD();

    result.x = op1.x + op2;
    result.y = op1.y + op2;
    result.z = op1.z + op2;

    return result;
  }

  // Перегружаем бинарный оператор "+" для варианта
  // "int-значение + объект".
  public static ThreeD operator +(int op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    result.x = op2.x + op1;
    result.y = op2.y + op1;
    result.z = op2.z + op1;

    return result;
  }

  // Отображаем координаты X, Y, Z.
  public void show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  public static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();

    Console.Write("Координаты точки a: ");
    a.show();
    Console.WriteLine();
    Console.Write("Координаты точки b: ");
    b.show();
    Console.WriteLine();

    c = a + b; // объект + объект
    Console.Write("Результат сложения a + b: ");
    c.show();
    Console.WriteLine();

    c = b + 10; // объект + int-значение
    Console.Write("Результат сложения b + 10: ");
    c.show();
    Console.WriteLine();

    c = 15 + b; // int-значение + объект
    Console.Write("Результат сложения 15 + b: ");
    c.show();
  }
}
```

Вот результаты выполнения этой программы:

```
Координаты точки a: 1, 2, 3

Координаты точки b: 10, 10, 10

Результат сложения a + b: 11, 12, 13

Результат сложения b + 10: 20, 20, 20

Результат сложения 15 + b: 25, 25, 25
```

Перегрузка операторов отношений

Операторы отношений (например, "==" или "<") также можно перегружать, причем сделать это совсем нетрудно. Как правило, перегруженный оператор отношения возвращает одно из двух возможных значений: `true` или `false`. Это позволяет использовать перегруженные операторы отношений в условных выражениях. Если бы они возвращали результат другого типа, это бы весьма ограничило круг их применения.

Рассмотрим версию класса `ThreeD`, который перегружает операторы "<" и ">".

```
// Перегрузка операторов "<" и ">".

using System;

// Класс трехмерных координат.
class ThreeD {
  int x, y, z; // 3-х-мерные координаты.

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) {
                                  x = i; y = j; z = k; }

  // Перегрузка оператора "<".
  public static bool operator <(ThreeD op1, ThreeD op2)
  {
    if((op1.x < op2.x) && (op1.y < op2.y) &&
                          (op1.z < op2.z))
      return true;
    else
      return false;
  }

  // Перегрузка оператора ">".
  public static bool operator >(ThreeD op1, ThreeD op2)
  {
    if((op1.x > op2.x) && (op1.y > op2.y) &&
                          (op1.z > op2.z))
      return true;
    else
      return false;
  }

  // Отображаем координаты X, Y, Z.
  public void show()
  {
```

```
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  public static void Main() {
    ThreeD a = new ThreeD(5, 6, 7);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD(1, 2, 3);

    Console.Write("Координаты точки a: ");
    a.show();
    Console.Write("Координаты точки b: ");
    b.show();
    Console.Write("Координаты точки c: ");
    c.show();
    Console.WriteLine();

    if(a > c) Console.WriteLine("a > c - ИСТИНА");
    if(a < c) Console.WriteLine("a < c - ИСТИНА");
    if(a > b) Console.WriteLine("a > b - ИСТИНА");
    if(a < b) Console.WriteLine("a < b - ИСТИНА");
  }
}
```

При выполнении эта программа генерирует такие результаты:

```
Координаты точки a: 5, 6, 7
Координаты точки b: 10, 10, 10
Координаты точки c: 1, 2, 3

a > c - ИСТИНА
a < b - ИСТИНА
```

На перегрузку операторов отношений налагается серьезное ограничение: их следует перегружать парами. Например, перегружая оператор "<", вы также должны перегрузить оператор ">", и наоборот. Вот что подразумевается под парами операторов отношений:

```
==      !=
<       >
<=      >=
```

Перегружая операторы "==" и "!=", следует перегрузить также методы `Object.Equals()` и `Object.GetHashCode()`. Эти методы (а также их перегрузка) рассматриваются в главе 11.

Перегрузка операторов `true` и `false`

Ключевые слова `true` и `false` в целях перегрузки также можно использовать в качестве унарных операторов. Перегруженные версии этих операторов обеспечивают специфическое определение понятий ИСТИНА и ЛОЖЬ в отношении создаваемых программистом классов. Если для класса реализовать таким образом ключевые слова `true` и `false`, то затем объекты этого класса можно использовать для управления инструкциями `if`, `while`, `for` и `do-while`, а также в `?`-выражении. Их можно даже использовать для реализации специальных типов логики (например, нечеткой логики).

Операторы `true` и `false` должны быть перегружены в паре. Нельзя перегружать только один из них. Оба они выполняют функцию унарных операторов и имеют такой формат:

```
public static bool operator true(тип_параметра op)
{
  // Возврат значения true или false.
}

public static bool operator false(тип_параметра op)
{
  // Возврат значения true или false.
}
```

Обратите внимание на то, что каждая форма возвращает результат типа `bool`.

В следующем примере демонстрируется один из возможных способов реализации операторов `true` и `false` для класса `ThreeD`. Предполагается, что `ThreeD`-объект истинен, если по крайней мере одна его координата не равна нулю. Если все три координаты равны нулю, объект считается ложным. В целях демонстрации здесь также реализован оператор декремента.

```
// Перегрузка операторов true и false для класса ThreeD.

using System;

// Класс трехмерных координат.
class ThreeD {
  int x, y, z; // 3-х-мерные координаты.

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) {
                                    x = i; y = j; z = k; }

  // Перегружаем оператор true.
  public static bool operator true(ThreeD op) {
    if((op.x != 0) || (op.y != 0) || (op.z != 0))
      return true; // Хотя бы одна координата не равна 0.
    else
      return false;
  }

  // Перегружаем оператор false.
  public static bool operator false(ThreeD op) {
    if((op.x == 0) && (op.y == 0) && (op.z == 0))
      return true; // Все координаты равны нулю.
    else
      return false;
  }

  // Перегружаем унарный оператор "--".
  public static ThreeD operator --(ThreeD op)
  {
    op.x--;
    op.y--;
    op.z--;

    return op;
  }

  // Отображаем координаты X, Y, Z.
```

```
  public void show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class TrueFalseDemo {
  public static void Main() {
    ThreeD a = new ThreeD(5, 6, 7);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD(0, 0, 0);

    Console.Write("Координаты точки a: ");
    a.show();
    Console.Write("Координаты точки b: ");
    b.show();
    Console.Write("Координаты точки c: ");
    c.show();
    Console.WriteLine();

    if(a) Console.WriteLine("a - это ИСТИНА.");
    else Console.WriteLine("a - это ЛОЖЬ.");

    if(b) Console.WriteLine("b - это ИСТИНА.");
    else Console.WriteLine("b - это ЛОЖЬ.");

    if(c) Console.WriteLine("c - это ИСТИНА.");
    else Console.WriteLine("c - это ЛОЖЬ.");

    Console.WriteLine();

    Console.WriteLine(
      "Управляем циклом, используя объект класса ThreeD.");
    do {
      b.show();
      b--;
    } while(b);

  }
}
```

Вот какие результаты генерирует эта программа:

```
Координаты точки a: 5, 6, 7
Координаты точки b: 10, 10, 10
Координаты точки c: 0, 0, 0

a - это ИСТИНА.
b - это ИСТИНА.
c - это ЛОЖЬ.

Управляем циклом, используя объект класса ThreeD.
10, 10, 10
9, 9, 9
8, 8, 8
7, 7, 7
6, 6, 6
5, 5, 5
4, 4, 4
3, 3, 3
2, 2, 2
1, 1, 1
```

Обратите внимание на то, что объекты класса ThreeD используются для управления if-инструкциями и while-цикла. Что касается if-инструкций, то ThreeD-объект оценивается с помощью ключевого слова true. В случае истинности результата этой операции выполняется соответствующая инструкция. В случае do-while-цикла каждая его итерация декрементирует значение объект b. Цикл выполняется до тех пор, пока значение объекта b оценивается как ИСТИНА (т.е. содержит по крайней мере одну ненулевую координату). Когда все координаты объекта b станут равными нулю, он (объект) будет считаться ложным (благодаря оператору true), и цикл прекратится.

Перегрузка логических операторов

Как вы знаете, в C# определены следующие логические операторы: &, |, !, && и ||. Безусловно, перегруженными могут быть только &, |, !. Однако при соблюдении определенных правил можно использовать и операторы && и ||, действующие по сокращенной схеме.

Простой случай перегрузки логических операторов

Начнем с рассмотрения простейшей ситуации. Если вы не планируете использовать логические операторы, работающие по сокращенной схеме, то можете перегружать операторы & и | по своему усмотрению, причем каждый вариант должен возвращать результат типа bool. Перегруженный оператор ! также, как правило, возвращает результат типа bool.

Рассмотрим пример перегрузки логических операторов &, |, ! для объектов типа ThreeD. Как и прежде, в каждом из них предполагается, что ThreeD-объект является истинным, если хотя бы одна его координата не равна нулю. Если же все три координаты равны нулю, объект считается ложным.

```
// Простой способ перегрузки операторов !, | и &
// для класса ThreeD.

using System;

// Класс трехмерных координат.
class ThreeD {
  int x, y, z; // 3-х-мерные координаты.

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) {
                                  x = i; y = j; z = k; }


  // Перегрузка оператора "|".
  public static bool operator |(ThreeD op1, ThreeD op2)
  {
    if( ((op1.x != 0) || (op1.y != 0) || (op1.z != 0)) |
        ((op2.x != 0) || (op2.y != 0) || (op2.z != 0)) )
      return true;
    else
      return false;
  }

  // Перегрузка оператора "&".
  public static bool operator &(ThreeD op1, ThreeD op2)
```

```
  {
    if( ((op1.x != 0) && (op1.y != 0) && (op1.z != 0)) &
       ((op2.x != 0) && (op2.y != 0) && (op2.z != 0)) )
      return true;
    else
      return false;
  }

  // Перегрузка оператора "!".
  public static bool operator !(ThreeD op)
  {
    if((op.x != 0) || (op.y != 0) || (op.z != 0))
      return false;
    else return true;
  }

  // Отображем координаты X, Y, Z.
  public void show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class TrueFalseDemo {
  public static void Main() {
    ThreeD a = new ThreeD(5, 6, 7);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD(0, 0, 0);

    Console.Write("Координаты точки a: ");
    a.show();
    Console.Write("Координаты точки b: ");
    b.show();
    Console.Write("Координаты точки c: ");
    c.show();
    Console.WriteLine();

    if(!a) Console.WriteLine("a - ЛОЖЬ.");
    if(!b) Console.WriteLine("b - ЛОЖЬ.");
    if(!c) Console.WriteLine("c - ЛОЖЬ.");

    Console.WriteLine();

    if(a & b) Console.WriteLine("a & b - ИСТИНА.");
    else Console.WriteLine("a & b - ЛОЖЬ.");

    if(a & c) Console.WriteLine("a & c - ИСТИНА.");
    else Console.WriteLine("a & c - ЛОЖЬ.");

    if(a | b) Console.WriteLine("a | b - ИСТИНА.");
    else Console.WriteLine("a | b - ЛОЖЬ.");

    if(a | c) Console.WriteLine("a | c - ИСТИНА.");
    else Console.WriteLine("a | c - ЛОЖЬ.");
  }
}
```

При выполнении эта программа генерирует следующие результаты:

```
Координаты точки a: 5, 6, 7
Координаты точки b: 10, 10, 10
```

```
Координаты точки c: 0, 0, 0

c - ЛОЖЬ.

a & b - ИСТИНА.
a & c - ЛОЖЬ.
a | b - ИСТИНА.
a | c - ИСТИНА.
```

В этом примере методы `operator |()`, `operator &()` и `operator !()` возвращают результат типа `bool`. Это необходимо в том случае, если перечисленные операторы должны использоваться в своем обычном "амплуа" (т.е. там, где ожидается результат типа `bool`). Вспомните, что для всех встроенных типов результат выполнения логической операции представляет собой значение типа `bool`. Таким образом, вполне логично, что перегруженные версии этих операторов должны возвращать значение типа `bool`. К сожалению, такая логика работает в случае, если нет необходимости в операторах, работающих по сокращенной схеме вычислений.

Включение операторов, действующих по сокращенной схеме вычислений

Чтобы иметь возможность использовать операторы `&&` и `||`, действующие по сокращенной схеме вычислений, необходимо соблюдать четыре правила. Во-первых, класс должен перегружать операторы `&` и `|`. Во-вторых, `&`- и `|`-методы должны возвращать объект класса, для которого перегружаются эти операторы. В-третьих, каждый параметр должен представлять собой ссылку на объект класса, для которого перегружаются эти операторы. В-четвертых, тот же класс должен перегружать операторы `true` и `false`. При соблюдении всех этих условий операторы сокращенной схемы действия автоматически становятся доступными для применения.

В следующей программе показано, как реализовать операторы `&` и `|` для класса `ThreeD`, чтобы можно было использовать операторы `&&` и `||`, действующие по сокращенной схеме вычислений.

```
/* Более удачный способ реализации операторов !, | и &
   для класса ThreeD. Эта версия автоматически делает
   работоспособными операторы && и ||. */

using System;

// Класс трехмерных координат.
class ThreeD {
  int x, y, z; // 3-х-мерные координаты.

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }


  // Перегружаем оператор "|" для вычислений по
  // сокращенной схеме.
  public static ThreeD operator |(ThreeD op1, ThreeD op2)
  {
    if( ((op1.x != 0) || (op1.y != 0) || (op1.z != 0)) |
        ((op2.x != 0) || (op2.y != 0) || (op2.z != 0)) )
      return new ThreeD(1, 1, 1);
    else
      return new ThreeD(0, 0, 0);
```

```
  }

// Перегружаем оператор "&" для вычислений по
// сокращенной схеме.
  public static ThreeD operator &(ThreeD op1, ThreeD op2)
  {
    if( ((op1.x != 0) && (op1.y != 0) && (op1.z != 0)) &
        ((op2.x != 0) && (op2.y != 0) && (op2.z != 0)) )
      return new ThreeD(1, 1, 1);
    else
      return new ThreeD(0, 0, 0);
  }

  // Перегружаем оператор "!".
  public static bool operator !(ThreeD op)
  {
    if(op) return false;
    else return true;
  }

  // Перегружаем оператор true.
  public static bool operator true(ThreeD op) {
    if((op.x != 0) || (op.y != 0) || (op.z != 0))
      return true; // Хотя бы одна координата не
                   // равна нулю.
    else
      return false;
  }

  // Перегружаем оператор false.
  public static bool operator false(ThreeD op) {
    if((op.x == 0) && (op.y == 0) && (op.z == 0))
      return true; // Все координаты равны нулю.
    else
      return false;
  }

  // Отображаем координаты X, Y, Z.
  public void show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class TrueFalseDemo {
  public static void Main() {
    ThreeD a = new ThreeD(5, 6, 7);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD(0, 0, 0);

    Console.Write("Координаты точки a: ");
    a.show();
    Console.Write("Координаты точки b: ");
    b.show();
    Console.Write("Координаты точки c: ");
    c.show();
    Console.WriteLine();
```

```
    if(a) Console.WriteLine("a - ИСТИНА.");
    if(b) Console.WriteLine("b - ИСТИНА.");
    if(c) Console.WriteLine("c - ИСТИНА.");

    if(!a) Console.WriteLine("a - ЛОЖЬ.");
    if(!b) Console.WriteLine("b - ЛОЖЬ.");
    if(!c) Console.WriteLine("c - ЛОЖЬ.");

    Console.WriteLine();

    Console.WriteLine("Используем операторы & и |");
    if(a & b) Console.WriteLine("a & b - ИСТИНА.");
    else Console.WriteLine("a & b - ЛОЖЬ.");

    if(a & c) Console.WriteLine("a & c - ИСТИНА.");
    else Console.WriteLine("a & c - ЛОЖЬ.");

    if(a | b) Console.WriteLine("a | b - ИСТИНА.");
    else Console.WriteLine("a | b - ЛОЖЬ.");

    if(a | c) Console.WriteLine("a | c - ИСТИНА.");
    else Console.WriteLine("a | c - ЛОЖЬ.");

    Console.WriteLine();

    // Теперь используем операторы && и ||, действующие
    // по сокращенной схеме вычислений.
    Console.WriteLine(
         "Используем "сокращенные" операторы && и ||");
    if(a && b) Console.WriteLine("a && b - ИСТИНА.");
    else Console.WriteLine("a && b - ЛОЖЬ.");

    if(a && c) Console.WriteLine("a && c - ИСТИНА.");
    else Console.WriteLine("a && c - ЛОЖЬ.");

    if(a || b) Console.WriteLine("a || b - ИСТИНА.");
    else Console.WriteLine("a || b - ЛОЖЬ.");

    if(a || c) Console.WriteLine("a || c - ИСТИНА.");
    else Console.WriteLine("a || c - ЛОЖЬ.");
  }
}
```

Эта программа при выполнении генерирует такие результаты:

```
Координаты точки a: 5, 6, 7
Координаты точки b: 10, 10, 10
Координаты точки c: 0, 0, 0

a - ИСТИНА.
b - ИСТИНА.
c - ЛОЖЬ.

Используем операторы & и |
a & b - ИСТИНА.
a & c - ЛОЖЬ.
a | b - ИСТИНА.
a | c - ИСТИНА.
```

```
Используем "сокращенные" операторы && и ||
a && b - ИСТИНА.
a && c - ЛОЖЬ.
a || b - ИСТИНА.
a || c - ИСТИНА.
```

Теперь остановимся подробнее на реализации операторов `&` и `|`. Для удобства приведем здесь их операторные методы.

```
// Перегружаем оператор "|" для вычислений по
// сокращенной схеме.
public static ThreeD operator |(ThreeD op1, ThreeD op2)
{
  if( ((op1.x != 0) || (op1.y != 0) || (op1.z != 0)) |
      ((op2.x != 0) || (op2.y != 0) || (op2.z != 0)) )
    return new ThreeD(1, 1, 1);
  else
    return new ThreeD(0, 0, 0);
}

// Перегружаем оператор "&" для вычислений по
// сокращенной схеме.
public static ThreeD operator &(ThreeD op1, ThreeD op2)
{
  if( ((op1.x != 0) && (op1.y != 0) && (op1.z != 0)) &
      ((op2.x != 0) && (op2.y != 0) && (op2.z != 0)) )
    return new ThreeD(1, 1, 1);
  else
    return new ThreeD(0, 0, 0);
}
```

Обратите внимание на то, что оба операторных метода сейчас возвращают объект типа `ThreeD`, а также на то, как генерируется этот объект. Если результатом операции оказывается значение ИСТИНА, то создается и возвращается истинный `ThreeD`-объект (т.е. объект, в котором не равна нулю хотя бы одна координата). Если же результатом операции оказывается значение ЛОЖЬ, то создается и возвращается ложный `ThreeD`-объект (т.е. объект, в котором равны нулю все координаты). Таким образом, в инструкции

```
if(a & b) Console.WriteLine("a & b - ИСТИНА.");
else Console.WriteLine("a & b - ЛОЖЬ.");
```

результатом операции `a & b` является `ThreeD`-объект, который в данном случае оказывается истинным. Поскольку в классе `ThreeD` определены операторы `true` и `false`, результирующий объект подвергается воздействию оператора `true`, вследствие чего возвращается результат типа `bool`. В данном случае результат равен значению `true`, и поэтому инструкция выводит сообщение `"a & b - ИСТИНА."`.

Поскольку в этом примере программы все необходимые правила соблюдены, для объектов класса `ThreeD` теперь доступны логические операторы сокращенного действия. Их работа заключается в следующем. Первый операнд тестируется с помощью операторного метода `operator true` (для оператора "`||`") или операторного метода `operator false` (для оператора "`&&`"). Если этот тест в состоянии определить результат всего выражения, то оставшиеся `&`- или `|`-операции уже не выполняются. В противном случае для определения результата используется соответствующий перегруженный оператор "`&`" или "`|`". Таким образом, использование `&&`- или `||`-операторов приводит к выполнению соответствующих `&`- или `|`-операций только в том случае, когда первый операнд не предопределяет результат всего выражения. Рассмотрим, например, следующую инструкцию из нашей программы:

```
if(a || c) Console.WriteLine("a || c - ИСТИНА.");
```

Сначала к объекту a применяется оператор true. Поскольку в данной ситуации a — истинный объект, в использовании операторного |-метода необходимости нет. Но эту инструкцию можно переписать и по-другому:

```
if(c || a) Console.WriteLine("c || a - ИСТИНА.");
```

В этом случае оператор true сначала будет применен к объекту c, а он в данной ситуации является ложным. Тогда будет вызван операторный |-метод, чтобы определить, является ли истинным объект a, что здесь как раз соответствует действительности.

Хотя на первый взгляд может показаться, что метод, используемый для разрешения использовать операторы сокращенного действия, несколько усложнен, то при более внимательном рассмотрении нетрудно убедиться в его логичности. Перегружая операторы true и false для класса, вы позволяете компилятору использовать операторы сокращенного действия без их перегрузки в явном виде. Более того, вы получаете возможность применять объекты в условных выражениях. Поэтому, если вам не нужна узкопрофильная реализация операторов "&" или "|", то лучше всего делать это в полном объеме.

Операторы преобразования

Иногда объект класса нужно использовать в выражении, включающем другие типы данных. Такие средства может обеспечить перегрузка одного или нескольких операторов. Но в некоторых случаях желаемого результата можно достичь за счет преобразования типов (из классового в нужный). В целях обработки подобных ситуаций C# позволяет создавать специальный тип операторного метода operator, именуемого *оператором преобразования*. Такой оператор преобразует объект некоторого класса в значение другого типа. По сути, оператор преобразования перегружает оператор приведения типов. Операторы преобразования способствуют полной интеграции классовых типов в C#-среду программирования, позволяя объектам класса свободно смешиваться с данными других типов при условии определения операторов преобразования в эти "другие типы".

Существуют две формы операторов преобразования: явная и неявная. В общем виде они записываются так:

```
public static explicit operator тип_результата(
                    исходный_тип v) [return значение;]
public static implicit operator тип_результата(
                    исходный_тип v) [return значение;]
```

Здесь элемент *тип_результата* представляет собой тип, в который вы хотите выполнить преобразование; элемент *исходный_тип* означает тип объекта, подлежащего преобразованию; элемент *v* — значение класса после преобразования. Операторы преобразования возвращают данные типа *тип_результата*, причем спецификатор типа здесь указывать не разрешается.

Если оператор преобразования определен с ключевым словом implicit, преобразование выполняется автоматически, т.е. при использовании объекта в выражении, включающем данные типа *тип_результата*. Если оператор преобразования определен с ключевым словом explicit, преобразование выполняется при использовании оператора приведения типов. Для одной и той же пары типов, участвующих в преобразовании, нельзя определить как explicit-, так и implicit-оператор преобразования.

Для иллюстрации определения и использования оператора преобразования создадим его для класса ThreeD. Предположим, необходимо преобразовать объект типа

ThreeD в целочисленное значение, чтобы его можно было использовать в выражениях типа int. Это преобразование будет заключаться в вычислении произведения значений всех трех координат объекта. Для реализации такого преобразования используем implicit-форму оператора, который будет иметь такой вид:

```
public static implicit operator int(ThreeD op1)
{
  return op1.x * op1.y * op1.z;
}
```

Ниже приведен текст программы, в которой иллюстрируется использование этого оператора преобразования.

```
// Пример использования implicit-оператора преобразования.

using System;

// Класс трехмерных координат.
class ThreeD {
  int x, y, z; // 3-х-мерные координаты.

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) {
                                  x = i; y = j; z = k; }

  // Перегружаем бинарный оператор "+".
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
  }

  // Неявное преобразование из типа ThreeD в тип int.
  public static implicit operator int(ThreeD op1)
  {
    return op1.x * op1.y * op1.z;
  }

  // Отображаем координаты X, Y, Z.
  public void show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  public static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();
    int i;

    Console.Write("Координаты точки a: ");
    a.show();
```

```
    Console.WriteLine();
    Console.Write("Координаты точки b: ");
    b.show();
    Console.WriteLine();

    c = a + b; // Суммируем координаты точек a и b.
    Console.Write("Результат сложения a + b: ");
    c.show();
    Console.WriteLine();

    i = a; // Преобразуем в значение типа int.
    Console.WriteLine(
           "Результат присваивания i = a: " + i);
    Console.WriteLine();

    i = a * 2 - b; // Преобразуем в значение типа int.
    Console.WriteLine(
       "Результат вычисления выражения a * 2 - b: " + i);
  }
}
```

При выполнении эта программа генерирует следующие результаты:

```
Координаты точки a: 1, 2, 3

Координаты точки b: 10, 10, 10

Результат сложения a + b: 11, 12, 13

Результат присваивания i = a: 6

Результат вычисления выражения a * 2 - b: -988
```

Как видно по результатам выполнения этой программы, если объект класса `ThreeD` используется в выражении целочисленного типа (например `i = a`), к этому объекту применяется оператор преобразования. В данном случае результатом этого преобразования будет число 6, полученное при умножении всех координат, хранимых в объекте `a`. Но если для выражения не требуется преобразование в `int`-значение, оператор преобразования не вызывается. Поэтому при вычислении выражения `c = a + b` операторный метод `operator int()` не вызывается.

Можно создавать различные методы преобразования, отвечающие вашим потребностям. Например, можно определить операторный метод преобразования объекта какого-либо класса в `double`- или `long`-значение. При этом каждое преобразование выполняется автоматически и независимо от других.

Оператор неявного преобразования применяется автоматически в том случае, когда в выражении требуется преобразование, когда методу передается объект, когда выполняется присваивание, а также когда используется явно заданная операция приведения объекта к результирующему типу. В качестве альтернативного варианта можно создать оператор явного преобразования, который вызывается только в случае явного приведения типов. Оператор явного преобразования не вызывается автоматически. Вот, например, как выглядит предыдущая программа, переработанная для использования оператора явного преобразования объекта в значение типа `int`:

```
// Использование оператора явного преобразования.

using System;

// Класс трехмерных координат.
```

```
class ThreeD {
  int x, y, z; // 3-х-мерные координаты.

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) {
                                    x = i; y = j; z = k; }

  // Перегружаем бинарный оператор "+".
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
  }

  // На этот раз перегружаем explicit-оператор.
  public static explicit operator int(ThreeD op1)
  {
    return op1.x * op1.y * op1.z;
  }

  // Отображаем координаты X, Y, Z.
  public void show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  public static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();
    int i;

    Console.Write("Координаты точки a: ");
    a.show();
    Console.WriteLine();
    Console.Write("Координаты точки b: ");
    b.show();
    Console.WriteLine();

    c = a + b; // Суммируем координаты объектов a и b.
    Console.Write("Результат сложения a + b: ");
    c.show();
    Console.WriteLine();

    i = (int) a; // Преобразуем объект в значение
                 // типа int, поскольку явно задана
                 // операция приведения типов.
    Console.WriteLine("Результат присваивания i = a: " + i);
    Console.WriteLine();

    i = (int)a * 2 - (int)b; // Требуется приведение типов.
```

```
    Console.WriteLine(
        "Результат вычисления выражения a * 2 - b: " + i);

  }
}
```

Поскольку оператор преобразования теперь определен с использованием ключевого слова `explicit`, преобразование объекта в значение типа `int` должно быть задано как оператор приведения типов. Например, если в строке кода

```
i = (int) a;
```

удалить оператор приведения типов, программа не скомпилируется.

Определение и использование операторов преобразования связано с рядом ограничений.

- Исходный тип объекта либо тип результата преобразования должен совпадать с создаваемым классом. Не разрешается переопределять такие преобразования, как из типа `double` в тип `int`.
- Нельзя определять преобразование в класс `Object` или из него.
- Нельзя задавать как явное, так и неявное преобразование одновременно для одной и той же пары исходного и результирующего типов.
- Нельзя задавать преобразование из базового класса в производный. (О базовых и производных классах см. главу 11.)
- Нельзя задавать преобразование из одного интерфейса в другой. (Об интерфейсах см. главу 12.)

Помимо перечисленных правил существуют также рекомендации, которым обычно следуют при выборе между явным и неявным операторами преобразования. Несмотря на определенные удобства к неявно заданным преобразованиям прибегают только в ситуациях, когда преобразование гарантированно лишено ошибок. Другими словами, неявные операторы преобразования следует создавать только при соблюдении следующих условий. Во-первых, такое преобразование должно гарантировать отсутствие потери данных, которое имеет место при усечении, переполнении или потере знака. Во-вторых, преобразование не должно стать причиной возникновения исключительных ситуаций. Если предполагаемое преобразование не отвечает этим требованиям, следует использовать преобразование явного типа.

Рекомендации и ограничения по созданию перегруженных операторов

Действие перегруженного оператора применительно к классу, для которого он определяется, не обязательно должно иметь отношение к стандартному действию этого оператора применительно к встроенным C#-типам. Но в целях структурированности и читабельности программного кода создаваемый перегруженный оператор должен по возможности отражать исходное назначение того или иного оператора. Например, оператор "+", перегруженный для класса `ThreeD`, концептуально подобен оператору "+", определенному для целочисленных типов. Ведь вряд ли есть логика в определении для класса, например, оператора "+", который по своему действию больше напоминает оператор деления (/). Таким образом, основная идея создания перегруженного оператора — наделить его новыми (нужными для вас) возможностями, которые тем не менее связаны с его первоначальным назначением.

На перегрузку операторов налагается ряд ограничений. Нельзя изменять приоритет оператора. Нельзя изменять количество операндов, принимаемых оператором, хотя операторный метод мог бы игнорировать любой операнд. Некоторые операторы вообще нельзя перегружать. Например, нельзя перегружать какие бы то ни было операторы присваивания (включая составные, например "+="). Ниже перечислены остальные операторы, перегрузка которых запрещена.

```
&&          ||          []      ()      new     is
sizeof      typeof      ?       ->      .       =
```

Несмотря на то что нельзя перегружать оператор приведения типов (()) в явном виде, можно создать операторы преобразования, которые, как было показано выше, успешно выполняют это.

Может показаться серьезным ограничением запрещение перегрузки таких составных операторов присваивания, как "+=". Если вы определили оператор, который используется в составном операторе присваивания, будет вызван соответствующий перегруженный операторный метод. Таким образом, использование "+=" в программе автоматически вызывает вашу версию метода `operator+()`. Например, возьмем снова класс `ThreeD`. При использовании фрагмента кода

```
ThreeD a = new ThreeD(1, 2, 3);
ThreeD b = new ThreeD(10, 10, 10);
b += a; // Суммируем a и b.
```

автоматически вызывается метод `operator+()` класса `ThreeD`, в результате чего объект b будет содержать координаты 11,12,13.

И еще. Несмотря на то что нельзя перегружать оператор доступа к элементам массива по индексу ([]), используя операторный метод (`operator()`), можно создавать индексаторы, которые описаны в следующей главе.

Еще один пример перегрузки операторов

На протяжении всей этой главы для демонстрации перегрузки операторов мы использовали класс `ThreeD`. Но прежде чем завершить эту главу, хотелось бы рассмотреть еще один пример. Несмотря на то что основные принципы перегрузки операторов не зависят от используемого класса, следующий пример поможет продемонстрировать мощь перегрузки операторов, особенно в случаях, связанных с расширяемостью типов.

В этом примере программы разрабатывается четырехразрядный целочисленный тип данных, для которого определяется ряд операций. Возможно, вам известно, что на заре компьютерной эры четырехразрядный тип был широко распространен, поскольку он представлял половину байта. Этот тип также позволял хранить одну шестнадцатеричную цифру. Так как четыре бита составляют половину байта, эту полубайтовую величину часто называли *nybble*. В период всеобщего распространения компьютеров с передней панелью, ввод данных в которые осуществлялся порциями объемом в 1 nybble, программисты привыкли оперировать величинами такого размера. Несмотря на то что полубайт нынче утратил былую популярность, этот четырехразрядный тип представляет интерес в качестве дополнения к другим целочисленным типам данных. По традиции nybble-значение рассматривается как значение без знака.

В следующем примере используется класс `Nybble`, в котором реализуется полубайтовый тип данных. Он основан на встроенном типе `int`, но ограничивает допустимые для хранения значения диапазоном 0–15. В этом классе определяются следующие операторы:

- сложение `Nybble`-объекта с `Nybble`-объектом;

- сложение int-значения с Nybble-объектом;
- сложение Nybble-объекта с int-значением;
- больше (>) и меньше (<);
- оператор инкремента;
- преобразование int-значения в Nybble-объект;
- преобразование Nybble-объекта в int-значение.

Этих операций вполне достаточно, чтобы показать, насколько полно тип класса может быть интегрирован с системой типов C#. Однако для полноты реализации типа Nybble, необходимо определить остальные операции. Это предлагается сделать читателю в качестве упражнения.

Ниже приведен код класса Nybble, а также класса NybbleDemo, который позволяет продемонстрировать его использование.

```
// Создание 4-битового типа с именем Nybble.

using System;

// 4-битовый тип.
class Nybble {
  int val; // Основа типа - встроенный тип int.

  public Nybble() { val = 0; }

  public Nybble(int i) {
    val = i;
    val = val & 0xF; // Выделяем младшие 4 бита.
  }

  // Перегружаем бинарный оператор "+" для
  // сложения Nybble + Nybble.
  public static Nybble operator +(Nybble op1, Nybble op2)
  {
    Nybble result = new Nybble();

    result.val = op1.val + op2.val;

    result.val = result.val & 0xF; // Оставляем младшие
                                   // 4 бита.

    return result;
  }

  // Перегружаем бинарный оператор "+" для
  // сложения Nybble + int.
  public static Nybble operator +(Nybble op1, int op2)
  {
    Nybble result = new Nybble();

    result.val = op1.val + op2;

    result.val = result.val & 0xF; // Оставляем младшие
                                   // 4 бита.

    return result;
  }
```

```
  // Перегружаем бинарный оператор "+" для
  // сложения int + Nybble.
  public static Nybble operator +(int op1, Nybble op2)
  {
    Nybble result = new Nybble();

    result.val = op1 + op2.val;

    result.val = result.val & 0xF; // Оставляем младшие
                                   // 4 бита.

    return result;
  }

  // Перегружаем оператор "++".
  public static Nybble operator ++(Nybble op)
  {
    op.val++;

    op.val = op.val & 0xF; // Оставляем младшие
                           // 4 бита.
    return op;
  }

  // Перегружаем оператор ">".
  public static bool operator >(Nybble op1, Nybble op2)
  {
    if(op1.val > op2.val) return true;
    else return false;
  }

  // Перегружаем оператор "<".
  public static bool operator <(Nybble op1, Nybble op2)
  {
    if(op1.val < op2.val) return true;
    else return false;
  }

  // Преобразование Nybble-объекта в значение типа int.
  public static implicit operator int (Nybble op)
  {
    return op.val;
  }

  // Преобразование int-значения в Nybble-объект.
  public static implicit operator Nybble (int op)
  {
    return new Nybble(op);
  }
}

class NybbleDemo {
  public static void Main() {
    Nybble a = new Nybble(1);
    Nybble b = new Nybble(10);
    Nybble c = new Nybble();
    int t;

    Console.WriteLine("a: " + (int) a);
```

```
    Console.WriteLine("b: " + (int) b);

    // Используем Nybble-объект в if-инструкции.
    if(a < b) Console.WriteLine("a меньше b\n");

    // Суммируем два Nybble-объекта.
    c = a + b;
    Console.WriteLine(
       "c после сложения c = a + b: " + (int) c);

    // Суммируем int-значение с Nybble-объектом.
    a += 5;
    Console.WriteLine("a после сложения a += 5: " + (int) a);

    Console.WriteLine();

    // Используем Nybble-объект в int-выражении.
    t = a * 2 + 3;
    Console.WriteLine(
          "Результат выражения a * 2 + 3: " + t);

    Console.WriteLine();

    // Иллюстрируем присваивание Nybble-объекту
    // int-значения и переполнение.
    a = 19;
    Console.WriteLine(
         "Результат присваивания a = 19: " + (int) a);

    Console.WriteLine();

    // Используем Nybble-объект для управления циклом.
    Console.WriteLine(
       "Управляем for-циклом с помощью Nybble-объекта.");
    for(a = 0; a < 10; a++)
      Console.Write((int) a + " ");

    Console.WriteLine();
  }
}
```

При выполнении эта программа генерирует следующие результаты:

```
a: 1
b: 10
a меньше b

c после сложения c = a + b: 11
a после сложения a += 5: 6

Результат выражения a * 2 + 3: 15

Результат присваивания a = 19: 3

Управляем for-циклом с помощью Nybble-объекта.
0 1 2 3 4 5 6 7 8 9
```

Несмотря на то что большинство реализованных здесь операций не нуждается в дополнительных комментариях, имеет смысл остановиться вот на чем. Операторы преобразования играют немаловажную роль в интеграции типа `Nybble` в систему ти-

пов C#. Поскольку в этом классе определены преобразования как из Nybble-объекта в int-значение, так из int-значения в Nybble-объект, то Nybble-объекты можно свободно смешивать в арифметических выражениях. Рассмотрим, например, такое выражение из этой программы:

```
t = a * 2 + 3;
```

Здесь переменная t имеет тип int, а переменная a представляет собой объект класса Nybble. Тем не менее эти два типа совместимы в одном выражении благодаря оператору неявного преобразования Nybble-объекта в int-значение. В данном случае, поскольку остальная часть выражения имеет тип int, объект a преобразуется в int-значение посредством вызова соответствующего метода преобразования.

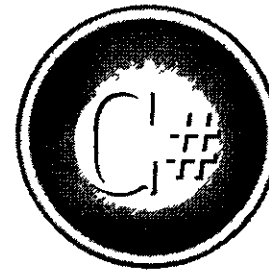
Преобразование int-значения в Nybble-объект позволяет присваивать Nybble-объекту int-значение. Например, инструкция

```
a = 19;
```

работает следующим образом. Сначала выполняется оператор преобразования int-значения в Nybble-объект. При этом создается новый Nybble-объект, который содержит младшие четыре (двоичных) разряда числа 19 ($19_{10} = 10011_2$). Это приводит к переполнению, поскольку значение 19 выходит за пределы диапазона, допустимого в классе Nybble. В результате получаем число 3 ($0011_2 = 3_{10}$), которое и присваивается объекту a. Без определения операторов преобразования такие выражения были бы недопустимы.

Преобразование Nybble-объекта в int-значение используется также в цикле for. Без определения этого преобразования было бы невозможно написать цикл for таким простым способом.

Полный справочник по



Глава 10

Индексаторы и свойства

В этой главе рассматриваются два специальных типа членов класса, которые тесно связаны друг с другом: индексаторы и свойства. Каждый из этих типов расширяет возможности класса, усиливая его интеграцию в систему типов языка C# и гибкость. Индексаторы обеспечивают механизм, посредством которого к объектам можно получать доступ по индексу подобно тому, как это реализовано в массивах. Свойства предлагают эффективный способ управления доступом к данным экземпляра класса. Эти типы связаны друг с другом, поскольку оба опираются на еще одно средство C#: аксессор, или средство доступа к данным.

Индексаторы

Как вы знаете, индексация массивов реализуется с использованием оператора "`[]`". В своих классах вы можете перегрузить его, но не прибегая к "услугам" метода `operator()`, а посредством создания *индексатора* (indexer). Индексатор позволяет обеспечить индексированный доступ к объекту. Главное назначение индексаторов — поддержать создание специализированных массивов, на которые налагается одно или несколько ограничений. При этом индексаторы можно использовать в синтаксисе, подобном реализованному в массивах. Индексаторы могут характеризоваться одной или несколькими размерностями, но мы начнем с одномерных индексаторов.

Создание одномерных индексаторов

Одномерный индексатор имеет следующий формат.

```
тип_элемента this[int индекс] {
    // Аксессор считывания данных.
    get {
        // Возврат значения, заданного
        // элементом индекс.
    }

    // Аксессор установки данных.
    set {
        // Установка значения, заданного
        // элементом индекс.
    }
}
```

Здесь *тип_элемента* — базовый тип индексатора. Таким образом, *тип_элемента* — это тип каждого элемента, к которому предоставляется доступ посредством индексатора. Он соответствует базовому типу массива. Параметр *индекс* получает индекс опрашиваемого (или устанавливаемого) элемента. Строго говоря, этот параметр не обязательно должен иметь тип `int`, но поскольку индексаторы обычно используются для обеспечения индексации массивов, целочисленный тип — наиболее подходящий.

В теле индексатора определяются два *аксессора* (средства доступа) с именами `get` и `set`. Аксессор подобен методу за исключением того, что в нем отсутствует объявление типа возвращаемого значения и параметров. При использовании индексатора аксессоры вызываются автоматически, и в качестве параметра оба аксессора принимают *индекс*. Если индексатор стоит слева от оператора присваивания, вызывается аксессор `set` и устанавливается элемент, заданный параметром *индекс*. В противном случае вызывается аксессор `get` и возвращается значение, соответствующее параметру *ин-*

декс. Метод set также получает значение (именуемое value), которое присваивается элементу массива, найденному по заданному индексу.

Одно из достоинств индексатора состоит в том, что он позволяет точно управлять характером доступа к массиву, "отбраковывая" попытки некорректного доступа. Рассмотрим пример. В следующей программе класс FailSoftArray реализует массив, который "вылавливает" ошибки нарушения границ, предотвращая возникновение исключительных ситуаций. Это достигается за счет инкапсуляции массива как закрытого члена класса и осуществления доступа к этому массиву только через индексатор. При таком подходе можно предотвратить любую попытку получить доступ к массиву за пределами его границ, причем последствия попытки нарушить границы в этом случае можно сравнить с "мягкой посадкой", а не с "катастрофическим падением". Поскольку в классе FailSoftArray используется индексатор, к массиву можно обращаться с помощью обычной формы записи ([]).

```
// Использование индексатора для создания
// отказоустойчивого массива.

using System;

class FailSoftArray {
  int[] a;    // Ссылка на массив.

  public int Length; // Length - открытый член.

  public bool errflag; // Индикатор результата
                       // последней операции.

  // Создаем массив заданного размера.
  public FailSoftArray(int size) {
    a = new int[size];
    Length = size;
  }

  // Это - индексатор для класса FailSoftArray.
  public int this[int index] {
    // Это - get-аксессор.
    get {
      if(ok(index)) {
        errflag = false;
        return a[index];
      } else {
        errflag = true;
        return 0;
      }
    }

    // Это - set-аксессор.
    set {
      if(ok(index)) {
        a[index] = value;
        errflag = false;
      }
      else errflag = true;
    }
  }

  // Метод возвращает значение true, если
```

```
  // индекс - в пределах границ.
  private bool ok(int index) {
   if(index >= 0 & index < Length) return true;
   return false;
  }
}

// Демонстрируем отказоустойчивый массив.
class FSDemo {
  public static void Main() {
    FailSoftArray fs = new FailSoftArray(5);
    int x;

    // Вот как выглядит "мягкая посадка" при ошибках.
    Console.WriteLine("Мягкое приземление.");
    for(int i=0; i < (fs.Length * 2); i++)
      fs[i] = i*10;

    for(int i=0; i < (fs.Length * 2); i++) {
      x = fs[i];
      if(x != -1) Console.Write(x + " ");
    }
    Console.WriteLine();

    // Теперь генерируем некорректный доступ.
    Console.WriteLine(
              "\nРабота с уведомлением об ошибках.");
    for(int i=0; i < (fs.Length * 2); i++) {
      fs[i] = i*10;
      if(fs.errflag)
        Console.WriteLine("fs[" + i + "] вне границ");
    }

    for(int i=0; i < (fs.Length * 2); i++) {
      x = fs[i];
      if(!fs.errflag) Console.Write(x + " ");
      else
        Console.WriteLine("fs[" + i + "] вне границ");
    }
  }
}
```

При выполнении этой программы получаем такие результаты:

```
"Мягкое приземление".
0 10 20 30 40 0 0 0 0 0

Работа с уведомлением об ошибках.
fs[5] вне границ
fs[6] вне границ
fs[7] вне границ
fs[8] вне границ
fs[9] вне границ
0 10 20 30 40 fs[5] вне границ
fs[6] вне границ
fs[7] вне границ
fs[8] вне границ
fs[9] вне границ
```

Созданный здесь индексатор предотвращает нарушение границ массива. Рассмотрим подробно каждую часть индексатора. Индексатор начинается с такой строки:

```
public int this[int index] {
```

Здесь объявляется индексатор, который оперирует элементами типа `int`. Индекс передается в параметре `index`. Сам индексатор — открытый для использования любым кодом вне его класса.

Теперь приведем код аксессора `get`.

```
get {
  if(ok(index)) {
    errflag = false;
    return a[index];
  } else {
     errflag = true;
     return 0;
  }
}
```

Аксессор `get` предотвращает ошибки нарушения границ. Если заданный индекс находится в пределах границ, аксессор `get` возвращает элемент, соответствующий этому индексу. А если переданный индекс выходит за пределы границ, операции с массивом не выполняются, но и ничего страшного при этом не происходит. В данной версии класса `FailSoftArray` переменная `errflag` содержит результат выполнения каждой операции. Чтобы узнать, как завершилась очередная операция, достаточно проанализировать это поле.

Ниже приведен код аксессора `set()`. Он также предотвращает ошибки нарушения границ.

```
set {
  if(ok(index)) {
    a[index] = value;
    errflag = false;
  }
  else errflag = true;
}
```

Если заданный индекс находится в пределах границ, значение, переданное через переменную `value`, присваивается соответствующему элементу массива. В противном случае признак ошибки `errflag` устанавливается равным значению `true`. Вспомните, что в любом аксессорном методе `value` — это автоматически устанавливаемый параметр, который содержит значение, подлежащее записи. Вам не нужно (да вы и не сможете) объявлять этот параметр самим.

Индексаторы необязательно создавать с поддержкой как `get`-, так и `set`-аксессоров. Можно создать индексатор, предназначенный только для чтения, реализовав лишь `get`-аксессор. И точно также можно создать индексатор, предназначенный только для записи, реализовав лишь `set`-аксессор.

Перегрузка индексаторов

Индексаторы можно перегружать. Здесь приведен пример определения класса `FailSoftArray`, в котором перегружается индексатор для индексов типа `double`. В действительности `double`-индексатор округляет индекс до ближайшего целого числа. Таким образом, из двух определенных в классе индексаторов будет выполняться тот, для которого окажется наилучшим соответствие типов параметра индексатора и его аргумента, используемого в качестве индекса.

```
// Перегрузка индексатора для класса FailSoftArray.

using System;

class FailSoftArray {
  int[] a;    // Ссылка на базовый массив.

  public int Length; // Length (длина) - открытый член.

  public bool errflag; // Индикатор результата
                       // последней операции.

  // Создаем массив заданной длины.
  public FailSoftArray(int size) {
    a = new int[size];
    Length = size;
  }

  // Это int-индексатор для класса FailSoftArray.
  public int this[int index] {
    // Это -- get-аксессор.
    get {
      if(ok(index)) {
        errflag = false;
        return a[index];
      } else {
        errflag = true;
        return 0;
      }
    }

    // Это -- set-аксессор.
    set {
      if(ok(index)) {
        a[index] = value;
        errflag = false;
      }
      else errflag = true;
    }
  }

  /* Это еще один индексатор для класса FailSoftArray.
     Здесь в качестве индекса принимается double-аргумент.
     Затем аргумент округляется до ближайшего целого
     индекса. */
  public int this[double idx] {
    // Это -- get-аксессор.
    get {
      int index;

      // Округление до ближайшего целого int-значения.
      if( (idx - (int) idx) < 0.5) index = (int) idx;
      else index = (int) idx + 1;

      if(ok(index)) {
        errflag = false;
        return a[index];
      } else {
```

```
        errflag = true;
        return 0;
      }
    }

    // Это -- set-аксессор.
    set {
      int index;

      // Округление до ближайшего целого int-значения.
      if( (idx - (int) idx) < 0.5) index = (int) idx;
      else index = (int) idx + 1;

      if(ok(index)) {
        a[index] = value;
        errflag = false;
      }
      else errflag = true;
    }
  }

  // Метод возвращает true, если индекс внутри границ.
  private bool ok(int index) {
   if(index >= 0 & index < Length) return true;
   return false;
  }
}

// Демонстрируем отказоустойчивый массив.
class FSDemo {
  public static void Main() {
    FailSoftArray fs = new FailSoftArray(5);

    // Помещаем в массив fs несколько значений.
    for(int i=0; i < fs.Length; i++)
      fs[i] = i;

    // Теперь используем в качестве индекса
    // int- и double-значения.
    Console.WriteLine("fs[1]: " + fs[1]);
    Console.WriteLine("fs[2]: " + fs[2]);

    Console.WriteLine("fs[1.1]: " + fs[1.1]);
    Console.WriteLine("fs[1.6]: " + fs[1.6]);

  }
}
```

Эта программа генерирует такие результаты:

```
fs[1]: 1
fs[2]: 2
fs[1.1]: 1
fs[1.6]: 2
```

Как подтверждают результаты выполнения этой программы, `double`-индексы округляются до ближайших целых значений. В частности, число `1.1` округляется до `1`, а число `1.6` — до `2`.

Несмотря на то что перегрузка индексатора, показанная в этой программе, вполне допустима, этот пример — нетипичен. Чаще всего индексатор перегружается, чтобы иметь возможность использовать объект класса в качестве индекса, значение которого вычисляется специальным образом.

Индексаторам не требуется базовый массив

Индексатор может не использовать базовый массив. Вполне достаточно, чтобы индексатор обеспечивал функционирование, которое для пользователя выглядело бы, как то, что обеспечивают массивы. Например, следующая программа включает индексатор, который действует подобно массиву, предназначенному только для чтения. Этот "массив" содержит степени числа 2 для чисел от 0 до 15. Однако в действительности никакого массива не существует. Вместо этого индексатор просто (и быстро) вычисляет соответствующее значение для заданного индекса.

```
// Индексаторы не обязательно должны использовать
// реальные массивы.

using System;

class PwrOfTwo {

  /* Доступ к логическому массиву, который содержит
     степени числа 2 для чисел от 0 до 15. */
  public int this[int index] {
    // Вычисляем и возвращаем степень числа 2.
    get {
      if((index >= 0) && (index < 16)) return pwr(index);
      else return -1;
    }

    // Здесь нет set-аксессора.
  }

  int pwr(int p) {
    int result = 1;

    for(int i=0; i<p; i++)
      result *= 2;

    return result;
  }
}

class UsePwrOfTwo {
  public static void Main() {
    PwrOfTwo pwr = new PwrOfTwo();

    Console.Write("Первые 8 степеней числа 2: ");
    for(int i=0; i < 8; i++)
      Console.Write(pwr[i] + " ");
    Console.WriteLine();

    Console.Write("А вот несколько ошибок: ");
    Console.Write(pwr[-1] + " " + pwr[17]);

    Console.WriteLine();
  }
}
```

Вот результаты выполнения этой программы:

```
Первые 8 степеней числа 2: 1 2 4 8 16 32 64 128
А вот несколько ошибок: -1 -1
```

Обратите внимание на то, что индексатор класса `UsePwrOfTwo` включает `get`-аксессор, но обходится без `set`-аксессора. Это означает, что индексатор предназначен только для чтения. Таким образом, объект класса `UsePwrOfTwo` можно использовать в правой части инструкции присвоения, но ни в коем случае не в левой. Например, попытка добавить эту инструкцию в предыдущую программу, обречена на неудачу:

```
pwr[0] = 11;  // не скомпилируется
```

Эта инструкция вызовет ошибку компиляции, поскольку в индексаторе не определен `set`-аксессор.

На использование индексаторов накладывается два ограничения. Во-первых, поскольку в индексаторе не определяется область памяти, получаемое индексатором значение нельзя передавать методу в качестве `ref`- или `out`-параметра. Во-вторых, индексатор должен быть членом экземпляра своего класса, поэтому его нельзя объявлять с использованием ключевого слова `static`.

Многомерные индексаторы

Можно создавать индексаторы и для многомерных массивов. Например, вот как работает двумерный отказоустойчивый массив. Обратите особое внимание на способ объявления индексатора в этом классе.

```
// Двумерный отказоустойчивый массив.

using System;

class FailSoftArray2D {
  int[,] a; // Ссылка на базовый двумерный массив.
  int rows, cols; // размерности
  public int Length; // Length - открытый член.

  public bool errflag; // Индикатор результата
                       // последней операции.

  // Создаем массив заданного размера.
  public FailSoftArray2D(int r, int c) {
    rows = r;
    cols = c;
    a = new int[rows, cols];
    Length = rows * cols;
  }

  // Это индексатор для класса FailSoftArray2D.
  public int this[int index1, int index2] {
    // Это -- get-аксессор.
    get {
      if(ok(index1, index2)) {
        errflag = false;
        return a[index1, index2];
      } else {
        errflag = true;
        return 0;
      }
    }
```

```
    // Это -- set-аксессор.
    set {
      if(ok(index1, index2)) {
        a[index1, index2] = value;
        errflag = false;
      }
      else errflag = true;
    }
  }

  // Метод возвращает значение true, если индексы
  // находятся внутри границ.
  private bool ok(int index1, int index2) {
   if(index1 >= 0 & index1 < rows &
      index2 >= 0 & index2 < cols)
         return true;

   return false;
  }
}

// Демонстрируем использование двумерного индексатора.
class TwoDIndexerDemo {
  public static void Main() {
    FailSoftArray2D fs = new FailSoftArray2D(3, 5);
    int x;

    // Демонстрируем "мягкую посадку" при ошибках.
    Console.WriteLine("Мягкое приземление.");
    for(int i=0; i < 6; i++)
      fs[i, i] = i*10;

    for(int i=0; i < 6; i++) {
      x = fs[i,i];
      if(x != -1) Console.Write(x + " ");
    }
    Console.WriteLine();

    // А теперь генерируем ошибки.
    Console.WriteLine(
              "\nРабота с уведомлением об ошибках.");
    for(int i=0; i < 6; i++) {
      fs[i,i] = i*10;
      if(fs.errflag)
        Console.WriteLine(
              "fs[" + i + ", " + i + "] вне границ");
    }

    for(int i=0; i < 6; i++) {
      x = fs[i,i];
      if(!fs.errflag) Console.Write(x + " ");
      else
        Console.WriteLine(
              "fs[" + i + ", " + i + "] вне границ");
    }
  }
}
```

Результаты, генерируемые этой программой:

```
Мягкое приземление.
0 10 20 0 0 0

Работа с уведомлением об ошибках.
fs[3, 3] вне границ
fs[4, 4] вне границ
fs[5, 5] вне границ
0 10 20 fs[3, 3] вне границ
fs[4, 4] вне границ
fs[5, 5] вне границ
```

Свойства

Свойство — это второй специальный тип членов класса, о котором мы собирались поговорить в этой главе. Свойство включает поле и методы доступа к этому полю. Часто требуется создать поле, которое должно быть доступно для пользователей объекта, но программист при этом хочет осуществлять управление операциями, разрешенными к выполнению над этим полем. Например, по некоторым обстоятельствам вы желаете ограничить диапазон значений, которые можно присваивать этому полю. Несмотря на то что этого можно достичь с помощью закрытой переменной и методов доступа к ней, свойство предлагает более удобный и простой способ решения этой задачи.

Свойства во многом напоминают индексаторы. Свойство состоит из имени и пары аксессоров (`get` и `set`). Аксессоры используются для чтения содержимого переменной и записи в нее нового значения. Основное достоинство свойства состоит в том, что его имя можно использовать в выражениях и инструкциях присваивания подобно обычной переменной, хотя в действительности здесь будут автоматически вызываться `get`- и `set`-аксессоры. Автоматический вызов аксессоров и роднит свойства с индексаторами.

Формат записи свойства таков:

```
тип имя{
  get{
    // код аксессора чтения поля
  }

  set{
    // код аксессора записи поля
  }
```

Здесь *тип* — это тип свойства (например, `int`), а *имя* — его имя. После определения свойства любое использование его имени означает вызов соответствующего аксессора. Аксессор `set` автоматически принимает параметр с именем `value`, который содержит значение, присваиваемое свойству.

Важно понимать, что свойства не определяют область памяти. Следовательно, свойство управляет доступом к полю, но самого поля не обеспечивает. Это поле должно быть задано независимо от свойства.

Рассмотрим простой пример, в котором определяется свойство `myprop`, используемое для доступа к полю `prop`. Это свойство позволяет присваивать полю только положительные числа.

```
// Пример использования свойства.

using System;
```

```
class SimpProp {
  int prop; // Это поле управляется свойством myprop.

  public SimpProp() { prop = 0; }

  /* Это свойство поддерживает доступ к закрытой
     переменной экземпляра prop.  Оно позволяет
     присваивать ей только положительные числа. */
  public int myprop {
    get {
      return prop;
    }
    set {
      if(value >= 0) prop = value;
    }
  }
}

// Демонстрируем использование свойства.
class PropertyDemo {
  public static void Main() {
    SimpProp ob = new SimpProp();

    Console.WriteLine("Исходное значение ob.myprop: " +
                      ob.myprop);

    ob.myprop = 100; // Присваиваем значение.
    Console.WriteLine("Значение ob.myprop: " + ob.myprop);

    // Переменной prop невозможно присвоить
    // отрицательное значение.
    Console.WriteLine(
              "Попытка присвоить -10 свойству ob.myprop");
    ob.myprop = -10;
    Console.WriteLine("Значение ob.myprop: " + ob.myprop);
  }
}
```

Результаты выполнения этой программы выглядят так:

```
Исходное значение ob.myprop: 0
Значение ob.myprop: 100
Попытка присвоить -10 свойству ob.myprop
Значение ob.myprop: 100
```

На этой программе стоит остановиться подробнее. В классе `SimpProp` определяется закрытое поле `prop` и свойство `myprop`, которое управляет доступом к полю `prop`. Как упоминалось выше, свойство само не определяет область хранения поля, а лишь управляет доступом к нему. Поэтому без определения базового поля определение свойства теряет всякий смысл. Более того, поскольку поле `prop` закрытое, к нему можно получить доступ только посредством свойства `myprop`.

Свойство `myprop` определяется как `public`-член класса, чтобы к нему можно было обратиться с помощью кода вне класса, включающего это свойство. В этом есть своя логика, поскольку свойство предоставляет доступ к закрытому полю `prop` с помощью аксессоров: `get`-аксессор просто возвращает значение `prop`, а `set`-аксессор устанавливает новое значение `prop`, если оно положительно. Таким образом, свойство

myprop управляет тем, какие значения может содержать поле prop. В этом и состоит важность свойств.

Свойство myprop предназначено для чтения и записи, поскольку позволяет как прочитать содержимое своего базового поля, так и записать в него новое значение. Но можно создавать свойства, предназначенные только для чтения (определив лишь get-аксессор) либо только для записи (определив лишь set-аксессор).

Мы можем использовать свойство для дальнейшего усовершенствования класса, определяющего отказоустойчивый массив. Как вы уже знаете, с каждым массивом связано свойство Length. До сих пор в классе FailSoftArray для этой цели просто использовалось открытое целочисленное поле Length. Такое решение — не самое лучшее, поскольку в этом случае можно записать в поле Length значение, отличное от реальной длины этого массива. (Например, какой-нибудь программист с дурными наклонностями мог умышленно ввести это значение.) Потенциально опасную ситуацию можно исправить, заменив открытую переменную Length свойством, предназначенным только для чтения, как показано в следующей версии класса FailSoftArray.

```
// Добавляем в класс FailSoftArray свойство Length.

using System;

class FailSoftArray {
  int[] a; // Ссылка на базовый массив.
  int len; // Длина массива, основа для свойства Length.

  public bool errflag; // Индикатор результата
                       // последней операции.

  // Создаем массив заданного размера.
  public FailSoftArray(int size) {
    a = new int[size];
    len = size;
  }

  // Свойство Length предназначено только для чтения.
  public int Length {
    get {
      return len;
    }
  }

  // Это -- индексатор класса FailSoftArray.
  public int this[int index] {
    // Это -- get-аксессор.
    get {
      if(ok(index)) {
        errflag = false;
        return a[index];
      } else {
        errflag = true;
        return 0;
      }
    }

    // Это -- set-аксессор.
    set {
      if(ok(index)) {
        a[index] = value;
```

```
        errflag = false;
      }
      else errflag = true;
    }
  }

  // Метод возвращает true, если индекс внутри границ.
  private bool ok(int index) {
   if(index >= 0 & index < Length) return true;
   return false;
  }
}

// Демонстрируем улучшенный отказоустойчивый массив.
class ImprovedFSDemo {
  public static void Main() {
    FailSoftArray fs = new FailSoftArray(5);
    int x;

    // Свойство Length можно считывать.
    for(int i=0; i < fs.Length; i++)
      fs[i] = i*10;

    for(int i=0; i < fs.Length; i++) {
      x = fs[i];
      if(x != -1) Console.Write(x + " ");
    }
    Console.WriteLine();

    // fs.Length = 10; // Ошибка, запись запрещена!
  }
}
```

Теперь `Length` — это свойство, которое в качестве области памяти использует закрытую переменную `len`. В этом свойстве определен только `get`-аксессор, и потому свойство `Length` можно только читать, но не изменять. Чтобы убедиться в этом, попробуйте убрать символы комментариев в начале следующей строки программы:

```
// fs.Length = 10; // Ошибка, запись запрещена!
```

При попытке скомпилировать программу с этой строкой кода вы получите сообщение об ошибке, уведомляющее о том, что свойство `Length` предназначено только для чтения.

Внесение в класс `FailSoftArray` свойства `Length` значительно улучшило его, но на этом рано ставить точку. Член `errflag` — еще один кандидат на "превращение" из обычной переменной экземпляра в свойство со всеми преимуществами, поскольку доступ к нему следует ограничить до определения "только для чтения". Приводим окончательную версию класса `FailSoftArray`, в которой создано свойство `Error`, использующее в качестве области хранения индикатора ошибки исходную переменную `errflag`.

```
// Превращаем переменную errflag в свойство.

using System;

class FailSoftArray {
  int[] a; // Ссылка на базовый массив.
  int len; // Длина массива.
```

```
  bool errflag; // Теперь этот член закрыт.

  // Создаем массив заданного размера.
  public FailSoftArray(int size) {
    a = new int[size];
    len = size;
  }

  // Свойство Length предназначено только для чтения.
  public int Length {
    get {
      return len;
    }
  }

  // Свойство Error предназначено только для чтения.
  public bool Error {
    get {
      return errflag;
    }
  }

  // Это - индексатор класса FailSoftArray.
  public int this[int index] {
    // Это -- get-аксессор.
    get {
      if(ok(index)) {
        errflag = false;
        return a[index];
      } else {
        errflag = true;
        return 0;
      }
    }

    // Это -- set-аксессор.
    set {
      if(ok(index)) {
        a[index] = value;
        errflag = false;
      }
      else errflag = true;
    }
  }

  // Метод возвращает true, если индекс внутри границ.
  private bool ok(int index) {
   if(index >= 0 & index < Length) return true;
   return false;
  }
}

// Демонстрируем улучшенный отказоустойчивый массив.
class FinalFSDemo {
  public static void Main() {
    FailSoftArray fs = new FailSoftArray(5);

    // Используем свойство Error.
```

```
    for(int i=0; i < fs.Length + 1; i++) {
      fs[i] = i*10;
      if(fs.Error)
        Console.WriteLine("Ошибка в индексе " + i);
    }

  }
}
```

Создание свойства `Error` заставило нас внести в класс `FailSoftArray` два изменения. Во-первых, переменную `errflag` пришлось сделать закрытой, поскольку она теперь используется в качестве базовой области памяти для свойства `Error`. В результате к члену `errflag` теперь нельзя обращаться напрямую. Во-вторых, добавлено свойство `Error`, предназначенное только для чтения. Отныне программы для выявления ошибок будут опрашивать не поле `errflag`, а свойство `Error`. Это продемонстрировано в методе `Main()`, в котором умышленно генерируется ошибка нарушения границ массива, а для ее обнаружения используется свойство `Error`.

Правила использования свойств

На использование свойств налагаются довольно серьезные ограничения. Во-первых, поскольку в свойстве не определяется область памяти, его нельзя передавать методу в качестве `ref`- или `out`-параметра. Во-вторых, свойство нельзя перегружать. (Но при необходимости вы *можете* иметь два различных свойства, которые используют одну и ту же базовую переменную, но к такой организации свойств прибегают нечасто.) Наконец, свойство не должно изменять состояние базовой переменной при вызове `get`-аксессора, хотя несоблюдение этого правила компилятор обнаружить не в состоянии. Другими словами, `get`-операция должна быть максимально простой.

Использование индексаторов и свойств

Несмотря на то что в предыдущих примерах продемонстрирован механизм работы индексаторов и свойств, в них не отражена в полной мере вся мощь этих программных средств. В заключение этой главы мы рассмотрим класс `RangeArray`, использующий индексаторы и свойства для создания массива такого типа, в котором индексный диапазон массива определяется программистом.

Как вы знаете, в C# индексация массивов начинается с нуля. Однако в некоторых приложениях было бы удобно начинать индексацию массивов с произвольного числа, например с единицы или даже с отрицательного числа, чтобы индексы изменялись в диапазоне от −5 до 5. Приведенный здесь класс `RangeArray` как раз и позволяет подобные способы индексации массивов.

При использовании класса `RangeArray` можно написать такие строки кода:

```
RangeArray ra = new RangeArray(-5, 10); // Массив с
                                        // индексами от -5 до 10.

for(int i = -5; i <= 10; i++) ra[i] = i; // Индекс
                                        // изменяется от -5 до 10.
```

Нетрудно догадаться, что первая строка создает объект класса `RangeArray` (массив `ra`), в котором индексы изменяются от −5 до 10 включительно. Первый аргумент задает начальный индекс, а второй — конечный.

Ниже представлены классы RangeArray и RangeArrayDemo, демонстрирующие использование этого массива. Класс RangeArray поддерживает массив int-элементов, но при необходимости можно заменить этот тип данных другим.

```
/* Создание класса для поддержки массива с заданным
   диапазоном индексации. Класс RangeArray позволяет
   начинать индексацию с числа, отличного от нуля.
   При создании объекта класса RangeArray необходимо
   задать индексы начала и конца диапазона.
   Отрицательные индексы также допустимы.  Например,
   можно создать массивы с диапазоном изменения индексов
   от -5 до 5, от 1 до 10 или от 50 до 56.
*/

using System;

class RangeArray {
  // Закрытые данные.
  int[] a; // Ссылка на базовый массив.
  int lowerBound; // Наименьший индекс.
  int upperBound; // Наибольший индекс.

  // Данные для свойств.
  int len; // Базовая переменная для свойства Length.
  bool errflag; // Базовая переменная для свойства Error.

  // Создаем массив с заданным размером.
  public RangeArray(int low, int high) {
    high++;
    if(high <= low) {
      Console.WriteLine("Неверные индексы.");
      high = 1; // Создаем минимальный массив для
                // безопасности.
      low = 0;
    }
    a = new int[high - low];
    len = high - low;

    lowerBound = low;
    upperBound = --high;
  }

  // Свойство Length, предназначенное только для чтения.
  public int Length {
    get {
      return len;
    }
  }

  // Свойство Error, предназначенное только для чтения.
  public bool Error {
    get {
      return errflag;
    }
  }

  // Это -- индексатор для класса RangeArray.
  public int this[int index] {
```

```
    // Это -- get-аксессор.
    get {
      if(ok(index)) {
        errflag = false;
        return a[index - lowerBound];
      } else {
        errflag = true;
        return 0;
      }
    }

    // Это -- set-аксессор.
    set {
      if(ok(index)) {
        a[index - lowerBound] = value;
        errflag = false;
      }
      else errflag = true;
    }
  }

  // Метод возвращает true, если индекс находится
  // внутри границ.
  private bool ok(int index) {
    if(index >= lowerBound & index <= upperBound)
      return true;
    return false;
  }
}

// Демонстрируем использование массива с произвольно
// заданным диапазоном индексов.
class RangeArrayDemo {
  public static void Main() {
    RangeArray ra = new RangeArray(-5, 5);
    RangeArray ra2 = new RangeArray(1, 10);
    RangeArray ra3 = new RangeArray(-20, -12);

    // Используем массив ra.
    Console.WriteLine("Длина массива ra: " + ra.Length);

    for(int i = -5; i <= 5; i++)
      ra[i] = i;

    Console.Write("Содержимое массива ra: ");
    for(int i = -5; i <= 5; i++)
      Console.Write(ra[i] + " ");

    Console.WriteLine("\n");

    // Используем массив ra2.
    Console.WriteLine("Длина массива ra2: " + ra2.Length);

    for(int i = 1; i <= 10; i++)
      ra2[i] = i;

    Console.Write("Содержимое массива ra2: ");
    for(int i = 1; i <= 10; i++)
```

```
      Console.Write(ra2[i] + " ");

    Console.WriteLine("\n");

    // Используем массив ra3
    Console.WriteLine("Длина массива ra3: " + ra3.Length);

    for(int i = -20; i <= -12; i++)
      ra3[i] = i;

    Console.Write("Содержимое массива ra3: ");
    for(int i = -20; i <= -12; i++)
      Console.Write(ra3[i] + " ");

    Console.WriteLine("\n");

  }
}
```

При выполнении эта программа генерирует такие результаты:

```
Длина массива ra: 11
Содержимое массива ra: -5 -4 -3 -2 -1 0 1 2 3 4 5

Длина массива ra2: 10
Содержимое массива ra2: 1 2 3 4 5 6 7 8 9 10

Длина массива ra3: 9
Содержимое массива ra3: -20 -19 -18 -17 -16 -15 -14 -13 -12
```

Как подтверждают результаты выполнения этой программы, объекты типа `RangeArray` могут быть индексированы не обязательно начиная с нуля. Рассмотрим, как же реализован класс `RangeArray`.

Определение этого класса начинается с определения закрытых переменных экземпляра:

```
// Закрытые данные.
int[] a; // Ссылка на базовый массив.
int lowerBound; // Наименьший индекс.
int upperBound; // Наибольший индекс.

// Данные для свойств.
int len; // Базовая переменная для свойства Length.
bool errflag; // Базовая переменная для свойства Error.
```

Базовый массив имеет имя `a`. Он размещается в памяти с помощью конструктора класса `RangeArray`. Индекс нижней границы массива сохраняется в закрытой переменной `lowerBound`, а индекс верхней границы — в закрытой переменной `upperBound`. Затем объявляются переменные элемента, которые поддерживают свойства `Length` и `Error`.

Конструктор класса `RangeArray` имеет такой вид:

```
// Создаем массив с заданным размером.
public RangeArray(int low, int high) {
  high++;
  if(high <= low) {
    Console.WriteLine("Неверные индексы.");
    high = 1; // Создаем минимальный массив для
              // безопасности.
    low = 0;
```

```
    }
    a = new int[high - low];
    len = high - low;

    lowerBound = low;
    upperBound = --high;
  }
```

Объект типа `RangeArray` создается в результате передачи нижнего граничного индекса в параметр `low` и верхнего граничного индекса — в параметр `high`. Значение `high` затем инкрементируется, чтобы вычислить размер массива, поскольку задаваемые индексы изменяются от `low` до `high` включительно. После этого проверяем, действительно ли верхний индекс больше нижнего. Если это не так, выводится сообщение об ошибке в индексах и создается одноэлементный массив. Затем выделяется область памяти для массива (либо корректно заданного, либо ошибочно), и ссылка на эту область присваивается переменной `a`. А переменная `len` (на которой основано свойство `Length`) устанавливается равной количеству элементов в массиве. Наконец, устанавливаются закрытые переменные `lowerBound` и `upperBound`.

В классе `RangeArray` затем реализуются свойства `Length` и `Error`. Ниже приведены их определения.

```
  // Свойство Length, предназначенное только для чтения.
  public int Length {
    get {
      return len;
    }
  }

  // Свойство Error, предназначенное только для чтения.
  public bool Error {
    get {
      return errflag;
    }
  }
```

Эти свойства аналогичны свойствам, используемым классом `FailSoftArray`, и их работа организована подобным образом.

Далее в классе `RangeArray` реализуется индексатор. Вот его определение:

```
// Это -- индексатор для класса RangeArray.
  public int this[int index] {
    // Это -- get-аксессор.
    get {
      if(ok(index)) {
        errflag = false;
        return a[index - lowerBound];
      } else {
        errflag = true;
        return 0;
      }
    }

    // Это -- set-аксессор.
    set {
      if(ok(index)) {
        a[index - lowerBound] = value;
        errflag = false;
      }
      else errflag = true;
```

```
    }
  }
```

Этот индексатор очень похож на индексатор класса `FailSoftArray`, но с одним важным отличием. Обратите внимание на выражение, которое служит в качестве значения индекса массива `a`.

```
index - lowerBound
```

Это выражение преобразует реальный индекс, переданный через параметр `index`, в "нормальный", т.е. в значение, которое имел бы индекс текущего элемента, если бы индексирование массива начиналась с нуля. Ведь только такое индексирование подходит для базового массива `a`. Это выражение работает при любом значении переменной `lowerBound`: положительном, отрицательном или нулевом.

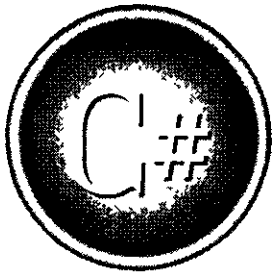
Теперь осталось рассмотреть определение метода `ok()`.

```
// Метод возвращает значение true, если индекс
// находится внутри границ.
private bool ok(int index) {
  if(index >= lowerBound & index <= upperBound)
    return true;
  return false;
}
```

Это определение аналогично тому, которое используется в классе `FailSoftArray` за исключением того, что попадание индекса в нужный диапазон проверяется с помощью значений переменных `lowerBound` и `upperBound`.

Класс `RangeArray` иллюстрирует только один вид массива, создаваемого "под заказ" с помощью индексаторов и свойств. Можно также создавать динамические массивы (которые расширяются и сокращаются по необходимости), ассоциативные и разреженные массивы. Попробуйте создать один из таких массивов в качестве упражнения.

Полный справочник по



Глава 11

Наследование

Наследование — один из трех фундаментальных принципов объектно-ориентированного программирования, поскольку именно благодаря ему возможно создание иерархических классификаций. Используя наследование, можно создать общий класс, который определяет характеристики, присущие множеству связанных элементов. Этот класс затем может быть унаследован другими, узкоспециализированными классами с добавлением в каждый из них своих, уникальных особенностей.

В языке C# класс, который наследуется, называется *базовым*. Класс, который наследует базовый класс, называется *производным*. Следовательно, производный класс — это специализированная версия базового класса. В производный класс, наследующий все переменные, методы, свойства, операторы и индексаторы, определенные в базовом классе, могут быть добавлены уникальные элементы.

Основы наследования

C# поддерживает наследование, позволяя в объявление класса встраивать другой класс. Это реализуется посредством задания базового класса при объявлении производного. Лучше всего начать с примера. Рассмотрим класс `TwoDShape`, в котором определяются атрибуты "обобщенной" двумерной геометрической фигуры (например, квадрата, прямоугольника, треугольника и т.д.).

```
// Класс двумерных объектов.
class TwoDShape {
  public double width;
  public double height;

  public void showDim() {
    Console.WriteLine("Ширина и высота равны " +
                      width + " и " + height);
  }
}
```

Класс `TwoDShape` можно использовать в качестве базового (т.е. как стартовую площадку) для классов, которые описывают специфические типы двумерных объектов. Например, в следующей программе класс `TwoDShape` используется для выведения класса `Triangle`. Обратите внимание на то, как объявляется класс `Triangle`.

```
// Простая иерархия классов.

using System;

// Класс двумерных объектов.
class TwoDShape {
  public double width;
  public double height;

  public void showDim() {
    Console.WriteLine("Ширина и высота равны " +
                      width + " и " + height);
  }
}

// Класс Triangle выводится из класса TwoDShape.
class Triangle : TwoDShape {
  public string style; // Тип треугольника.
```

```
  // Метод возвращает площадь треугольника.
  public double area() {
    return width * height / 2;
  }

  // Отображаем тип треугольника.
  public void showStyle() {
    Console.WriteLine("Треугольник " + style);
  }
}

class Shapes {
  public static void Main() {
    Triangle t1 = new Triangle();
    Triangle t2 = new Triangle();

    t1.width = 4.0;
    t1.height = 4.0;
    t1.style = "равнобедренный";

    t2.width = 8.0;
    t2.height = 12.0;
    t2.style = "прямоугольный";

    Console.WriteLine("Информация о t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Площадь равна " + t1.area());

    Console.WriteLine();

    Console.WriteLine("Информация о t2: ");
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Площадь равна " + t2.area());
  }
}
```

Вот результаты работы этой программы.

```
Информация о t1:
Треугольник равнобедренный
Ширина и высота равны 4 и 4
Площадь равна 8

Информация о t2:
Треугольник прямоугольный
Ширина и высота равны 8 и 12
Площадь равна 48
```

В классе `Triangle` создается специфический тип объекта класса `TwoDShape`, в данном случае треугольник. Класс `Triangle` содержит все элементы класса `TwoDShape` и, кроме того, поле `style`, метод `area()` и метод `showStyle()`. В переменной `style` хранится описание типа треугольника, метод `area()` вычисляет и возвращает его площадь, а метод `showStyle()` отображает данные о типе треугольника.

Ниже приведен синтаксис, который используется в объявлении класса `Triangle`, чтобы сделать его производным от класса `TwoDShape`.

```
class Triangle : TwoDShape {
```

Этот синтаксис можно обобщить. Если один класс наследует другой, то имя базового класса указывается после имени производного, причем имена классов разделяются двоеточием. В C# синтаксис наследования класса очень прост для запоминания и использования.

Поскольку класс `Triangle` включает все члены базового класса, `TwoDShape`, он может обращаться к членам `width` и `height` внутри метода `area()`. Кроме того, внутри метода `Main()` объекты `t1` и `t2` могут прямо ссылаться на члены `width` и `height`, как если бы они были частью класса `Triangle`. Включение класса `TwoDShape` в класс `Triangle` схематически показано на рис. 11.1.

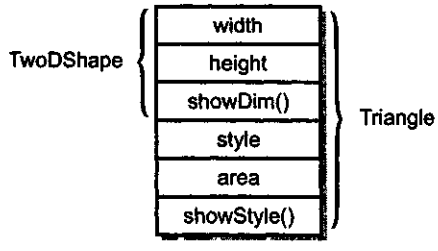


Рис. 11.1. Схематическое представление класса `Triangle`

Несмотря на то что класс `TwoDShape` является базовым для класса `Triangle`, это совершенно независимый и автономный класс. То, что его использует в качестве базового производный класс (классы), не означает невозможность использования его самого. Например, следующий фрагмент кода абсолютно легален:

```
TwoDShape shape = new TwoDShape();

shape.width = 10;
shape.height = 20;

shape.showDim();
```

Безусловно, объект класса `TwoDShape` "ничего не знает" и не имеет права доступа к классу, производному от `TwoDShape`.

Общая форма объявления класса, который наследует базовый класс, имеет такой вид:

```
class имя_производного_класса : имя_базового_класса {
  // тело класса
}
```

Для создаваемого производного класса можно указать только один базовый класс. В C# (в отличие от C++) не поддерживается наследование нескольких базовых классов в одном производном классе. Этот факт необходимо учитывать при переводе C++-кода на C#. Однако можно создать иерархию наследования, в которой один производный класс становится базовым для другого производного класса. И конечно же, ни один класс не может быть базовым (ни прямо, ни косвенно) для самого себя.

Основное достоинство наследования состоит в том, что, создав базовый класс, который определяет общие атрибуты для множества объектов, его можно использовать для создания любого числа более специализированных производных классов. В каждом производном классе можно затем точно "настроить" собственную классификацию. Вот, например, как из базового класса `TwoDShape` можно вывести производный класс, который инкапсулирует прямоугольники:

```
// Класс прямоугольников Rectangle, производный
// от класса TwoDShape.
class Rectangle : TwoDShape {
  // Метод возвращает значение true, если
  // прямоугольник является квадратом.
  public bool isSquare() {
    if(width == height) return true;
    return false;
  }

  // Метод возвращает значение площади прямоугольника.
  public double area() {
    return width * height;
  }
}
```

Класс `Rectangle` включает класс `TwoDShape` и добавляет метод `isSquare()`, который определяет, является ли прямоугольник квадратом, и метод `area()`, вычисляющий площадь прямоугольника.

Доступ к членам класса и наследование

Как разъяснялось в главе 8, члены класса часто объявляются закрытыми, чтобы предотвратить несанкционированное использование и внесение изменений. Наследование класса *не* отменяет ограничения, связанные с закрытым доступом. Таким образом, несмотря на то, что производный класс включает все члены базового класса, он не может получить доступ к тем из них, которые объявлены закрытыми. Например, как показано в следующем коде, если члены `width` и `height` являются `private`-членами в классе `TwoDShape`, то класс `Triangle` не сможет получить к ним доступ.

```
// Доступ к закрытым членам не наследуется.

// Этот пример не скомпилируется.
using System;

// Класс двумерных объектов.
class TwoDShape {
  double width;  // Теперь это private-член.
  double height; // Теперь это private-член.

  public void showDim() {
    Console.WriteLine("Ширина и высота равны " +
                      width + " и " + height);
  }
}

// Класс Triangle выводится из класса TwoDShape.
class Triangle : TwoDShape {
  public string style; // Тип треугольника.

  // Метод возвращает значение площади треугольника.
  public double area() {
    return width * height / 2; // Ошибка, нельзя получить
                               // прямой доступ к закрытым
                               // членам.
  }
```

```
  // Отображаем тип треугольника.
  public void showStyle() {
    Console.WriteLine("Треугольник " + style);
  }
}
```

Класс `Triangle` не скомпилируется, поскольку ссылка на члены `width` и `height` внутри метода `area()` вызовет ошибку нарушения прав доступа. Поскольку `width` и `height` — закрытые члены, они доступны только для членов их собственного класса. На производные классы эта доступность не распространяется.

Совет

Закрытый член класса остается закрытым в рамках этого класса. К нему нельзя получить доступ из кода, определенного вне этого класса, включая производные классы.

На первый взгляд может показаться, что невозможность доступа к закрытым членам базового класса со стороны производного — серьезное ограничение. Однако это не так, поскольку в C# предусмотрены возможности решения этой проблемы. Одна из них — `protected`-члены, о которых пойдет речь в следующем разделе. Вторая возможность — использование открытых свойств и методов, позволяющих получить доступ к закрытым данным. Как было показано в предыдущих главах, C#-программисты обычно предоставляют доступ к закрытым членам класса посредством открытых методов или путем превращения их в свойства. Перед вами — новая версия класса `TwoDShape`, в котором бывшие члены `width` и `height` стали свойствами.

```
// Использование свойств для записи и чтения закрытых
// членов класса.

using System;

// Класс двумерных объектов.
class TwoDShape {
  double pri_width;  // Теперь это private-член.
  double pri_height; // Теперь это private-член.

  // Свойства width и height.
  public double width {
     get { return pri_width; }
     set { pri_width = value; }
  }

  public double height {
     get { return pri_height; }
     set { pri_height = value; }
  }

  public void showDim() {
    Console.WriteLine("Ширина и высота равны " +
                      width + " и " + height);
  }
}

// Класс треугольников - производный от класса TwoDShape.
class Triangle : TwoDShape {
  public string style; // Тип треугольника.

  // Метод возвращает значение площади треугольника.
  public double area() {
    return width * height / 2;
```

```
  }

  // Отображаем тип треугольника.
  public void showStyle() {
    Console.WriteLine("Треугольник " + style);
  }
}

class Shapes2 {
  public static void Main() {
    Triangle t1 = new Triangle();
    Triangle t2 = new Triangle();

    t1.width = 4.0;
    t1.height = 4.0;
    t1.style = "равнобедренный";

    t2.width = 8.0;
    t2.height = 12.0;
    t2.style = "прямоугольный";

    Console.WriteLine("Информация о t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Площадь равна " + t1.area());

    Console.WriteLine();

    Console.WriteLine("Информация о t2: ");
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Площадь равна " + t2.area());
  }
}
```

Базовый и производный классы иногда называют *суперклассом* и *подклассом*. Эти термины пришли из программирования на языке Java. Суперкласс в Java — это базовый класс в C#. Подкласс в Java — это производный класс в C#. Вероятно, вам приходилось слышать эти термины, но мы будем придерживаться стандартных C#-терминов. В C++ также используются термины "базовый класс/производный класс".

Использование защищенного доступа

Как упоминалось выше, закрытый член базового класса недоступен для производного класса. Казалось бы, это означает, что, если производный класс должен иметь доступ к члену базового класса, его нужно сделать открытым. При этом придется смириться с тем, что открытый член будет доступным для любого другого кода, что иногда нежелательно. К счастью, таких ситуаций можно избежать, поскольку C# позволяет создавать *защищенные члены*. Защищенным является член, который открыт для своей иерархии классов, но закрыт вне этой иерархии.

Защищенный член создается с помощью модификатора доступа `protected`. При объявлении `protected`-члена он по сути является закрытым, но с одним исключением. Это исключение вступает в силу, когда защищенный член наследуется. В этом случае защищенный член базового класса становится защищенным членом производного класса, а следовательно, и доступным для производного класса. Таким образом, используя модификатор доступа `protected`, можно создавать закрытые (для "внеш-

него мира") члены класса, но вместе с тем они будут наследоваться с возможностью доступа со стороны производных классов.

Рассмотрим простой пример использования защищенных членов класса.

```
// Демонстрация использования защищенных членов класса.

using System;

class B {
  protected int i, j; // Закрыт внутри класса B,
                      // но доступен для класса D.

  public void set(int a, int b) {
    i = a;
    j = b;
  }

  public void show() {
    Console.WriteLine(i + " " + j);
  }
}

class D : B {
  int k; // Закрытый член.

  // Класс D получает доступ к членам i и j класса B.
  public void setk() {
     k = i * j;
  }

  public void showk() {
    Console.WriteLine(k);
  }
}

class ProtectedDemo {
  public static void Main() {
    D ob = new D();

    ob.set(2, 3); // OK, так как D "видит" B-члены i и j.
    ob.show();    // OK, так как D "видит" B-члены i и j.

    ob.setk();  // OK, так как это часть самого класса D.
    ob.showk(); // OK, так как это часть самого класса D.
  }
}
```

Поскольку в этом примере класс `B` наследуется классом `D` и члены `i` и `j` объявлены защищенными в классе `B` (т.е. с использованием модификатора доступа `protected`), метод `setk()` может получить к ним доступ. Если бы члены `i` и `j` были объявлены в классе `B` закрытыми, класс `D` не имел бы к ним права доступа, и программа не скомпилировалась бы.

Подобно модификаторам `public` и `private` модификатор `protected` остается со своим членом независимо от реализуемого количества уровней наследования. Следовательно, при использовании производного класса в качестве базового для создания другого производного класса любой защищенный член исходного базового класса, который наследуется первым производным классом, также наследуется в статусе защищенного и вторым производным классом.

Конструкторы и наследование

В иерархии классов как базовые, так и производные классы могут иметь собственные конструкторы. При этом возникает важный вопрос: какой конструктор отвечает за создание объекта производного класса? Конструктор базового или конструктор производного класса, или оба одновременно? Ответ таков: конструктор базового класса создает часть объекта, соответствующую базовому классу, а конструктор производного класса — часть объекта, соответствующую производному классу. И это вполне логично, потому что базовый класс "не видит" или не имеет доступа к элементам производного класса. Поэтому их конструкции должны быть раздельными. В предыдущих примерах классы опирались на конструкторы по умолчанию, создаваемые автоматически средствами C#, и поэтому мы не сталкивались с подобной проблемой. Но на практике большинство классов имеет конструкторы, и вы должны знать, как справляться с подобной ситуацией.

Если конструктор определяется только в производном классе, процесс создания объекта несложен: просто создается объект производного класса. Часть объекта, соответствующая базовому классу, создается автоматически с помощью конструктора по умолчанию. Например, рассмотрим переработанную версию класса `Triangle`, в которой определяется конструктор. Здесь член `style` объявлен `private`-членом, поскольку теперь он устанавливается конструктором.

```
// Добавление конструктора в класс Triangle.

using System;

// Класс двумерных объектов.
class TwoDShape {
  double pri_width;  // Закрытый член.
  double pri_height; // Закрытый член.

  // Свойства width и height.
  public double width {
     get { return pri_width; }
     set { pri_width = value; }
  }

  public double height {
     get { return pri_height; }
     set { pri_height = value; }
  }

  public void showDim() {
    Console.WriteLine("Ширина и высота равны " +
                      width + " и " + height);
  }
}

// Класс треугольников - производный от класса TwoDShape.
class Triangle : TwoDShape {
  string style; // Закрытый член.

  // Конструктор.
  public Triangle(string s, double w, double h) {
    width = w;  // Инициализирует член базового класса.
    height = h; // Инициализирует член базового класса.
```

```
    style = s;  // Инициализирует член своего класса.
  }

  // Метод возвращает значение площади треугольника.
  public double area() {
    return width * height / 2;
  }

  // Отображаем тип треугольника.
  public void showStyle() {
    Console.WriteLine("Треугольник " + style);
  }
}

class Shapes3 {
  public static void Main() {
    Triangle t1 = new Triangle("равнобедренный",
                               4.0, 4.0);
    Triangle t2 = new Triangle("прямоугольный",
                               8.0, 12.0);

    Console.WriteLine("Информация о t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Площадь равна " + t1.area());

    Console.WriteLine();

    Console.WriteLine("Информация о t2: ");
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Площадь равна " + t2.area());
  }
}
```

Здесь конструктор класса `Triangle` инициализирует наследуемые им члены класса `TwoDShape`, а также собственное поле `style`.

Если конструкторы определены и в базовом, и в производном классе, процесс создания объектов несколько усложняется, поскольку должны выполниться конструкторы обоих классов. В этом случае необходимо использовать еще одно ключевое слово C# `base`, которое имеет два назначения: вызвать конструктор базового класса и получить доступ к члену базового класса, который скрыт "за" членом производного класса. Сначала рассмотрим первое назначение слова `base`.

Вызов конструкторов базового класса

Производный класс может вызывать конструктор, определенный в его базовом классе, используя расширенную форму объявления конструктора производного класса и ключевое слово `base`. Формат расширенного объявления таков:

```
конструктор_производного_класса(
    список_параметров) : base(список_аргументов) {
  // тело конструктора
}
```

Здесь с помощью элемента *список_аргументов* задаются аргументы, необходимые конструктору в базовом классе.

Чтобы понять, как используется ключевое слово base, рассмотрим в следующей программе еще одну версию класса TwoDShape. В ней определяется конструктор, который инициализирует свойства width и height.

```
// Добавление конструкторов в класс TwoDShape.

using System;

// Класс двумерных объектов.
class TwoDShape {
  double pri_width;  // Закрытый член.
  double pri_height; // Закрытый член.

  // Конструктор класса TwoDShape.
  public TwoDShape(double w, double h) {
    width = w;
    height = h;
  }

  // Свойства width и height.
  public double width {
     get { return pri_width; }
     set { pri_width = value; }
  }

  public double height {
     get { return pri_height; }
     set { pri_height = value; }
  }

  public void showDim() {
    Console.WriteLine("Ширина и высота равны " +
                      width + " и " + height);
  }
}

 // Класс треугольников, производный от класса TwoDShape.
class Triangle : TwoDShape {
  string style; // Закрытый член.

  // Вызываем конструктор базового класса.
  public Triangle(string s,
                  double w,
                  double h) : base(w, h) {
    style = s;
  }

  // Метод возвращает площадь треугольника.
  public double area() {
    return width * height / 2;
  }

  // Отображаем тип треугольника.
  public void showStyle() {
    Console.WriteLine("Треугольник " + style);
  }
}
```

```
class Shapes4 {
  public static void Main() {
    Triangle t1 = new Triangle("равнобедренный", 4.0, 4.0);
    Triangle t2 = new Triangle("прямоугольный", 8.0, 12.0);

    Console.WriteLine("Информация о t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Площадь равна " + t1.area());

    Console.WriteLine();

    Console.WriteLine("Информация о t2: ");
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Площадь равна " + t2.area());
  }
}
```

Здесь конструктор `Triangle()` вызывает "метод" `base()` с параметрами `w` и `h`, что в действительности означает вызов конструктора `TwoDShape()`, который инициализирует свойства `width` и `height` значениями `w` и `h`, соответственно. Класс `Triangle` больше не инициализирует эти значения сам. Ему остается инициализировать только одно значение, уникальное для класса треугольников, а именно член `style` (тип треугольника). Такой подход дает классу `TwoDShape` свободу выбора среди возможных способов построения подобъектов. Более того, со временем класс `TwoDShape` может расширять свои функции, но об этом расширении ранее созданные производные классы не будут "знать", что предотвратит существующий код от разрушения.

С помощью ключевого слова `base` можно вызвать конструктор любой формы, определенный в базовом классе. Реально же выполнится тот конструктор, параметры которого будут соответствовать переданным при вызове аргументам. Например, вот как выглядят расширенные версии классов `TwoDShape` и `Triangle`, которые включают конструкторы по умолчанию и конструкторы, принимающие один аргумент:

```
// Добавляем в класс TwoDShape конструкторы.

using System;

class TwoDShape {
  double pri_width;  // Закрытый член.
  double pri_height; // Закрытый член.

  // Конструктор по умолчанию.
  public TwoDShape() {
    width = height = 0.0;
  }

  // Конструктор класса TwoDShape с параметрами.
  public TwoDShape(double w, double h) {
    width = w;
    height = h;
  }

  // Создаем объект, у которого ширина равна высоте.
  public TwoDShape(double x) {
    width = height = x;
  }
```

```
  // Свойства width и height.
  public double width {
     get { return pri_width; }
     set { pri_width = value; }
  }

  public double height {
     get { return pri_height; }
     set { pri_height = value; }
  }

  public void showDim() {
    Console.WriteLine("Ширина и высота равны " +
                      width + " и " + height);
  }
}

// Класс треугольников, производный от TwoDShape.
class Triangle : TwoDShape {
  string style; // Закрытый член.

  /* Конструктор по умолчанию. Он автоматически вызывает
     конструктор по умолчанию класса TwoDShape. */
  public Triangle() {
    style = "null";
  }

  // Конструктор, который принимает три аргумента.
  public Triangle(string s, double w, double h) : base(w, h) {
    style = s;
  }

  // Создаем равнобедренный треугольник.
  public Triangle(double x) : base(x) {
    style = "равнобедренный";
  }

  // Метод возвращает площадь треугольника.
  public double area() {
    return width * height / 2;
  }

  // Отображаем тип треугольника.
  public void showStyle() {
    Console.WriteLine("Треугольник " + style);
  }
}

class Shapes5 {
  public static void Main() {
    Triangle t1 = new Triangle();
    Triangle t2 = new Triangle("прямоугольный", 8.0, 12.0);
    Triangle t3 = new Triangle(4.0);

    t1 = t2;

    Console.WriteLine("Информация о t1: ");
    t1.showStyle();
```

```
    t1.showDim();
    Console.WriteLine("Площадь равна " + t1.area());

    Console.WriteLine();

    Console.WriteLine("Информация о t2: ");
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Площадь равна " + t2.area());

    Console.WriteLine();

    Console.WriteLine("Информация о t3: ");
    t3.showStyle();
    t3.showDim();
    Console.WriteLine("Площадь равна " + t3.area());

    Console.WriteLine();
  }
}
```

При выполнении этой версии программы получаем такие результаты:

```
Информация о t1:
Треугольник прямоугольный
Ширина и высота равны 8 и 12
Площадь равна 48

Информация о t2:
Треугольник прямоугольный
Ширина и высота равны 8 и 12
Площадь равна 48

Информация о t3:
Треугольник равнобедренный
Ширина и высота равны 4 и 4
Площадь равна 8
```

Рассмотрим ключевые концепции `base`-механизма. При задании производным классом `base`-"метода" вызывается конструктор непосредственного базового класса. Таким образом, ключевое слово `base` всегда отсылает к базовому классу, стоящему в иерархии классов непосредственно над вызывающим классом. Это справедливо и для многоуровневой иерархии. Чтобы передать аргументы конструктору базового класса, достаточно указать их в качестве аргументов "метода" `base()`. При отсутствии ключевого слова `base` автоматически вызывается конструктор базового класса, действующий по умолчанию.

Наследование и сокрытие имен

Производный класс может определить член, имя которого совпадает с именем члена базового класса. В этом случае член базового класса становится скрытым в производном классе. Поскольку с точки зрения формального синтаксиса языка C# эта ситуация не является ошибочной, компилятор выразит свое "недоумение" всего лишь предупреждающим сообщением. Это предупреждение должно послужить напоминанием о факте сокрытия имени. Если вы действительно собирались скрыть член базового класса, то для предотвращения этого предупреждения перед членом производ-

ного класса необходимо поставить ключевое слово new. Необходимо понимать, что эта функция слова new совершенно отличается от его использования при создании экземпляра объекта.

Рассмотрим пример сокрытия имени.

```
// Пример сокрытия имени в связи с наследованием.

using System;

class A {
  public int i = 0;
}

// Создаем производный класс.
class B : A {
  new int i; // Этот член i скрывает член i класса A.

  public B(int b) {
    i = b; // Член i в классе B.
  }

  public void show() {
    Console.WriteLine(
                "Член i в производном классе: " + i);
  }
}

class NameHiding {
  public static void Main() {
    B ob = new B(2);

    ob.show();
  }
}
```

Во-первых, обратите внимание на использование ключевого слова new при объявлении члена i в классе B. По сути, он сообщает компилятору о том, что вы знаете, что создается новая переменная с именем i, которая скрывает переменную i в базовом классе A. Если убрать слово new, компилятор сгенерирует предупреждающее сообщение.

Результаты выполнения этой программы выглядят так:

```
Член i в производном классе: 2
```

Поскольку в классе B определяется собственная переменная экземпляра с именем i, она скрывает переменную i, определенную в классе A. Следовательно, при вызове метода show() для объекта типа B, отображается значение переменной i, соответствующее ее определению в классе B, а не в классе A.

Использование ключевого слова base для доступа к скрытому имени

Существует вторая форма использования ключевого слова base, которая действует подобно ссылке this, за исключением того, что ссылка base всегда указывает на базовый класс производного класса, в котором она используется. В этом случае формат ее записи такой:

```
base.член
```

Здесь в качестве элемента *член* можно указывать либо метод, либо переменную экземпляра. Эта форма ссылки base наиболее применима в тех случаях, когда имя члена в производном классе скрывает член с таким же именем в базовом классе. Рассмотрим следующую версию иерархии классов из предыдущего примера:

```
// Использование ссылки base для доступа к скрытому имени.

using System;

class A {
  public int i = 0;
}

// Создаем производный класс.
class B : A {
  new int i; // Эта переменная i скрывает i класса A.

  public B(int a, int b) {
    base.i = a; // Так можно обратиться к i класса A.
    i = b; // Переменная i в классе B.
  }

  public void show() {
    // Эта инструкция отображает переменную i в классе A.
    Console.WriteLine("i в базовом классе: " + base.i);

    // Эта инструкция отображает переменную i в классе B.
    Console.WriteLine("i в производном классе: " + i);
  }
}

class UncoverName {
  public static void Main() {
    B ob = new B(1, 2);

    ob.show();
  }
}
```

Результаты выполнения этой программы выглядят так:

```
i в базовом классе: 1
i в производном классе: 2
```

Несмотря на то что переменная экземпляра i в классе B скрывает переменную i в классе A, ссылка base позволяет получить доступ к i в базовом классе.

С помощью ссылки base также можно вызывать скрытые методы. Рассмотрим пример.

```
// Вызов скрытого метода.

using System;

class A {
  public int i = 0;

  // Метод show() в классе A.
  public void show() {
    Console.WriteLine("i в базовом классе: " + i);
  }
```

```
}

// Создаем производный класс.
class B : A {
  new int i; // Эта переменная i скрывает
             // одноименную переменную класса A.

  public B(int a, int b) {
    base.i = a; // Так можно обратиться к
                // переменной i класса A.
    i = b; // Переменная i в классе B.
  }

  // Этот метод скрывает метод show(), определенный в
  // классе A. Обратите внимание на использование
  // ключевого слова new.
  new public void show() {
    base.show(); // Вызов метода show() класса A.

    // Отображаем значение переменной i класса B.
    Console.WriteLine("i в производном классе: " + i);
  }
}

class UncoverName {
  public static void Main() {
    B ob = new B(1, 2);

    ob.show();
  }
}
```

Вот результаты выполнения этой программы:

```
i в базовом классе: 1
i в производном классе: 2
```

Как видите, при вызове `base.show()` происходит обращение к версии метода `show()`, определенной в базовом классе.

Обратите внимание на то, что назначение ключевого слова `new` в этой программе — сообщить компилятору о том, что вы сознательно создаете в классе `B` новый метод с именем `show()`, который скрывает метод `show()`, определенный в классе `A`.

Создание многоуровневой иерархии

До сих пор мы использовали простые иерархии, состоящие только из базового и производного классов. Но можно построить иерархии, которые содержат любое количество уровней наследования. Как упоминалось выше, один производный класс вполне допустимо использовать в качестве базового для другого производного класса. Например, из трех классов (`A`, `B` и `C`) `C` может быть производным от `B`, который, в свою очередь, может быть производным от `A`. В подобной ситуации каждый производный класс наследует содержимое всех своих базовых классов. В данном случае класс `C` наследует все члены классов `B` и `A`.

Чтобы понять, какую пользу можно получить от многоуровневой иерархии, рассмотрим следующую программу. В ней производный класс `Triangle` используется в качестве базового для создания производного класса с именем `ColorTriangle`. Класс

ColorTriangle наследует все члены классов Triangle и TwoDShape и добавляет собственное поле color, которое содержит цвет треугольника.

```
// Многоуровневая иерархия.

using System;

class TwoDShape {
  double pri_width;  // Закрытый член.
  double pri_height; // Закрытый член.

  // Конструктор по умолчанию.
  public TwoDShape() {
    width = height = 0.0;
  }

  // Конструктор класса TwoDShape.
  public TwoDShape(double w, double h) {
    width = w;
    height = h;
  }

  // Конструктор, создающий объекты, у которых
  // ширина равна высоте.
  public TwoDShape(double x) {
    width = height = x;
  }

  // Свойства width и height.
  public double width {
     get { return pri_width; }
     set { pri_width = value; }
  }

  public double height {
     get { return pri_height; }
     set { pri_height = value; }
  }

  public void showDim() {
    Console.WriteLine("Ширина и высота равны " +
                      width + " и " + height);
  }
}

// Класс треугольников, производный от класса TwoDShape.
class Triangle : TwoDShape {
  string style; // Закрытый член.

  /* Конструктор по умолчанию. Он вызывает конструктор
     по умолчанию класса TwoDShape. */
  public Triangle() {
    style = "null";
  }

  // Конструктор с параметрами.
  public Triangle(string s, double w, double h) : base(w, h) {
    style = s;
```

```
  }

  // Создаем равнобедренный треугольник.
  public Triangle(double x) : base(x) {
    style = "равнобедренный";
  }

  // Метод возвращает значение площади треугольника.
  public double area() {
    return width * height / 2;
  }

  // Метод отображает тип треугольника.
  public void showStyle() {
    Console.WriteLine("Треугольник " + style);
  }
}

// Продолжаем иерархию классов треугольников.
class ColorTriangle : Triangle {
  string color;

  public ColorTriangle(
              string c, string s,
              double w, double h) : base(s, w, h) {
    color = c;
  }

  // Метод отображает цвет треугольника.
  public void showColor() {
    Console.WriteLine("Цвет " + color);
  }
}

class Shapes6 {
  public static void Main() {
    ColorTriangle t1 =
         new ColorTriangle("синий", "прямоугольный",
                           8.0, 12.0);
    ColorTriangle t2 =
         new ColorTriangle("красный", "равнобедренный",
                           2.0, 2.0);

    Console.WriteLine("Информация о t1: ");
    t1.showStyle();
    t1.showDim();
    t1.showColor();
    Console.WriteLine("Площадь равна " + t1.area());

    Console.WriteLine();

    Console.WriteLine("Информация о t2: ");
    t2.showStyle();
    t2.showDim();
    t2.showColor();
    Console.WriteLine("Площадь равна " + t2.area());
  }
}
```

При выполнении этой программы получаем следующие результаты.

```
Информация о t1:
Треугольник прямоугольный
Ширина и высота равны 8 и 12
Цвет синий
Площадь равна 48

Информация о t2:
Треугольник равнобедренный
Ширина и высота равны 2 и 2
Цвет красный
Площадь равна 2
```

Благодаря наследованию класс `ColorTriangle` может использовать ранее определенные классы `Triangle` и `TwoDShape`, добавляя только ту информацию, которая необходима для собственного (специального) применения. В этом и состоит ценность наследования: оно позволяет использовать код многократно.

Этот пример иллюстрирует еще один важный момент: `base` всегда ссылается на конструктор "ближайшего" производного класса. Так, в классе `ColorTriangle` ссылка `base` вызывает конструктор, определенный в классе `Triangle`. В классе `Triangle` ссылка `base` вызывает конструктор, определенный в классе `TwoDShape`. Если в иерархии классов конструктору базового класса требуются параметры, все производные классы должны передавать эти параметры, независимо от того, нужны ли эти параметры самому производному классу.

Последовательность вызова конструкторов

У читателя может возникнуть вопрос: какой конструктор выполнится первым при создании объекта производного класса — определенный в производном классе или в базовом? Например, если класс `B` — производный от класса `A`, то конструктор класса `A` будет вызван до вызова конструктора класса `B`, или наоборот? Ответ звучит так. В иерархии классов конструкторы вызываются в порядке выведения классов, т.е. начиная с конструктора базового класса и заканчивая конструктором производного класса. Более того, этот порядок не нарушается, независимо от использования ссылки `base`. Если ссылка `base` не используется, будут выполнены конструкторы по умолчанию (т.е. конструкторы без параметров) всех базовых классов. Порядок выполнения конструкторов демонстрируется в следующей программе:

```
// Демонстрация порядка выполнения конструкторов.

using System;

// Создаем базовый класс.
class A {
  public A() {
    Console.WriteLine("Создание класса A.");
  }
}

// Создаем класс, производный от A.
class B : A {
  public B() {
    Console.WriteLine("Создание класса B.");
  }
```

```
}

// Создаем класс, производный от B.
class C : B {
  public C() {
    Console.WriteLine("Создание класса C.");
  }
}

class OrderOfConstruction {
  public static void Main() {
    C c = new C();
  }
}
```

Вот результаты, сгенерированные программой:

```
Создание класса A.
Создание класса B.
Создание класса C.
```

Как видите, конструкторы вызываются в порядке выведения классов.

И в этом есть логика. Поскольку базовый класс "ничего не знает" о производном, то действия по инициализации, которые он должен выполнить, никак не связаны с существованием производного класса. Более того, они (действия) могут быть необходимы как обязательное условие (предпосылка) инициализации, выполняемой производным классом в форме вызова его конструктора. Потому-то конструктор базового класса выполняется первым.

Ссылки на базовый класс и объекты производных классов

Как вы знаете, C# — строго типизированный язык. За исключением стандартного и автоматического преобразований, которые применяются к простым типам, совместимость типов строго соблюдается. Следовательно, ссылочная переменная одного "классового" типа обычно не может ссылаться на объект другого "классового" типа. Рассмотрим, например, следующую программу:

```
// Эта программа не скомпилируется.

class X {
  int a;

  public X(int i) { a = i; }
}

class Y {
  int a;

  public Y(int i) { a = i; }
}

class IncompatibleRef {
  public static void Main() {
    X x = new X(10);
    X x2;
```

```
    Y y = new Y(5);

    x2 = x; // OK, обе переменные имеют одинаковый тип.

    x2 = y; // Ошибка, здесь переменные разного типа.
  }
}
```

Несмотря на то что здесь классы `X` и `Y` физически представляют собой одно и то же, невозможно присвоить объект класса `Y` ссылочной переменной типа `X`, поскольку они имеют разные типы. В общем случае ссылочная переменная может ссылаться только на объекты своего типа.

Однако существует важное исключение из C#-требования строгой совместимости типов. Ссылочную переменную базового класса можно присвоить ссылке на объект любого класса, выведенного из этого базового класса. Рассмотрим пример.

```
// Ссылка на базовый класс может указывать на
// объект производного класса.

using System;

class X {
  public int a;

  public X(int i) {
    a = i;
  }
}

class Y : X {
  public int b;

  public Y(int i, int j) : base(j) {
    b = i;
  }
}

class BaseRef {
  public static void Main() {
    X x = new X(10);
    X x2;
    Y y = new Y(5, 6);

    x2 = x; // OK, обе переменные имеют одинаковый тип.
    Console.WriteLine("x2.a: " + x2.a);

    x2 = y; // Все равно ok, поскольку класс Y
            // выведен из класса X.
    Console.WriteLine("x2.a: " + x2.a);

    // X-ссылки "знают" только о членах класса X.
    x2.a = 19; // OK
//    x2.b = 27; // Ошибка, в классе X нет члена b.
  }
}
```

На этот раз класс `Y` — производный от класса `X`, поэтому допустимо ссылке `x2` присвоить ссылку на объект класса `Y`.

Важно понимать, что именно тип ссылочной переменной (а не тип объекта, на который она ссылается) определяет, какие члены могут быть доступны. Другими словами, когда ссылка на производный класс присваивается ссылочной переменной базового класса, вы получаете доступ только к тем частям объекта, которые определены базовым классом. Вот почему ссылка `x2` не может получить доступ к члену `b` класса `Y` даже при условии, что она указывает на объект класса `Y`. И это вполне логично, поскольку базовый класс "не имеет понятия" о том, что добавил в свой состав производный класс. Поэтому последняя строка программы представлена как комментарий.

И хотя последний абзац может показаться несколько "эзотерическим", он имеет ряд важных практических приложений. Одно из них описано в этом разделе, а другое — ниже в этой главе при рассмотрении виртуальных методов.

Важность присвоения ссылок на производный класс ссылочным переменным базового класса ощущается в случае, когда в иерархии классов вызываются конструкторы. Как вы знаете, считается нормальным определить для класса конструктор, который в качестве параметра принимает объект своего класса. Это позволяет классу создать копию объекта. Классы, выведенные из такого класса, могут из этого факта извлечь определенную пользу. Рассмотрим, например, следующие версии классов `TwoDShape` и `Triangle`. В оба класса добавлены конструкторы, которые в качестве параметра принимают объект.

```
// Передача ссылки на производный класс
// ссылке на базовый класс.

using System;

class TwoDShape {
  double pri_width;  // Закрытый член.
  double pri_height; // Закрытый член.

  // Конструктор по умолчанию.
  public TwoDShape() {
    width = height = 0.0;
  }

  // Конструктор класса TwoDShape.
  public TwoDShape(double w, double h) {
    width = w;
    height = h;
  }

  // Создаем объект, в котором ширина равна высоте.
  public TwoDShape(double x) {
    width = height = x;
  }

  // Создаем объект из объекта.
  public TwoDShape(TwoDShape ob) {
    width = ob.width;
    height = ob.height;
  }

  // Свойства width и height.
  public double width {
     get { return pri_width; }
     set { pri_width = value; }
  }
```

```
  public double height {
     get { return pri_height; }
     set { pri_height = value; }
  }

  public void showDim() {
    Console.WriteLine("Ширина и высота равны " +
                      width + " и " + height);
  }
}

// Класс треугольников, производный от класса TwoDShape.
class Triangle : TwoDShape {
  string style; // Закрытый член.

  // Конструктор по умолчанию.
  public Triangle() {
    style = "null";
  }

  // Конструктор класса Triangle.
  public Triangle(string s,
                  double w,
                  double h) : base(w, h) {
    style = s;
  }

  // Создаем равнобедренный треугольник.
  public Triangle(double x) : base(x) {
    style = "равнобедренный";
  }

  // Создаем объект из объекта.
  public Triangle(Triangle ob) : base(ob) {
    style = ob.style;
  }

  // Метод возвращает площадь треугольника.
  public double area() {
    return width * height / 2;
  }

  // Метод отображает тип треугольника.
  public void showStyle() {
    Console.WriteLine("Треугольник " + style);
  }
}

class Shapes7 {
  public static void Main() {
    Triangle t1 = new Triangle("прямоугольный", 8.0, 12.0);

    // Создаем копию объекта t1.
    Triangle t2 = new Triangle(t1);

    Console.WriteLine("Информация о t1: ");
    t1.showStyle();
    t1.showDim();
```

```
    Console.WriteLine("Площадь равна " + t1.area());

    Console.WriteLine();

    Console.WriteLine("Информация о t2: ");
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Площадь равна " + t2.area());
  }
}
```

В этой программе объект `t2` создается из объекта `t1` и является идентичным ему. Вот результаты выполнения этой программы:

```
Информация о t1:
Треугольник прямоугольный
Ширина и высота равны 8 и 12
Площадь равна 48

Информация о t2:
Треугольник прямоугольный
Ширина и высота равны 8 и 12
Площадь равна 48
```

Обратите внимание на этот конструктор класса `Triangle`:

```
// Создаем объект из объекта.
  public Triangle(Triangle ob) : base(ob) {
    style = ob.style;
  }
```

Он принимает объект типа `Triangle` и передает его (посредством `base`-механизма) этому конструктору класса `TwoDShape`:

```
// Создаем объект из объекта.
  public TwoDShape(TwoDShape ob) {
    width = ob.width;
    height = ob.height;
  }
```

Ключевым моментом здесь является то, что конструктор `TwoDShape()` ожидает объект класса `TwoDShape`. Однако конструктор `Triangle()` передает ему объект класса `Triangle`. Как разъяснялось выше, такой "номер проходит" благодаря тому, что ссылка на базовый класс может указывать на объект производного класса. Следовательно, вполне допустимо передать конструктору `TwoDShape()` ссылку на объект класса, выведенного из класса `TwoDShape`. Поскольку конструктор `TwoDShape()` инициализирует только те части объекта производного класса, которые являются членами класса `TwoDShape`, не имеет значения, что объект может содержать и другие члены, добавленные производным классом.

Виртуальные методы и их переопределение

Виртуальным называется метод, объявляемый с помощью ключевого слова `virtual` в базовом классе и переопределяемый в одном или нескольких производных классах. Таким образом, каждый производный класс может иметь собственную версию виртуального метода. Виртуальные методы представляют интерес с такой позиции: что произойдет, если виртуальный метод будет вызван посредством ссылки на базовый класс. Какую именно версию метода нужно вызвать, C# определяет по *типу*

объекта, *на который указывает* эта ссылка, причем решение принимается динамически, *во время выполнения программы*. Следовательно, если имеются ссылки на *различные* объекты, будут выполняться *различные* версии виртуального метода. Другими словами, именно тип объекта, на который указывает ссылка (а не тип ссылки) определяет, какая версия виртуального метода будет выполнена. Таким образом, если базовый класс содержит виртуальный метод и из этого класса выведены производные классы, то при наличии ссылки на различные типы объектов (посредством ссылки на базовый класс) будут выполняться различные версии этого виртуального метода.

Чтобы объявить метод в базовом классе виртуальным, его объявление необходимо предварить ключевым словом `virtual`. При переопределении виртуального метода в производном классе используется модификатор `override`. Итак, процесс переопределения виртуального метода в производном классе иногда называется *замещением метода* (method overriding). При переопределении метода сигнатуры типа у виртуального и метода-заменителя должны совпадать. Кроме того, виртуальный метод нельзя определять как статический (с использованием слова `static`) или абстрактный (с использованием слова `abstract`, о котором пойдет речь ниже в этой главе).

Переопределение виртуального метода формирует базу для одной из самых мощных концепций C#: *динамической диспетчеризации методов*. Динамическая диспетчеризация методов — это механизм вызова переопределенного метода во время выполнения программы, а не в период компиляции. Именно благодаря механизму диспетчеризации методов в C# реализуется динамический полиморфизм.

Рассмотрим пример, который иллюстрирует виртуальные методы и их переопределение.

```
// Демонстрация виртуального метода.

using System;

class Base {
  // Создаем виртуальный метод в базовом классе.
  public virtual void who() {
    Console.WriteLine("Метод who() в классе Base.");
  }
}

class Derived1 : Base {
  // Переопределяем метод who() в производном классе.
  public override void who() {
    Console.WriteLine("Метод who() в классе Derived1");
  }
}

class Derived2 : Base {
  // Снова переопределяем метод who()
  // в другом производном классе.
  public override void who() {
    Console.WriteLine("Метод who() в классе Derived2");
  }
}

class OverrideDemo {
  public static void Main() {
    Base baseOb = new Base();
    Derived1 dOb1 = new Derived1();
    Derived2 dOb2 = new Derived2();
```

```
    Base baseRef; // Ссылка на базовый класс.

    baseRef = baseOb;
    baseRef.who();

    baseRef = dOb1;
    baseRef.who();

    baseRef = dOb2;
    baseRef.who();
  }
}
```

Вот результаты выполнения этой программы:

```
Метод who() в классе Base.
Метод who() в классе Derived1
Метод who() в классе Derived2
```

В программе создается базовый класс `Base` и два производных класса — `Derived1` и `Derived2`. В классе `Base` объявляется метод с именем `who()`, а производные классы его переопределяют. В методе `Main()` объявляются объекты типа `Base`, `Derived1` и `Derived2`, а также ссылка `baseRef` типа `Base`. Затем программа поочередно присваивает ссылку на объект каждого типа ссылке `baseRef` и использует эту ссылку для вызова метода `who()`. Как показывают результаты выполнения этой программы, нужная для выполнения версия определяется типом объекта, адресуемого в момент вызова, а не "классовым" типом ссылки `baseRef`.

Виртуальный метод переопределять необязательно. Если производный класс не предоставляет собственную версию виртуального метода, используется версия, определенная в базовом классе. Вот пример:

```
/* Если виртуальный метод не переопределен
   в производном классе, используется метод
   базового класса. */

using System;

class Base {
  // Создаем виртуальный метод в базовом классе.
  public virtual void who() {
    Console.WriteLine("Метод who() в классе Base");
  }
}

class Derived1 : Base {
  // Переопределяем метод who() в производном классе.
  public override void who() {
    Console.WriteLine("Метод who() в классе Derived1");
  }
}

class Derived2 : Base {
  // Этот класс не переопределяет метод who().
}

class NoOverrideDemo {
  public static void Main() {
    Base baseOb = new Base();
    Derived1 dOb1 = new Derived1();
```

```
    Derived2 dOb2 = new Derived2();

    Base baseRef; // Ссылка на базовый класс.

    baseRef = baseOb;
    baseRef.who();

    baseRef = dOb1;
    baseRef.who();

    baseRef = dOb2;
    baseRef.who(); // Вызывает метод who() класса Base.
  }
}
```

Вот результаты выполнения этой программы:

```
Метод who() в классе Base
Метод who() в классе Derived1
Метод who() в классе Base
```

Здесь класс `Derived2` не переопределяет метод `who()`. Поэтому при вызове метода `who()` для объекта класса `Derived2` выполняется метод `who()`, определенный в классе `Base`.

Если производный класс не переопределяет виртуальный метод в случае многоуровневой иерархии, то будет выполнен первый переопределенный метод, который обнаружится при просмотре иерархической лестницы в направлении снизу вверх. Рассмотрим пример.

```
/* Если производный класс не переопределяет виртуальный
   метод в случае многоуровневой иерархии, будет выполнен
   первый переопределенный метод, который обнаружится
   при просмотре иерархической лестницы в направлении
   снизу вверх. */

using System;

class Base {
  // Создаем виртуальный метод в базовом классе.
  public virtual void who() {
    Console.WriteLine("Метод who() в классе Base");
  }
}

class Derived1 : Base {
  // Переопределяем метод who() в производном классе.
  public override void who() {
    Console.WriteLine("Метод who() в классе Derived1");
  }
}

class Derived2 : Derived1 {
  // Этот класс не переопределяет метод who().
}

class Derived3 : Derived2 {
  // Этот класс также не переопределяет метод who().
}

class NoOverrideDemo2 {
```

```
  public static void Main() {
    Derived3 dOb = new Derived3();
    Base baseRef; // Ссылка на базовый класс.

    baseRef = dOb;
    baseRef.who(); // Вызывает метод who()
                   // из класса Derived1.
  }
}
```

Результаты выполнения этой программы таковы:

```
Метод who() в классе Derived1
```

Здесь класс `Derived3` наследует класс `Derived2`, который наследует класс `Derived1`, который в свою очередь наследует класс `Base`. Как подтверждают результаты выполнения этой программы, поскольку метод `who()` не переопределяется ни в классе `Derived3`, ни в классе `Derived2`, но переопределяется в классе `Derived1`, то именно эта версия метода `who()` (из класса `Derived1`) и выполняется, так как она является первой обнаруженной в иерархии классов.

Еще одно замечание. Свойства также можно модифицировать с помощью ключевого слова `virtual`, а затем переопределять с помощью ключевого слова `override`.

Зачем переопределять методы

Переопределение методов позволяет C# поддерживать динамический полиморфизм. Без полиморфизма объектно-ориентированное программирование невозможно, поскольку он позволяет исходному классу определять общие методы, которыми будут пользоваться все производные классы, и в которых при этом можно будет задать собственную реализацию некоторых или всех этих методов. Переопределенные методы представляют собой еще один способ реализации в C# аспекта полиморфизма, который можно выразить как "один интерфейс — много методов".

Ключ (вернее, его первый "поворот") к успешному применению полиморфизма лежит в понимании того, что базовые и производные классы образуют иерархию, которая развивается в сторону более узкой специализации. При корректном использовании базовый класс предоставляет производному классу все элементы "пригодными к употреблению", т.е. для прямого их использования. Кроме того, он определяет методы, которые производный класс должен реализовать самостоятельно. Это делает определение производными классами собственных методов более гибким, по-прежнему оставляя в силе требование согласующегося интерфейса. Таким образом, сочетая наследование с возможностью переопределения (замещения) методов, в базовом классе можно определить общую форму методов, которые будут использованы производными классами.

Применение виртуальных методов

Чтобы лучше почувствовать силу виртуальных методов, применим их к классу `TwoDShape`. В предыдущих примерах каждый класс, выведенный из класса `TwoDShape`, определяет метод с именем `area()`. Это наводит нас на мысль о том, не лучше ли сделать метод вычисления площади фигуры `area()` виртуальным в классе `TwoDShape`, получив возможность переопределить его в производных классах таким образом, чтобы он вычислял площадь согласно типу конкретной геометрической фигуры, которую инкапсулирует класс. Эта мысль и реализована в следующей программе. Для удобства в класс `TwoDShape` вводится свойство `name`, которое упрощает демонстрацию этих классов.

```
// Использование виртуальных методов и полиморфизма.

using System;

class TwoDShape {
  double pri_width;  // Закрытый член.
  double pri_height; // Закрытый член.
  string pri_name;   // Закрытый член.

  // Конструктор по умолчанию.
  public TwoDShape() {
    width = height = 0.0;
    name = "null";
  }

  // Конструктор с параметрами.
  public TwoDShape(double w, double h, string n) {
    width = w;
    height = h;
    name = n;
  }

  // Создаем объект, у которого ширина равна высоте.
  public TwoDShape(double x, string n) {
    width = height = x;
    name = n;
  }

  // Создаем объект из объекта.
  public TwoDShape(TwoDShape ob) {
    width = ob.width;
    height = ob.height;
    name = ob.name;
  }

  // Свойства width, height и name.
  public double width {
    get { return pri_width; }
    set { pri_width = value; }
  }

  public double height {
    get { return pri_height; }
    set { pri_height = value; }
  }

  public string name {
    get { return pri_name; }
    set { pri_name = value; }
  }

  public void showDim() {
    Console.WriteLine("Ширина и высота равны " +
                      width + " и " + height);
  }

  public virtual double area() {
    Console.WriteLine(
```

```
                "Метод area() необходимо переопределить.");
    return 0.0;
  }
}

// Класс треугольников, производный от класса TwoDShape.
class Triangle : TwoDShape {
  string style; // Закрытый член.

  // Конструктор по умолчанию.
  public Triangle() {
    style = "null";
  }

  // Конструктор с параметрами.
  public Triangle(string s, double w, double h) :
    base(w, h, "треугольник") {
      style = s;
  }

  // Создаем равнобедренный треугольник.
  public Triangle(double x) : base(x, "треугольник") {
    style = "равнобедренный";
  }

  // Создаем объект из объекта.
  public Triangle(Triangle ob) : base(ob) {
    style = ob.style;
  }

  // Переопределяем метод area() для класса Triangle.
  public override double area() {
    return width * height / 2;
  }

  // Метод отображает тип треугольника.
  public void showStyle() {
    Console.WriteLine("Треугольник " + style);
  }
}

// Класс прямоугольников, производный от класса TwoDShape.
class Rectangle : TwoDShape {
  // Конструктор с параметрами.
  public Rectangle(double w, double h) :
    base(w, h, "прямоугольник"){ }

  // Создаем квадрат.
  public Rectangle(double x) :
    base(x, "прямоугольник") { }

  // Создаем объект из объекта.
  public Rectangle(Rectangle ob) : base(ob) { }

  // Метод возвращает true, если прямоугольник - квадрат.
  public bool isSquare() {
    if(width == height) return true;
    return false;
```

```
  }

  // Переопределяем метод area() для класса Rectangle.
  public override double area() {
    return width * height;
  }
}

class DynShapes {
  public static void Main() {
    TwoDShape[] shapes = new TwoDShape[5];

    shapes[0] = new Triangle("прямоугольный", 8.0, 12.0);
    shapes[1] = new Rectangle(10);
    shapes[2] = new Rectangle(10, 4);
    shapes[3] = new Triangle(7.0);
    shapes[4] = new TwoDShape(10, 20,
                              "заготовка для фигуры");

    for(int i=0; i < shapes.Length; i++) {
      Console.WriteLine("Объектом является " +
                        shapes[i].name);
      Console.WriteLine("Площадь равна " +
                        shapes[i].area());

      Console.WriteLine();
    }
  }
}
```

При выполнении программа генерирует следующие результаты:

```
Объектом является треугольник
Площадь равна 48

Объектом является прямоугольник
Площадь равна 100

Объектом является прямоугольник
Площадь равна 40

Объектом является треугольник
Площадь равна 24.5

Объектом является заготовка для фигуры
Метод area() необходимо переопределить.
Площадь равна 0
```

Рассмотрим программу подробнее. Во-первых, метод `area()` объявляется в классе `TwoDShape` с использованием ключевого слова `virtual` и переопределяется в классах `Triangle` и `Rectangle`. В классе `TwoDShape` метод `area()` представляет собой своего рода "заглушку", которая просто информирует пользователя о том, что в производном классе этот метод необходимо переопределить. Каждое переопределение метода `area()` реализует вариант вычисления площади, соответствующий типу объекта, инкапсулируемому производным классом. Таким образом, если бы вы реализовали класс эллипсов, то метод `area()` в этом классе вычислял бы площадь эллипса.

В предыдущей программе проиллюстрирован еще один важный момент. Обратите внимание на то, что в методе `Main()` член `shapes` объявляется как массив объектов

типа TwoDShape. Однако элементам этого массива присваиваются ссылки на объекты классов Triangle, Rectangle и TwoDShape. Это вполне допустимо, поскольку ссылка на базовый класс может указывать на объект производного класса. Затем программа в цикле опрашивает массив shapes, отображая информацию о каждом объекте. Несмотря на простоту, этот цикл иллюстрирует силу как наследования, так и переопределения методов. Конкретный тип объекта, хранимый в ссылочной переменной базового класса, определяется во время выполнения программы, что позволяет принять соответствующие меры, т.е. выполнить действия, соответствующие объекту данного типа. Если объект выведен из класса TwoDShape, его площадь можно узнать посредством вызова метода area(). Интерфейс для выполнения этой операции одинаков для всех производных классов, независимо от типа используемой фигуры.

Использование абстрактных классов

Иногда полезно создать базовый класс, определяющий только своего рода "пустой бланк", который унаследуют все производные классы, причем каждый из них заполнит этот "бланк" собственной информацией. Такой класс определяет "суть" методов, которые производные классы должны реализовать, но сам при этом не обеспечивает реализации одного или нескольких методов. Подобная ситуация может возникнуть, когда базовый класс попросту не в состоянии реализовать метод. Этот случай был проиллюстрирован версией класса TwoDShape (из предыдущей программы), в которой определение метода area() представляло собой "заглушку", поскольку в нем площадь фигуры не вычислялась и, естественно, не отображалась.

В будущем, создавая собственные библиотеки классов, вы убедитесь, что отсутствие у метода четкого определения в контексте своего (базового) класса, не является чем-то необычным. Описанную ситуацию можно обработать двумя способами. Один из них, который продемонстрирован в предыдущем примере, — вывод предупреждающего сообщения. И хотя такой подход может быть полезным в определенных обстоятельствах (например, при отладке программы), все же он не соответствует уровню профессионального программирования. Существует и другой способ. Наша цель — заставить производные классы переопределить методы, которые в базовом классе не имеют никакого смысла. Рассмотрим класс Triangle. Им нельзя пользоваться, если не определен метод area(). Необходимо иметь средство, благодаря которому производный класс обязательно переопределит все необходимые методы. Этим средством в C# является *абстрактный метод*.

Абстрактный метод создается с помощью модификатора типа abstract. Абстрактный метод не содержит тела и, следовательно, не реализуется базовым классом. Поэтому производный класс *должен* его переопределить, поскольку он не может использовать версию, предложенную в базовом классе. Нетрудно догадаться, что абстрактный метод автоматически является виртуальным, поэтому и нет необходимости в использовании модификатора virtual. Более того, совместное использование модификаторов virtual и abstract считается ошибкой.

Для объявления абстрактного метода используйте следующий формат записи.

```
abstract тип имя(список_параметров);
```

Как видите, тело абстрактного метода отсутствует. Модификатор abstract можно использовать только применительно к обычным, а не к static-методам. Свойства также могут быть абстрактными.

Класс, содержащий один или несколько абстрактных методов, также должен быть объявлен как абстрактный с помощью спецификатора abstract, который ставится перед объявлением class. Поскольку абстрактный класс нереализуем в полном объе-

ме, невозможно создать его экземпляры, или объекты. Таким образом, попытка создать объект абстрактного класса с помощью оператора new приведет к возникновению ошибки времени компиляции.

Если производный класс выводится из абстрактного, он может реализовать все абстрактные методы базового класса. В противном случае такой производный класс также должен быть определен как абстрактный. Таким образом, атрибут abstract наследуется до тех пор, пока реализация класса не будет полностью достигнута.

Используя абстрактный класс, можно усовершенствовать определение класса TwoDShape. Поскольку для не определенной заранее двумерной фигуры понятие площади не имеет смысла, в следующей версии предыдущей программы метод area() в классе TwoDShape объявляется как абстрактный, как, впрочем, и сам класс TwoDShape. Безусловно, это означает, что все классы, выведенные из TwoDShape, должны переопределить метод area().

```
// Создание абстрактного класса.

using System;

abstract class TwoDShape {
  double pri_width;  // Закрытый член.
  double pri_height; // Закрытый член.
  string pri_name;   // Закрытый член.

  // Конструктор по умолчанию.
  public TwoDShape() {
    width = height = 0.0;
    name = "null";
  }

  // Конструктор с параметрами.
  public TwoDShape(double w, double h, string n) {
    width = w;
    height = h;
    name = n;
  }

  // Создаем объект, у которого ширина равна высоте.
  public TwoDShape(double x, string n) {
    width = height = x;
    name = n;
  }

  // Создаем объект из объекта.
  public TwoDShape(TwoDShape ob) {
    width = ob.width;
    height = ob.height;
    name = ob.name;
  }

  // Свойства width, height и name.
  public double width {
    get { return pri_width; }
    set { pri_width = value; }
  }

  public double height {
    get { return pri_height; }
    set { pri_height = value; }
```

```
  }

  public string name {
    get { return pri_name; }
    set { pri_name = value; }
  }

  public void showDim() {
    Console.WriteLine("Ширина и высота равны " +
                      width + " и " + height);
  }

  // Теперь метод area() абстрактный.
  public abstract double area();
}

// Класс треугольников, производный от класса TwoDShape.
class Triangle : TwoDShape {
  string style; // Закрытый член.

  // Конструктор по умолчанию.
  public Triangle() {
    style = "null";
  }

  // Конструктор с параметрами.
  public Triangle(string s, double w, double h) :
    base(w, h, "triangle") {
      style = s;
  }

  // Создаем равнобедренный треугольник.
  public Triangle(double x) : base(x, "треугольник") {
    style = "равнобедренный";
  }

  // Создаем объект из объекта.
  public Triangle(Triangle ob) : base(ob) {
    style = ob.style;
  }

  // Переопределяем метод area() для класса Triangle.
  public override double area() {
    return width * height / 2;
  }

  // Отображаем тип треугольника.
  public void showStyle() {
    Console.WriteLine("Треугольник " + style);
  }
}

// Класс прямоугольников, производный от класса TwoDShape.
class Rectangle : TwoDShape {
  // Конструктор с параметрами.
  public Rectangle(double w, double h) :
    base(w, h, "прямоугольник"){ }

  // Создаем квадрат.
```

```
  public Rectangle(double x) :
    base(x, "прямоугольник") { }

  // Создаем объект из объекта.
  public Rectangle(Rectangle ob) : base(ob) { }

  // Метод возвращает значение true, если
  // прямоугольник является квадратом.
  public bool isSquare() {
    if(width == height) return true;
    return false;
  }

  // Переопределяем метод area() для класса Rectangle.
  public override double area() {
    return width * height;
  }
}

class AbsShape {
  public static void Main() {
    TwoDShape[] shapes = new TwoDShape[4];

    shapes[0] = new Triangle("прямоугольный", 8.0, 12.0);
    shapes[1] = new Rectangle(10);
    shapes[2] = new Rectangle(10, 4);
    shapes[3] = new Triangle(7.0);

    for(int i=0; i < shapes.Length; i++) {
      Console.WriteLine("Объектом является " +
                        shapes[i].name);
      Console.WriteLine("Площадь равна " +
                        shapes[i].area());

      Console.WriteLine();
    }
  }
}
```

Как продемонстрировано этой программой, все производные классы *должны* или переопределить метод `area()`, или также объявить себя абстрактными. Чтобы убедиться в этом, попробуйте создать производный класс, который не переопределяет метод `area()`. Вы тут же (т.е. во время компиляции) получите сообщение об ошибке. Конечно, мы можем создать объектную ссылку типа `TwoDShape`, что и делается в программе. Однако теперь нельзя объявить объект типа `TwoDShape`. Поэтому в методе `Main()` размер массива `shapes` сокращен до 4, и больше не создается "заготовка для фигуры" в виде объекта класса `TwoDShape`.

Обратите также внимание на то, что класс `TwoDShape` по-прежнему включает метод `showDim()`, объявления которого не коснулся модификатор `abstract`. Ведь вполне допустимо для абстрактного класса содержать конкретные (а не только абстрактные) методы, которые производный класс может использовать "как есть". И только методы, объявленные с использованием ключевого слова `abstract`, должны переопределяться производными классами.

Использование ключевого слова sealed для предотвращения наследования

Каким бы мощным и полезным ни был механизм наследования, все же иногда необходимо его отключать. Например, у вас может быть класс, который инкапсулирует последовательность действий при инициализации такого специализированного устройства, как медицинский монитор. В этом случае необходимо запретить пользователям изменять характер инициализации этого монитора, чтобы исключить возможную некорректность этой процедуры. Специально для подобных ситуаций в C# предусмрена возможность предотвращения наследования класса с помощью ключевого слова `sealed`.

Чтобы запретить наследование класса, предварите его объявление ключевым словом `sealed`. Нетрудно догадаться, что нельзя объявлять класс одновременно с помощью двух модификаторов — `abstract` и `sealed`, поскольку абстрактный класс сам по себе "полуфабрикат" и его полная реализация возможна только в следующих "поколениях", т.е. после создания производных классов.

Рассмотрим пример `sealed`-класса.

```
sealed class A {
  // ...
}

// Следующий класс создать невозможно.
class B : A { // ОШИБКА! Класс A не может иметь наследников.
  // ...
}
```

Класс `B` не может быть производным от класса `A`, так как последний объявлен `sealed`-классом.

Класс object

В C# определен специальный класс с именем `object`, который является неявным базовым классом всех других классов и типов (включая типы значений). Другими словами, все остальные типы выводятся из класса `object`. Это означает, что ссылочная переменная типа `object` может указывать на объект любого типа. Кроме того, поскольку C#-массивы реализованы как классы, переменная типа `object` также может ссылаться на любой массив. Строго говоря, C#-имя `object` — еще одно имя для класса `System.Object`, который является частью библиотеки классов .NET Framework.

Класс `object` определяет методы, перечисленные в табл. 11.1. Эти методы доступны для каждого объекта.

Таблица 11.1. Методы класса `object`

Метод	Назначение
`public virtual bool Equals(object ob)`	Определяет, является ли вызывающий объект таким же, как объект, адресуемый ссылкой *ob*
`public static bool Equals(object ob1, object ob2)`	Определяет, является ли объект *ob1* таким же, как объект *ob2*

Окончание табл. 11.1

Метод	Назначение
`protected Finalize()`	Выполняет завершающие действия перед процессом сбора мусора. В C# метод `Finalize()` доступен через деструктор
`public virtual int GetHashCode()`	Возвращает хеш-код, связанный с вызывающим объектом
`public Type GetType()`	Получает тип объекта во время выполнения программы
`protected object MemberwiseClone()`	Выполняет "поверхностное копирование" объекта, т.е. копируются члены, но не объекты, на которые ссылаются эти члены
`public static bool ReferenceEquals(object ob1, object ob2)`	Определяет, ссылаются ли объекты *ob1* и *ob2* на один и тот же объект
`public virtual string ToString()`	Возвращает строку, которая описывает объект

Назначение некоторых из перечисленных выше методов требует дополнительных разъяснений. По умолчанию метод `Equals(object)` определяет, ссылаются ли вызывающий объект и объект, адресуемый аргументом, на один и тот же объект, т.е. метод определяет, являются ли эти две ссылки одинаковыми. Метод возвращает значение `true`, если объекты совпадают, и `false` — в противном случае. Этот метод можно переопределить в создаваемых им классах. Это позволит уточнить, что означает равенство для вашего класса. Например, вы можете так определить метод `Equals(object)`, чтобы он сравнивал содержимое двух объектов (и давал ответ на вопрос, равны ли они). Метод `Equals(object, object)` для получения результата вызывает метод `Equals(object)`.

Метод `GetHashCode()` возвращает хеш-код, связанный с вызывающим объектом. Этот хеш-код можно использовать с любым алгоритмом, который применяет хеширование как средство доступа к объектам, хранимым в памяти.

Как упоминалось в главе 9, при перегрузке оператора "==" необходимо переопределить методы `Equals(object)` и `GetHashCode()`, поскольку функции оператора "==" и метода `Equals(object)`, как правило, должны быть идентичными. Переопределив метод `Equals(object)`, необходимо переопределить и метод `GetHashCode()`, чтобы они были совместимы.

Метод `ToString()` возвращает строку, содержащую описание объекта, для которого вызывается этот метод. Кроме того, метод `ToString()` автоматически вызывается при выводе объекта с помощью метода `WriteLine()`. Метод `ToString()` переопределяется во многих классах, что позволяет подобрать описание специально для типов объектов, которые они создают. Вот пример:

```
// Демонстрация использования метода ToString().

using System;

class MyClass {
  static int count = 0;
  int id;

  public MyClass() {
    id = count;
    count++;
  }

  public override string ToString() {
    return "Объект класса MyClass #" + id;
```

```
  }
}

class Test {
  public static void Main() {
    MyClass ob1 = new MyClass();
    MyClass ob2 = new MyClass();
    MyClass ob3 = new MyClass();

    Console.WriteLine(ob1);
    Console.WriteLine(ob2);
    Console.WriteLine(ob3);

  }
}
```

Эта программа генерирует следующие результаты:

```
Объект класса MyClass #0
Объект класса MyClass #1
Объект класса MyClass #2
```

Приведение к объектному типу и восстановление значения

Как упоминалось выше, все C#-типы, включая типы значений, выведены из класса `object`. Следовательно, ссылку типа `object` можно использовать в качестве ссылки на любой другой тип, включая типы значений. Если ссылку типа `object` заставляют указывать на значение нессылочного типа, этот процесс называют *приведением к объектному типу* (boxing). В результате этого процесса значение нессылочного типа должно храниться подобно объекту, или экземпляру класса. Другими словами, "необъектное" значение помещается в объектную оболочку. Такой "необъектный" объект можно затем использовать подобно любому другому объекту. В любом случае приведение к объектному типу происходит автоматически. Для этого достаточно присвоить значение ссылке на объект класса `object`. Все остальное доделает C#.

Восстановление значения из "объектного образа" (unboxing) — это по сути процесс извлечения значения из объекта. Это действие выполняется с помощью операции приведения типа, т.е. приведения ссылки на объект класса `object` к значению желаемого типа.

Рассмотрим простой пример, который иллюстрирует приведение значения к объектному типу и его восстановление.

```
// Простой пример "объективизации" и "дезобъективизации".
using System;

class BoxingDemo {
  public static void Main() {
    int x;
    object obj;

    x = 10;
    obj = x; // "Превращаем" x в объект.

    int y = (int)obj; // Обратное "превращение"
                      // объекта obj в int-значение.
    Console.WriteLine(y);
  }
}
```

Эта программа отображает значение 10. Обратите внимание на то, что значение переменной x приводится к объектному типу простым ее присваиванием переменной obj, которая является ссылкой на объект типа object. Целочисленное значение, хранимое в ссылочной переменной obj, извлекается с помощью операции приведения к типу int.

А вот еще один (более интересный) пример приведения значения к объектному типу. На этот раз int-значение передается в качестве аргумента методу sqr(), который использует параметр типа object.

```
// Приведение значений к объектному типу возможно
// при передаче значений методам.

using System;

class BoxingDemo {
  public static void Main() {
    int x;

    x = 10;
    Console.WriteLine("Значение x равно: " + x);

    // Переменная x автоматически приводится
    // к объектному типу при передаче методу sqr().
    x = BoxingDemo.sqr(x);
    Console.WriteLine("Значение x в квадрате равно: " + x);
  }

  static int sqr(object o) {
    return (int)o * (int)o;
  }
}
```

При выполнении этой программы получаются такие результаты:

```
Значение x равно: 10
Значение x в квадрате равно: 100
```

Здесь значение x при передаче методу sqr() автоматически приводится к объектному типу.

Приведение к объектному типу и обратный процесс делает C#-систему типов полностью унифицированной. Все типы выводятся из класса object. Ссылке типа object можно присвоить ссылку на любой тип. В процессе приведения значения к объектному типу и его восстановления из объекта автоматически обрабатываются все детали, соответствующие нессылочным типам. Более того, поскольку все типы выведены из класса object, все они имеют доступ к методам этого класса. Рассмотрим, например, следующую маленькую, но удивительную программу.

```
// "Объективизация" позволяет вызывать методы
// класса object для значений нессылочного типа!

using System;

class MethOnValue {
  public static void Main() {

    Console.WriteLine(10.ToString());

  }
}
```

Эта программа отображает значение 10. Дело в том, что метод `ToString()` возвращает строковое представление объекта, для которого он вызывается. В данном случае строковое представление числа 10 выглядит как 10!

Использование класса `object` в качестве обобщенного типа данных

С учетом того, что `object` — базовый класс для всех остальных C#-типов и что приведение значения к объектному типу и его восстановление из объекта происходят автоматически, класс `object` можно использовать в качестве обобщенного типа данных. Например, рассмотрим следующую программу, которая создает массив объектов класса `object`, а затем присваивает его элементам данные различных типов.

```
// Использование класса object для создания
// массива обобщенного типа.

using System;

class GenericDemo {
  public static void Main() {
    object[] ga = new object[10];

    // Сохраняем int-значения.
    for(int i=0; i < 3; i++)
      ga[i] = i;

    // Сохраняем double-значения.
    for(int i=3; i < 6; i++)
      ga[i] = (double) i / 2;


    // Сохраняем две строки, bool- и char-значения.
    ga[6] = "Массив обобщенного типа";
    ga[7] = true;
    ga[8] = 'X';
    ga[9] = "Конец";

    for(int i = 0; i < ga.Length; i++)
      Console.WriteLine("ga[" + i + "]: " + ga[i] + " ");

  }
}
```

Вот результаты выполнения этой программы:

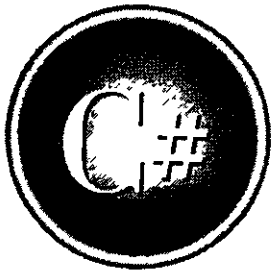
```
ga[0]: 0
ga[1]: 1
ga[2]: 2
ga[3]: 1.5
ga[4]: 2
ga[5]: 2.5
ga[6]: Массив обобщенного типа
ga[7]: True
ga[8]: X
ga[9]: Конец
```

Как видно по результатам выполнения программы, ссылку на объект класса `object` можно использовать в качестве ссылки на данные других типов. Таким обра-

зом, массив типа object в этой программе может хранить значения любого типа. Это говорит о том, что object-массив по сути представляет собой обобщенный список. Развивая эту идею, нетрудно создать класс стека, который бы хранил ссылки на объекты класса object. Это позволило бы стеку сохранять данные любого типа.

Несмотря на эффективность такого обобщенного типа, как класс object, в некоторых ситуациях, было бы ошибкой думать, что класс object следует использовать как способ обойти строгий C#-контроль соответствия типов. Другими словами, для хранения int-значения используйте int-переменную, для хранения string-значения — string-ссылку и т.д. Оставьте обобщенную природу класса object для особых ситуаций.

Полный справочник по



Глава 12

Интерфейсы, структуры и перечисления

Эта глава посвящена одному из самых важных средств языка C#: интерфейсу. *Интерфейс* определяет набор методов, которые будут реализованы классом. Сам интерфейс не реализует методы. Таким образом, интерфейс — это логическая конструкция, которая описывает методы, не устанавливая жестко способ их реализации. В этой главе рассматриваются еще два типа данных C#: структуры и перечисления. *Структуры* подобны классам, за исключением того, что они обрабатываются как типы значений, а не как ссылочные типы. *Перечисления* — это списки именованных целочисленных констант. Структуры и перечисления вносят существенный вклад в общую копилку средств и инструментов, составляющих среду программирования C#.

Интерфейсы

В объектно-ориентированном программировании иногда требуется определить, *что* класс должен делать, а не *как* он будет это делать. Вы уже видели такой подход на примере абстрактного метода. Абстрактный метод определяет сигнатуру для метода, но не обеспечивает его реализации. В производном классе каждый абстрактный метод, определенный базовым классом, реализуется по-своему. Таким образом, абстрактный метод задает *интерфейс* для метода, но не способ его *реализации*. Несмотря на всю полезность абстрактных классов и методов, эту идею можно развить. В C# предусмотрена возможность полностью отделить интерфейс класса от его реализации с помощью ключевого слова `interface`.

Интерфейсы синтаксически подобны абстрактным классам. Однако в интерфейсе ни один метод не может включать тело, т.е. интерфейс в принципе не предусматривает какой бы то ни было реализации. Он определяет, *что* должно быть сделано, но не уточняет, *как*. Коль скоро интерфейс определен, его может реализовать любое количество классов. При этом один класс может реализовать любое число интерфейсов.

Для реализации интерфейса класс должен обеспечить тел*а* (способы реализации) методов, описанных в интерфейсе. Каждый класс может определить собственную реализацию. Таким образом, два класса могут реализовать один и тот же интерфейс различными способами, но все классы поддерживают одинаковый набор методов. Следовательно, код, "осведомленный" о наличии интерфейса, может использовать объекты любого класса, поскольку интерфейс для всех объектов одинаков. Предоставляя программистам возможность применения такого средства программирования, как интерфейс, C# позволяет в полной мере использовать аспект полиморфизма, выражаемый как "один интерфейс — много методов".

Интерфейсы объявляются с помощью ключевого слова `interface`. Вот как выглядит упрощенная форма объявления интерфейса:

```
interface имя{
  тип_возврата имя_метода1(список_параметров);
  тип_возврата имя_метода2(список_параметров);
  // ...
  тип_возврата имя_методаN(список_параметров);
}
```

Имя интерфейса задается элементом *имя*. Методы объявляются с использованием лишь типа возвращаемого ими значения и сигнатуры. Все эти методы, по сути, — абстрактные. Как упоминалось выше, для методов в интерфейсе не предусмотрены способы реализации. Следовательно. каждый класс, который включает интерфейс, должен реализовать все его методы. В интерфейсе методы неявно являются открытыми (`public`-методами), при этом не разрешается явным образом указывать спецификатор доступа.

Рассмотрим пример интерфейса для класса, который генерирует ряд чисел.

```
public interface ISeries {
  int getNext(); // Возвращает следующее число ряда.
  void reset();  // Выполняет перезапуск.
  void setStart(int x); // Устанавливает начальное
                        // значение.
}
```

Этот интерфейс имеет имя `ISeries`. Хотя префикс "I" необязателен, многие программисты его используют, чтобы отличать интерфейсы от классов. Интерфейс `ISeries` объявлен открытым, поэтому он может быть реализован любым классом в любой программе.

Помимо сигнатур методов интерфейсы могут объявлять сигнатуры свойств, индексаторов и событий. События рассматриваются в главе 15, поэтому здесь мы остановимся на методах, индексаторах и свойствах. Интерфейсы не могут иметь членов данных. Они не могут определять конструкторы, деструкторы или операторные методы. Кроме того, ни один член интерфейса не может быть объявлен статическим.

Реализация интерфейсов

Итак, если интерфейс определен, один или несколько классов могут его реализовать. Чтобы реализовать интерфейс, нужно указать его имя после имени класса подобно тому, как при создании производного указывается базовый класс. Формат записи класса, который реализует интерфейс, таков:

```
class имя_класса : имя_интерфейса {
  // тело класса
}
```

Нетрудно догадаться, что имя реализуемого интерфейса задается с помощью элемента *имя_интерфейса*.

Если класс реализует интерфейс, он должен это сделать в полном объеме, т.е. реализация интерфейса не может быть выполнена частично.

Классы могут реализовать несколько интерфейсов. В этом случае имена интерфейсов отделяются запятыми. Класс может наследовать базовый класс и реализовать один или несколько интерфейсов. В этом случае список интерфейсов должно возглавлять имя базового класса.

Методы, которые реализуют интерфейс, должны быть объявлены открытыми. Дело в том, что методы внутри интерфейса неявно объявляются открытыми, поэтому их реализации также должны быть открытыми. Кроме того, сигнатура типа в реализации метода должна в точности совпадать с сигнатурой типа, заданной в определении интерфейса.

Рассмотрим пример реализации интерфейса `ISeries`, объявление которого приведено выше. Здесь создается класс с именем `ByTwos`, генерирующий ряд чисел, в котором каждое следующее число больше предыдущего на два.

```
// Реализация интерфейса ISeries.
class ByTwos : ISeries {
  int start;
  int val;

  public ByTwos() {
    start = 0;
    val = 0;
  }

  public int getNext() {
```

```
    val += 2;
    return val;
  }

  public void reset() {
    val = start;
  }

  public void setStart(int x) {
    start = x;
    val = start;
  }
}
```

Как видите, класс `ByTwos` реализует все три метода, определенные интерфейсом `ISeries`. Иначе и быть не может, поскольку классу не разрешается создавать частичную реализацию интерфейса.

Рассмотрим пример, демонстрирующий использование класса `ByTwos`. Вот его код:

```
// Демонстрация использования интерфейса,
// реализованного классом ByTwos.

using System;

class SeriesDemo {
  public static void Main() {
    ByTwos ob = new ByTwos();

    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее значение равно " +
                        ob.getNext());

    Console.WriteLine("\nПереход в исходное состояние.");
    ob.reset();
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее значение равно " +
                        ob.getNext());

    Console.WriteLine("\nНачинаем с числа 100.");
    ob.setStart(100);
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее значение равно " +
                        ob.getNext());
  }
}
```

Чтобы скомпилировать программу `SeriesDemo`, необходимо включить в процесс компиляции файлы, которые содержат классы `ISeries`, `ByTwos` и `SeriesDemo`. Для создания выполняемой программы компилятор автоматически скомпилирует все три файла. Если эти файлы называются, например, `ISeries.cs`, `ByTwos.cs` и `SeriesDemo.cs`, то программа скомпилируется посредством выполнения такой командной строки:

```
>csc SeriesDemo.cs ISeries.cs ByTwos.cs
```

Если вы используете интегрированную среду (IDE) Visual Studio, добавьте все эти три файла в свой C#-проект. Вполне допустимо также поместить их в один файл.

Вот результаты выполнения этой программы:

```
Следующее значение равно 2
Следующее значение равно 4
Следующее значение равно 6
Следующее значение равно 8
Следующее значение равно 10

Переход в исходное состояние.
Следующее значение равно 2
Следующее значение равно 4
Следующее значение равно 6
Следующее значение равно 8
Следующее значение равно 10

Начинаем с числа 100.
Следующее значение равно 102
Следующее значение равно 104
Следующее значение равно 106
Следующее значение равно 108
Следующее значение равно 110
```

В классах, которые реализуют интерфейсы, можно определять дополнительные члены. Например, в представленную ниже версию класса `ByTwos` добавлен метод `getPrevious()`, который возвращает предыдущее значение ряда.

```
// Реализация интерфейса ISeries с дополнительно
// определяемым методом getPrevious().

class ByTwos : ISeries {
  int start;
  int val;
  int prev;

  public ByTwos() {
    start = 0;
    val = 0;
    prev = -2;
  }

  public int getNext() {
    prev = val;
    val += 2;
    return val;
  }

  public void reset() {
    val = start;
    prev = start - 2;
  }

  public void setStart(int x) {
    start = x;
    val = start;
    prev = val - 2;
  }

  // Метод, не объявленный в интерфейсе ISeries.
  public int getPrevious() {
    return prev;
  }
}
```

Обратите внимание на то, что добавление метода `getPrevious()` потребовало внесения изменений в реализацию методов, определенных интерфейсом `ISeries`. Но поскольку интерфейс для этих методов остается прежним, при изменении не разрушается код, написанный ранее. В этом и заключается одно из достоинств использования интерфейсов.

Как упоминалось выше, интерфейс может реализовать любое количество классов. Рассмотрим, например, класс `Primes`, который генерирует ряд простых чисел. Обратите внимание на то, что его способ реализации интерфейса `ISeries` в корне отличается от используемого классом `ByTwos`.

```
// Использование интерфейса ISeries для реализации
// ряда простых чисел.

class Primes : ISeries {
  int start;
  int val;

  public Primes() {
    start = 2;
    val = 2;
  }

  public int getNext() {
    int i, j;
    bool isprime;

    val++;
    for(i = val; i < 1000000; i++) {
      isprime = true;
      for(j = 2; j < (i/j + 1); j++) {
        if((i%j)==0) {
          isprime = false;
          break;
        }
      }
      if(isprime) {
        val = i;
        break;
      }
    }
    return val;
  }

  public void reset() {
    val = start;
  }

  public void setStart(int x) {
    start = x;
    val = start;
  }
}
```

Здесь важно понимать, что, хотя классы `Primes` и `ByTwos` генерируют разные ряды чисел, оба они реализуют один и тот же интерфейс `ISeries`. И в этом нет ничего удивительного, поскольку каждый класс волен решить эту задачу так, как "считает" нужным.

Использование интерфейсных ссылок

Возможно, вы будете несколько удивлены, узнав, что можно объявить ссылочную переменную интерфейсного типа. Другими словами, можно создать переменную-ссылку на интерфейс. Такая переменная может ссылаться на любой объект, который реализует ее интерфейс. При вызове метода для объекта посредством интерфейсной ссылки будет выполнена та версия указанного метода, которая реализована этим объектом. Этот процесс аналогичен использованию ссылки на базовый класс для доступа к объекту производного класса (см. главу 11).

Использование интерфейсной ссылки демонстрируется в следующем примере. Здесь используется одна и та же интерфейсная переменная-ссылка, чтобы вызывать методы для объектов как класса `ByTwos`, так и класса `Primes`.

```
// Демонстрация использования интерфейсных ссылок.

using System;

// Определение интерфейса.
public interface ISeries {
  int getNext(); // Возвращает следующее число ряда.
  void reset();  // Выполняет перезапуск.
  void setStart(int x); // Устанавливает начальное
                        // значение.
}

// Используем интерфейс ISeries для генерирования
// последовательности четных чисел.
class ByTwos : ISeries {
  int start;
  int val;

  public ByTwos() {
    start = 0;
    val = 0;
  }

  public int getNext() {
    val += 2;
    return val;
  }

  public void reset() {
    val = start;
  }

  public void setStart(int x) {
    start = x;
    val = start;
  }
}

// Используем интерфейс ISeries для построения
// ряда простых чисел.
class Primes : ISeries {
  int start;
  int val;
```

```
  public Primes() {
    start = 2;
    val = 2;
  }

  public int getNext() {
    int i, j;
    bool isprime;

    val++;
    for(i = val; i < 1000000; i++) {
      isprime = true;
      for(j = 2; j < (i/j + 1); j++) {
        if((i%j)==0) {
          isprime = false;
          break;
        }
      }
      if(isprime) {
        val = i;
        break;
      }
    }
    return val;
  }

  public void reset() {
    val = start;
  }

  public void setStart(int x) {
    start = x;
    val = start;
  }
}

class SeriesDemo2 {
  public static void Main() {
    ByTwos twoOb = new ByTwos();
    Primes primeOb = new Primes();
    ISeries ob;

    for(int i=0; i < 5; i++) {
      ob = twoOb;
      Console.WriteLine(
                  "Следующее четное число равно " +
                  ob.getNext());
      ob = primeOb;
      Console.WriteLine(
                  "Следующее простое число равно " +
                  ob.getNext());
    }
  }
}
```

Вот результаты выполнения этой программы:

```
Следующее четное число равно 2
Следующее простое число равно 3
Следующее четное число равно 4
Следующее простое число равно 5
```

```
Следующее четное число равно 6
Следующее простое число равно 7
Следующее четное число равно 8
Следующее простое число равно 11
Следующее четное число равно 10
Следующее простое число равно 13
```

В методе `Main()` объявляется переменная `ob` как ссылка на интерфейс `ISeries`. Это означает, что ее можно использовать для хранения ссылок на любой объект, который реализует интерфейс `ISeries`. В данном случае она служит для ссылки на объекты `twoOb` и `primeOb`, которые являются экземплярами классов `ByTwos` и `Primes`, соответственно, причем оба класса реализуют один и тот же интерфейс, `ISeries`.

Важно понимать, что интерфейсная ссылочная переменная "осведомлена" только о методах, объявленных "под сенью" ключевого слова `interface`. Следовательно, интерфейсную ссылочную переменную нельзя использовать для доступа к другим переменным или методам, которые может определить объект, реализующий этот интерфейс.

Интерфейсные свойства

Как и методы, свойства определяются в интерфейсе без тела. Ниже приведен формат спецификации свойства.

```
// Интерфейсное свойство
тип имя{
  get;
  set;
}
```

Свойства, предназначенные только для чтения или только для записи, содержат только `get`- или `set`-элемент, соответственно.

Рассмотрим еще одну версию интерфейса `ISeries` и класса `ByTwos`, в котором для получения следующего элемента ряда и его установки используется свойство.

```
// Использование свойства в интерфейсе.

using System;

public interface ISeries {
  // Интерфейсное свойство.
  int next {
    get; // Возвращает следующее число ряда.
    set; // Устанавливает следующее число ряда.
  }
}

// Реализация интерфейса ISeries.
class ByTwos : ISeries {
  int val;

  public ByTwos() {
    val = 0;
  }

  // Получаем или устанавливаем значение ряда.
  public int next {
    get {
```

```
      val += 2;
      return val;
    }
    set {
      val = value;
    }
  }
}

// Демонстрируем использование интерфейсного свойства.
class SeriesDemo3 {
  public static void Main() {
    ByTwos ob = new ByTwos();

    // Получаем доступ к ряду через свойство.
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее значение равно " +
                        ob.next);

    Console.WriteLine("\nНачинаем с числа 21");
    ob.next = 21;
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее значение равно " +
                        ob.next);
  }
}
```

Результаты выполнения этой программы таковы:

```
Следующее значение равно 2
Следующее значение равно 4
Следующее значение равно 6
Следующее значение равно 8
Следующее значение равно 10

Начинаем с числа 21
Следующее значение равно 23
Следующее значение равно 25
Следующее значение равно 27
Следующее значение равно 29
Следующее значение равно 31
```

Интерфейсные индексаторы

В интерфейсе можно определить и индексатор. Объявление индексатора в интерфейсе имеет следующий формат записи:

```
 // Интерфейсный индексатор
тип_элемента this[int индекс]{
  get;
  set;
}
```

Индексаторы, предназначенные только для чтения или только для записи, содержат только `get`- или `set`-метод, соответственно.

Предлагаем еще одну версию интерфейса `ISeries`, в который добавлен индексатор, предназначенный только для чтения элемента ряда.

```
// Добавление в интерфейс индексатора.

using System;

public interface ISeries {
  // Интерфейсное свойство.
  int next {
    get; // Возвращает следующее число ряда.
    set; // Устанавливает следующее число ряда.
  }

  // Интерфейсный индексатор.
  int this[int index] {
    get; // Возвращает заданный член ряда.
  }
}

// Реализация интерфейса ISeries.
class ByTwos : ISeries {
  int val;

  public ByTwos() {
    val = 0;
  }

  // Получаем или устанавливаем значение с помощью
  // свойства.
  public int next {
    get {
      val += 2;
      return val;
    }
    set {
      val = value;
    }
  }

  // Получаем значение с помощью индексатора.
  public int this[int index] {
    get {
      val = 0;
      for(int i=0; i<index; i++)
        val += 2;
      return val;
    }
  }
}

// Демонстрируем использование интерфейсного индексатора.
class SeriesDemo4 {
  public static void Main() {
    ByTwos ob = new ByTwos();

    // Получаем доступ к ряду посредством свойства.
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее значение равно " +
                        ob.next);
```

```
    Console.WriteLine("\nНачинаем с числа 21");
    ob.next = 21;
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее значение равно " +
                        ob.next);

    Console.WriteLine("\nПереход в исходное состояние.");
    ob.next = 0;

    // Получаем доступ к ряду посредством индексатора.
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее значение равно " +
                        ob[i]);
  }
}
```

Вот результаты, сгенерированные этой программой:

```
Следующее значение равно 2
Следующее значение равно 4
Следующее значение равно 6
Следующее значение равно 8
Следующее значение равно 10

Начинаем с числа 21
Следующее значение равно 23
Следующее значение равно 25
Следующее значение равно 27
Следующее значение равно 29
Следующее значение равно 31

Переход в исходное состояние.
Следующее значение равно 0
Следующее значение равно 2
Следующее значение равно 4
Следующее значение равно 6
Следующее значение равно 8
```

Наследование интерфейсов

Один интерфейс может унаследовать "богатство" другого. Синтаксис этого механизма аналогичен синтаксису, используемому для наследования классов. Если класс реализует интерфейс, который наследует другой интерфейс, этот класс должен обеспечить способы реализации для всех членов, определенных внутри цепочки наследования интерфейсов. Рассмотрим такой пример:

```
// Один интерфейс может наследовать другой.

using System;

public interface A {
  void meth1();
  void meth2();
}

// Интерфейс B теперь включает методы meth1() и meth2(),
// а также добавляет метод meth3().
```

```
public interface B : A {
  void meth3();
}

// Этот класс должен реализовать все методы
// интерфейсов A и B.
class MyClass : B {
  public void meth1() {
    Console.WriteLine("Реализация метода meth1().");
  }

  public void meth2() {
    Console.WriteLine("Реализация метода meth2().");
  }

  public void meth3() {
    Console.WriteLine("Реализация метода meth3().");
  }
}

class IFExtend {
  public static void Main() {
    MyClass ob = new MyClass();

    ob.meth1();
    ob.meth2();
    ob.meth3();
  }
}
```

Если бы в качестве эксперимента вы попытались удалить метод `meth1()`, реализованный в классе `MyClass`, то сразу же получили бы от компилятора сообщение об ошибке. Как упоминалось выше, любой класс, который реализует интерфейс, должен реализовать все методы, определенные этим интерфейсом, включая методы, которые унаследованы от других интерфейсов.

Сокрытие имен с помощью наследования интерфейсов

В производном интерфейсе можно объявить член, который скрывает член, определенный в базовом интерфейсе. Это происходит при совпадении их сигнатур. Такое совпадение вызовет предупреждающее сообщение, если член производного интерфейса не модифицировать с помощью ключевого слова `new`.

Явная реализация членов интерфейса

При реализации члена интерфейса можно квалифицировать его имя с использованием имени интерфейса. В этом случае говорят, что *член интерфейса реализуется явным образом*, или имеет место его *явная реализация*. Например, при определении интерфейса

```
interface IMyIF {
  int myMeth(int x);
}
```

вполне допустимо реализовать интерфейс `IMyIF` следующим образом:

```
class MyClass : IMyIF{
  int IMyIF.myMeth(int x) {
    return x / 3;
  }
}
```

Как видите, при реализации метода `myMeth()` члена интерфейса `IMyIF` указывается его полное имя, включающее имя интерфейса.

Явная реализация членов интерфейса может понадобиться по двум причинам. Во-первых, реализуя метод с использованием полностью квалифицированного имени, вы тем самым обозначаете части закрытой реализации, которые не "видны" коду, определенному вне класса. Во-вторых, класс может реализовать два интерфейса, которые объявляют методы с одинаковыми именами и типами. Полная квалификация имен позволяет избежать неопределенности ситуации. Рассмотрим примеры.

Закрытая реализация

Следующая программа содержит интерфейс с именем `IEven`, который определяет два метода `isEven()` и `isOdd()`, устанавливающие факт четности и нечетности числа, соответственно. Класс `MyClass` реализует интерфейс `IEven`, причем его член `isOdd()` реализуется явным образом.

```
// Явная реализация члена интерфейса.

using System;

interface IEven {
  bool isOdd(int x);
  bool isEven(int x);
}

class MyClass : IEven {
  // Явная реализация.
  bool IEven.isOdd(int x) {
    if((x%2) != 0) return true;
    else return false;
  }

  // Обычная реализация.
  public bool isEven(int x) {
    IEven o = this; // Ссылка на вызывающий объект.

    return !o.isOdd(x);
  }
}

class Demo {
  public static void Main() {
    MyClass ob = new MyClass();
    bool result;

    result = ob.isEven(4);
    if(result) Console.WriteLine("4 - четное число.");
```

```
    else Console.WriteLine("3 - нечетное число.");

    // result = ob.isOdd(); // Ошибка, член не виден.
  }
}
```

Поскольку метод `isOdd()` реализован в явном виде, он недоступен вне класса `MyClass`. Такой способ реализации делает его надежно закрытым. Внутри класса `MyClass` к методу `isOdd()` можно получить доступ только через ссылку на интерфейс. Вот почему он прекрасно вызывается для объекта `o` в реализации метода `isEven()`.

Как избежать неопределенности с помощью явной реализации

Рассмотрим пример, в котором реализовано два интерфейса, причем оба объявляют метод с именем `meth()`. В этой ситуации явная реализация используется для того, чтобы избежать неопределенности.

```
// Использование явной реализации для того, чтобы избежать
// неоднозначности.

using System;

interface IMyIF_A {
  int meth(int x);
}

interface IMyIF_B {
  int meth(int x);
}

// В классе MyClass реализованы оба интерфейса.
class MyClass : IMyIF_A, IMyIF_B {

  // Явным образом реализуем два метода meth().
  int IMyIF_A.meth(int x) {
    return x + x;
  }
  int IMyIF_B.meth(int x) {
    return x * x;
  }

  // Вызываем метод meth() посредством ссылки на интерфейс.
  public int methA(int x){
    IMyIF_A a_ob;
    a_ob = this;
    return a_ob.meth(x); // Имеется в виду
                         // интерфейс IMyIF_A.
  }

  public int methB(int x){
    IMyIF_B b_ob;
    b_ob = this;
    return b_ob.meth(x); // Имеется в виду
                         // интерфейс IMyIF_B
  }
}
```

```
class FQIFNames {
  public static void Main() {
    MyClass ob = new MyClass();

    Console.Write("Вызов метода IMyIF_A.meth(): ");
    Console.WriteLine(ob.methA(3));

    Console.Write("Вызов метода IMyIF_B.meth(): ");
    Console.WriteLine(ob.methB(3));
  }
}
```

Вот результаты выполнения этой программы:

```
Вызов метода IMyIF_A.meth(): 6
Вызов метода IMyIF_B.meth(): 9
```

Обратите внимание на то, что метод `meth()` имеет одинаковую сигнатуру в интерфейсах `IMyIF_A` и `IMyIF_B`. Следовательно, если класс `MyClass` реализует оба эти интерфейса, он должен реализовать каждый метод в отдельности, полностью указав его имя (с использованием имени соответствующего интерфейса). Поскольку единственный способ вызова явно заданного метода состоит в использовании интерфейсной ссылки, метод `meth()`, объявленный в интерфейсе `IMyIF_A`, создает ссылку на интерфейс `IMyIF_A`, а метод `meth()`, объявленный в интерфейсе `IMyIF_B`, создает ссылку на интерфейс `IMyIF_B`. Созданные ссылки затем используются при вызове этих методов, благодаря чему можно избежать неоднозначности.

Выбор между интерфейсом и абстрактным классом

В программировании на C# при необходимости описать функции, а не способ их реализации, важно знать, когда следует использовать интерфейс, а когда — абстрактный класс. Общее правило таково. Если вы полностью описываете действия класса и не нужно уточнять, как он это делает, следует использовать интерфейс. Если же требуется включить в описание детали реализации, имеет смысл представить концепцию программы (или ее части) в виде абстрактного класса.

Стандартные интерфейсы среды .NET Framework

В среде .NET Framework определено множество интерфейсов, которые могут использовать C#-программы. Например, интерфейс `System.IComparable` определяет метод `CompareTo()`, который позволяет сравнивать объекты. Интерфейсы также образуют важную часть коллекции классов, которая обеспечивает различные типы для хранения групп объектов (например, стеки и очереди). Так, например, интерфейс `System.Collections.ICollection` определяет функциональность, общую для всех коллекций. Интерфейс `System.Collections.IEnumerator` предлагает способ опроса элементов в коллекции. Эти и другие интерфейсы мы рассмотрим в части II.

Учебный проект: создание интерфейса

Прежде чем продолжать изучение других средств программирования на C#, было бы полезно рассмотреть пример использования интерфейса. В этом разделе мы создадим интерфейс ICipher, который определяет методы поддержки шифрования строк. Для этой задачи использование интерфейса вполне оправданно, поскольку здесь можно полностью отделить код с описанием "что" от кода, где указано, "как".

Интерфейс ICipher имеет такой вид:

```
// Интерфейс шифрования и дешифрирования строк.
public interface ICipher {
  string encode(string str);
  string decode(string str);
}
```

В интерфейсе ICipher объявляются два метода: encode() и decode(), которые используются для шифрования и дешифрирования строк, соответственно. При этом другие детали не уточняются. Это значит, что классы, которые будут реализовывать эти методы, могут выбирать любой метод шифрования. Например, один класс мог бы шифровать строку на основе ключа, определенного пользователем. Другой мог бы использовать защиту с помощью системы паролей. У третьего механизм действия шифра мог бы опираться на побитовую обработку, а у четвертого — на простую перестановку кода (реализация перестановочного шифра). Главное то, что интерфейс операции шифрования и дешифрирования строк не зависит от используемого способа шифрования. А поскольку здесь нет необходимости определять даже часть механизма шифрования, то для его представления мы выбираем средство интерфейса.

В нашем учебном проекте интерфейс ICipher реализуют сразу два класса: SimpleCipher и BitCipher. Класс SimpleCipher шифрует строку посредством сдвига каждого символа на одну "алфавитную" позицию выше. Например, в результате такого сдвига буква *А* становится буквой *Б*, а буква *Б* — буквой *В* и т.д. Класс BitCipher шифрует строку по-другому: каждый символ заменяется результатом операции исключающего ИЛИ, примененной к этому символу и некоторому 16-разрядному значению, которое используется в качестве ключа.

```
/* Простая реализация интерфейса ICipher, которая кодирует
   сообщение посредством сдвига каждого символа на
   1 позицию вверх.  Так, буква А превратится в
   букву Б и т.д. */
class SimpleCipher : ICipher {

  // Метод возвращает зашифрованную строку, заданную
  // открытым текстом.
  public string encode(string str) {
    string ciphertext = "";

    for(int i=0; i < str.Length; i++)
      ciphertext = ciphertext + (char) (str[i] + 1);

    return ciphertext;
  }

  // Метод возвращает дешифрированную строку, заданную
  // зашифрованным текстом.
  public string decode(string str) {
    string plaintext = "";
```

```
    for(int i=0; i < str.Length; i++)
      plaintext = plaintext + (char) (str[i] - 1);

    return plaintext;
  }
}

/* В этой реализации интерфейса ICipher используется
   побитовая обработка и ключ. */
class BitCipher : ICipher {
  ushort key;

  // Определяем ключ при построении объектов
  // класса BitCipher.
  public BitCipher(ushort k) {
    key = k;
  }

  // Метод возвращает зашифрованную строку, заданную
  // открытым текстом.
  public string encode(string str) {
    string ciphertext = "";

    for(int i=0; i < str.Length; i++)
      ciphertext = ciphertext + (char) (str[i] ^ key);

    return ciphertext;
  }

  // Метод возвращает дешифрированную строку, заданную
  // зашифрованным текстом.
  public string decode(string str) {
    string plaintext = "";

    for(int i=0; i < str.Length; i++)
      plaintext = plaintext + (char) (str[i] ^ key);

    return plaintext;
  }
}
```

Как видите, оба класса `SimpleCipher` и `BitCipher` реализуют один и тот же интерфейс `ICipher`, хотя используют при этом различные способы его реализации. В следующей программе демонстрируется функционирование классов `SimpleCipher` и `BitCipher`.

```
// Демонстрация использования интерфейса ICipher.

using System;

class ICipherDemo {
  public static void Main() {
    ICipher ciphRef;
    BitCipher bit = new BitCipher(27);
    SimpleCipher sc = new SimpleCipher();

    string plain;
    string coded;
```

```
    // Сначала переменная ciphRef ссылается на объект
    // класса SimpleCipher (простое шифрование).
    ciphRef = sc;

    Console.WriteLine("Использование простого шифра.");

    plain = "testing";
    coded = ciphRef.encode(plain);
    Console.WriteLine("Зашифрованный текст: " + coded);

    plain = ciphRef.decode(coded);
    Console.WriteLine("Открытый текст: " + plain);

    // Теперь переменная ciphRef refer ссылается на
    // объект класса BitCipher (поразрядное шифрование).
    ciphRef = bit;

    Console.WriteLine(
            "\nИспользование поразрядного шифрования.");

    plain = "testing";
    coded = ciphRef.encode(plain);
    Console.WriteLine("Зашифрованный текст: " + coded);

    plain = ciphRef.decode(coded);
    Console.WriteLine("Открытый текст: " + plain);
  }
}
```

Вот результаты выполнения этой программы:

```
Использование простого шифра.
Зашифрованный текст: uftujoh
Открытый текст: testing

Использование поразрядного шифрования.
Зашифрованный текст: o~horu|
Открытый текст: testing
```

Одно из достоинств создания интерфейса шифрования состоит в том, что доступ к любому классу, который реализует этот интерфейс, осуществляется одинаково, независимо от того, как реализован процесс шифрования. Например, рассмотрим следующую программу, в которой класс `UnlistedPhone` используется для хранения телефонных номеров в зашифрованном формате. При необходимости имена и цифры номера автоматически дешифруются.

```
// Использование интерфейса ICipher.

using System;

// Класс для хранения телефонных номеров.
class UnlistedPhone {
  string pri_name;   // Поддерживает свойство Name.
  string pri_number; // Поддерживает свойство Number.

  ICipher crypt; // Ссылка на объект шифрования.

  public UnlistedPhone(string name, string number,
                       ICipher c)
  {
```

```
    crypt = c; // Хранит объект шифрования.

    pri_name = crypt.encode(name);
    pri_number = crypt.encode(number);
  }

  public string Name {
    get {
      return crypt.decode(pri_name);
    }
    set {
      pri_name = crypt.encode(value);
    }
  }

  public string Number {
    get {
      return crypt.decode(pri_number);
    }
    set {
      pri_number = crypt.encode(value);
    }
  }
}

// Демонстрируем использование класса UnlistedPhone.
class UnlistedDemo {
  public static void Main() {
    UnlistedPhone phone1 =
        new UnlistedPhone("Том", "555-3456",
                          new BitCipher(27));
    UnlistedPhone phone2 =
        new UnlistedPhone("Мэри", "555-8891",
                          new BitCipher(9));


    Console.WriteLine("Телефонный номер абонента по имени "
                      + phone1.Name + " : "
                      + phone1.Number);

    Console.WriteLine("Телефонный номер абонента по имени "
                      + phone2.Name + " : "
                      + phone2.Number);
  }
}
```

Вот результаты выполнения этой программы:

```
Телефонный номер абонента по имени Том : 555-3456
Телефонный номер абонента по имени Мэри : 555-8891
```

Рассмотрим, как реализован класс `UnlistedPhone`. Обратите внимание на то, что он содержит три поля. Первые два представляют собой закрытые переменные для хранения имени и соответствующего ему телефонного номера. Третье поле — это ссылка на объект интерфейса `ICipher`. Объекту класса `UnlistedPhone` при создании передаются три ссылки. Первые две ссылаются на строки, содержащие имя и телефонный номер, а третья — на объект шифрования, который используется для кодирования имени и номера. Ссылка на объект шифрования хранится в переменной `crypt`. Здесь допустим объект шифрования любого типа, если, конечно, он реализует интер-

фейс ICipher. В данном случае используется объект типа BitCipher. Таким образом, объект класса UnlistedPhone может вызывать методы encode() и decode() для объекта BitCipher через ссылку crypt.

Теперь обратите внимание на организацию работы свойств Name и Number. При выполнении set-операции имя или телефонный номер автоматически шифруются посредством вызова метода encode() для объекта, определяемого ссылкой crypt. При выполнении get-операции имя или телефонный номер автоматически дешифруются посредством вызова метода decode(). Ни свойству Name, ни свойству Number не известен используемый для них метод шифрования. Они просто получают доступ к его телу через интерфейс.

Поскольку интерфейс шифрования стандартизирован описанием интерфейса ICipher, можно изменить объект шифрования, не изменяя внутренний код класса UnlistedPhone. Например, в следующей программе при создании объектов класса UnlistedPhone используется SimpleCipher-объект, а не BitCipher-объект. Сюда внесено единственное изменение, связанное с передачей объекта шифрования конструктору класса UnlistedPhone.

```
// Эта версия программы использует класс SimpleCipher.

using System;

class UnlistedDemo {
  public static void Main() {

    // На этот раз вместо класса BitCipher используем
    // класс SimpleCipher.
    UnlistedPhone phone1 =
        new UnlistedPhone("Tom", "555-3456",
                          new SimpleCipher());
    UnlistedPhone phone2 =
        new UnlistedPhone("Mary", "555-8891",
                          new SimpleCipher());

     Console.WriteLine(
                "Телефонный номер абонента по имени " +
                phone1.Name + " : " +
                phone1.Number);

    Console.WriteLine(
                "Телефонный номер абонента по имени " +
                phone2.Name + " : " +
                phone2.Number);
  }
}
```

Как показывает эта программа, поскольку интерфейс ICipher реализуют оба класса — SimpleCipher и BitCipher, для создания объектов класса UnlistedPhone можно использовать любой из них.

И последнее. Код класса UnlistedPhone также демонстрирует возможность доступа к объектам, которые реализуют интерфейс, посредством ссылки на него. Поскольку на объект шифрования можно указывать с помощью ссылочной переменной типа ICipher, для реализации процесса шифрования можно использовать любой объект, который реализует интерфейс ICipher. Это позволяет совершенно безболезненно и безопасно заменить один метод шифрования другим, не изменяя код класса UnlistedPhone. Но если бы в классе UnlistedPhone для данных типа crypt был

жестко определен тип объекта шифрования (например, класс `BitCipher`), то при необходимости заменить схему шифрования в код класса `UnlistedPhone` пришлось бы вносить изменения.

Структуры

Как вы уже знаете, классы — это ссылочные типы. Это означает, что к объектам классов доступ осуществляется через ссылку. Этим они отличаются от типов значений, к которым в C# реализован прямой доступ. Но иногда желательно получать прямой доступ и к объектам, как в случае нессылочных типов. Одна из причин для этого — эффективность. Ведь очевидно, что доступ к объектам классов через ссылки увеличивает расходы системных ресурсов, в том числе и памяти. Даже для очень маленьких объектов требуются существенные объемы памяти. Для компенсации упомянутых расходов времени и пространства в C# предусмотрены структуры. *Структура* подобна классу, но она относится к типу значений, а не к ссылочным типам.

Структуры объявляются с использованием ключевого слова `struct` и синтаксически подобны классам. Формат записи структуры таков:

```
struct имя : интерфейсы {
  // объявления членов
}
```

Элемент *имя* означает имя структуры.

Структуры не могут наследовать другие структуры или классы. Структуры не могут использоваться в качестве базовых для других структур или классов. (Однако, подобно другим C#-типам, структуры наследуют класс `object`). Структура может реализовать один или несколько интерфейсов. Они указываются после имени структуры и отделяются запятыми. Как и у классов, членами структур могут быть методы, поля, индексаторы, свойства, операторные методы и события. Структуры могут также определять конструкторы, но не деструкторы. Однако для структуры нельзя определить конструктор по умолчанию (без параметров). Дело в том, что конструктор по умолчанию автоматически определяется для всех структур, и его изменить нельзя. Поскольку структуры не поддерживают наследования, члены структуры нельзя определять с использованием модификаторов `abstract`, `virtual` или `protected`.

Объект структуры можно создать с помощью оператора `new`, подобно любому объекту класса, но это не обязательно. Если использовать оператор `new`, вызывается указанный конструктор, а если не использовать его, объект все равно будет создан, но не инициализирован. В этом случае вам придется выполнить инициализацию вручную. Рассмотрим пример использования структуры для хранения информации о книге.

```
// Демонстрация использования структуры.

using System;

// Определение структуры.
struct Book {
  public string author;
  public string title;
  public int copyright;

  public Book(string a, string t, int c) {
    author = a;
    title = t;
    copyright = c;
  }
```

```
}

// Демонстрируем использование структуры Book.
class StructDemo {
  public static void Main() {
    Book book1 = new Book("Herb Schildt",
                          "C# A Beginner's Guide",
                          2001); // Вызов явно заданного
                                 // конструктора.

    Book book2 = new Book(); // Вызов конструктора
                             // по умолчанию.
    Book book3; // Создание объекта без вызова
                // конструктора.

    Console.WriteLine(book1.title + ", автор " +
                      book1.author +
                      ", (c) " + book1.copyright);
    Console.WriteLine();

    if(book2.title == null)
      Console.WriteLine("Член book2.title содержит null.");
    // Теперь поместим в структуру book2 данные.
    book2.title = "Brave New World";
    book2.author = "Aldous Huxley";
    book2.copyright = 1932;
    Console.Write("Теперь структура book2 содержит:\n    ");
    Console.WriteLine(book2.title + ", автор " +
                      book2.author +
                      ", (c) " + book2.copyright);

    Console.WriteLine();

// Console.WriteLine(book3.title); // Ошибка: сначала
                                   // необходима
                                   // инициализация.
    book3.title = "Red Storm Rising";

    Console.WriteLine(book3.title); // Теперь все Ok!
  }
}
```

Вот результаты выполнения этой программы:

```
C# A Beginner's Guide, автор Herb Schildt, (c) 2001

Член book2.title содержит null.
Теперь структура book2 содержит:
    Brave New World, автор Aldous Huxley, (c) 1932

Red Storm Rising
```

Как видно из результатов выполнения этой программы, структура может быть создана либо с помощью оператора `new`, который вызывает соответствующий конструктор, либо простым объявлением объекта. При использовании оператора `new` поля структуры будут инициализированы, причем это сделает либо конструктор по умолчанию (он инициализирует все поля значениями по умолчанию), либо конструктор, определенный пользователем. Если оператор `new` не используется, как в случае объекта

book3, созданный таким образом объект остается неинициализированным, и его поля должны быть установлены до начала использования.

При присваивании одной структуры другой создается копия этого объекта. Это — очень важное отличие struct-объекта от class-объекта. Как упоминалось выше, присваивая одной ссылке на класс другую, вы просто меняете объект, на который ссылается переменная, стоящая с левой стороны от оператора присваивания. А присваивая одной struct-переменной другую, вы создаете копию объекта, указанного с правой стороны от оператора присваивания. Рассмотрим, например, следующую программу:

```
// Копирование структуры.

using System;

// Определяем структуру.
struct MyStruct {
  public int x;
}

// Демонстрируем присваивание структуры.
class StructAssignment {
  public static void Main() {
    MyStruct a;
    MyStruct b;

    a.x = 10;
    b.x = 20;

    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);

    a = b;
    b.x = 30;

    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);
  }
}
```

Эта программа генерирует следующие результаты.

```
a.x 10, b.x 20
a.x 20, b.x 30
```

Как подтверждают результаты выполнения этой программы, после присваивания

```
a = b;
```

структурные переменные a и b по-прежнему не зависят одна от другой. Другими словами, переменная a никак не связана с переменной b, если не считать, что переменная a содержит копию значения переменной b. Будь a и b ссылками на классы, все обстояло бы по-другому. Рассмотрим теперь class-версию предыдущей программы.

```
// Копирование класса.

using System;

// Определяем класс.
class MyClass {
  public int x;
}

// Теперь покажем присваивание объектов класса.
```

```
class ClassAssignment {
  public static void Main() {
    MyClass a = new MyClass();
    MyClass b = new MyClass();

    a.x = 10;
    b.x = 20;

    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);

    a = b;
    b.x = 30;

    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);
  }
}
```

Вот какие результаты получены при выполнении этой программы:

```
a.x 10, b.x 20
a.x 30, b.x 30
```

Как видите, после присваивания объекта b переменной a обе переменные ссылаются на один и тот же объект, т.е. на тот, на который изначально ссылалась переменная b.

Зачем нужны структуры

Вы могли бы выразить удивление, почему C# включает тип `struct`, если, казалось бы, он представляет собой "слаборазвитую" версию типа `class`. Ответ следует искать в эффективности и производительности. Поскольку структуры — это типы значений, они обрабатываются напрямую, а не через ссылки. Таким образом, тип `struct` не требует отдельной ссылочной переменной. Это означает, что при использовании структур расходуется меньший объем памяти. Более того, благодаря прямому доступу к структурам, при работе с ними не снижается производительность, что имеет место при доступе к объектам классов. Поскольку классы — ссылочные типы, доступ к их объектам осуществляется через ссылки. Такая косвенность увеличивает затраты системных ресурсов при каждом доступе. Структуры этим не страдают. В общем случае, если вам нужно хранить небольшую группу связанных данных, но не нужно обеспечивать наследование и использовать другие достоинства ссылочных типов, тип `struct` может оказаться более предпочтительным вариантом.

Рассмотрим еще один пример, демонстрирующий, как использовать структуру на практике. Эта программа имитирует запись транзакции. Одна запись содержит заголовок пакета с его номером и длиной. За этими данными указывается номер счета и объем транзакции. Поскольку заголовок пакета представляет собой самодостаточную единицу информации, он организован в виде структуры. Эту структуру затем можно использовать для создания записи транзакции или любого другого типа информационного пакета.

```
// Структуры прекрасно работают при группировании данных.

using System;

// Определяем структуру пакета.
struct PacketHeader {
  public uint packNum;   // Номер пакета.
  public ushort packLen; // Длина пакета.
}
```

```
// Используем структуру PacketHeader для создания
// электронной записи транзакции.
class Transaction {
  static uint transacNum = 0;

  PacketHeader ph;  // Включаем в транзакцию
                    // структуру PacketHeader.
  string accountNum;
  double amount;

  public Transaction(string acc, double val) {
    // Создаем заголовок пакета.
    ph.packNum = transacNum++;
    ph.packLen = 512;  // arbitrary length

    accountNum = acc;
    amount = val;
  }

  // Имитируем транзакцию.
  public void sendTransaction() {
    Console.WriteLine("Пакет #: " + ph.packNum +
                      ", Длина: " + ph.packLen +
                      ",\n    Счет #: " + accountNum +
                      ", Сумма: {0:C}\n", amount);
  }
}

// Демонстрируем использование пакетной обработки.
class PacketDemo {
  public static void Main() {
    Transaction t = new Transaction("31243", -100.12);
    Transaction t2 = new Transaction("AB4655", 345.25);
    Transaction t3 = new Transaction("8475-09", 9800.00);

    t.sendTransaction();
    t2.sendTransaction();
    t3.sendTransaction();
  }
}
```

При выполнении этой программы были получены следующие результаты.

```
Пакет #: 0, Длина: 512,
    Счет #: 31243, Сумма: ($100.12)

Пакет #: 1, Длина: 512,
    Счет #: AB4655, Сумма: $345.25

Пакет #: 2, Длина: 512,
    Счет #: 8475-09, Сумма: $9,800.00
```

Выбор для данных `PacketHeader` типа структуры вполне оправдан, поскольку эти данные имеют небольшой объем, не наследуются и даже не содержат методов. В качестве структуры объект типа `PacketHeader` не требует дополнительных затрат системных ресурсов на доступ через ссылку, как в случае класса. Таким образом, структуру `PacketHeader` можно использовать для записей транзакций любого типа без ущерба для эффективности.

Интересно отметить, что в языке C++ также можно определять структуры с помощью ключевого слова `struct`. Однако C#- и C++-структуры имеют кардинальное отличие. В C++ тип `struct` — эти тип класса, причем типы `struct` и `class` практически эквивалентны (различие между ними выражается в доступе к их членам по умолчанию: для класса он принят закрытым, а для структуры — открытым). В C# ключевое слово `struct` используется для определения типов значений, а `class` — ссылочных типов.

Перечисления

Перечисление (enumeration) — это множество именованных целочисленных констант. Ключевое слово `enum` объявляет перечислимый тип. Формат записи перечисления таков:

```
enum имя {список_перечисления};
```

Здесь с помощью элемента *имя* указывается имя типа перечисления. Элемент *список_перечисления* представляет собой список идентификаторов, разделенных запятыми.

Рассмотрим пример. В следующем фрагменте кода определяется перечисление `apple`, которое содержит список названий различных сортов яблок.

```
enum apple { Jonathan, GoldenDel, RedDel, Winsap,
             Cortland, McIntosh };
```

Здесь важно понимать, что каждый символ списка перечисления означает целое число, причем каждое следующее число (представленное идентификатором) на единицу больше предыдущего. Поскольку значение первого символа перечисления равно нулю, следовательно, идентификатор `Jonathan` имеет значение 0, `GoldenDel` — значение 1 и т.д.

Константу перечисления можно использовать везде, где допустимо целочисленное значение. Однако между типом `enum` и встроенным целочисленным типом неявные преобразования не определены, поэтому при необходимости должна использоваться явно заданная операция приведения типов. В случае преобразования одного типа перечисления в другой также необходимо использовать приведение типов.

К членам перечисления доступ осуществляется посредством имени типа и оператора "точка". Например, при выполнении инструкции

```
Console.WriteLine(apple.RedDel + " имеет значение " +
                  (int)apple.RedDel);
```

будет отображено следующее.

```
RedDel имеет значение 2
```

Как подтверждает результат выполнения этой инструкции, при отображении значения перечислимого типа используется его имя. А для получения его целочисленного значения необходимо использовать операцию приведения к типу `int`. (В этом заключается отличие от ранних версий C#, в которых по умолчанию отображалось целочисленное представление значения перечислимого типа, а не его имя.)

Теперь рассмотрим программу, которая демонстрирует использование перечисления `apple`.

```
// Демонстрация использования перечисления.

using System;

class EnumDemo {
```

```
enum apple { Jonathan, GoldenDel, RedDel, Winsap,
             Cortland, McIntosh };

public static void Main() {
  string[] color = {
    "красный",
    "желтый",
    "красный",
    "красный",
    "красный",
    "красно-зеленый"
  };

  apple i; // Объявляем переменную перечислимого типа.

  // Используем переменную i для обхода всех
  // членов перечисления.
  for(i = apple.Jonathan; i <= apple.McIntosh; i++)
    Console.WriteLine(i + " имеет значение " + (int)i);

  Console.WriteLine();

  // Используем перечисление для индексации массива.
  for(i = apple.Jonathan; i <= apple.McIntosh; i++)
    Console.WriteLine("Цвет сорта " + i + " - " +
                      color[(int)i]);

}
}
```

Результаты выполнения этой программы таковы:

```
Jonathan имеет значение 0
GoldenDel имеет значение 1
RedDel имеет значение 2
Winsap имеет значение 3
Cortland имеет значение 4
McIntosh имеет значение 5

Цвет сорта Jonathan - красный
Цвет сорта GoldenDel - желтый
Цвет сорта RedDel - красный
Цвет сорта Winsap - красный
Цвет сорта Cortland - красный
Цвет сорта McIntosh - красно-зеленый
```

Обратите внимание на то, как `for`-циклы управляются переменной типа `apple`. Поскольку перечисление — это целочисленный тип, значение перечисления может быть использовано везде, где допустимы целые значения. Поскольку значения перечислимого типа начинаются с нуля, их можно использовать для индексирования массива `color` (чтобы получить цвет яблок). Заметьте: в этом случае необходимо выполнить приведение типа. Как упоминалось выше, неявные преобразования между целочисленными и перечислимыми типами не определены. Поэтому без явно заданного приведения типа здесь не обойтись.

И еще. Поскольку перечислимые типы представляют собой целочисленные значения, их можно использовать для управления `switch`-инструкцией (соответствующий пример приведен ниже).

Инициализация перечислений

Одно или несколько символов в перечислении можно определить с помощью инициализатора. Это реализуется путем использования знака "равно" и последующего целого значения. Символам, стоящим после инициализатора, присваиваются значения, превышающие предыдущее значение инициализации. Например, следующий фрагмент кода присваивает число 10 символу RedDel.

```
enum apple { Jonathan, GoldenDel, RedDel = 10, Winsap,
             Cortland, McIntosh };
```

Вот какие значения имеют теперь эти символы:

Jonathan	0
GoldenDel	1
RedDel	10
Winsap	11
Cortland	12
McIntosh	13

Задание базового типа перечисления

По умолчанию перечисления используют тип int, но можно также создать перечисление любого другого целочисленного типа, за исключением типа char. Чтобы задать тип, отличный от int, укажите этот базовый тип после имени перечисления и двоеточия. Например, следующая инструкция создает перечисление apple с базовым типом byte.

```
enum apple : byte { Jonathan, GoldenDel, RedDel, Winsap,
                    Cortland, McIntosh };
```

Теперь член apple.Winsap, например, представляет собой byte-значение.

Использование перечислений

На первый взгляд может показаться, что перечисления, хотя и представляют определенный интерес, но вряд ли заслуживают большого внимания C#-программиста. Это не так. Перечисления незаменимы, когда в программе необходимо использовать один или несколько специализированных символов. Например, представьте, что вы пишете программу для управления лентой конвейера на фабрике. Вы могли бы создать метод conveyor(), который в качестве параметров принимает следующие параметры: старт, стоп, вперед и назад. Вместо того чтобы передавать методу conveyor() числа (1 для команды "старт", 2 для команды "стоп" и т.д.), что чревато ошибками, вы можете создать перечисление, которое этим числам присваивает слова. Вот пример такого решения:

```
// Управление лентой конвейера.

using System;

class ConveyorControl {
  // Перечисляем команды, управляющие конвейером.
  public enum action { старт, стоп, вперед, назад };

  public void conveyor(action com) {
    switch(com) {
      case action.старт:
        Console.WriteLine("Запуск конвейера.");
```

```
        break;
      case action.стоп:
        Console.WriteLine("Останов конвейера.");
        break;
      case action.вперед:
        Console.WriteLine("Перемещение вперед.");
        break;
      case action.назад:
        Console.WriteLine("Перемещение назад.");
        break;
    }
  }
}

class ConveyorDemo {
  public static void Main() {
    ConveyorControl c = new ConveyorControl();

    c.conveyor(ConveyorControl.action.старт);
    c.conveyor(ConveyorControl.action.вперед);
    c.conveyor(ConveyorControl.action.назад);
    c.conveyor(ConveyorControl.action.стоп);

  }
}
```

Вот какие результаты генерирует эта программа:

```
Запуск конвейера.
Перемещение вперед.
Перемещение назад.
Останов конвейера.
```

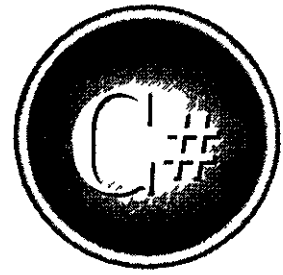
Поскольку метод `conveyor()` принимает аргумент типа `action`, этому методу могут передаваться только значения, имеющие тип `action`. Вот, например, попытка передать методу `conveyor()` значение 22.

```
c.conveyor(22);  // Ошибка!
```

Эта инструкция не скомпилируется, поскольку встроенного преобразования из типа `int` в тип `action` не существует. Это — своего рода защита от передачи методу `conveyor()` некорректных команд. Конечно, чтобы "настоять" на таком преобразовании, можно было бы использовать операцию приведения типов, но это потребовало бы заранее продуманных действий и вряд ли привело к случайной ошибке. Кроме того, поскольку команды задаются словами, а не числами, маловероятно, что пользователь метода `conveyor()` по небрежности передаст неверное значение.

В этом примере хотелось бы еще обратить ваше внимание вот на что. Тип перечисления используется здесь для управления `switch`-инструкцией. Как уже упоминалось, поскольку перечисления считаются целочисленными типами, их можно использовать в инструкции `switch`.

Полный справочник по



Глава 13

Обработка исключительных ситуаций

Исключительная ситуация (или исключение) — это ошибка, которая возникает во время выполнения программы. Используя C#-подсистему обработки исключительных ситуаций, с такими ошибками можно справляться. Эта подсистема в C# включает в себя усовершенствованные методы, используемые в языках C++ и Java. Поэтому эта тема будет знакомой для C++- и Java-программистов. Однако обработка исключений в C# отличается ясностью и полнотой реализации.

Преимущество подсистемы обработки исключений состоит в автоматизации создания большей части кода, который ранее необходимо было вводить в программы "вручную". Например, в любом компьютерном языке при отсутствии такой подсистемы практически каждый метод возвращал коды ошибок, и эти значения проверялись вручную при каждом вызове метода. Такой подход довольно утомителен, кроме того, при этом возможно возникновение ошибок. Обработка исключений упрощает "работу над ошибками", позволяя в программах определять блок кода, именуемый *обработчиком исключений*, который будет автоматически выполняться при возникновении определенной ошибки. В этом случае не обязательно проверять результат выполнения каждой конкретной операции или метода вручную. Если ошибка возникнет, ее должным образом обработает обработчик исключений.

Еще одним преимуществом обработки исключительных ситуаций в C# является определение стандартных исключений для таких распространенных программных ошибок, как деление на нуль или попадание вне диапазона определения индекса. Чтобы отреагировать на возникновение таких ошибок, программа должна отслеживать и обрабатывать эти исключения.

Без знания возможностей C#-подсистемы обработки исключений успешное программирование на C# попросту невозможно.

Класс `System.Exception`

В C# исключения представляются классами. Все классы исключений должны быть выведены из встроенного класса исключений `Exception`, который является частью пространства имен `System`. Таким образом, все исключения — подклассы класса `Exception`.

Из класса `Exception` выведены классы `SystemException` и `ApplicationException`. Они поддерживают две общие категории исключений, определенные в C#: те, которые генерируются C#-системой динамического управления, или общеязыковым средством управления (Common Language Runtime — CLR), и те, которые генерируются прикладными программами. Но ни класс `SystemException`, ни класс `ApplicationException` не привносят ничего нового в дополнение к членам класса `Exception`. Они просто определяют вершины двух различных иерархий классов исключений.

C# определяет встроенные исключения, которые выводятся из класса `SystemException`. Например, при попытке выполнить деление на нуль генерируется исключение класса `DivideByZeroException`. Как будет показано ниже в этой главе, вы сможете создавать собственные классы исключений, выводя их из класса `ApplicationException`.

Основы обработки исключений

Управление C#-механизмом обработки исключений зиждется на четырех ключевых словах: `try`, `catch`, `throw` и `finally`. Они образуют всзимосвязанную подсисте-

му, в которой использование одного из них предполагает использование другого. В этой главе каждое слово рассматривается подробно. Однако для начала будет полезно получить общее представление о роли, которую они играют в обработке исключительных ситуаций. Если кратко, то их работа состоит в следующем.

Программные инструкции, которые нужно проконтролировать на предмет исключений, помещаются в try-блок. Если исключение таки возникает в этом блоке, оно дает знать о себе *выбросом* определенного рода информации. Это *выброшенное исключение* может быть перехвачено программным путем с помощью catch-блока и обработано соответствующим образом. Системные исключения автоматически генерируются C#-системой динамического управления. Чтобы сгенерировать исключение вручную, используется ключевое слово throw. Любой код, который должен быть обязательно выполнен при выходе из try-блока, помещается в блок finally.

Использование try- и catch-блоков

Ядром обработки исключений являются блоки try и catch. Эти ключевые слова работают "в одной связке"; нельзя использовать слово try без catch или catch без try. Вот каков формат записи try/catch-блоков обработки исключений:

```
try {
// Блок кода, подлежащий проверке на наличие ошибок.
}
catch (ExcepType1 exOb) {
// Обработчик для исключения типа ExcepType1.
}
catch (ExcepType2 exOb) {
// Обработчик для исключения типа ExcepType2.
}...
```

Здесь *ExcepType* — это тип сгенерированного исключения. После "выброса" исключение перехватывается соответствующей инструкцией catch, которая его обрабатывает. Как видно из формата записи try/catch-блоков, с try-блоком может быть связана не одна, а несколько catch-инструкций. Какая именно из них будет выполнена, определит тип исключения. Другими словами, будет выполнена та catch-инструкция, тип исключения которой совпадает с типом сгенерированного исключения (а все остальные будут проигнорированы). После перехвата исключения параметр *exOb* примет его значение.

Задавать параметр *exOb* необязательно. Если обработчику исключения не нужен доступ к объекту исключения (как это часто бывает), в задании параметра *exOb* нет необходимости. Поэтому во многих примерах этой главы параметр *exOb* не задан.

Важно понимать следующее. Если исключение не генерируется, try-блок завершается нормально, и все его catch-инструкции игнорируются. Выполнение программы продолжается с первой инструкции, которая стоит после последней инструкции catch. Таким образом, catch-инструкция (из предложенных после try-блока) выполняется только в случае, если сгенерировано соответствующее исключение.

Пример обработки исключения

Ниже приведен простой пример, демонстрирующий, как отследить и перехватить исключение. Известно, что попытка индексировать массив за пределами его границ вызывает ошибку нарушения диапазона. В этом случае C#-система динамического управления генерирует исключение типа IndexOutOfRangeException, которое представляет собой стандартное исключение, определенное языком C#. В следующей программе такое исключение намеренно генерируется, а затем перехватывается.

```
// Демонстрация обработки исключений.

using System;

class ExcDemo1 {
  public static void Main() {
    int[] nums = new int[4];

    try {
      Console.WriteLine(
                  "Перед генерированием исключения.");

      // Генерируем исключение, связанное с попаданием
      // индекса вне диапазона.
      for(int i=0; i < 10; i++) {
        nums[i] = i;
        Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
      }

      Console.WriteLine("Этот текст не отображается.");
    }
    catch (IndexOutOfRangeException) {
      // Перехватываем исключение.
      Console.WriteLine("Индекс вне диапазона!");
    }
    Console.WriteLine("После catch-инструкции.");
  }
}
```

При выполнении этой программы получаем такие результаты:

```
Перед генерированием исключения.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
Индекс вне диапазона!
После catch-инструкции.
```

Обратите внимание на то, что `nums` — это `int`-массив для хранения четырех элементов. Однако в цикле `for` делается попытка индексировать этот массив от 0 до 9, и как только значение индекса устанавливается равным четырем, генерируется исключение типа `IndexOutOfRangeException`.

Несмотря на небольшой размер, предыдущая программа иллюстрирует ряд ключевых аспектов обработки исключений. Во-первых, проверяемый код содержится внутри `try`-блока. Во-вторых, при возникновении исключения (в данном случае из-за попытки внутри `for`-цикла индексировать массив `nums` за границами его диапазона) выполнение `try`-блока прекращается, а само исключение перехватывается `catch`-инструкцией. Другими словами, управление программой передается `catch`-инструкции, независимо от того, все ли инструкции `try`-блока выполнились. При этом важно то, что инструкция `catch` *не* вызывается, а ей передается управление программой. Поэтому инструкция

```
Console.WriteLine("После catch-инструкции.");
```

никогда не выполнится. После выполнения `catch`-инструкции программа продолжится со следующей инструкции. Следовательно, чтобы ваша программа могла нормально продолжить свое выполнение, обработчик должен устранить проблему, которая стала причиной возникновения исключительной ситуации.

Обратите внимание на то, что в инструкции `catch` параметр отсутствует. Как упоминалось выше, параметр необходим только в тех случаях, когда требуется доступ к объекту исключения. В некоторых случаях значение объекта исключения используется обработчиком для получения дополнительной информации об ошибке, но чаще всего достаточно просто знать о том, что исключение имело место. Следовательно, в отсутствии `catch`-параметра в обработчике исключения нет ничего необычного, как в случае, проиллюстрированном предыдущей программой.

Как уже упоминалось, если `try`-блоком исключение не сгенерировано, ни одна из `catch`-инструкций не выполняется, и управление программой будет передано инструкции, следующей за `catch`-инструкцией. Чтобы убедиться в этом, замените в предыдущей программе эту инструкцию `for`-цикла

```
for(int i=0; i < 10; i++) {
```

такой:

```
for(int i=0; i < nums.Length; i++) {
```

Теперь цикл `for` не нарушает границы индексирования массива `nums`. Поэтому исключение не генерируется, и `catch`-блок не выполняется.

Второй пример исключения

Весь код, выполняемый внутри `try`-блока, проверяется на предмет возникновения исключительной ситуации. Сюда также относятся исключения, которые могут сгенерировать методы, вызываемые из блока `try`. Исключение, сгенерированное методом, вызванным из `try`-блока, может быть перехвачено этим `try`-блоком, если, конечно, метод сам не перехватит это исключение. Рассмотрим пример.

```
/* Исключение может сгенерировать один метод, а
   перехватить его -- другой. */

using System;

class ExcTest {
  // Генерируем исключение.
  public static void genException() {
    int[] nums = new int[4];

    Console.WriteLine("Перед генерированием исключения.");

    // Генерируем исключение, связанное с попаданием
    // индекса вне диапазона.
    for(int i=0; i < 10; i++) {
      nums[i] = i;
      Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
    }

    Console.WriteLine("Этот текст не будет отображаться.");
  }
}

class ExcDemo2 {
  public static void Main() {

    try {
      ExcTest.genException();
    }
    catch (IndexOutOfRangeException) {
```

```
      // Перехватываем исключение.
      Console.WriteLine("Индекс вне диапазона!");
    }
    Console.WriteLine("После catch-инструкции.");
  }
}
```

Эта программа показывает результаты, которые не отличаются от результатов выполнения предыдущей ее версии:

```
Перед генерированием исключения.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
Индекс вне диапазона!
После catch-инструкции.
```

Поскольку метод `genException()` вызывается из блока `try`, исключение, которое он генерирут (и не перехватывает), перехватывается инструкцией `catch` в методе `Main()`. Но если бы метод `genException()` перехватывал это исключение, оно бы никогда не вернулось в метод `Main()`.

Последствия возникновения неперехватываемых исключений

Перехват одного из стандартных C#-исключений, как показала предыдущая программа, имеет побочный эффект: он предотвращает аварийное окончание программы. При генерировании исключения оно должно быть перехвачено программным кодом. Если программа не перехватывает исключение, оно перехвачивается C#-системой динамического управления. Но дело в том, что система динамического управления сообщит об ошибке и завершит программу. Например, в следующем примере исключение, связанное с нарушением границ диапазона, программой не перехватывается.

```
// Предоставим возможность обработать ошибку
// C#-системе динамического управления.

using System;

class NotHandled {
  public static void Main() {
    int[] nums = new int[4];

    Console.WriteLine("Перед генерированием исключения.");

    // Генерируем исключение, связанное с попаданием
    // индекса вне диапазона.
    for(int i=0; i < 10; i++) {
      nums[i] = i;
      Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
    }

  }
}
```

При неверном индексировании массива выполнение программы останавливается, и на экране отображается следующее сообщение об ошибке:

```
Unhandled Exception: System.IndexOutOfRangeException:
Index was outside the bounds of the array.
at NotHandled.Main()
```

Это сообщение уведомляет об обнаружении в методе `NotHandled.Main()` необработанного исключения типа `System.IndexOutOfRangeException`, которое связано с выходом индекса массива за границы диапазона.

Несмотря на то что такое сообщение может быть полезным во время отладки программы, вряд ли вы захотите, чтобы его увидели пользователи! Поэтому важно, чтобы программы сами обрабатывали подобные исключения.

Как упоминалось выше, тип исключения должен совпадать с типом, заданным в `catch`-инструкции. В противном случае это исключение не будет перехвачено. Например, в следующей программе делается попытка перехватить ошибку нарушения индексом массива границ диапазона с помощью `catch`-инструкции для класса `DivideByZeroException` (это еще одно из встроенных C#-исключений). При нарушении границ диапазона, допустимого для индекса массива, генерируется исключение типа `IndexOutOfRangeException`, которое не перехватывается предусмотренной в программе `catch`-инструкцией. В результате программа завершается аварийно.

```
// Эта программа работать не будет!

using System;

class ExcTypeMismatch {
  public static void Main() {
    int[] nums = new int[4];

    try {
      Console.WriteLine("Перед генерированием исключения.");

      // Генерируем исключение, связанное с попаданием
      // индекса вне диапазона.
      for(int i=0; i < 10; i++) {
        nums[i] = i;
        Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
      }

      Console.WriteLine("Этот текст не отображается.");
    }

    /* Если в catch-инструкции указан тип исключения
       DivideByZeroException, то с ее помощью невозможно
       перехватить ошибку нарушения границ массива. */
    catch (DivideByZeroException) {
      // Перехватываем исключение.
      Console.WriteLine("Индекс вне границ диапазона!");
    }
    Console.WriteLine("После catch-инструкции.");
  }
}
```

Вот как выглядят результаты выполнения этой программы:

```
Перед генерированием исключения.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
```

```
Unhandled Exception: System.IndexOutOfRangeException:
Index was outside the bounds of the array.
   at ExcTypeMismatch.Main()
```

Как подтверждают результаты выполнения этой программы, `catch`-инструкция, предназначенная для перехвата исключения типа `DivideByZeroException`, не в состоянии перехватить исключение типа `IndexOutOfRangeException`.

Возможность красиво выходить из ошибочных ситуаций

Одно из основных достоинств обработки исключений состоит в том, что она позволяет программе отреагировать на ошибку и продолжить выполнение. Рассмотрим, например, следующую программу, которая делит элементы одного массива на элементы другого. Если при этом встречается деление на нуль, генерируется исключение типа `DivideByZeroException`. В программе это исключение обрабатывается выдачей сообщения об ошибке, после чего выполнение программы продолжается. Следовательно, попытка разделить на нуль не вызывает внезапную динамическую ошибку, в результате которой прекращается выполнение программы. Вместо аварийного останова исключение позволяет красиво выйти из ошибочной ситуации и продолжить выполнение программы.

```
// Достойная реакция на ошибку и продолжение работы --
// вот что значит с толком использовать исключения!

using System;

class ExcDemo3 {
  public static void Main() {
    int[] numer = { 4, 8, 16, 32, 64, 128 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i < numer.Length; i++) {
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " равно " +
                          numer[i]/denom[i]);
      }
      catch (DivideByZeroException) {
        // Перехватываем исключение.
        Console.WriteLine("Делить на нуль нельзя!");
      }
    }
  }
}
```

При выполнении эта программа демонстрирует следующие результаты:

```
4 / 2 равно 2
Делить на нуль нельзя!
16 / 4 равно 4
32 / 4 равно 8
Делить на нуль нельзя!
128 / 8 равно 16
```

Эта программа демонстрирует еще один важный аспект обработки исключений. После обработки исключение удаляется из системы. Таким образом, в этой программе

при каждом проходе через цикл заново вводится try-блок, обеспечивая полную "готовность" к обработке следующих исключений. Такая организация позволяет обрабатывать в программах повторяющиеся ошибки.

Использование нескольких catch-инструкций

С try-блоком можно связать не одно, а несколько catch-инструкций. И это — довольно распространенная практика программирования. Однако все catch-инструкции должны перехватывать исключения различного типа. Например, следующая программа перехватывает как ошибку нарушения границ массива, так и ошибку деления на нуль.

```
// Использование нескольких catch-инструкций.

using System;

class ExcDemo4 {
  public static void Main() {
    // Здесь массив numer длинне массива denom.
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i < numer.Length; i++) {
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " равно " +
                          numer[i]/denom[i]);
      }
      catch (DivideByZeroException) {
        // Перехватываем исключение.
        Console.WriteLine("Делить на нуль нельзя!");
      }
      catch (IndexOutOfRangeException) {
        // Перехватываем исключение.
        Console.WriteLine("Нет соответствующего элемента.");
      }
    }
  }
}
```

Эта программа генерирует следующие результаты:

```
4 / 2 равно 2
Делить на нуль нельзя!
16 / 4 равно 4
32 / 4 равно 8
Делить на нуль нельзя!
128 / 8 равно 16
Нет соответствующего элемента.
Нет соответствующего элемента.
```

Как подтверждают результаты выполнения этой программы, каждая catch-инструкция реагирует только на собственный тип исключения.

В общем случае catch-выражения проверяются в том порядке, в котором они встречаются в программе. Выполняется только инструкция, тип исключения которой совпадает со сгенерированным исключением. Все остальные catch-блоки игнорируются.

Перехват всех исключений

Иногда требуется перехватывать все исключения, независимо от их типа. Для этого используйте `catch`-инструкцию без параметров. В этом случае создается обработчик "глобального перехвата", который используется, чтобы программа гарантированно обработала все исключения. В следующей программе приведен пример использования такого обработчика, который успешно перехватывает генерируемые здесь исключение типа `IndexOutOfRangeException` и исключение типа `DivideByZeroException`.

```
// Использование catch-инструкции для
// "глобального перехвата".

using System;

class ExcDemo5 {
  public static void Main() {
    // Здесь массив numer длиннее массива denom.
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i < numer.Length; i++) {
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " равно " +
                          numer[i]/denom[i]);
      }
      catch {
        Console.WriteLine(
                    "Произошло некоторое исключение.");
      }
    }
  }
}
```

Вот как выглядят результаты выполнения этой программы:

```
4 / 2 равно 2
Произошло некоторое исключение.
16 / 4 равно 4
32 / 4 равно 8
Произошло некоторое исключение.
128 / 8 равно 16
Произошло некоторое исключение.
Произошло некоторое исключение.
```

В отношении `catch`-инструкции, предназначенной для "глобального перехвата", необходимо запомнить следующее: она должна быть последней в последовательности `catch`-инструкций.

Вложение `try`-блоков

Один `try`-блок можно вложить в другой. Исключение, сгенерированное во внутреннем `try`-блоке и не перехваченное `catch`-инструкцией, которая связана с этим `try`-блоком, передается во внешний `try`-блок. Например, в следующей программе исключение типа `IndexOutOfRangeException` перехватывается не внутренним `try`-блоком, а внешним.

```
// Использование вложенного try-блока.

using System;

class NestTrys {
  public static void Main() {
    // Здесь массив numer длиннее массива denom.
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    try { // Внешний try-блок.
      for(int i=0; i < numer.Length; i++) {
        try { // Вложенный try-блок.
          Console.WriteLine(numer[i] + " / " +
                            denom[i] + " равно " +
                            numer[i]/denom[i]);
        }
        catch (DivideByZeroException) {
          // Перехватываем исключение.
          Console.WriteLine("На нуль делить нельзя!");
        }
      }
    }
    catch (IndexOutOfRangeException) {
      // Перехватываем исключение.
      Console.WriteLine("Нет соответствующего элемента.");
      Console.WriteLine(
           "Неисправимая ошибка -- программа завершена.");
    }
  }
}
```

Вот результаты выполнения этой программы:

```
4 / 2 равно 2
На нуль делить нельзя!
16 / 4 равно 4
32 / 4 равно 8
На нуль делить нельзя!
128 / 8 равно 16
Нет соответствующего элемента.
Неисправимая ошибка -- программа завершена.
```

Исключение, которое может быть обработано внутренним `try`-блоком (в данном случае это деление на нуль), позволяет программе продолжать работу. Однако нарушение границ массива перехватывается внешним `try`-блоком и заставляет программу завершиться.

В предыдущей программе хочется обратить ваше внимание вот на что. Чаще всего использование вложенных `try`-блоков обусловлено желанием обрабатывать различные категории ошибок различными способами. Одни типы ошибок носят катастрофический характер и не подлежат исправлению. Другие — неопасны для дальнейшего функционирования программы, и с ними можно справиться прямо на месте их возникновения. Многие программисты используют внешний `try`-блок для перехвата самых серьезных ошибок, позволяя внутренним `try`-блокам обрабатывать менее опасные. Внешние `try`-блоки можно также использовать в качестве механизма "глобального перехвата" для обработки тех ошибок, которые не перехватываются внутренним блоком.

Генерирование исключений вручную

В предыдущих примерах демонстрировался перехват исключений, сгенерированных автоматически средствами C#. Однако можно сгенерировать исключение вручную, используя инструкцию `throw`. Формат ее записан таков:

```
throw exceptOb;
```

Элемент *exceptOb* — это объект класса исключений, производного от класса `Exception`.

Рассмотрим пример, который демонстрирует использование инструкции `throw` для генерирования исключения типа `DivideByZeroException` вручную.

```
// Генерирование исключения вручную.

using System;

class ThrowDemo {
  public static void Main() {
    try {
      Console.WriteLine("До генерирования исключения.");
      throw new DivideByZeroException();
    }
    catch (DivideByZeroException) {
      // Перехватываем исключение.
      Console.WriteLine("Исключение перехвачено.");
    }
    Console.WriteLine("После try/catch-блока.");
  }
}
```

Результаты выполнения этой программы имеют такой вид:

```
До генерирования исключения.
Исключение перехвачено.
После try/catch-блока.
```

Обратите внимание на то, как был создан объект исключения типа `DivideByZeroException`, а именно: с помощью оператора `new` в инструкции `throw`. Помните, что инструкция `throw` генерирует *объект*, ведь нельзя просто сгенерировать "тип исключения". В данном случае при создании объекта класса `DivideByZeroException` использовался конструктор по умолчанию, но для генерирования исключений предусмотрены и другие конструкторы.

Чаще всего генерируемые исключения являются экземплярами классов исключений, создаваемых в программе. Как будет показано далее в этой главе, создание собственных классов исключений позволяет обрабатывать ошибки в коде, и эта процедура может стать частью общей стратегии обработки исключений программы.

Повторное генерирование исключений

Исключение, перехваченное одной `catch`-инструкцией, можно перегенерировать, чтобы обеспечить возможность его перехвата другой (внешней) `catch`-инструкцией. Самая распространенная причина для повторного генерирования исключения — позволить нескольким обработчикам получить доступ к исключению. Например, возможна такая ситуация, что один обработчик исключений управляет одним аспектом исключения, а второй — другим. Чтобы повторно сгенерировать исключение, доста-

точно использовать ключевое слово throw, не указывая исключения. Другими словами, используйте следующую форму инструкции throw.

```
throw ;
```

Помните, что при повторном генерировании исключения оно не будет повторно перехватываться той же catch-инструкцией, а передается следующей catch-инструкции.

Повторное генерирование исключений демонстрируется в следующей программе. В данном случае она перегенерирует исключение типа IndexOutOfRangeException.

```
// Повторное генерирование исключения.

using System;

class Rethrow {
  public static void genException() {
    // Здесь массив numer длиннее массива denom.
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i<numer.Length; i++) {
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " равно " +
                          numer[i]/denom[i]);
      }
      catch (DivideByZeroException) {
        // Перехватываем исключение.
        Console.WriteLine("Делить на нуль нельзя!");
      }
      catch (IndexOutOfRangeException) {
        // Перехватываем исключение.
        Console.WriteLine(
                    "Нет соответствующего элемента.");
        throw; // Генерируем исключение повторно.
      }
    }
  }
}

class RethrowDemo {
  public static void Main() {
    try {
      Rethrow.genException();
    }
    catch(IndexOutOfRangeException) {
      // Перехватываем повторно
      // сгенерированное исключение.
     Console.WriteLine("Неисправимая ошибка -- " +
                       "программа завершена.");
    }
  }
}
```

В этой программе ошибки деления на нуль обрабатываются локально (по месту), т.е. в самом методе genException(), но ошибка нарушения границ массива генерируется повторно. В данном случае исключение типа IndexOutOfRangeException обрабатывается функцией Main().

Использование блока `finally`

Иногда возникает потребность определить программный блок, который должен выполняться по выходу из try/catch-блока. Например, исключение может вызвать ошибку, которая завершает текущий метод и, следовательно, является причиной преждевременного возврата. Однако такой метод может оставить открытым файл или соединение с сетью, которые необходимо закрыть. Подобные обстоятельства — обычное явление в программировании, и C# предоставляет удобный путь выхода из них: блок finally.

Чтобы определить блок кода, подлежащий выполнению по выходу из try/catch-блока, включите в конец try/catch-последовательности блок finally. Общая форма записи последовательности try/catch-блоков, содержащей блок finally, выглядит следующим образом.

```
try {
// Блок кода, предназначенный для обработки ошибок.
}
catch (ExcepType1 exOb) {
// Обработчик для исключения типа ExcepType1.
}
catch (ExcepType2 exOb) {
// Обработчик для исключения типа ExcepType2.
}
.
.
.
finally {
// Код завершения обработки исключений.
}
```

Блок finally будет выполнен после выхода из try/catch-блока, независимо от условий его выполнения. Другими словами, при нормальном завершении try-блока или в условиях возникновения исключения содержимое finally-блока будет безусловно отработано. Блок finally выполнится и в том случае, если любой код внутри try-блока или любая из его catch-инструкций определены внутри метода.

Вот пример использования блока finally:

```
// Использование блока finally.

using System;

class UseFinally {
  public static void genException(int what) {
    int t;
    int[] nums = new int[2];

    Console.WriteLine("Получаем " + what);
    try {
      switch(what) {
        case 0:
          t = 10 / what; // Генерируем ошибку
                         // деления на нуль.
          break;
        case 1:
          nums[4] = 4; // Генерируем ошибку
                       // индексирования массива.
```

```
          break;
        case 2:
          return; // Возврат из try-блока.
      }
    }
    catch (DivideByZeroException) {
      // Перехватываем исключение.
      Console.WriteLine("На нуль делить нельзя!");
      return; // Возврат из catch-блока.
    }
    catch (IndexOutOfRangeException) {
      // Перехватываем исключение.
      Console.WriteLine("Нет соответствующего элемента.");
    }
    finally {
      Console.WriteLine("По окончании try-блока.");
    }
  }
}

class FinallyDemo {
  public static void Main() {

    for(int i=0; i < 3; i++) {
      UseFinally.genException(i);
      Console.WriteLine();
    }
  }
}
```

Вот какие результаты получены при выполнении этой программы:

```
Получаем 0
На нуль делить нельзя!
По окончании try-блока.

Получаем 1
Нет соответствующего элемента.
По окончании try-блока.

Получаем 2
По окончании try-блока.
```

Как подтверждают результаты выполнения этой программы, независимо от итога завершения `try`-блока, блок `finally` выполняется обязательно.

Исключения "под микроскопом"

До сих пор мы перехватывали исключения, но ничего не делали с самим объектом исключения. Как упоминалось выше, `catch`-инструкция позволяет задать тип исключения и параметр. Параметр предназначен для принятия объекта исключения. Поскольку все классы исключений являются производными от класса `Exception`, все исключения поддерживают члены, определенные в этом классе. В этой главе мы рассмотрим наиболее полезные члены и конструкторы класса `Exception` и узнаем преимущества использования параметра `catch`-инструкции.

В классе `Exception` определен ряд свойств. Самые интересные из них: `Message`, `StackTrace` и `TargetSite`. Все они предназначены только для чтения. Свойство

Message содержит строку, которая описывает причину ошибки, а свойство StackTrace — строку со стеком вызовов, приведших к возникновению исключений. Свойство TargetSite принимает объект, который задает метод, сгенерировавший исключение.

В классе Exception также определен ряд методов. Чаще всего используется метод ToString(), который возвращает строку с описанием исключения. Метод ToString() автоматически вызывается, если некоторое исключение отображается, например, с помощью метода WriteLine(). Использование свойств и методов класса Exception демонстрируется в следующей программе.

```
// Использование членов класса Exception.

using System;

class ExcTest {
  public static void genException() {
    int[] nums = new int[4];

    Console.WriteLine("Перед генерированием исключения.");

    // Генерируем исключение, связанное с попаданием
    // индекса вне диапазона.
      for(int i=0; i < 10; i++) {
        nums[i] = i;
        Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
      }

    Console.WriteLine("Этот текст не отображается.");
  }
}

class UseExcept {
  public static void Main() {

    try {
      ExcTest.genException();
    }
    catch (IndexOutOfRangeException exc) {
      // Перехватываем исключение.
      Console.WriteLine("Стандартное сообщение таково: ");
      Console.WriteLine(exc); // Вызов метода ToString().
      Console.WriteLine("Свойство StackTrace: " +
                        exc.StackTrace);
      Console.WriteLine("Свойство Message: " +
                        exc.Message);
      Console.WriteLine("Свойство TargetSite: " +
                        exc.TargetSite);
    }
    Console.WriteLine("После catch-инструкции.");
  }
}
```

Вот результаты выполнения этой программы:

```
Перед генерированием исключения.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
```

```
Стандартное сообщение таково:
System.IndexOutOfRangeException: Index was outside
the bounds of the array.
   at ExcTest.genException()
   at UseExcept.Main()
Свойство StackTrace:    at ExcTest.genException()
   at UseExcept.Main()
Свойство Message: Index was outside the bounds of the array.
Свойство TargetSite: Void genException()
После catch-инструкции.
```

В классе `Exception` определено четыре конструктора. Наиболее часто используются такие:

```
Exception()
Exception(string str)
```

Первый — это конструктор по умолчанию. Второй принимает значение свойства `Message`, связанное с исключением. При создании собственных классов исключений необходимо реализовать оба этих конструктора.

Наиболее употребительные исключения

В пространстве имен `System` определено несколько стандартных встроенных исключений. Все они выведены из класса `SystemException`, поскольку генерируются системой динамического управления (Common Language Runtime) при возникновении динамических ошибок. Некоторые из самых употребительных стандартных исключений, определенных в C#, приведены в табл. 13.1.

Таблица 13.1. Наиболее употребительные исключения, определенные в пространстве имен `System`

Исключение	*Значение*
`ArrayTypeMismatchException`	Тип сохраняемого значения несовместим с типом массива
`DivideByZeroException`	Попытка деления на нуль
`IndexOutOfRangeException`	Индекс массива оказался вне диапазона
`InvalidCastException`	Неверно выполнено динамическое приведение типов
`OutOfMemoryException`	Обращение к оператору `new` оказалось неудачным из-за недостаточного объема свободной памяти
`OverflowException`	Имеет место арифметическое переполнение
`NullReferenceException`	Была сделана попытка использовать нулевую ссылку, т.е. ссылку, которая не указывает ни на какой объект
`StackOverflowException`	Переполнение стека

Большинство исключений, перечисленных в табл. 13.1, не нуждается в дополнительных разъяснениях, за исключением класса `NullReferenceException`. Это исключение генерируется при попытке использовать нулевую ссылку, например, при попытке вызвать метод, передав ему вместо ссылки на объект нулевую ссылку. *Нулевая ссылка* не указывает ни на какой объект. Один из способов создать нулевую ссылку — явно присвоить ссылочной переменной `null`-значение, используя ключевое слово `null`. Нулевые ссылки можно получить и другими путями, которые, однако, менее очевидны. Рассмотрим программу, в которой демонстрируется возникновение исключения типа `NullReferenceException`.

```
// Использование исключения типа NullReferenceException.

using System;

class X {
  int x;
  public X(int a) {
    x = a;
  }

  public int add(X o) {
    return x + o.x;
  }
}

// Демонстрируем исключение типа NullReferenceException.
class NREDemo {
  public static void Main() {
    X p = new X(10);
    X q = null; // Переменной q явно присваивается
                // значение null.
    int val;

    try {
      val = p.add(q); // Такой вызов метода
                      // приведет к исключению.
    } catch (NullReferenceException) {
      Console.WriteLine("NullReferenceException!");
      Console.WriteLine("Исправляем ошибку...\n");

      // Исправляем ошибку.
      q = new X(9);
      val = p.add(q);
    }

    Console.WriteLine("Значение val равно {0}", val);

  }
}
```

При выполнении этой программы получаем такие результаты:

```
NullReferenceException!
Исправляем ошибку...

Значение val равно 19
```

Эта программа создает класс `X`, в котором определяется член `x` и метод `add()`, предназначенный для сложения значения `x`, принадлежащего вызывающему объекту, с членом `x`, который определен в объекте, переданном в качестве параметра. В методе `Main()` создаются два объекта класса `X`. Первый, `p`, инициализируется, а второй, `q`, — нет (ему явным образом присваивается значение `null`). Затем вызывается метод `p.meth()`, которому значение `q` передается как аргумент. Поскольку переменная `q` не ссылается ни на один объект, при попытке получить значение члена `q.x` генерируется исключение типа `NullReferenceException`.

Заслуживает внимания исключение типа `StackOverflowException`, которое генерируется при переполнении системного стека. Оно может возникнуть при некорректном определении рекурсивного метода. Программисту, который увлекается рекурсией,

возможно, стоит внимательно отследить появление исключения этого типа, приняв соответствующие меры в случае его обнаружения. Однако здесь следует проявить осторожность. Если уж это исключение сгенерировано, значит, системный стек исчерпал свои возможности, поэтому лучше всего просто начать анализ с рекурсивного вызова.

Наследование классов исключений

Несмотря на то что встроенные C#-исключения обрабатывают самые распространенные ошибки, C#-механизм обработки исключений не ограничивается этими ошибками. В C# имеется возможность обрабатывать исключения, создаваемые программистом. В своих программах вы можете использовать для обработки ошибок "собственные" исключения. В создании исключения нет ничего сложного. Для этого достаточно определить класс как производный от класса `Exception`. Как правило, определяемые программистом исключения, должны быть производными от класса `ApplicationException`, "родоначальника" иерархии, зарезервированной для исключений, связанных с прикладными программами. Созданные вами производные классы не должны ничего реализовывать, поскольку одно лишь их существование в системе типов уже позволит использовать их в качестве исключений.

Классы исключений, создаваемые программистом, будут автоматически иметь свойства и методы, определенные в классе `Exception` и доступные для них. Конечно, один или несколько членов в новых классах можно переопределить.

Рассмотрим пример, в котором создается "пользовательский" тип исключения. В конце главы 10 был приведен пример разработки класса массива с именем `RangeArray`. Вспомним, что класс `RangeArray` поддерживает одномерные `int`-массивы, в которых начальный и конечный индексы задаются пользователем. Например, массив, индексы которого лежат в диапазоне от −5 до 27, абсолютно легален для объектов класса `RangeArray`. В главе 10 было показано, что при попадании индекса за пределы диапазона, устанавливалась переменная ошибки, определенная в классе `RangeArray`. Это означает, что переменную ошибки необходимо было проверять после каждой операции, в которой участвовал объект класса `RangeArray`. Безусловно, такое решение связано с ошибками и лишено "изящества". Предпочтительней, чтобы объект класса `RangeArray` при возникновении ошибки нарушения границ диапазона генерировал "свое" исключение. Именно такое решение и реализовано в следующей версии класса `RangeArray`.

```
// Создание пользовательского исключения для
// обнаружения ошибок при работе с объектами класса
// RangeArray.

using System;

// Создаем исключение для класса RangeArray.
class RangeArrayException : ApplicationException {
  // Реализуем стандартные конструкторы.
  public RangeArrayException() : base() { }
  public RangeArrayException(string str) : base(str) { }

  // Переопределяем метод ToString() для класса
  // RangeArrayException.
  public override string ToString() {
    return Message;
  }
}
```

```
// Улучшенная версия класса RangeArray.
class RangeArray {
  // Закрытые данные.
  int[] a; // Ссылка на базовый массив.
  int lowerBound; // Наименьший индекс.
  int upperBound; // Наибольший индекс.

  int len; // Базовая переменная для свойства Length.

  // Создаем массив с заданным размером.
  public RangeArray(int low, int high) {
    high++;
    if(high <= low) {
      throw new RangeArrayException(
                        "Нижний индекс не меньше верхнего.");
    }
    a = new int[high - low];
    len = high - low;

    lowerBound = low;
    upperBound = --high;
  }

  // Свойство Length, предназначенное только для чтения.
  public int Length {
    get {
      return len;
    }
  }

  // Индексатор для объекта класса RangeArray.
  public int this[int index] {
    // Средство для чтения элемента массива.
    get {
      if(ok(index)) {
        return a[index - lowerBound];
      } else {
        throw new RangeArrayException(
                  "Ошибка нарушения границ диапазона.");
      }
    }

    // Средство для записи элемента массива.
    set {
      if(ok(index)) {
        a[index - lowerBound] = value;
      }
      else throw new RangeArrayException(
                  "Ошибка нарушения границ диапазона.");
    }
  }

  // Метод возвращает значение true,
  // если индекс в пределах диапазона.
  private bool ok(int index) {
    if(index >= lowerBound & index <= upperBound)
      return true;
    return false;
  }
}
```

```
// Демонстрируем использование массива с заданным
// диапазоном изменения индекса.
class RangeArrayDemo {
  public static void Main() {
    try {
      RangeArray ra = new RangeArray(-5, 5);
      RangeArray ra2 = new RangeArray(1, 10);

      // Демонстрируем использование объекта-массива ra.
      Console.WriteLine("Длина массива ra: " + ra.Length);

      for(int i = -5; i <= 5; i++)
        ra[i] = i;

      Console.Write("Содержимое массива ra: ");
      for(int i = -5; i <= 5; i++)
        Console.Write(ra[i] + " ");

      Console.WriteLine("\n");

      // Демонстрируем использование объекта-массива ra2.
      Console.WriteLine("Длина массива ra2: " + ra2.Length);

      for(int i = 1; i <= 10; i++)
        ra2[i] = i;

      Console.Write("Содержимое массива ra2: ");
      for(int i = 1; i <= 10; i++)
        Console.Write(ra2[i] + " ");

      Console.WriteLine("\n");

    } catch (RangeArrayException exc) {
       Console.WriteLine(exc);
    }

    // Теперь демонстрируем "работу над ошибками".
    Console.WriteLine(
           "Сгенерируем ошибки непопадания в диапазон.");

    // Используем неверно заданный конструктор.
    try {
      RangeArray ra3 = new RangeArray(100, -10); // Ошибка!

    } catch (RangeArrayException exc) {
       Console.WriteLine(exc);
    }

    // Используем неверно заданный индекс.
    try {
      RangeArray ra3 = new RangeArray(-2, 2);

      for(int i = -2; i <= 2; i++)
        ra3[i] = i;

      Console.Write("Содержимое массива ra3: ");
      for(int i = -2; i <= 10; i++) // Ошибка непопадания
                                    // в диапазон.
        Console.Write(ra3[i] + " ");
```

```
    } catch (RangeArrayException exc) {
      Console.WriteLine(exc);
    }
  }
}
```

При выполнении этой программы получаем такие результаты:

```
Длина массива ra: 11
Содержимое массива ra: -5 -4 -3 -2 -1 0 1 2 3 4 5

Длина массива ra2: 10
Содержимое массива ra2: 1 2 3 4 5 6 7 8 9 10

Сгенерируем ошибки непопадания в диапазон.
Нижний индекс не меньше верхнего.
Содержимое массива ra3: -2 -1 0 1 2 Ошибка нарушения границ диапазона.
```

При возникновении ошибки нарушения границ диапазона `RangeArray`-объект генерирует объект типа `RangeArrayException`. Этот класс — производный от класса `ApplicationException`. Как упоминалось выше, класс исключений, создаваемый программистом, обычно выводится из класса `ApplicationException`. Обратите внимание на то, что подобная ошибка может обнаружиться во время создания `RangeArray`-объекта. Чтобы перехватывать такие исключения, объекты класса `RangeArray` должны создаваться внутри блока `try`, как это показано в программе. С использованием исключений для сообщений об ошибках класс `RangeArray` теперь напоминает один из встроенных C#-типов и может быть полностью интегрирован в механизм обработки исключений любой программы.

Прежде чем переходить к следующему разделу, "поиграйте" с этой программой. Например, попробуйте закомментировать переопределение метода `ToString()` и посмотрите результат. Попробуйте также создать исключение, используя конструктор по умолчанию, и посмотрите, какое сообщение в этом случае сгенерирует C#.

Перехват исключений производных классов

При перехвате исключений, типы которых включают базовые и производные классы, необходимо обращать внимание на порядок `catch`-инструкций, поскольку `catch`-инструкция для базового класса соответствует любой `catch`-инструкции производных классов. Например, поскольку базовым классом для всех исключений является `Exception`, при перехвате исключения типа `Exception` будут перехватываться все возможные исключения. Конечно, как упоминалось выше, использование `catch`-инструкции без аргумента обеспечивает более ясный способ перехвата всех исключений. Однако проблема перехвата исключений производных классов очень важна в других контекстах, особенно в случае создания собственных классов исключений.

Если нужно перехватывать исключения и базового, и производного класса, поместите первой в `catch`-последовательности инструкцию с заданием производного класса. В противном случае `catch`-инструкция с заданием базового класса будет перехватывать все исключения производных классов. Это правило — вынужденная мера, поскольку размещение `catch`-инструкции для базового класса перед остальными `catch`-инструкциями сделает их недостижимыми, и они никогда не будут выполнены. В C# недостижимая `catch`-инструкция считается ошибкой.

Следующая программа создает два класса исключений с именами `ExceptA` и `ExceptB`. Класс `ExceptA` выводится из класса `ApplicationException`, а класс `ExceptB` — из класса `ExceptA`. Затем программа генерирует исключение каждого типа.

```
// Инструкции перехвата исключений производных классов
// должны стоять перед инструкциями перехвата исключений
// базовых классов.

using System;

// Создаем класс исключения.
class ExceptA : ApplicationException {
  public ExceptA() : base() { }
  public ExceptA(string str) : base(str) { }

  public override string ToString() {
    return Message;
  }
}

// Создаем класс исключения как производный
// от класса ExceptA.
class ExceptB : ExceptA {
  public ExceptB() : base() { }
  public ExceptB(string str) : base(str) { }

  public override string ToString() {
    return Message;
  }
}

class OrderMatters {
  public static void Main() {
    for(int x = 0; x < 3; x++) {
      try {
        if(x==0) throw new ExceptA(
                      "Перехват исключения типа ExceptA.");
        else if(x==1) throw new ExceptB(
                      "Перехват исключения типа ExceptB.");
        else throw new Exception();
      }
      catch (ExceptB exc) {
        // Перехватываем исключение.
        Console.WriteLine(exc);
      }
      catch (ExceptA exc) {
        // Перехватываем исключение.
        Console.WriteLine(exc);
      }
      catch (Exception exc) {
        Console.WriteLine(exc);
      }
    }
  }
}
```

Вот результаты выполнения этой программы:

```
Перехват исключения типа ExceptA.
Перехват исключения типа ExceptB.
System.Exception: Exception of type System.Exception was thrown.
   at OrderMatters.Main()
```

Обратите внимание на порядок следования `catch`-инструкций. Это — единственно правильный вариант. Поскольку класс `ExceptB` выведен из класса `ExceptA`, `catch`-инструкция для исключений типа `ExceptB` должна стоять перед инструкцией, предна-

значенной для перехвата исключений типа ExceptA. Точно так же catch-инструкция для исключений класса Exception (который является базовым для всех исключений) должна стоять последней. Чтобы убедиться в этом, попробуйте переставить catch-инструкции в другом порядке. Это приведет к ошибке при компиляции.

Часто catch-инструкцию, определенную для исключений базового класса, успешно используют для перехвата целой категории исключений. Предположим, вы создаете набор исключений для некоторого устройства. Если вывести все эти исключения из общего базового класса, то приложения, которым не требуется подробные сведения о возникшей проблеме, могли бы просто перехватывать исключение, настроенное на базовый класс, избегнув тем самым ненужного дублирования кода.

Использование ключевых слов checked и unchecked

В C# предусмотрено специальное средство, которое связано с генерированием исключений, связанных с переполнением в арифметических вычислениях. Как вы знаете, в некоторых случаях при вычислении арифметических выражений получается результат, который выходит за пределы диапазона, определенного для типа данных в выражении. В этом случае говорят, что произошло переполнение результата. Рассмотрим, например, такой фрагмент программы:

```
byte a, b, result;
a = 127;
b = 127;

result = (byte)(a * b);
```

Здесь произведение значений a и b превышает диапазон представления значений типа byte. Следовательно, результат вычисления этого выражения вызвал переполнение для типа переменной result.

C# позволяет управлять генерированием исключений при возникновении переполнения с помощью ключевых слов checked и unchecked. Чтобы указать, что некоторое выражение должно быть проконтролировано на предмет переполнения, используйте ключевое слово checked. А чтобы проигнорировать переполнение, используйте ключевое слово unchecked. В последнем случае результат будет усечен так, чтобы его тип соответствовал типу-результату выражения.

Ключевое слово checked имеет две формы. Одна проверяет конкретное выражение и называется *операторной* checked-*формой*. Другая же проверяет блок инструкций.

```
checked (expr)

checked {
  // Инструкции, подлежащие проверке.
}
```

Здесь *expr* — выражение, которое необходимо контролировать. Если значение контролируемого выражения переполнилось, генерируется исключение типа OverflowException.

Ключевое слово unchecked имеет две формы. Одна из них — операторная форма, которая позволяет игнорировать переполнение для заданного выражения. Вторая игнорирует переполнение, которое возможно в блоке инструкций.

```
unchecked (expr)
```

```
unchecked {
  // Инструкции, для которых переполнение игнорируется.
}
```

Здесь *expr* — выражение, которое не проверяется на предмет переполнения. В случае переполнения это выражение усекается.

Рассмотрим программу, которая демонстрирует использование как слова checked, так и слова unchecked.

```
// Использование ключевых слов checked и unchecked.

using System;

class CheckedDemo {
  public static void Main() {
    byte a, b;
    byte result;

    a = 127;
    b = 127;

    try {
      result = unchecked((byte)(a * b));
      Console.WriteLine("Unchecked-результат: " + result);

      result = checked((byte)(a * b)); // Эта инструкция
                                       // вызывает исключение.
      Console.WriteLine("Checked-результат: " +
                        result); // Инструкция не будет
                                 // выполнена.
    }
    catch (OverflowException exc) {
      // Перехватываем исключение.
      Console.WriteLine(exc);
    }
  }
}
```

При выполнении этой программы получаются такие результаты:

```
Unchecked-результат: 1
System.OverflowException: Arithmetic operation resulted in an
overflow.
   at CheckedDemo.Main()
```

Как видите, результат непроверяемого выражения усекается. В случае разрешения проверки переполнения было бы сгенерировано исключение.

В предыдущей программе было продемонстрировано использование ключевых слов checked и unchecked для одного выражения. На примере следующей программы показано, как можно избежать переполнения при выполнении блока инструкций.

```
// Использование ключевых слов checked и unchecked
// для блоков инструкций.

using System;

class CheckedBlocks {
  public static void Main() {
    byte a, b;
    byte result;
```

```
    a = 127;
    b = 127;

    try {
      unchecked {
        a = 127;
        b = 127;
        result = unchecked((byte)(a * b));
        Console.WriteLine("Unchecked-результат: " + result);

        a = 125;
        b = 5;
        result = unchecked((byte)(a * b));
        Console.WriteLine("Unchecked-результат: " + result);
      }

      checked {
        a = 2;
        b = 7;
        result = checked((byte)(a * b)); // Все в порядке.
        Console.WriteLine("Checked-результат: " + result);

        a = 127;
        b = 127;
        result = checked((byte)(a * b)); // Здесь должно
                                         // быть сгенерировано
                                                // исключение.
        Console.WriteLine("Checked-результат: " +
                          result); // Эта инструкция не
                                   // выполнится.
      }
    }
    catch (OverflowException exc) {
      // Перехватываем исключение.
      Console.WriteLine(exc);
    }
  }
}
```

Вот как выглядят результаты выполнения этой программы:

```
Unchecked-результат: 1
Unchecked-результат: 113
Checked-результат: 14
System.OverflowException: Arithmetic operation resulted in an overflow.
   at CheckedBlocks.Main()
```

Как видите, при выполнении unchecked-блока результат вычисления выражения при наличии переполнения усекается. При возникновении переполнения в checked-блоке генерируется исключение.

Управление генерированием исключений с помощью ключевых слов checked или unchecked может быть полезным в том случае, когда checked/unchecked-статус переполнения определяется использованием соответствующей опции компилятора и настройкой самой среды выполнения. Таким образом, для некоторых типов программ лучше всего явно задавать статус контроля переполнения.

Полный справочник по

Глава 14

Использование средств ввода-вывода

С самого начала книги мы использовали часть C#-системы ввода-вывода — метод `Console.WriteLine()`, но не давали подробных пояснений по этому поводу. Поскольку C#-система ввода-вывода построена на иерархии классов, то ее теорию и детали невозможно освоить, не рассмотрев сначала классы, наследование и исключения. Теперь настало время для подробного изучения C#-средств ввода-вывода. Поскольку в C# используется система ввода-вывода и классы, определенные средой .NET Framework, в эту тему включено рассмотрение в общих чертах системы ввода-вывода .NET-среды.

В этой главе рассматриваются средства как консольного, так и файлового ввода-вывода. Необходимо сразу отметить, что C#-система ввода-вывода — довольно обширная тема, и здесь описаны лишь самые важные и часто применяемые средства.

Организация C#-системы ввода-вывода

C#-программы выполняют операции ввода-вывода посредством потоков. *Поток* (stream) — это абстракция, которая либо синтезирует информацию, либо потребляет ее. Поток связывается с физическим устройством с помощью C#-системы ввода-вывода. Характер поведения всех потоков одинаков, несмотря на различные физические устройства, с которыми они связываются. Следовательно, классы и методы ввода-вывода можно применить ко многим типам устройств. Например, методы, используемые для записи данных на консольное устройство, также можно использовать для записи в дисковый файл.

Байтовые и символьные потоки

На самом низком уровне все C#-системы ввода-вывода оперируют байтами. И это логично, поскольку многие устройства при выполнении операций ввода-вывода ориентированы на байты. Однако для человека привычнее оперировать символами. Вспомните, что в C# `char` — это 16-разрядный тип, а `byte` — 8-разрядный. Если вы используете набор символов ASCII, то в преобразовании типов `char` и `byte` нет ничего сложного: достаточно проигнорировать старший байт `char`-значения. Но такой подход не сработает для остальных Unicode-символов, которым необходимы оба байта. Таким образом, байтовые потоки не вполне подходят для обработки текстового (ориентированного на символы) ввода-вывода. Для решения этой проблемы в C# определен ряд классов, которые преобразуют байтовый поток в символьный, выполняя `byte-char`- и `char-byte`-перевод автоматически.

Встроенные потоки

Тремя встроенными потоками, доступ к которым осуществляется через свойства `Console.In`, `Console.Out` и `Console.Error`, могут пользоваться все программы, работающие в пространстве имен `System`. Свойство `Console.Out` относится к стандартному выходному потоку. По умолчанию это консоль. Например, при вызове метода `Console.WriteLine()` информация автоматически передается в поток `Console.Out`. Свойство `Console.In` относится к стандартному входному потоку, источником которого по умолчанию является клавиатура. Свойство `Console.Error` относится к ошибкам в стандартном потоке, источником которого также по умолчанию является консоль. Однако эти потоки могут быть перенаправлены на любое совместимое устройство ввода-вывода. Стандартные потоки являются символьными. Следовательно, эти потоки считывают и записывают символы.

Классы потоков

В C# определены как байтовые, так и символьные классы потоков. Однако символьные классы потоков в действительности представляют собой оболочки, которые преобразуют базовый байтовый поток в символьный, причем любое преобразование выполняется автоматически. Таким образом, символьные потоки построены на основе байтовых, несмотря на то, что они логически отделены друг от друга.

Все классы потоков определены в пространстве имен `System.IO`. Чтобы иметь возможность использовать эти классы, в начало программы необходимо включить следующую инструкцию:

```
using System.IO;
```

Для ввода и вывода на консоль не нужно задавать пространство имен `System.IO`, поскольку класс `Console` определен в пространстве имен `System`.

Класс `Stream`

Центральную часть потоковой C#-системы занимает класс `System.IO.Stream`. Класс `Stream` представляет байтовый поток и является базовым для всех остальных потоковых классов. Этот класс также абстрактный, т.е. мы не можем создать его объект. В классе `Stream` определен набор стандартных потоковых операций. Наиболее применяемые методы, определенные в классе `Stream`, перечислены в табл. 14.1.

Таблица 14.1. Некоторые методы класса `Stream`

Метод	*Описание*
`void Close()`	Закрывает поток
`void Flush()`	Записывает содержимое потока в физическое устройство
`int ReadByte()`	Возвращает целочисленное представление следующего доступного байта потока. При обнаружении конца файла возвращает значение -1
`int Read(byte[] buf, int offset, int numBytes)`	Делает попытку прочитать *numBytes* байт в массив *buf*, начиная с элемента *buf[offset]*, возвращает количество успешно прочитанных байтов
`long Seek(long offset, SeekOrigin origin)`	Устанавливает текущую позицию потока равной указанному значению смещения от заданного начала отсчета
`Void WriteByte(byte b)`	Записывает один байт в выходной поток
`Void Write(byte[] buf, int offset, int numBytes)`	Записывает поддиапазон размером *numBytes* байт из массива *buf*, начиная с элемента *buf[offset]*

В общем случае при возникновении ошибки ввода-вывода методы, представленные в табл. 14.1, генерируют исключение типа `IOException`. При попытке выполнить некорректную операцию, например, записать данные в поток, предназначенный только для чтения, генерируется исключение типа `NotSupportedException`.

Обратите внимание на то, что в классе `Stream` определены методы, которые считывают и записывают данные. Однако не все потоки поддерживают все эти операции, поскольку возможен вариант, когда поток открывается только для чтения или только для записи. Кроме того, не все потоки поддерживают функцию установки в заданную позицию с помощью метода `Seek()`. Чтобы определить возможности потока, используйте одно или несколько свойств класса `Stream`. Они представлены в табл. 14.2. В этой таблице также описаны свойства `Length` и `Position`, которые содержат длину и текущую позицию потока, соответственно.

Таблица 14.2. Свойства, определенные в классе `Stream`

Свойство	*Описание*
`bool CanRead`	Свойство равно значению `true`, если из потока можно считывать данные. Это свойство предназначено только для чтения
`bool CanSeek`	Свойство равно значению `true`, если поток поддерживает функцию установки в заданную позицию. Это свойство предназначено только для чтения
`bool CanWrite`	Свойство равно значению `true`, если в поток можно записывать данные. Это свойство предназначено только для чтения
`long Length`	Свойство содержит длину потока. Это свойство предназначено только для чтения
`long Position`	Свойство представляет текущую позицию потока. Это свойство можно как читать, так и устанавливать

Байтовые классы потоков

Из класса `Stream` выведены такие байтовые классы потоков.

Класс потока	*Описание*
`BufferedStream`	Заключает в оболочку байтовый поток и добавляет буферизацию. Буферизация во многих случаях увеличивает производительность
`FileStream`	Байтовый поток, разработанный для файлового ввода-вывода
`MemoryStream`	Байтовый поток, который использует память для хранения данных

Программист может также вывести собственные потоковые классы. Однако для подавляющего большинства приложений достаточно встроенных потоков.

Символьные классы потоков

Чтобы создать символьный поток, поместите байтовый поток в одну из символьных потоковых C#-оболочек. В вершине иерархии символьных потоков находятся абстрактные классы `TextReader` и `TextWriter`. Класс `TextReader` предназначен для обработки операций ввода данных, а класс `TextWriter` — для обработки операций вывода данных. Методы, определенные этими двумя абстрактными классами, доступны для всех их подклассов. Следовательно, они образуют минимальный набор функций ввода-вывода, который будут иметь все символьные потоки.

В табл. 14.3 перечислены методы ввода данных, принадлежащие классу `TextReader`. В случае ошибки эти методы могут генерировать исключение типа `IOException`. (Некоторые методы могут также генерировать и другие типы исключений.) Особого внимания заслуживает метод `ReadLine()`, который считывает целую строку текста, возвращая ее в качестве `string`-значения. Этот метод полезен при считывании входных данных, которые содержат пробелы.

Таблица 14.3. Методы ввода данных, определенные в классе `TextReader`

Метод	*Описание*
`void Close()`	Закрывает источник ввода данных
`int Peek()`	Получает следующий символ из входного потока, но не удаляет его. Возвращает значение -1, если ни один символ не доступен
`int Read()`	Возвращает целочисленное представление следующего доступного символа из вызывающего объекта входного потока. При обнаружении конца файла возвращает значение -1

Окончание табл. 14.3

Метод	Описание
`int Read(char[] buf, int offset, int numChars)`	Делает попытку прочитать `numChars` символов в массив `buf`, начиная с элемента `buf[offset]`, и возвращает количество успешно прочитанных символов
`int ReadBlock(char[] buf, int offset, int numChars)`	Делает попытку прочитать `numChars` символов в массив `buf`, начиная с элемента `buf[offset]`, и возвращает количество успешно прочитанных символов
`string ReadLine()`	Считывает следующую строку текста и возвращает ее как `string`-значение. При попытке прочитать признак конца файла возвращает `null`-значение
`string ReadToEnd()`	Считывает все символы, оставшиеся в потоке, и возвращает их как `string`-значение

В классе `TextWriter` определены версии методов `Write()` и `WriteLine()`, которые могут выводить данные всех встроенных типов. Приведем, например, только некоторые их перегруженные версии.

Метод	Описание
`void Write(int val)`	Записывает значение типа `int`
`void Write(double val)`	Записывает значение типа `double`
`void Write(bool val)`	Записывает значение типа `bool`
`void WriteLine(string val)`	Записывает значение типа `string` с последующим символом новой строки
`void WriteLine(uint val)`	Записывает значение типа `uint` с последующим символом новой строки
`void WriteLine(char val)`	Записывает символ с последующим символом новой строки

Помимо методов `Write()` и `WriteLine()`, в классе `TextWriter` также определены методы `Close()` и `Flush()`:

```
virtual void Close()
virtual void Flush()
```

Метод `Flush()` записывает все данные, оставшиеся в выходном буфере, на физический носитель информации. Метод `Close()` закрывает поток.

Из классов `TextReader` и `TextWriter` выведен ряд символьно-ориентированных потоковых классов, в том числе и те, что перечислены в следующей таблице. Следовательно, эти потоковые классы используют методы и свойства, определенные в классах `TextReader` и `TextWriter`.

Потоковый класс	Описание
`StreamReader`	Предназначен для чтения символов из байтового потока. Этот класс является оболочкой для байтового входного потока
`StreamWriter`	Предназначен для записи символов в байтовый поток. Этот класс является оболочкой для байтового выходного потока
`StringReader`	Предназначен для чтения символов из строки
`StringWriter`	Предназначен для записи символов в строку

Двоичные потоки

Помимо байтовых и символьных потоков, в C# определены два двоичных потоковых класса, которые можно использовать для прямого считывания и записи двоичных данных. Эти классы, которые называются `BinaryReader` и `BinaryWriter`, будут рассмотрены ниже в этой главе в теме двоичного файлового ввода-вывода.

Теперь, когда вы получили общее представление о C#-системе ввода-вывода, можно переходить к детальному изучению этой темы, которую, пожалуй, стоит начать с консольного ввода-вывода.

Консольный ввод-вывод данных

Консольный ввод-вывод данных реализуется посредством стандартных потоков `Console.In`, `Console.Out` и `Console.Error`. Консольные классы ввода-вывода использовались, начиная с главы 2, поэтому вы уже с ними знакомы. Как вы убедитесь ниже, они обладают и другими возможностями.

Прежде всего, важно отметить, что большинство реальных C#-приложений являются не текстовыми, или консольными, а графическими программами или компонентами, которые опираются на оконный интерфейс, предназначенный для взаимодействия с пользователем. Таким образом, часть C#-системы ввода-вывода, которая связана с консольным вводом-выводом данных, не относится к широко используемым средствам. Несмотря на то что текстовые программы — прекрасные учебные примеры коротких утилит и некоторых типов компонентов, они не годятся для большинства реальных приложений.

Считывание данных из консольного входного потока

Поток `Console.In` — экземпляр класса `TextReader`, поэтому для доступа к нему можно использовать методы и свойства, определенные в классе `TextReader`. Однако обычно используют методы, определенные в классе `Console`, которые автоматически считывают значение свойства `Console.In`. В классе `Console` определено два метода ввода информации: `Read()` и `ReadLine()`.

Метод `Read()` используется для считывания одного символа.

```
static int Read()
```

Метод `Read()` возвращает следующий символ, считанный с консоли. Он ожидает, пока пользователь не нажмет какую-нибудь клавишу, а затем возвращает результат. Считанный символ возвращается как значение типа `int`, которое должно быть приведено к типу `char`. При возникновении ошибки метод `Read()` возвращает −1, а в случае неудачного исхода операции генерирует исключение типа `IOException`. По умолчанию консольный ввод данных буферизован (с ориентацией на строки), поэтому, прежде чем введенный с клавиатуры символ будет послан программе, необходимо нажать клавишу <Enter>.

Рассмотрим пример программы, которая считывает символ с клавиатуры с помощью метода `Read()`.

```
// Считывание символа с клавиатуры.

using System;

class KbIn {
  public static void Main() {
    char ch;
```

```
      Console.Write(
          "Нажмите любую клавишу, а затем -- <ENTER>: ");

      ch = (char) Console.Read(); // Считывание
                                  // char-значения.

      Console.WriteLine("Вы нажали клавишу: " + ch);
    }
}
```

Вот как могут выглядеть результаты выполнения этой программы:

```
Нажмите любую клавишу, а затем -- <ENTER>: ю
Вы нажали клавишу: ю
```

Тот факт, что метод `Read()` буферизирует строки, является порой источником досадных недоразумений. При нажатии клавиши <Enter> во входной поток вводится последовательность, состоящая из символов возврата каретки и перевода строки. Более того, эти символы остаются во входном потоке до тех пор, пока вы их не прочитаете. Таким образом, в некоторых приложениях перед выполнением следующей операции ввода их нужно удалить (просто считыванием из потока).

Чтобы прочитать строку символов, используйте метод `ReadLine()`.

```
static string ReadLine()
```

Метод `ReadLine()` считывает символы до тех пор, пока не будет нажата клавиша <Enter>, и возвращает объект типа `string`. При неудачном завершении метод генерирует исключение типа `IOException`.

Рассмотрим программу, которая демонстрирует считывание строки из потока `Console.In` с помощью метода `ReadLine()`.

```
// Ввод данных с консоли с помощью метода ReadLine().

using System;

class ReadString {
  public static void Main() {
    string str;

    Console.WriteLine("Введите несколько символов.");
    str = Console.ReadLine();
    Console.WriteLine("Вы ввели: " + str);
  }
}
```

Вот такие результаты можно получить при выполнении этой программы.

```
Введите несколько символов.
Это всего лишь тест.
Вы ввели: Это всего лишь тест.
```

Несмотря на то что методы класса `Console` являются самым простым способом считывания данных из потока `Console.In`, можно с таким же успехом вызывать методы базового класса `TextReader`. Например, перепишем предыдущую программу с использованием методов, определенных в классе `TextReader`.

```
// Считывание строки с клавиатуры благодаря
// непосредственному использованию свойства Console.In.

using System;

class ReadChars2 {
  public static void Main() {
    string str;
```

```
      Console.WriteLine("Введите несколько символов.");

    str = Console.In.ReadLine();

    Console.WriteLine("Вы ввели: " + str);
  }
}
```

Обратите внимание на то, что метод `ReadLine()` теперь напрямую вызывается для потока `Console.In`. Здесь важно то, что при необходимости доступа к методам, определенным в классе `TextReader`, который является базовым для объекта `Console.In`, их можно вызывать так, как показано в этом примере.

Запись данных в консольный входной поток

Потоки `Console.Out` и `Console.Error` — объекты типа `TextWriter`. Проще всего консольный вывод данных выполнить с помощью методов `Write()` и `WriteLine()`, с которыми вы уже знакомы. Для каждого из встроенных типов предусмотрены отдельные версии этих методов. В классе `Console` определены собственные версии методов `Write()` и `WriteLine()`, поэтому их можно вызывать непосредственно для класса `Console`, как мы и делали это в примерах этой книги. Однако при желании можно вызывать эти (и другие) методы класса `TextWriter`, который является базовым для объектов `Console.Out` и `Console.Error`.

Рассмотрим программу, которая демонстрирует запись данных в потоки `Console.Out` и `Console.Error`.

```
// запись данных в потоки Console.Out и Console.Error.

using System;

class ErrOut {
  public static void Main() {
    int a=10, b=0;
    int result;

    Console.Out.WriteLine(
          "Деление на нуль сгенерирует исключение.");
    try {
      result = a / b; // Генерируем исключение.
    } catch(DivideByZeroException exc) {
      Console.Error.WriteLine(exc.Message);
    }
  }
}
```

При выполнении этой программы получим следующие результаты:

```
Деление на нуль сгенерирует исключение.
Attempted to divide by zero.
```

Иногда неопытные программисты путаются, не зная точно, когда следует использовать поток `Console.Error`. Если и `Console.Out`, и `Console.Error` по умолчанию выводят данные на консоль, то почему существует два различных потока? Ответ прост: стандартные потоки можно перенаправить на другие устройства. Например, поток `Console.Error` можно перенаправить для записи данных в дисковый файл, а не на экран. Следовательно, и поток ошибок можно перенаправить, например, в системный журнал (log file), не затрагивая при этом консольный вывод. И наоборот, если консольный вывод данных перенаправить в файл, а вывод ошибок — нет, то сообще-

ния об ошибках будут появляться на консольном устройстве, и их легко увидеть. Но подробнее о перенаправлении потоков мы поговорим ниже, когда рассмотрим файловый ввод-вывод.

Класс FileStream и файловый ввод-вывод на побайтовой основе

В C# предусмотрены классы, которые позволяют считывать содержимое файлов и записывать в них информацию. Конечно же, дисковые файлы — самый распространенный тип файлов. На уровне операционной системы все файлы обрабатываются на побайтовой основе. Нетрудно предположить, что в C# определены методы, предназначенные для считывания байтов из файла и записи байтов в файл. Таким образом, файловые операции чтения и записи с использованием байтовых потоков очень востребованы. C# также позволяет поместить файловый поток с ориентацией на побайтовую обработку в символьный объект. Файловые операции, .ориентированные на символы, используются в случае текстовых файлов. Символьные потоки рассматриваются ниже в этой главе. А пока изучим ввод-вывод данных на побайтовой основе.

Чтобы создать байтовый поток с привязкой к файлу, используйте класс `FileStream`. Класс `FileStream` — производный от `Stream` и потому обладает функциональными возможностями базового класса. Помните, что потоковые классы, включая `FileStream`, определены в пространстве имен `System.IO`. Следовательно, при их использовании в начало программы вы должны включить следующую инструкцию:

```
using System.IO;
```

Как открыть и закрыть файл

Чтобы создать байтовый поток, связанный с файлом, создайте объект класса `FileStream`. В классе `FileStream` определено несколько конструкторов. Чаще всего из них используется следующий:

`FileStream(string` *`filename`*`, FileMode` *`mode`*`)`

Здесь элемент *`filename`* означает имя файла, который необходимо открыть, причем оно может включать полный путь к файлу. Элемент *`mode`* означает, как именно должен быть открыт этот файл, или *режим открытия*. Элемент *`mode`* может принимать одно из значений, определенных перечислением `FileMode` (они описаны в табл. 14.4). Этот конструктор открывает файл для доступа с разрешением чтения и записи.

Таблица 14.4. Значения перечисления `FileMode`

Значение	*Описание*
`FileMode.Append`	Добавляет выходные данные в конец файла
`FileMode.Create`	Создает новый выходной файл. Существующий файл с таким же именем будет разрушен
`FileMode.CreateNew`	Создает новый выходной файл. Файл с таким же именем не должен существовать
`FileMode.Open`	Открывает существующий файл
`FileMode.OpenOrCreate`	Открывает файл, если он существует. В противном случае создает новый
`FileMode.Truncate`	Открывает существующий файл, но усекает его длину до нуля

Если попытка открыть файл оказалось неуспешной, генерируется исключение. Если файл невозможно открыть по причине его отсутствия, генерируется исключение типа `FileNotFoundException`. Если файл невозможно открыть из-за ошибки ввода-вывода, генерируется исключение типа `IOException`. Возможны также исключения следующих типов: `ArgumentNullException` (если имя файла представляет собой `null`-значение), `ArgumentException` (если некорректен параметр *mode*), `SecurityException` (если пользователь не обладает правами доступа) и `DirectoryNotFoundException` (если некорректно задан каталог).

В следующем фрагменте программы показан один из способов открыть файл `test.dat` для ввода данных.

```
FileStream fin;

try {
  fin = new FileStream("test.dat", FileMode.Open);
}
catch(FileNotFoundException exc) {
  Console.WriteLine(exc.Message);
  return;
}
catch {
  Console.WriteLine("Не удается открыть файл.");
  return;
}
```

Здесь первая `catch`-инструкция перехватывает ошибку, связанную с отсутствием файла. Вторая, предназначенная для "всеобщего перехвата", обрабатывает другие ошибки, которые возможны при работе с файлами. Конечно, можно было бы отслеживать возникновение каждой ошибки в отдельности, сообщая о возникшей проблеме. Но ради простоты во всех примерах этой книги организован перехват исключений только типа `FileNotFoundException` или `IOException`, но в реальных приложениях (в зависимости от обстоятельств) вам, скорее всего, придется обрабатывать другие возможные исключения.

Как уже упоминалось, приведенный выше конструктор `FileStream` открывает файл с доступом для чтения и записи. Если необходимо ограничить доступ только чтением или только записью, используйте следующий конструктор:

```
FileStream(string filename, FileMode mode,
           FileAccess how)
```

Как и прежде, элемент *filename* означает имя открываемого файла, а *mode* — способ его открытия. Значение, передаваемое с помощью параметра *how*, определяет способ доступа к файлу. Этот параметр может принимать одно из значений, определенных перечислением `FileAccess`, а именно:

`FileAccess.Read` `FileAccess.Write` `FileAccess.ReadWrite`

Например, при выполнении следующей инструкции файл `test.dat` будет открыт только для чтения:

```
FileStream fin = new FileStream("test.dat", FileMode.Open,
                                FileAccess.Read);
```

По завершении работы с файлом его необходимо закрыть. Для этого достаточно вызвать метод `Close()`. Его общая форма вызова имеет такой вид:

```
void Close()
```

При закрытии файла освобождаются системные ресурсы, ранее выделенные для этого файла, что дает возможность использовать их для других файлов. Метод `Close()` может генерировать исключение типа `IOException`.

Считывание байтов из объекта класса FileStream

В классе FileStream определены два метода, которые считывают байты из файла: ReadByte() и Read(). Чтобы прочитать из файла один байт, используйте метод ReadByte(), общая форма вызова которого имеет следующий вид:

```
int ReadByte()
```

При каждом вызове этого метода из файла считывается один байт, и метод возвращает его как целочисленное значение. При обнаружении конца файла метод возвращает −1. Метод может генерировать исключения типов NotSupportedExeption (поток не открыт для ввода) и ObjectDisposedException (поток закрыт).

Чтобы считать блок байтов, используйте метод Read(), общая форма вызова которого такова:

```
int Read(byte[] buf, int offset, int numBytes)
```

Метод Read() пытается считать *numBytes* байтов в массив *buf*, начиная с элемента *buf*[*offset*]. Он возвращает количество успешно считанных байтов. При возникновении ошибки ввода-вывода генерируется исключение типа IOException. Помимо прочих, возможно также генерирование исключения типа NotSupportedException, если используемый поток не поддерживает операцию считывания данных.

В следующей программе метод ReadByte() используется для ввода содержимого текстового файла и его отображения. Имя файла задается в качестве аргумента командной строки. Обратите внимание на try/catch-блоки, которые обрабатывают две ошибки, возможные при первоначальном выполнении этой программы: "указанный файл не найден" или "пользователь забыл указать имя файла". Такой подход обычно полезен при использовании аргументов командной строки.

```
/* Отображение содержимого текстового файла.

   Чтобы использовать эту программу, укажите имя
   файла, содержимое которого вы хотите увидеть.
   Например, чтобы увидеть содержимое файла TEST.CS,
   используйте следующую командную строку:

   ShowFile TEST.CS
*/

using System;
using System.IO;

class ShowFile {
  public static void Main(string[] args) {
    int i;
    FileStream fin;

    try {
      fin = new FileStream(args[0], FileMode.Open);
    } catch(FileNotFoundException exc) {
      Console.WriteLine(exc.Message);
      return;
    } catch(IndexOutOfRangeException exc) {
      Console.WriteLine(exc.Message +
                        "\nПрименение: ShowFile Файл");
      return;
    }
```

```
    // Считываем байты до тех пор, пока не встретится EOF.
    do {
      try {
        i = fin.ReadByte();
      } catch(Exception exc) {
        Console.WriteLine(exc.Message);
        return;
      }
      if(i != -1) Console.Write((char) i);
    } while(i != -1);

    fin.Close();
  }
}
```

Запись данных в файл

Чтобы записать байт в файл, используйте метод `WriteByte()`. Простейшая его форма имеет следующий вид:

```
void WriteByte(byte val)
```

Этот метод записывает в файл байт, заданный параметром *val*. При возникновении во время записи ошибки генерируется исключение типа `IOException`. Если соответствующий поток не открыт для вывода данных, генерируется исключение типа `NotSupportedException`.

С помощью метода `Write()` можно записать в файл массив байтов. Это делается следующим образом:

```
void Write(byte[] buf, int offset, int numBytes)
```

Метод `Write()` записывает в файл *numBytes* байтов из массива *buf*, начиная с элемента *buf*[*offset*]. При возникновении во время записи ошибки генерируется исключение типа `IOException`. Если соответствующий поток не открыт для вывода данных, генерируется исключение типа `NotSupportedException`. Возможны и другие исключения.

Вероятно, вы уже знаете, что при выполнении операции вывода в файл выводимые данные зачастую не записываются немедленно на реальное физическое устройство, а буферизируются операционной системой до тех пор, пока не накопится порция данных достаточного размера, чтобы ее можно было всю сразу переписать на диск. Такой способ выполнения записи данных на диск повышает эффективность системы. Например, дисковые файлы организованы по секторам, которые могут иметь размер от 128 байт. Данные, предназначенные для вывода, обычно буферизируются до тех пор, пока не накопится такой их объем, который позволяет заполнить сразу весь сектор. Но если вы хотите записать данные на физическое устройство вне зависимости от того, полон буфер или нет, вызовите следующий метод `Flush()`:

```
void Flush()
```

В случае неудачного исхода операции записи генерируется исключение типа `IOException`.

Завершив работу с выходным файлом, вы должны его закрыть с помощью метода `Close()`. Это гарантирует, что любые данные, оставшиеся в дисковом буфере, будут переписаны на диск. Поэтому перед закрытием файла нет необходимости специально вызывать метод `Flush()`.

Рассмотрим простой пример записи данных в файл.

```
// Запись данных в файл.

using System;
using System.IO;

class WriteToFile {
  public static void Main(string[] args) {
    FileStream fout;

    // Открываем выходной файл.
    try {
      fout = new FileStream("test.txt", FileMode.Create);
    } catch(IOException exc) {
      Console.WriteLine(
                exc.Message +
                "\nОшибка при открытии выходного файла.");
      return;
    }

    // Записываем в файл алфавит.
    try {
      for(char c = 'A'; c <= 'Я'; c++)
        fout.WriteByte((byte) c);
    } catch(IOException exc) {
      Console.WriteLine(exc.Message +
                        "Ошибка при записи в файл.");
    }

    fout.Close();
  }
}
```

Эта программа сначала открывает для вывода файл с именем `test.txt`. Затем в этот файл записывается алфавит английского языка, после чего файл закрывается. Обратите внимание на то, как обрабатываются возможные ошибки с помощью блоков `try/catch`. После выполнения этой программы файл `test.txt` будет иметь такое содержимое:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Использование класса `FileStream` для копирования файла

Одно из достоинств байтового ввода-вывода с использованием класса `FileStream` заключается в том, что этот класс можно использовать для всех типов файлов, а не только текстовых. Например, следующая программа копирует файл любого типа, включая выполняемые файлы. Имена исходного и приемного файлов указываются в командной строке.

```
/* Копирование файла.

   Для использования этой программы укажите имя
   исходного и приемного файлов.
   Например, чтобы скопировать файл FIRST.DAT
   в файл SECOND.DAT, используйте следующую
   командную строку:

   CopyFile FIRST.DAT SECOND.DAT
*/
```

```
using System;
using System.IO;

class CopyFile {
  public static void Main(string[] args) {
    int i;
    FileStream fin;
    FileStream fout;

    try {
      // Открываем входной файл.
      try {
        fin = new FileStream(args[0], FileMode.Open);
      } catch(FileNotFoundException exc) {
        Console.WriteLine(exc.Message +
                          "\nВходной файл не найден.");
        return;
      }

      // Открываем выходной файл.
      try {
        fout = new FileStream(args[1], FileMode.Create);
      } catch(IOException exc) {
        Console.WriteLine(
                  exc.Message +
                  "\nОШибка при открытии выходного файла.");
        return;
      }
    } catch(IndexOutOfRangeException exc) {
      Console.WriteLine(exc.Message +
                        "\nПрименение: CopyFile ИЗ КУДА");
      return;
    }

    // Копируем файл.
    try {
      do {
        i = fin.ReadByte();
        if(i != -1) fout.WriteByte((byte)i);
      } while(i != -1);
    } catch(IOException exc) {
      Console.WriteLine(exc.Message +
                        "Ошибка при чтении файла. ");
    }

    fin.Close();
    fout.Close();
  }
}
```

Файловый ввод-вывод с ориентацией на символы

Несмотря на то что байтовая обработка файлов получила широкое распространение, C# также поддерживает символьные потоки. Символьные потоки работают непосредственно с Unicode-символами (это их достоинство). Поэтому, если вы хотите сохранить Unicode-текст, лучше всего выбрать именно символьные потоки. В общем случае, чтобы выполнять файловые операции на символьной основе, поместите объект класса `FileStream` внутрь объекта класса `StreamReader` или класса `StreamWriter`. Эти классы автоматически преобразуют байтовый поток в символьный и наоборот.

Помните, что на уровне операционной системы файл представляет собой набор байтов. Использование классов `StreamReader` или `StreamWriter` не влияет на этот факт.

Класс `StreamWriter` — производный от класса `TextWriter`, а `StreamReader` — производный от `TextReader`. Следовательно, классы `StreamWriter` и `StreamReader` имеют доступ к методам и свойствам, определенным их базовыми классами.

Использование класса `StreamWriter`

Чтобы создать выходной поток для работы с символами, поместите объект класса `Stream` (например, `FileStream`) в объект класса `StreamWriter`. В классе `StreamWriter` определено несколько конструкторов. Самый популярный из них выглядит следующим образом:

```
StreamWriter(Stream stream)
```

Здесь элемент *stream* означает имя открытого потока. Этот конструктор генерирует исключение типа `ArgumentException`, если поток *stream* не открыт для вывода, и исключение типа `ArgumentNullException`, если он (поток) имеет `null`-значение. Созданный объект класса `StreamWriter` автоматически выполняет преобразование символов в байты.

Рассмотрим простую утилиту "клавиатура-диск", которая считывает строки текста, вводимые с клавиатуры, и записывает их в файл `test.txt`. Текст считывается до тех пор, пока пользователь не введет слово "стоп". Здесь используется объект класса `FileStream`, помещенный в оболочку класса `StreamWriter` для вывода данных в файл.

```
/* Простая утилита "клавиатура-диск", которая
   демонстрирует использование класса StreamWriter. */

using System;
using System.IO;

class KtoD {
  public static void Main() {
    string str;
    FileStream fout;

    try {
      fout = new FileStream("test.txt", FileMode.Create);
    }
    catch(IOException exc) {
```

```
      Console.WriteLine(exc.Message +
                        "Не удается открыть файл.");
      return ;
    }
    StreamWriter fstr_out = new StreamWriter(fout);

    Console.WriteLine(
                "Введите текст ('стоп' для завершения).");
    do {
      Console.Write(": ");
      str = Console.ReadLine();

      if(str != "стоп") {
        str = str + "\r\n"; // Добавляем символ
                            // новой строки.
        try {
          fstr_out.Write(str);
        } catch(IOException exc) {
          Console.WriteLine(exc.Message +
                            "Ошибка при работе с файлом.");
          return ;
        }
      }
    } while(str != "стоп");

    fstr_out.Close();
  }
}
```

Иногда удобнее открывать файл с помощью класса `StreamWriter`. Для этого используйте один из следующих конструкторов:

```
StreamWriter(string filename)
StreamWriter(string filename, bool appendFlag)
```

Здесь элемент *filename* означает имя открываемого файла, причем имя может включать полный путь к файлу. Во второй форме используется параметр *appendFlag* типа `bool`: если *appendFlag* равен значению `true`, выводимые данные добавляются в конец существующего файла. В противном случае заданный файл перезаписывается. В обоих случаях, если файл не существует, он создается, а при возникновении ошибки ввода-вывода генерируется исключение типа `IOException` (также возможны и другие исключения).

Перед вами новая версия предыдущей утилиты "клавиатура-диск", в которой для открытия выходного файла используется класс `StreamWriter`.

```
// Открытие файла с использованием класса StreamWriter.

using System;
using System.IO;

class KtoD {
  public static void Main() {
    string str;
    StreamWriter fstr_out;

    // Открываем файл напрямую, используя
    // класс StreamWriter.
    try {
      fstr_out = new StreamWriter("test.txt");
```

```
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message +
                        "Не удается открыть файл.");
      return ;
    }

    Console.WriteLine(
                "Введите текст ('стоп' для завершения).");
    do {
      Console.Write(": ");
      str = Console.ReadLine();

      if(str != "стоп") {
        str = str + "\r\n"; // Добавляем символ
                            // новой строки.
        try {
          fstr_out.Write(str);
        } catch(IOException exc) {
          Console.WriteLine(
                          exc.Message +
                          "Ошибка при работе с файлом. ");
          return ;
        }
      }
    } while(str != "стоп");

    fstr_out.Close();
  }
}
```

Использование класса `StreamReader`

Чтобы создать входной поток с ориентацией на обработку символов, поместите байтовый поток в класс-оболочку `StreamReader`. В классе `StreamReader` определено несколько конструкторов. Чаще всего используется следующий конструктор:

`StreamReader(Stream` *`stream`*`)`

Здесь элемент *`stream`* означает имя открытого потока. Этот конструктор генерирует исключение типа `ArgumentNullException`, если поток *`stream`* имеет null-значение, и исключение типа `ArgumentException`, если поток *`stream`* не открыт для ввода. После создания объект класса `StreamReader` автоматически преобразует байты в символы.

Следующая программа создает простую утилиту "клавиатура-диск", которая считывает текстовый файл `test.txt` и отображает его содержимое на экране. Таким образом, эта программа представляет собой дополнение к утилите, представленной в предыдущем разделе.

```
/* Простая утилита "клавиатура-диск", которая
   демонстрирует использование класса FileReader. */

using System;
using System.IO;

class DtoS {
  public static void Main() {
    FileStream fin;
```

```
    string s;

    try {
      fin = new FileStream("test.txt", FileMode.Open);
    }
    catch(FileNotFoundException exc) {
      Console.WriteLine(exc.Message +
                        "Не удается открыть файл.");
      return ;
    }

    StreamReader fstr_in = new StreamReader(fin);

    // Считываем файл построчно.
    while((s = fstr_in.ReadLine()) != null) {
      Console.WriteLine(s);
    }

    fstr_in.Close();
  }
}
```

Обратите внимание на то, как определяется конец файла. Если ссылка, возвращаемая методом `ReadLine()`, равна значению `null`, значит, конец файла достигнут.

Как и в случае класса `StreamWriter`, иногда проще открыть файл, напрямую используя класс `StreamReader`. Для этого обратитесь к этому конструктору:

`StreamReader(string` *`filename`*`)`

Здесь элемент *`filename`* означает имя открываемого файла, которое может включать полный путь к файлу. Указанный файл должен существовать. В противном случае генерируется исключение типа `FileNotFoundException`. Если параметр *`filename`* равен значению `null`, генерируется исключение типа `ArgumentNullException`, а если он представляет собой пустую строку, — исключение типа `ArgumentException`.

Перенаправление стандартных потоков

Как упоминалось выше, такие стандартные потоки, как `Console.In`, можно перенаправлять. Безусловно, чаще всего они перенаправляются в какой-нибудь файл. При перенаправлении стандартного потока входные и/или выходные данные автоматически направляются в новый поток. При этом устройства, действующие по умолчанию, игнорируются. Благодаря перенаправлению стандартных потоков программа может считывать команды из дискового файла, создавать системные журналы или даже считывать входные данные с сетевых устройств.

Перенаправить стандартный поток можно двумя способами. Во-первых, при выполнении программы из командной строки можно использовать операторы "<" и ">", чтобы перенаправить потоки `Console.In` и/или `Console.Out`, соответственно. Рассмотрим, например, следующую программу:

```
using System;

class Test {
  public static void Main() {
    Console.WriteLine("Это тест.");
  }
}
```

При выполнении ее с помощью командной строки

```
Test > log
```

текстовая строка "Это тест." будет записана в файл `log`. Входной поток можно перенаправить аналогичным способом. При перенаправлении входного потока важно позаботиться о том, чтобы задаваемый источник входных данных содержал информацию, удовлетворяющую требованиям программы. В противном случае программа зависнет.

Операторы перенаправления "<" и ">" являются частью не языка C#, а операционной системы. Таким образом, если среда поддерживает функцию перенаправления потоков ввода-вывода (как это реализовано в Windows), вы сможете перенаправить стандартные входные и выходные потоки, не внося изменений в программы. Однако существует и второй способ, который позволяет перенаправлять стандартные потоки именно программно. Для этого понадобятся следующие методы `SetIn()`, `SetOut()` и `SetError()`, которые являются членами класса `Console`:

```
static void SetIn(TextReader input)
static void SetOut(TextWriter output)
static void SetError(TextWriter output)
```

Таким образом, чтобы перенаправить входной поток, вызовите метод `SetIn()`, указав в качестве параметра желаемый поток. Вы можете использовать любой входной поток, если он является производным от класса `TextReader`. Чтобы перенаправить выходной поток в файл, задайте `FileStream`-объект, помещенный в оболочку `StreamWriter`-объекта. Пример перенаправления потоков проиллюстрирован следующей программой:

```
// Перенаправление потока Console.Out.

using System;
using System.IO;

class Redirect {
  public static void Main() {
    StreamWriter log_out;

    try {
      log_out = new StreamWriter("logfile.txt");
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message +
                        "Не удается открыть файл.");
      return ;
    }

    // Направляем стандартный выходной поток в
    // системный журнал.
    Console.SetOut(log_out);
    Console.WriteLine("Это начало системного журнала.");

    for(int i=0; i<10; i++) Console.WriteLine(i);

    Console.WriteLine("Это конец системного журнала.");
    log_out.Close();
  }
}
```

При выполнении этой программы на экране не появится ни одного символа, но файл `logfile.txt` будет иметь такое содержимое:

```
Это начало системного журнала.
0
1
2
3
4
5
6
7
8
9
Это конец системного журнала.
```

При желании вы можете поэкспериментировать, перенаправляя другие встроенные потоки ввода-вывода.

Считывание и запись двоичных данных

До сих пор мы считывали и записывали байты или символы, но эти операции ввода-вывода можно выполнять и с другими типами данных. Например, вы могли бы создать файл, содержащий `int`-, `double`- или `short`-значения. Для считывания и записи двоичных значений встроенных C#-типов используйте классы `BinaryReader` и `BinaryWriter`. Важно понимать, что эти данные считываются и записываются с использованием внутреннего двоичного формата, а не в текстовой форме, понятной человеку.

Класс `BinaryWriter`

Класс `BinaryWriter` представляет собой оболочку для байтового потока, которая управляет записью двоичных данных. Его наиболее употребительный конструктор имеет следующий вид:

```
BinaryWriter(Stream outputStream)
```

Здесь элемент *outputStream* означает поток, в который будут записываться данные. Чтобы записать выходные данные в файл, можно использовать для этого параметра объект, созданный классом `FileStream`. Если поток *outputStream* имеет `null`-значение, генерируется исключение типа `ArgumentNullException`, а если поток *outputStream* не открыт для записи, — исключение типа `ArgumentException`.

В классе `BinaryWriter` определены методы, способные записывать значения всех встроенных C#-типов (некоторые из них перечислены в табл. 14.5). Обратите внимание: значение типа `string` записывается с использованием внутреннего формата, который включает спецификатор длины. В классе `BinaryWriter` также определены стандартные методы `Close()` и `Flush()`, работа которых описана выше.

Таблица 14.5. Методы вывода информации, определенные в классе `BinaryWriter`

Метод	*Описание*
`void Write(sbyte val)`	Записывает `byte`-значение (со знаком)
`void Write(byte val)`	Записывает `byte`-значение (без знака)

Метод	Описание
`void Write(byte[] buf)`	Записывает массив `byte`-значений
`void Write(short val)`	Записывает целочисленное значение типа `short` (короткое целое)
`void Write(ushort val)`	Записывает целочисленное `ushort` -значение (короткое целое без знака)
`void Write(int val)`	Записывает целочисленное значение типа `int`
`void Write(uint val)`	Записывает целочисленное `uint`-значение (целое без знака)
`void Write(long val)`	Записывает целочисленное значение типа `long` (длинное целое)
`void Write(ulong val)`	Записывает целочисленное `ulong`-значение (длинное целое без знака)
`void Write(float val)`	Записывает `float`-значение
`void Write(double val)`	Записывает `double`-значение
`void Write(char val)`	Записывает символ
`void Write(char[] buf)`	Записывает массив символов
`void Write(string val)`	Записывает `string`-значение с использованием его внутреннего представления, которое включает спецификатор длины

Класс `BinaryReader`

Класс `BinaryReader` представляет собой оболочку для байтового потока, которая управляет чтением двоичных данных. Его наиболее употребительный конструктор имеет такой вид:

```
BinaryReader(Stream inputStream)
```

Здесь элемент *inputStream* означает поток, из которого считываются данные. Чтобы выполнить чтение из файла, можно использовать для этого параметра объект, созданный классом `FileStream`. Если поток *inputStream* имеет `null`-значение, генерируется исключение типа `ArgumentNullException`, а если поток *inputStream* не открыт для чтения,— исключение типа `ArgumentException`.

В классе `BinaryReader` предусмотрены методы для считывания всех простых C#-типов. Наиболее употребимые из них показаны в табл. 14.6. Обратите внимание на то, что метод `ReadString()` считывает строку, которая хранится с использованием внутреннего формата, включающего спецификатор длины. При обнаружении конца потока все эти методы генерируют исключение типа `EndOfStreamException`, а при возникновении ошибки — исключение типа `IOException`. В классе `BinaryReader` также определены следующие версии метода `Read()`:

Метод	Описание
`int Read()`	Возвращает целочисленное представление следующего доступного символа из вызывающего входного потока. При обнаружении конца файла возвращает значение -1
`int Read(byte[] buf, int offset, int num)`	Делает попытку прочитать *num* байтов в массив *buf*, начиная с элемента *buf*`[`*offset*`]`, и возвращает количество успешно считанных байтов
`int Read(char[] buf, int offset, int num)`	Делает попытку прочитать *num* символов в массив *buf*, начиная с элемента *buf*`[`*offset*`]`, и возвращает количество успешно считанных символов

В случае неудачного исхода операции чтения эти методы генерируют исключение типа `IOException`.

В классе `BinaryReader` также определен стандартный метод `Close()`.

Таблица 14.6. Методы ввода данных, определенные в классе `BinaryReader`

Метод	*Описание*
`bool ReadBoolean()`	Считывает `bool`-значение
`byte ReadByte()`	Считывает `byte`-значение
`sbyte ReadSByte()`	Считывает `sbyte`-значение
`byte[] ReadBytes(int num)`	Считывает *num* байтов и возвращает их в виде массива
`char ReadChar()`	Считывает `char`-значение
`char[] ReadChars(int num)`	Считывает *num* символов и возвращает их в виде массива
`double ReadDouble()`	Считывает `double`-значение
`float ReadSingle()`	Считывает `float`-значение
`short ReadInt16()`	Считывает `short`-значение
`int ReadInt32()`	Считывает `int`-значение
`long ReadInt64()`	Считывает `long`-значение
`ushort ReadUInt16()`	Считывает `ushort`-значение
`uint ReadUInt32()`	Считывает `uint`-значение
`ulong ReadUInt64()`	Считывает `ulong`-значение
`string ReadString()`	Считывает `string`-значение, представленное во внутреннем двоичном формате, который включает спецификатор длины. Этот метод следует использовать для считывания строки, которая была записана с помощью объекта класса `BinaryWriter`

Демонстрация использования двоичного ввода-вывода

Рассмотрим программу, которая иллюстрирует использование классов `BinaryReader` и `BinaryWriter`. Она записывает в файл данные различных типов, а затем считывает их.

```
// Запись в файл двоичных данных с последующим
// их считыванием.

using System;
using System.IO;

class RWData {
  public static void Main() {
    BinaryWriter dataOut;
    BinaryReader dataIn;

    int i = 10;
    double d = 1023.56;
    bool b = true;

    try {
      dataOut = new
        BinaryWriter(new FileStream("testdata",
                                    FileMode.Create));
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message +
                        "\nНе удается открыть файл.");
```

```
      return;
    }

    try {
      Console.WriteLine("Запись " + i);
      dataOut.Write(i);

      Console.WriteLine("Запись " + d);
      dataOut.Write(d);

      Console.WriteLine("Запись " + b);
      dataOut.Write(b);

      Console.WriteLine("Запись " + 12.2 * 7.4);
      dataOut.Write(12.2 * 7.4);

    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message +
                        "\nОшибка при записи.");
    }

    dataOut.Close();

    Console.WriteLine();

    // Теперь попробуем прочитать эти данные.
    try {
      dataIn = new
          BinaryReader(new FileStream("testdata",
                                      FileMode.Open));
    }
    catch(FileNotFoundException exc) {
      Console.WriteLine(exc.Message +
                        "\nНе удается открыть файл.");
      return;
    }

    try {
      i = dataIn.ReadInt32();
      Console.WriteLine("Считывание " + i);

      d = dataIn.ReadDouble();
      Console.WriteLine("Считывание " + d);

      b = dataIn.ReadBoolean();
      Console.WriteLine("Считывание " + b);

      d = dataIn.ReadDouble();
      Console.WriteLine("Считывание " + d);
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message +
                        "Ошибка при считывании.");
    }

    dataIn.Close();
  }
}
```

При выполнении этой программы были получены следующие результаты:

```
Запись 10
Запись 1023.56
Запись True
Запись 90.28

Считывание 10
Считывание 1023.56
Считывание True
Считывание 90.28
```

Если вы попробуете просмотреть содержимое файла testdata, созданного этой программой, то увидите, что в нем содержатся двоичные данные, а не понятный для человека текст.

А вот более практичный пример, который демонстрирует возможности C#-средств двоичного ввода-вывода. Следующая программа реализует очень простую программу инвентаризации. Для каждого элемента описи программа хранит соответствующее наименование, имеющееся в наличии количество и стоимость. Программа предлагает пользователю ввести наименование элемента описи, а затем выполняет поиск в базе данных. Если элемент найден, на экране отображается соответствующая информация.

```
/* Использование классов BinaryReader и BinaryWriter
   для реализации простой программы инвентаризации. */

using System;
using System.IO;

class Inventory {
  public static void Main() {
    BinaryWriter dataOut;
    BinaryReader dataIn;

    string item; // Наименование элемента.
    int onhand;  // Количество, имеющееся в наличии.
    double cost; // Цена.

    try {
      dataOut = new
        BinaryWriter(new FileStream("inventory.dat",
                                    FileMode.Create));
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message +
                        "\nНе удается открыть файл.");
      return;
    }

    // Записываем некоторые инвентаризационные данные
    // в файл.
    try {
      dataOut.Write("Молотки");
      dataOut.Write(10);
      dataOut.Write(3.95);

      dataOut.Write("Отвертки");
      dataOut.Write(18);
      dataOut.Write(1.50);
```

```
    dataOut.Write("Плоскогубцы");
    dataOut.Write(5);
    dataOut.Write(4.95);

    dataOut.Write("Пилы");
    dataOut.Write(8);
    dataOut.Write(8.95);
  }
  catch(IOException exc) {
    Console.WriteLine(exc.Message +
                      "\nОшибка при записи.");
  }

  dataOut.Close();

  Console.WriteLine();

  // Теперь откроем файл инвентаризации
  // для чтения информации.
  try {
    dataIn = new
        BinaryReader(new FileStream("inventory.dat",
                                    FileMode.Open));
  }
  catch(FileNotFoundException exc) {
    Console.WriteLine(exc.Message +
                      "\nНе удается открыть файл.");
    return;
  }

  // Поиск элемента, введенного пользователем.
  Console.Write("Введите наименование для поиска: ");
  string what = Console.ReadLine();
  Console.WriteLine();

  try {
    for(;;) {
      // Считываем запись из базы данных.
      item = dataIn.ReadString();
      onhand = dataIn.ReadInt32();
      cost = dataIn.ReadDouble();

      /* Если элемент в базе данных совпадает с элементом
         из запроса, отображаем найденную информацию. */
      if(item.CompareTo(what) == 0) {
        Console.WriteLine(item + ": " + onhand +
                          " штук в наличии. " +
                          "Цена: {0:C} за каждую единицу.",
                          cost);
        Console.WriteLine(
          "Общая стоимость по наименованию <{0}>: {1:C}." ,
          item, cost * onhand);
        break;
      }
    }
  }
  catch(EndOfStreamException) {
    Console.WriteLine("Элемент не найден.");
```

```
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message + "Ошибка при чтении.");
    }

    dataIn.Close();
  }
}
```

Вот результаты выполнения этой программы:

```
Введите наименование для поиска: Отвертки

Отвертки: 18 штук в наличии. Цена: $1.50 за каждую единицу.
Общая стоимость по наименованию <Отвертки>: $27.00.
```

В этой программе стоит обратить внимание на то, как хранится информация о наличии товаров на складе, а именно — на двоичный формат хранения данных. Следовательно, количество товаров, имеющихся в наличии, и их стоимость хранятся с использованием двоичного формата, а не в текстовом виде, удобном для восприятия человеком. Это позволяет выполнять вычисления над числовыми данными, не делая дополнительных преобразований.

Хотелось бы также обратить ваше внимание на то, как обнаруживается здесь конец файла. Поскольку при достижении конца потока методы ввода двоичной информации генерируют исключение типа `EndOfStreamException`, эта программа просто считывает содержимое файла до тех пор, пока либо не найдет нужный элемент, либо не сгенерируется это исключение. Таким образом, для обнаружения конца файла в данном случае специального механизма не требуется.

Файлы с произвольным доступом

До сих пор мы использовали последовательные файлы, т.е. файлы, доступ к содержимому которых организован строго линейно, байт за байтом. Но в C# также возможен доступ к файлу, осуществляющийся случайным образом. В этом случае необходимо использовать метод `Seek()`, определенный в классе `FileStream`. Этот метод позволяет установить индикатор позиции (или указатель позиции) в любое место файла.

Заголовочная информация о методе `Seek()` имеет следующий вид:

```
long Seek(long newPos, SeekOrigin origin)
```

Здесь элемент *newPos* означает новую позицию, выраженную в байтах, файлового указателя относительно позиции, заданной элементом *origin*. Элемент *origin* может принимать одно из значений, определенных перечислением `SeekOrigin`.

Значение	*Описание*
`SeekOrigin.Begin`	Поиск от начала файла
`SeekOrigin.Current`	Поиск от текущей позиции
`SeekOrigin.End`	Поиск от конца файла

После обращению к методу `Seek()` следующая операция чтения или записи данных будет выполняться на новой позиции в файле. Если при выполнении поиска возникнет какая-либо ошибка, генерируется исключение типа `IOException`. Если базовый поток не поддерживает функцию запроса нужной позиции, генерируется исключение типа `NotSupportedException`.

Рассмотрим пример, который демонстрирует выполнение операций ввода-вывода с произвольным доступом. Следующая программа записывает в файл алфавит прописными буквами, а затем беспорядочно считывает его.

```
// Демонстрация произвольного доступа к файлу.

using System;
using System.IO;

class RandomAccessDemo {
  public static void Main() {
    FileStream f;
    char ch;

    try {
      f = new FileStream("random.dat", FileMode.Create);
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message);
      return ;
    }

    // Записываем в файл алфавит.
    for(int i=0; i < 26; i++) {
      try {
        f.WriteByte((byte)('A'+i));
      }
      catch(IOException exc) {
        Console.WriteLine(exc.Message);
        return ;
      }
    }

    try {
      // Теперь считываем отдельные значения.
      f.Seek(0, SeekOrigin.Begin); // Поиск первого байта.
      ch = (char) f.ReadByte();
      Console.WriteLine("Первое значение равно " + ch);

      f.Seek(1, SeekOrigin.Begin); // Поиск второго байта.
      ch = (char) f.ReadByte();
      Console.WriteLine("Второе значение равно " + ch);

      f.Seek(4, SeekOrigin.Begin); // Поиск пятого байта.
      ch = (char) f.ReadByte();
      Console.WriteLine("Пятое значение равно " + ch);

      Console.WriteLine();

      // Теперь считываем значения через одно.
      Console.WriteLine("Выборка значений через одно: ");
      for(int i=0; i < 26; i += 2) {
        f.Seek(i, SeekOrigin.Begin); // Переход
                                     // к i-му байту.
        ch = (char) f.ReadByte();
        Console.Write(ch + " ");
      }
    }
```

```
    catch(IOException exc) {
      Console.WriteLine(exc.Message);
    }

    Console.WriteLine();
    f.Close();
  }
}
```

При выполнении этой программы получены такие результаты:

```
Первое значение равно A
Второе значение равно B
Пятое значение равно E

Выборка значений через одно:
A C E G I K M O Q S U W Y
```

Использование класса MemoryStream

Не всегда удобно выполнять операции ввода-вывода непосредственно с помощью физического устройства. Иногда полезно считывать входные данные из массива или записывать их в массив. В этом случае стоит воспользоваться классом `MemoryStream`. Класс `MemoryStream` — это реализация класса `Stream`, в которой для операций ввода-вывода используются массивы байтов. Вот как выглядит конструктор этого класса:

```
MemoryStream(byte[] buf)
```

Здесь элемент *buf* — это массив байтов, который предполагается использовать в операциях ввода-вывода в качестве источника и/или приемника информации. В поток, создаваемый этим конструктором, можно записывать данные или считывать их в него. Этот поток поддерживает метод `Seek()`. Перед использованием этого конструктора необходимо позаботиться о достаточном размере массива *buf*, чтобы он позволил сохранить все направляемые в него данные.

Вот пример программы, которая демонстрирует использование класса `MemoryStream`:

```
// Демонстрация использования класса MemoryStream.

using System;
using System.IO;

class MemStrDemo {
  public static void Main() {
    byte[] storage = new byte[255];

    // Создаем поток с ориентацией на память.
    MemoryStream memstrm = new MemoryStream(storage);

    // Помещаем объект memstrm в оболочки StreamWriter
    // и StreamReader.
    StreamWriter memwtr = new StreamWriter(memstrm);
    StreamReader memrdr = new StreamReader(memstrm);

    // Записываем данные в память с помощью
    // объекта memwtr.
    for(int i=0; i < 10; i++)
```

```
        memwtr.WriteLine("byte [" + i + "]: " + i);

    // Ставим в конце точку.
    memwtr.Write('.');

    memwtr.Flush();

    Console.WriteLine(
          "Считываем данные прямо из массива storage: ");

    // Отображаем напрямую содержимое памяти.
    foreach(char ch in storage) {
      if (ch == '.') break;
      Console.Write(ch);
    }

    Console.WriteLine(
      "\nСчитываем данные посредством объекта memrdr: ");

    // Считываем данные из объекта memstrm, используя
    // средство чтения потоков.
    memstrm.Seek(0, SeekOrigin.Begin); // Установка
                 // указателя позиции в начало потока.

    string str = memrdr.ReadLine();
    while(str != null) {
      str = memrdr.ReadLine();
      if(str.CompareTo(".") == 0) break;
      Console.WriteLine(str);
    }

  }
}
```

Вот как выглядят результаты выполнения этой программы:

```
Считываем данные прямо из массива storage:
byte [0]: 0
byte [1]: 1
byte [2]: 2
byte [3]: 3
byte [4]: 4
byte [5]: 5
byte [6]: 6
byte [7]: 7
byte [8]: 8
byte [9]: 9

Считываем данные посредством объекта memrdr:
byte [1]: 1
byte [2]: 2
byte [3]: 3
byte [4]: 4
byte [5]: 5
byte [6]: 6
byte [7]: 7
byte [8]: 8
byte [9]: 9
```

В этой программе создается байтовый массив storage. Этот массив затем используется в качестве базовой области памяти для объекта memstrm класса MemoryStream. На основе объекта memstrm создаются объект класса StreamReader с именем memrdr и объект класса StreamWriter с именем memwtr. Через объект memwtr данные записываются в поток, ориентированный на конкретную область памяти. Обратите внимание на то, что после записи выходных данных для объекта memwtr вызывается метод flush(). Тем самым гарантируется, что содержимое буфера, связанного с потоком memwtr, реально перепишется в базовый массив. Затем содержимое этого байтового массива отображается "вручную", т.е. с помощью цикла foreach. После этого посредством метода Seek() указатель позиции устанавливается в начало потока, и его содержимое считывается с использованием объекта memrdr.

Потоки, ориентированные на память, весьма полезны в программировании. Например, можно заблаговременно составить выходные данные и хранить их в массиве до тех пор, пока в них не отпадет необходимость. Такой подход особенно полезен в программировании для такой GUI-среды, как Windows. Можно также перенаправить стандартный поток для считывания данных из массива. Это полезно, например, при вводе тестовой информации в программу.

Использование классов StringReader и StringWriter

В некоторых приложениях при выполнении операций ввода-вывода, ориентированных на использование памяти в качестве базовой области хранения данных, проще работать не с байтовыми (byte-) массивами, а со строковыми (string-). В этом случае используйте классы StringReader и StringWriter. Класс StringReader наследует класс TextReader, а класс StringWriter — класс TextWriter. Следовательно, эти потоки имеют доступ к методам, определенным в этих классах. Например, вы можете вызывать метод ReadLine() для объекта класса StringReader и метод WriteLine() для объекта класса StringWriter.

Конструктор класса StringReader имеет следующий вид:

```
StringReader(string str)
```

Здесь параметр *str* представляет собой строку, из которой должны считываться данные.

В классе StringWriter определено несколько конструкторов. Мы будем использовать такой:

```
StringWriter()
```

Этот конструктор создает "записывающий" механизм, который помещает выходные данные в строку. Эта строка автоматически создается объектом класса StringWriter. Содержимое строки можно получить, вызвав метод ToString().

Рассмотрим пример использования классов StringReader и StringWriter.

```
// Демонстрация использования классов StringReader
// и StringWriter.

using System;
using System.IO;

class StrRdrDemo {
  public static void Main() {
    // Создаем объект класса StringWriter.
```

```
    StringWriter strwtr = new StringWriter();

    // Записываем данные в StringWriter-объект.
    for(int i=0; i < 10; i++)
       strwtr.WriteLine("Значение i равно: " + i);

    // Создаем объект класса StringReader.

    StringReader strrdr = new StringReader(
                                    strwtr.ToString());

    // Теперь считываем данные из StringReader-объекта.
    string str = strrdr.ReadLine();
    while(str != null) {
      str = strrdr.ReadLine();
      Console.WriteLine(str);
    }

  }
}
```

Результаты выполнения этой программы имеют такой вид:

```
Значение i равно: 1
Значение i равно: 2
Значение i равно: 3
Значение i равно: 4
Значение i равно: 5
Значение i равно: 6
Значение i равно: 7
Значение i равно: 8
Значение i равно: 9
```

Эта программа сначала создает объект класса `StringWriter` с именем `strwtr` и записывает в него данные с помощью метода `WriteLine()`. Затем создается объект класса `StringReader` с использованием строки, содержащейся в объекте `strwtr`, и метода `ToString()`. Наконец, содержимое строки считывается с помощью метода `ReadLine()`.

Преобразование числовых строк во внутреннее представление

Прежде чем завершить тему ввода-вывода, рассмотрим метод, который будет весьма полезен программистам при считывании числовых строк. Как вы знаете, C#-метод `WriteLine()` предоставляет удобный способ вывода данных различных типов (включая числовые значения таких встроенных типов, как `int` и `double`) на консольное устройство. Следовательно, метод `WriteLine()` автоматически преобразует числовые значения в удобную для восприятия человеком форму. Однако C# не обеспечивает обратную функцию, т.е. метод ввода, который бы считывал и преобразовывал строковые представления числовых значений во внутренний двоичный формат. Например, не существует метода ввода данных, который бы считывал такую строку, как "100", и автоматически преобразовывал ее в соответствующее двоичное значение, которое можно было бы хранить в `int`-переменной. Для решения этой задачи понадобится метод, определенный для всех встроенных числовых типов, — `Parse()`.

Приступая к решению этой задачи, необходимо отметить такой важный факт. Все встроенные C#-типы (например, `int` и `double`) в действительности являются лишь псевдонимами (т.е. другими именами) для структур, определенных в среде .NET Framework. Компания Microsoft заявляет, что понятия C#-типа и .NET-типа структуры неразличимы. Первое — просто еще одно имя для другого. Поскольку C#-типы значений поддерживаются структурами, они имеют члены, определенные для этих структур.

Ниже представлены .NET-имена структур и их C#-эквиваленты (в виде ключевых слов) для числовых типов.

.NET-имя структуры	*C#-имя*
`Decimal`	`decimal`
`Double`	`double`
`Single`	`float`
`Int16`	`short`
`Int32`	`int`
`Int64`	`long`
`UInt16`	`ushort`
`UInt32`	`uint`
`UInt64`	`ulong`
`Byte`	`byte`
`Sbyte`	`sbyte`

Эти структуры определены в пространстве имен `System`. Таким образом, составное имя для структуры `Int32` "звучит" как `System.Int32`. Для этих структур определен широкий диапазон методов, которые способствуют полной интеграции типов значений в C#-иерархию объектов. В качестве дополнительного "вознаграждения" эти числовые структуры также определяют статические методы, которые преобразуют числовую строку в соответствующий двоичный эквивалент. Эти методы преобразования представлены в следующей таблице. Каждый метод возвращает двоичное значение, которое соответствует строке.

Структура	*Метод преобразования*
`Decimal`	`static decimal Parse(string str)`
`Double`	`static double Parse(string str)`
`Single`	`static float Parse(string str)`
`Int64`	`static long Parse(string str)`
`Int32`	`static int Parse(string str)`
`Int16`	`static short Parse(string str)`
`UInt64`	`static ulong Parse(string str)`
`UInt32`	`static uint Parse(string str)`
`UInt16`	`static ushort Parse(string str)`
`Byte`	`static byte Parse(string str)`
`SByte`	`static sbyte Parse(string str)`

Методы `Parse()` генерируют исключение типа `FormatException`, если параметр `str` не содержит числа, допустимого для типа вызывающего объекта. Если параметр `str` имеет `null`-значение, генерируется исключение типа `ArgumentNullException`, а

если значение параметра *str* превышает диапазон, допустимый для типа вызывающего объекта, — исключение типа `OverflowException`.

Методы синтаксического анализа позволяют легко преобразовать числовое значение, прочитанное в виде строки с клавиатуры или текстового файла, в соответствующий внутренний формат. Например, следующая программа вычисляет среднее арифметическое от чисел, введенных пользователем в виде списка. Сначала пользователю предлагается ввести количество усредняемых чисел, а затем программа считывает эти числа с помощью метода `ReadLine()` и с помощью метода `Int32.Parse()` преобразует строки в целочисленное значение. Затем она вводит значения, используя метод `Double.Parse()` для преобразования строк в их `double`-эквиваленты.

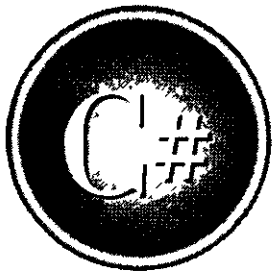
```
// Эта программа усредняет список чисел,
// введенных пользователем.

using System;
using System.IO;

class AvgNums {
  public static void Main() {
    string str;
    int n;
    double sum = 0.0;
    double avg, t;

    Console.Write("Сколько чисел вы собираетесь ввести: ");
    str = Console.ReadLine();
    try {
      n = Int32.Parse(str);
    }
    catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      n = 0;
    }
    catch(OverflowException exc) {
      Console.WriteLine(exc.Message);
      n = 0;
    }

    Console.WriteLine("Введите " + n + " чисел.");
    for(int i=0; i < n ; i++)  {
      Console.Write(": ");
      str = Console.ReadLine();
      try {
        t = Double.Parse(str);
      } catch(FormatException exc) {
        Console.WriteLine(exc.Message);
        t = 0.0;
      }
      catch(OverflowException exc) {
        Console.WriteLine(exc.Message);
        t = 0;
      }
      sum += t;
    }
    avg = sum / n;
    Console.WriteLine("Среднее равно " + avg);
  }
}
```

Вот как могут выглядеть результаты выполнения этой программы:

```
Сколько чисел вы собираетесь ввести: 5
Введите 5 чисел.
: 1.1
: 2.2
: 3.3
: 4.4
: 5.5
Среднее равно 3.3
```

И еще. Вы должны использовать надлежащий метод анализа для типа значения, которое вы пытаетесь преобразовать. Например, попытка использовать метод `Int32.Parse()` для строки, содержащей значение с плавающей точкой, желаемого результата не даст.

Полный справочник по



Глава 15

Делегаты и события

В этой главе рассматриваются два новых C#-средства: делегаты и события. Делегат предоставляет возможность инкапсулировать метод, а событие — это своего рода уведомление о том, что имело место некоторое действие. Делегаты и события связанны между собой, поскольку событие создается на основе делегата. Эти средства расширяют диапазон задач программирования, к которым можно применить язык C#.

Делегаты

Начнем с определения термина *делегат* (delegate). Делегат — это объект, который может ссылаться на метод. Таким образом, создавая делегат, вы по сути создаете объект, который может содержать ссылку на метод. Более того, этот метод можно вызвать посредством соответствующей ссылки. Таким образом, делегат может вызывать метод, на который он ссылается.

На первый взгляд идея ссылки на метод может показаться странной, поскольку обычно мы имеем дело с ссылками, которые указывают на объекты, но в действительности здесь разница небольшая. Как разъяснялось выше, ссылка по существу представляет собой адрес памяти. Следовательно, ссылка на объект — это адрес объекта. Даже несмотря на то что метод не является объектом, он тоже имеет отношение к физической области памяти, а адрес его точки входа — это адрес, к которому происходит обращение при вызове метода. Этот адрес можно присвоить делегату. Если уж делегат ссылается на метод, этот метод можно вызвать посредством данного делегата.

Если вы знакомы с C/C++, то вам будет полезно узнать, что делегат в C# аналогичен указателю на функцию в C/C++

Важно понимать, что во время выполнения программы один и тот же делегат можно использовать для вызова различных методов, просто заменив метод, на который ссылается этот делегат. Таким образом, метод, который будет вызван делегатом, определяется не в период компиляции программы, а во время ее работы. В этом и состоит достоинство делегата.

Делегат объявляется с помощью ключевого слова `delegate`. Общая форма объявления делегата имеет следующий вид:

```
delegate тип_возврата имя(список_параметров);
```

Здесь элемент *тип_возврата* представляет собой тип значений, возвращаемых методами, которые этот делегат будет вызывать. Имя делегата указывается элементом *имя*. Параметры, принимаемые методами, которые вызываются посредством делегата, задаются с помощью элемента *список_параметров*. Делегат может вызывать только такие методы, у которых тип возвращаемого значения и список параметров (т.е. его сигнатура) совпадают с соответствующими элементами объявления делегата.

Делегат может вызывать либо метод экземпляра класса, связанный с объектом, или статический метод, связанный с классом.

Чтобы увидеть делегат в действии, начнем со следующего простого примера:

```
// Простой пример использования делегата.

using System;

// Объявляем делегат.
delegate string strMod(string str);

class DelegateTest {
  // Метод заменяет пробелы дефисами.
```

```
  static string replaceSpaces(string a) {
    Console.WriteLine("Замена пробелов дефисами.");
    return a.Replace(' ', '-');
  }

  // Метод удаляет пробелы.
  static string removeSpaces(string a) {
    string temp = "";
    int i;

    Console.WriteLine("Удаление пробелов.");
    for(i=0; i < a.Length; i++)
      if(a[i] != ' ') temp += a[i];

    return temp;
  }

  // Метод реверсирует строку.
  static string reverse(string a) {
    string temp = "";
    int i, j;

    Console.WriteLine("Реверсирование строки.");
    for(j=0, i=a.Length-1; i >= 0; i--, j++)
      temp += a[i];

    return temp;
  }

  public static void Main() {
    // Создание делегата.
    strMod strOp = new strMod(replaceSpaces);
    string str;

    // Вызываем методы посредством делегата.
    str = strOp("Это простой тест.");
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();

    strOp = new strMod(removeSpaces);
    str = strOp("Это простой тест.");
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();

    strOp = new strMod(reverse);
    str = strOp("Это простой тест.");
    Console.WriteLine("Результирующая строка: " + str);
  }
}
```

Результаты выполнения этой программы выглядят так:

```
Замена пробелов дефисами.
Результирующая строка: Это-простой-тест.

Удаление пробелов.
Результирующая строка: Этопростойтест.

Реверсирование строки.
Результирующая строка: .тсет йотсорп отЭ
```

Итак, в программе объявляется делегат с именем strMod, который принимает один параметр типа string и возвращает string-значение. В классе DelegateTest объявляны три статических метода, сигнатура которых совпадает с сигнатурой, заданной делегатом. Эти методы предназначены для модификации строк определенного вида. Обратите внимание на то, что метод replaceSpaces() для замены пробелов дефисами использует метод Replace() — один из методов класса string.

В методе Main() создается ссылка типа strMod с именем strOp, и ей присваивается ссылка на метод replaceSpaces(). Внимательно рассмотрите следующую строку:

```
strMod strOp = new strMod(replaceSpaces);
```

Обратите внимание на то, что метод replaceSpaces() передается делегату в качестве параметра. Здесь используется только имя метода (параметры не указываются). Это наблюдение можно обобщить: при реализации делегата задается только имя метода, на который должен ссылаться этот делегат. Кроме того, объявление метода должно соответствовать объявлению делегата. В противном случае вы получите сообщение об ошибке еще во время компиляции.

Затем метод replaceSpaces() вызывается посредством экземпляра делегата с именем strOp, как показано в следующей строке:

```
str = strOp("Это простой тест.");
```

Поскольку экземпляр strOp ссылается на метод replaceSpaces(), то вызывается именно метод replaceSpaces(). Затем экземпляру делегата strOp присваивается ссылка на метод removeSpaces(), после чего strOp вызывается снова. На этот раз вызывается метод removeSpaces().

Наконец, экземпляру делегата strOp присваивается ссылка на метод reverse(), и strOp вызывается еще раз. Это, как нетрудно догадаться, приводит к вызову метода reverse().

Главное в этом примере то, что вызов экземпляра делегата strOp трансформируется в обращение к методу, на который ссылается strOp при вызове. Таким образом, решение о вызываемом методе принимается во время выполнения программы, а не в период компиляции.

Несмотря на то что в предыдущем примере используются статические методы, делегат может также ссылаться на методы экземпляров класса. Однако он должен при этом использовать объектную ссылку. Например, вот как выглядит предыдущая программа, переписанная с целью инкапсуляции операций над строками внутри класса StringOps:

```
// Делегаты могут ссылаться также на методы
// экземпляров класса.

using System;

// Объявляем делегат.
delegate string strMod(string str);

class StringOps {
  // Метод заменяет пробелы дефисами.
  public string replaceSpaces(string a) {
    Console.WriteLine("замена пробелов дефисами.");
    return a.Replace(' ', '-');
  }

  // Метод удаляет пробелы.
  public string removeSpaces(string a) {
    string temp = "";
    int i;
```

```
    Console.WriteLine("Удаление пробелов.");
    for(i=0; i < a.Length; i++)
      if(a[i] != ' ') temp += a[i];

    return temp;
  }

  // Метод реверсирует строку.
  public string reverse(string a) {
    string temp = "";
    int i, j;

    Console.WriteLine("Реверсирование строки.");
    for(j=0, i=a.Length-1; i >= 0; i--, j++)
      temp += a[i];

    return temp;
  }
}

class DelegateTest {
  public static void Main() {
    StringOps so = new StringOps(); // Создаем экземпляр
                                    // класса StringOps.

    // Создаем делегат.
    strMod strOp = new strMod(so.replaceSpaces);
    string str;

    // Вызываем методы с использованием делегатов.
    str = strOp("Это простой тест.");
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();

    strOp = new strMod(so.removeSpaces);
    str = strOp("Это простой тест.");
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();

    strOp = new strMod(so.reverse);
    str = strOp("Это простой тест.");
    Console.WriteLine("Результирующая строка: " + str);
  }
}
```

Результаты выполнения этой программы совпадают с результатами предыдущей версии, но в этом случае делегат ссылается на методы экземпляра класса `StringOps`.

Многоадресатная передача

Одна из самых интересных возможностей делегата — поддержка *многоадресатной передачи* (multicasting). Выражаясь простым языком, Многоадресатная передача — это способность создавать список вызовов (или цепочку вызовов) методов, которые должны автоматически вызываться при вызове делегата. Такую цепочку создать нетрудно. Достаточно создать экземпляр делегата, а затем для добавления методов в эту цепочку использовать оператор "+=". Для удаления метода из цепочки используется оператор "-=". (Можно также для добавления и удаления методов использовать в отдельности операторы "+", "-" и "=", но чаще применяются составные операторы "+=" и "-=".)

Делегат с многоадресатной передачей имеет одно ограничение: он должен возвращать тип void.

Рассмотрим следующий пример многоадресатной передачи. Это — переработанный вариант предыдущих примеров, в котором тип string для значений, возвращаемых методами обработки строк, заменен типом void, а для возврата модифицированных строк используется ref-параметр.

```
// Демонстрация использования многоадресатной передачи.

using System;

// Объявляем делегат.
delegate void strMod(ref string str);

class StringOps {
  // Метод заменяет пробелы дефисами.
  static void replaceSpaces(ref string a) {
    Console.WriteLine("Замена пробелов дефисами.");
    a = a.Replace(' ', '-');
  }

  // Метод удаляет пробелы.
  static void removeSpaces(ref string a) {
    string temp = "";
    int i;

    Console.WriteLine("Удаление пробелов.");
    for(i=0; i < a.Length; i++)
      if(a[i] != ' ') temp += a[i];

    a = temp;
  }

  // Метод реверсирует строку.
  static void reverse(ref string a) {
    string temp = "";
    int i, j;

    Console.WriteLine("Реверсирование строки.");
    for(j=0, i=a.Length-1; i >= 0; i--, j++)
      temp += a[i];

    a = temp;
  }

  public static void Main() {
    // Создаем экземпляры делегатов.
    strMod strOp;
    strMod replaceSp = new strMod(replaceSpaces);
    strMod removeSp = new strMod(removeSpaces);
    strMod reverseStr = new strMod(reverse);
    string str = "Это простой тест.";

    // Организация многоадресатной передачи.
    strOp = replaceSp;
    strOp += reverseStr;

    // Вызов делегата с многоадресатной передачей.
    strOp(ref str);
```

```
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();

    // Удаляем метод замены пробелов и
    // добавляем метод их удаления.
    strOp -= replaceSp;
    strOp += removeSp;

    str = "Это простой тест."; // Восстановление
                               // исходной строки.

    // Вызов делегата с многоадресатной передачей.
    strOp(ref str);
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();
  }
}
```

Вот как выглядят результаты выполнения этой программы:

```
Замена пробелов дефисами.
Реверсирование строки.
Результирующая строка: .тсет-йотсорп-отЭ

Реверсирование строки.
Удаление пробелов.
Результирующая строка: .тсетйотсорпотЭ
```

В методе `Main()` создаются четыре экземпляра делегата. Первый, `strOp`, имеет `null`-значение. Три других ссылаются на методы модификации строк. Затем организуется делегат для многоадресатной передачи, который вызывает методы `removeSpaces()` и `reverse()`. Это достигается благодаря следующим строкам программы:

```
strOp = replaceSp;
strOp += reverseStr;
```

Сначала делегату `strOp` присваивается ссылка `replaceSp`. Затем, с помощью оператора "+=", в цепочку вызовов добавляется ссылка `reverseStr`. При вызове делегата `strOp` в этом случае вызываются оба метода, заменяя пробелы дефисами и реверсируя строку.

Затем при выполнении строки программы

```
strOp -= replaceSp;
```

из цепочки вызовов удаляется ссылка `replaceSp`, а с помощью строки

```
strOp += removeSp;
```

в цепочку вызовов добавляется ссылка `removeSp`.

Затем делегат `StrOp` вызывается снова. На этот раз из исходной строки удаляются пробелы, после чего она реверсируется.

Цепочки вызовов, организованные с помощью делегата, — мощный механизм, который позволяет определять набор методов, выполняемых "единым блоком". Как будет показано ниже, цепочки делегатов имеют особое значение для событий.

Класс `System.Delegate`

Все делегаты представляют собой классы, которые неявным образом выводятся из класса `System.Delegate`. Обычно его члены не используются напрямую, и в этой книге не показано явное использование класса `System.Delegate`. Все же в некоторых ситуациях его члены могут оказаться весьма полезными.

Назначение делегатов

Несмотря на то что предыдущие примеры программ продемонстрировали, "как" работают делегаты, они не содержали ответа на вопрос "зачем это нужно?". Так вот, делегаты используются по двум основным причинам. Во-первых, как будет показано в следующем разделе, делегаты обеспечивают поддержку функционирования событий. Во-вторых, делегаты позволяют во время выполнения программы выполнить метод, который точно не известен в период компиляции. Эта возможность особенно полезна, когда нужно создать оболочку, к которой могли бы подключаться программные компоненты. Например, представьте графическую программу (наподобие стандартной утилиты Windows Paint). Используя делегат, можно было бы разрешить пользователю подключать специальные цветные светофильтры или анализаторы изображений. Более того, пользователь мог бы создавать "свои" последовательности этих фильтров или анализаторов. С помощью делегатов организовать такой алгоритм очень легко.

 События

На основе делегатов построено еще одно важное средство C#: *событие* (event). Событие — это по сути автоматическое уведомление о выполнении некоторого действия. События работают следующим образом. Объект, которому необходима информация о некотором событии, регистрирует обработчик для этого события. Когда ожидаемое событие происходит, вызываются все зарегистрированные обработчики. А теперь внимание: обработчики событий представляются делегатами.

События — это члены класса, которые объявляются с использованием ключевого слова `event`. Наиболее распространенная форма объявления события имеет следующий вид:

```
event событийный_делегат объект;
```

Здесь элемент *событийный_делегат* означает имя делегата, используемого для поддержки объявляемого события, а элемент *объект* — это имя создаваемого событийного объекта.

Начнем с рассмотрения очень простого примера.

```
// Демонстрация использования простейшего события.

using System;

// Объявляем делегат для события.
delegate void MyEventHandler();

// Объявляем класс события.
class MyEvent {
  public event MyEventHandler SomeEvent;

  // Этот метод вызывается для генерирования события.
  public void OnSomeEvent() {
    if(SomeEvent != null)
      SomeEvent();
  }
}

class EventDemo {
  // Обработчик события.
  static void handler() {
```

```
    Console.WriteLine("Произошло событие.");
  }

  public static void Main() {
    MyEvent evt = new MyEvent();

    // Добавляем метод handler() в список события.
    evt.SomeEvent += new MyEventHandler(handler);

    // Генерируем событие.
    evt.OnSomeEvent();
  }
}
```

При выполнении программа отображает следующие результаты:

```
Произошло событие.
```

Несмотря на простоту, программа содержит все элементы, необходимые для надлежащей обработки события. Рассмотрим их по порядку.

Программа начинается с такого объявления делегата для обработчика события:

```
delegate void MyEventHandler();
```

Все события активизируются посредством делегата. Следовательно, событийный делегат определяет сигнатуру для события. В данном случае параметры отсутствуют, однако событийные параметры разрешены. Поскольку события обычно предназначены для многоадресатной передачи, они должны возвращать значение типа `void`.

Затем создается класс события `MyEvent`. При выполнении следующей строки кода, принадлежащей этому классу, объявляется событийный объект `SomeEvent`:

```
public event MyEventHandler SomeEvent;
```

Обратите внимание на синтаксис. Именно так объявляются события всех типов.

Кроме того, внутри класса `MyEvent` объявляется метод `OnSomeEvent()`, который в этой программе вызывается, чтобы сигнализировать о событии. (Другими словами, этот метод вызывается, когда происходит событие.) Как показано в следующем фрагменте кода, он вызывает обработчик события посредством делегата `SomeEvent`.

```
if(SomeEvent != null)
  SomeEvent();
```

Обратите внимание на то, что обработчик события вызывается только в том случае, если делегат `SomeEvent` не равен `null`-значению. Поскольку другие части программы, чтобы получить уведомлении о событии, должны зарегистрироваться, можно сделать так, чтобы метод `OnSomeEvent()` был вызван до регистрации любого обработчика события. Чтобы предотвратить вызов `null`-объекта, событийный делегат необходимо протестировать и убедиться в том, что он не равен `null`-значению.

Внутри класса `EventDemo` создается обработчик события `handler()`. В этом примере обработчик события просто отображает сообщение, но ясно, что другие обработчики могли бы выполнять более полезные действия. Как показано в следующем фрагменте кода, в методе `Main()` создается объект класса `MyEvent`, а метод `handler()` регистрируется в качестве обработчика этого события.

```
MyEvent evt = new MyEvent();

// Добавляем метод handler() в список события.
evt.SomeEvent += new MyEventHandler(handler);
```

Обратите внимание на то, что обработчик добавляется в список с использованием составного оператора "+=". Следует отметить, что события поддерживают только операторы "+=" и "-=". В нашем примере метод `handler()` является статическим, но в

общем случае обработчики событий могут быть методами экземпляров классов. Наконец, при выполнении следующей инструкции "происходит" событие, о котором мы так много говорили.

```
// Генерируем событие.
evt.OnSomeEvent();
```

При вызове метода `OnSomeEvent()` вызываются все зарегистрированные обработчики событий. В данном случае зарегистрирован только один обработчик, но, как вы увидите в следующем разделе, их могло бы быть и больше.

Пример события для многоадресатной передачи

Подобно делегатам события могут предназначаться для многоадресатной передачи. В этом случае на одно уведомление о событии может отвечать несколько объектов. Рассмотрим пример.

```
// Демонстрация использования события, предназначенного
// для многоадресатной передачи.

using System;

// Объявляем делегат для события.
delegate void MyEventHandler();

// Объявляем класс события.
class MyEvent {
  public event MyEventHandler SomeEvent;

  // Этот метод вызывается для генерирования события.
  public void OnSomeEvent() {
    if(SomeEvent != null)
      SomeEvent();
  }
}

class X {
  public void Xhandler() {
    Console.WriteLine("Событие, полученное объектом X.");
  }
}

class Y {
  public void Yhandler() {
    Console.WriteLine("Событие, полученное объектом Y.");
  }
}

class EventDemo {
  static void handler() {
    Console.WriteLine(
              "Событие, полученное классом EventDemo.");
  }

  public static void Main() {
    MyEvent evt = new MyEvent();
    X xOb = new X();
    Y yOb = new Y();
```

```
    // Добавляем обработчики в список события.
    evt.SomeEvent += new MyEventHandler(handler);
    evt.SomeEvent += new MyEventHandler(xOb.Xhandler);
    evt.SomeEvent += new MyEventHandler(yOb.Yhandler);

    // Генерируем событие.
    evt.OnSomeEvent();
    Console.WriteLine();

    // Удаляем один обработчик.
    evt.SomeEvent -= new MyEventHandler(xOb.Xhandler);
    evt.OnSomeEvent();
  }
}
```

Результаты выполнения этой программы имеют следующий вид:

```
Событие, полученное классом EventDemo.
Событие, полученное объектом X.
Событие, полученное объектом Y.

Событие, полученное классом EventDemo.
Событие, полученное объектом Y.
```

В этом примере создается два дополнительных класса `X` и `Y`, в которых также определяются обработчики событий, совместимые с сигнатурой делегата `MyEventHandler`. Следовательно, эти обработчики могут стать частью цепочки событийных вызовов. Обратите внимание на то, что обработчики в классах `X` и `Y` не являются статическими. Это значит, что сначала должны быть созданы объекты каждого класса, после чего в цепочку событийных вызовов должен быть добавлен обработчик, связанный с каждым экземпляром класса. Различие между статическими обработчиками и обработчиками экземпляров классов рассматривается в следующем разделе.

Сравнение методов экземпляров классов со статическими методами, используемыми в качестве обработчиков событий

Несмотря на то что и методы экземпляров классов, и статические методы могут служить обработчиками событий, в их использовании в этом качестве есть существенные различия. Если в качестве обработчика используется статический метод, уведомление о событии применяется к классу (и неявно ко всем объектам этого класса). Если же в качестве обработчика событий используется метод экземпляра класса, события посылаются к конкретным экземплярам этого класса. Следовательно, каждый объект класса, который должен получать уведомление о событии, необходимо регистрировать в отдельности. На практике в большинстве случаев "роль" обработчиков событий "играют" методы экземпляров классов, но, безусловно, все зависит от конкретной ситуации. Теперь перейдем к рассмотрению примеров.

В следующей программе создается класс `X`, в котором в качестве обработчика событий определен метод экземпляра. Это значит, что для получения информации о событиях каждый объект класса `X` необходимо регистрировать отдельно. Для демонстрации этого факта программа готовит уведомление о событии для многоадресатной передачи трем обьектам типа `X`.

```
/* При использовании в качестве обработчиков событий
   методов экземпляров уведомление о событиях принимают
   отдельные объекты. */

using System;
```

```
// Объявляем делегат для события.
delegate void MyEventHandler();

// Объявляем класс события.
class MyEvent {
  public event MyEventHandler SomeEvent;

  // Этот метод вызывается для генерирования события.
  public void OnSomeEvent() {
    if(SomeEvent != null)
      SomeEvent();
  }
}

class X {
  int id;

  public X(int x) { id = x; }

  // Метод экземпляра, используемый в качестве
  // обработчика событий.
  public void Xhandler() {
    Console.WriteLine("Событие принято объектом " + id);
  }
}

class EventDemo {
  public static void Main() {
    MyEvent evt = new MyEvent();
    X o1 = new X(1);
    X o2 = new X(2);
    X o3 = new X(3);

    evt.SomeEvent += new MyEventHandler(o1.Xhandler);
    evt.SomeEvent += new MyEventHandler(o2.Xhandler);
    evt.SomeEvent += new MyEventHandler(o3.Xhandler);

    // Генерируем событие.
    evt.OnSomeEvent();
  }
}
```

Результаты выполнения этой программы имеют такой вид:

```
Событие принято объектом 1
Событие принято объектом 2
Событие принято объектом 3
```

Как подтверждают эти результаты, каждый объект заявляет о своей заинтересованности в событии и получает о нем отдельное уведомление.

Если же в качестве обработчика событий используется статический метод, то, как показано в следующей программе, события обрабатываются независимо от объекта.

```
/* При использовании в качестве обработчиков событий
   статического метода уведомление о событиях получает
   класс. */

using System;

// Объявляем делегат для события.
```

```
delegate void MyEventHandler();

// Объявляем класс события.
class MyEvent {
  public event MyEventHandler SomeEvent;

  // Этот метод вызывается для генерирования события.
  public void OnSomeEvent() {
    if(SomeEvent != null)
      SomeEvent();
  }
}

class X {

  /* Это статический метод, используемый в качестве
     обработчика события. */
  public static void Xhandler() {
    Console.WriteLine("Событие получено классом.");
  }
}

class EventDemo {
  public static void Main() {
    MyEvent evt = new MyEvent();

    evt.SomeEvent += new MyEventHandler(X.Xhandler);

    // Генерируем событие.
    evt.OnSomeEvent();
  }
}
```

Вот как выглядят результаты выполнения программы:

```
Событие получено классом.
```

Обратите внимание на то, что в программе не создается ни одного объекта типа `X`. Но поскольку `handler()` — статический метод класса `X`, его можно связать с событием `SomeEvent` и обеспечить его выполнение при вызове метода `OnSomeEvent()`.

Использование событийных средств доступа

Предусмотрены две формы записи инструкций, связанных с событиями. Форма, используемая в предыдущих примерах, обеспечивала создание событий, которые автоматически управляют списком вызова обработчиков, включая такие операции, как добавление обработчиков в список и удаление их из списка. Таким образом, можно было не беспокоиться о реализации операций по управлению этим списком. Поэтому такие типы событий, безусловно, являются наиболее применимыми. Однако можно и самим организовать ведение списка обработчиков событий, чтобы, например, реализовать специализированный механизм хранения событий.

Чтобы управлять списком обработчиков событий, используйте вторую форму `event`-инструкции, которая позволяет использовать средства доступа к событиям. Эти средства доступа дают возможность управлять реализацией списка обработчиков событий. Упомянутая форма имеет следующий вид:

```
event событийный_делегат имя_события {
  add {
```

```
    // Код добавления события в цепочку событий.
  }

  remove {
    // Код удаления события из цепочки событий.
  }
}
```

Эта форма включает два средства доступа к событиям: `add` и `remove`. Средство доступа `add` вызывается в случае, когда с помощью оператора "+=" в цепочку событий добавляется новый обработчик, а средство доступа `remove` вызывается, когда с помощью оператора "-=" из цепочки событий удаляется новый обработчик.

Средство доступа `add` или `remove` при вызове получает обработчик, который необходимо добавить или удалить, в качестве параметра. Этот параметр, как и в случае использования других средств доступа, называется `value`. При реализации средств доступа `add` и `remove` можно задать собственную схему хранения обработчиков событий. Например, для этого вы могли бы использовать массив, стек или очередь.

Рассмотрим пример использования событийных средств доступа. Здесь для хранения обработчиков событий взят массив. Поскольку этот массив содержит три элемента, в любой момент времени в событийной цепочке может храниться только три обработчика событий.

```
// Создание собственных средств управления списком событий.

using System;

// Объявляем делегат для события.
delegate void MyEventHandler();

// Объявляем класс события для хранения трех
// обработчиков событий.
class MyEvent {
  MyEventHandler[] evnt = new MyEventHandler[3];

  public event MyEventHandler SomeEvent {
    // Добавляем обработчик события в список.
    add {
      int i;

      for(i=0; i < 3; i++)
        if(evnt[i] == null) {
          evnt[i] = value;
          break;
        }
      if (i == 3)
        Console.WriteLine(
                  "Список обработчиков событий полон.");
    }

    // Удаляем обработчик события из списка.
    remove {
      int i;

      for(i=0; i < 3; i++)
        if(evnt[i] == value) {
          evnt[i] = null;
          break;
        }
```

```
      if (i == 3)
        Console.WriteLine("Обработчик события не найден.");
    }
  }

  // Этот метод вызывается для генерирования событий.
  public void OnSomeEvent() {
      for(int i=0; i < 3; i++)
        if(evnt[i] != null) evnt[i]();
  }

}

// Создаем классы, которые используют
// делегат MyEventHandler.
class W {
  public void Whandler() {
    Console.WriteLine("Событие получено объектом W.");
  }
}

class X {
  public void Xhandler() {
    Console.WriteLine("Событие получено объектом X.");
  }
}

class Y {
  public void Yhandler() {
    Console.WriteLine("Событие получено объектом Y.");
  }
}

class Z {
  public void Zhandler() {
    Console.WriteLine("Событие получено объектом Z.");
  }
}

class EventDemo {
  public static void Main() {
    MyEvent evt = new MyEvent();
    W wOb = new W();
    X xOb = new X();
    Y yOb = new Y();
    Z zOb = new Z();

    // Добавляем обработчики в список.
    Console.WriteLine("Добавление обработчиков событий.");
    evt.SomeEvent += new MyEventHandler(wOb.Whandler);
    evt.SomeEvent += new MyEventHandler(xOb.Xhandler);
    evt.SomeEvent += new MyEventHandler(yOb.Yhandler);

    // Этот обработчик сохранить нельзя -- список полон.
    evt.SomeEvent += new MyEventHandler(zOb.Zhandler);
    Console.WriteLine();

    // Генерируем события.
```

```
    evt.OnSomeEvent();
    Console.WriteLine();

    // Удаляем обработчик из списка.
    Console.WriteLine("Удаляем обработчик xOb.Xhandler.");
    evt.SomeEvent -= new MyEventHandler(xOb.Xhandler);
    evt.OnSomeEvent();

    Console.WriteLine();

    // Пытаемся удалить его еще раз.
    Console.WriteLine(
      "Попытка повторно удалить обработчик xOb.Xhandler.");
    evt.SomeEvent -= new MyEventHandler(xOb.Xhandler);
    evt.OnSomeEvent();

    Console.WriteLine();

    // Теперь добавляем обработчик Zhandler.
    Console.WriteLine("Добавляем обработчик zOb.Zhandler.");
    evt.SomeEvent += new MyEventHandler(zOb.Zhandler);
    evt.OnSomeEvent();

  }
}
```

Вот результаты выполнения программы:

```
Добавление обработчиков событий.
Список обработчиков событий полон.

Событие получено объектом W.
Событие получено объектом X.
Событие получено объектом Y.

Удаляем обработчик xOb.Xhandler.
Событие получено объектом W.
Событие получено объектом Y.

Попытка повторно удалить обработчик xOb.Xhandler.
Обработчик события не найден.
Событие получено объектом W.
Событие получено объектом Y.

Добавляем обработчик zOb.Zhandler.
Событие получено объектом W.
Событие получено объектом Z.
Событие получено объектом Y.
```

Рассмотрим внимательно код этой программы. Сначала определяется делегат обработчика события `MyEventHandler`. Код класса `MyEvent`, как показано в следующей инструкции, начинается с определения трехэлементного массива обработчиков событий `evnt`.

```
MyEventHandler[] evnt = new MyEventHandler[3];
```

Этот массив предназначен для хранения обработчиков событий, которые добавлены в цепочку событий. Элементы массива `evnt` инициализируются `null`-значениями по умолчанию.

Приведем `event`-инструкцию, в которой используются событийные средства доступа.

```
public event MyEventHandler SomeEvent {
  // Добавляем обработчик события в список.
  add {
    int i;

    for(i=0; i < 3; i++)
      if(evnt[i] == null) {
        evnt[i] = value;
        break;
      }
    if (i == 3)
      Console.WriteLine(
               "Список обработчиков событий полон.");
  }

  // Удаляем обработчик события из списка.
  remove {
    int i;

    for(i=0; i < 3; i++)
      if(evnt[i] == value) {
        evnt[i] = null;
        break;
      }
    if (i == 3)
      Console.WriteLine("Обработчик события не найден.");
  }
}
```

При добавлении в список обработчика событий вызывается add-средство, и ссылка на этот обработчик (содержащаяся в параметре value) помещается в первый встретившийся неиспользуемый элемент массива evnt. Если свободных элементов нет, выдается сообщение об ошибке. Поскольку массив evnt рассчитан на хранение лишь трех элементов, он может принять только три обработчика событий. При удалении заданного обработчика событий вызывается remove-средство, и в массиве evnt выполняется поиск ссылки на обработчик, переданной в параметре value. Если ссылка найдена, в соответствующий элемент массива помещается значение null, что равнозначно удалению обработчика из списка.

При генерировании события вызывается метод OnSomeEvent(). Он в цикле просматривает массив evnt, по очереди вызывая каждый обработчик событий.

Как показано в предыдущих примерах, при необходимости относительно нетрудно реализовать собственный механизм хранения обработчиков событий. Для большинства приложений все же лучше использовать стандартный механизм хранения, в котором не используются событийные средства доступа. Однако в определенных ситуациях форма event-инструкции, ориентированной на событийные средства доступа, может оказаться весьма полезной. Например, если в программе обработчики событий должны выполняться в порядке уменьшения приоритетов, а не в порядке их добавления в событийную цепочку, то для хранения таких обработчиков можно использовать очередь по приоритету.

Смешанные средства обработки событий

События можно определять в интерфейсах. "Поставкой" событий должны заниматься соответствующие классы. События можно определять как абстрактные. Обеспечить реализацию такого события должен производный класс. Однако события, реа-

лизованные с использованием средств доступа add и remove, абстрактными быть не могут. Любое событие можно определить с помощью ключевого слова sealed. Событие может быть виртуальным, т.е. его можно переопределить в производном классе.

Рекомендации по обработке событий в среде .NET Framework

C# позволяет программисту создавать события любого типа. Однако в целях компонентной совместимости со средой .NET Framework необходимо следовать рекомендациям, подготовленным Microsoft специально для этих целей. Центральное место в этих рекомендациях занимает требование того, чтобы обработчики событий имели два параметра. Первый должен быть ссылкой на объект, который будет генерировать событие. Второй должен иметь тип EventArgs и содержать остальную информацию, необходимую обработчику. Таким образом, .NET-совместимые обработчики событий должны иметь следующую общую форму записи:

```
void handler(object source, EventArgs arg) {
  // ...
}
```

Обычно параметр *source* передается вызывающим кодом. Параметр типа EventArgs содержит дополнительную информацию, которую в случае ненадобности можно проигнорировать.

Класс EventArgs не содержит полей, которые используются при передаче дополнительных данных обработчику; он используется в качестве базового класса, из которого можно выводить класс, содержащий необходимые поля. Но поскольку многие обработчики обходятся без дополнительных данных, в класс EventArgs включено статическое поле Empty, которое задает объект, не содержащий никаких данных.

Ниже приведен пример, в котором создается .NET-совместимое событие.

```
// A .NET-совместимое событие.

using System;

// Создаем класс, производный от класса EventArgs.
class MyEventArgs : EventArgs {
  public int eventnum;
}

// Объявляем делегат для события.
delegate void MyEventHandler(object source,
                             MyEventArgs arg);

// Объявляем класс события.
class MyEvent {
  static int count = 0;

  public event MyEventHandler SomeEvent;

  // Этот метод генерирует SomeEvent-событие.
  public void OnSomeEvent() {
    MyEventArgs arg = new MyEventArgs();

    if(SomeEvent != null) {
```

```
      arg.eventnum = count++;
      SomeEvent(this, arg);
    }
  }
}

class X {
  public void handler(object source, MyEventArgs arg) {
    Console.WriteLine("Событие " + arg.eventnum +
                      " получено объектом X.");
    Console.WriteLine("Источником является класс " +
                      source + ".");
    Console.WriteLine();
  }
}

class Y {
  public void handler(object source, MyEventArgs arg) {
    Console.WriteLine("Событие " + arg.eventnum +
                      " получено объектом Y.");
    Console.WriteLine("Источником является класс " +
                      source + ".");
    Console.WriteLine();
  }
}


class EventDemo {
  public static void Main() {
    X ob1 = new X();
    Y ob2 = new Y();
    MyEvent evt = new MyEvent();

    // Добавляем обработчик handler() в список событий.
    evt.SomeEvent += new MyEventHandler(ob1.handler);
    evt.SomeEvent += new MyEventHandler(ob2.handler);

    // Генерируем событие.
    evt.OnSomeEvent();
    evt.OnSomeEvent();
  }
}
```

Вот как выглядят результаты выполнения этой программы:

```
Событие 0 получено объектом X.
Источником является класс MyEvent.

Событие 0 получено объектом Y.
Источником является класс MyEvent.

Событие 1 получено объектом X.
Источником является класс MyEvent.

Событие 1 получено объектом Y.
Источником является класс MyEvent.
```

В этом примере класс `MyEventArgs` выводится из класса `EventArgs`. В классе `MyEventArgs` добавлено только одно "собственное" поле — `eventnum`. В соответствии

с требованиями .NET Framework делегат для обработчика событий MyEventHandler теперь принимает два параметра. Как разъяснялось выше, первый из них представляет собой объектную ссылку на генератор событий, а второй — ссылку на класс EventArgs или производный от класса EventArgs. В данном случае здесь используется ссылка на объект типа MyEventArgs.

Использование встроенного делегата EventHandler

Для многих событий параметр типа EventArgs не используется. Для упрощения процесса создания кода в таких ситуациях среда .NET Framework включает встроенный тип делегата, именуемый EventHandler. Его можно использовать для объявления обработчиков событий, которым не требуется дополнительная информация. Рассмотрим пример использования типа EventHandler.

```
// Использование встроенного делегата EventHandler.

using System;

// Объявляем класс события.
class MyEvent {
  public event EventHandler SomeEvent; // Объявление
                  // использует делегат EventHandler.

  // Этот метод вызывается для генерирования
  // SomeEvent-событие.
  public void OnSomeEvent() {
    if(SomeEvent != null)
      SomeEvent(this, EventArgs.Empty);
  }
}

class EventDemo {
  static void handler(object source, EventArgs arg) {
    Console.WriteLine("Событие произошло.");
    Console.WriteLine("Источником является класс " +
                      source + ".");
  }

  public static void Main() {
    MyEvent evt = new MyEvent();

    // Добавляем обработчик handler() в список событий.
    evt.SomeEvent += new EventHandler(handler);

    // Генерируем событие.
    evt.OnSomeEvent();
  }
}
```

В данном случае параметр типа EventArgs не используется и вместо него передается объект-заполнитель EventArgs.Empty. Результаты выполнения этой программы весьма лаконичны:

```
Событие произошло.
Источником является класс MyEvent.
```

Учебный проект: использование событий

События часто используются в таких средах с ориентацией на передачу сообщений, как Windows. В подобной среде программа просто ожидает до тех пор, пока не получит сообщение, а затем выполняет соответствующие действия. Такая архитектура прекрасно подходит для обработки событий в стиле языка C#, позволяя создавать обработчики событий для различных сообщений и просто вызывать обработчик при получении определенного сообщения. Например, с некоторым событием можно было бы связать сообщение, получаемое в результате щелчка левой кнопкой мыши. Тогда после щелчка левой кнопкой мыши все зарегистрированные обработчики будут уведомлены о приходе этого сообщения.

Несмотря на то что разработка Windows-программ, в которых демонстрируется такой подход, выходит за рамки этой главы, все же обрисуем в общих чертах работу этого механизма. В следующей программе создается обработчик событий нажатия клавиш. Событие называется `KeyPress`, и при каждом нажатии клавиши оно генерируется посредством вызова метода `OnKeyPress()`.

```
// Пример обработки события, связанного с нажатием
// клавиши на клавиатуре.

using System;

// Выводим собственный класс EventArgs, который
// будет хранить код клавиши.
class KeyEventArgs : EventArgs {
  public char ch;
}

// Объявляем делегат для события.
delegate void KeyHandler(object source, KeyEventArgs arg);

// Объявляем класс события, связанного с нажатием
// клавиши на клавиатуре.
class KeyEvent {
  public event KeyHandler KeyPress;

  // Этот метод вызывается при нажатии
  // какой-нибудь клавиши.
  public void OnKeyPress(char key) {
    KeyEventArgs k = new KeyEventArgs();

    if(KeyPress != null) {
      k.ch = key;
      KeyPress(this, k);
    }
  }
}

// Класс, который принимает уведомления о нажатии клавиши.
class ProcessKey {
  public void keyhandler(object source, KeyEventArgs arg) {
    Console.WriteLine(
        "Получено сообщение о нажатии клавиши: " + arg.ch);
  }
}
```

```
// Еще один класс, который принимает уведомления
// о нажатии клавиши.
class CountKeys {
  public int count = 0;

  public void keyhandler(object source, KeyEventArgs arg) {
    count++;
  }
}

// Демонстрируем использование класса KeyEvent.
class KeyEventDemo {
  public static void Main() {
    KeyEvent kevt = new KeyEvent();
    ProcessKey pk = new ProcessKey();
    CountKeys ck = new CountKeys();
    char ch;

    kevt.KeyPress += new KeyHandler(pk.keyhandler);
    kevt.KeyPress += new KeyHandler(ck.keyhandler);

    Console.WriteLine("Введите несколько символов. " +
                      "Для останова введите точку.");
    do {
      ch = (char) Console.Read();
      kevt.OnKeyPress(ch);
    } while(ch != '.');
    Console.WriteLine("Было нажато " +
                      ck.count + " клавиш.");
  }
}
```

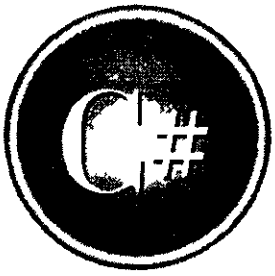
При выполнении этой программы можно получить такие результаты:

```
Введите несколько символов. Для останова введите точку.
тест.
Получено сообщение о нажатии клавиши: т
Получено сообщение о нажатии клавиши: е
Получено сообщение о нажатии клавиши: с
Получено сообщение о нажатии клавиши: т
Получено сообщение о нажатии клавиши: .
Было нажато 5 клавиш.
```

Эта программа начинается с выведения класса `KeyEventArgs`, который используется для передачи сообщения о нажатии клавиши обработчику событий. Затем делегат `KeyHandler` определяет обработчик для событий, связанных с нажатием клавиши на клавиатуре. Эти события инкапсулируются в классе `KeyEvent`.

Программа для обработки нажатий клавиш создает два класса: `ProcessKey` и `CountKeys`. Класс `ProcessKey` включает обработчик с именем `keyhandler()`, который отображает сообщение о нажатии клавиши. Класс `CountKeys` предназначен для хранения текущего количества нажатых клавиш. В методе `Main()` создается объект класса `KeyEvent`. Затем создаются объекты классов `ProcessKey` и `CountKeys`, а ссылки на их методы `keyhandler()` добавляются в список вызовов, реализуемый с помощью событийного объекта `kevt.KeyPress`. Затем начинает работать цикл, в котором при каждом нажатии клавиши вызывается метод `kevt.OnKeyPress()`, в результате чего зарегистрированные обработчики уведомляются о событии.

Полный справочник по



Глава 16

Пространства имен, препроцессор и компоновочные файлы

В этой главе рассматриваются три C#-средства, которые позволяют влиять на организацию и доступность программы. Речь пойдет о пространствах имен, препроцессоре и компоновочных файлах.

Пространства имен

О пространствах имен кратко упоминалось в главе 2, поскольку это одно из основополагающих понятий C#: каждая C#-программа так или иначе использует некоторое пространство имен. До сих пор мы не затрагивали эту тему, поскольку C# автоматически предоставляет программе пространство имен по умолчанию. Таким образом, программы, приведенные в предыдущих главах, просто использовали стандартное пространство имен. Но реальным программам придется создавать собственные или взаимодействовать с другими пространствами имен. Поэтому настало время поговорить о них более подробно.

Пространство имен определяет декларативную область, которая позволяет отдельно хранить множества имен. По существу, имена, объявленные в одном пространстве имен, не будут конфликтовать с такими же именами, объявленными в другом. Библиотека .NET Framework (которая является C#-библиотекой) использует пространство имен `System`. Поэтому в начало каждой программы мы включали следующую инструкцию:

```
using System;
```

Как было показано в главе 14, классы ввода-вывода определяются внутри пространства имен, подчиненного `System`, и именуемого `System.IO`. Существуют и другие пространства имен, подчиненные `System`, которые включают иные части C#-библиотеки.

Возникновение пространств имен продиктовано самой жизнью, поскольку в течение последних лет для программирования характерен взрывоподобный рост количества имен переменных, методов, свойств и классов, которые используются в библиотечных процедурах, приложениях сторонних изготовителей ПО и программах, написанных отдельными программистами. Без использования пространств имен все эти имена боролись бы за место "под солнцем" в глобальном пространстве имен, что привело бы к росту числа конфликтов. Например, если в программе определяется класс `Finder`, это имя обязательно будет конфликтовать с именем другого класса, `Finder` из библиотеки стороннего приложения, которое использует ваша программа. К счастью, благодаря пространствам имен, проблем такого рода можно избежать, поскольку пространство имен локализует видимость имен, объявленных внутри него.

Объявление пространства имен

Пространство имен объявляется с помощью ключевого слова `namespace`. Общая форма объявления пространства имен имеет следующий вид:

```
namespace имя {
  // Члены
}
```

Здесь элемент *имя* означает имя пространства имен. Все, что определено внутри пространства имен, находится в пределах его области видимости. Следовательно, пространство имен определяет область видимости. Внутри пространства имен можно объявлять классы, структуры, делегаты, перечисления, интерфейсы или другое пространство имен.

Рассмотрим пример использования ключевого слова namespace, которое создает пространство имен Counter. Оно ограничивает распространение имени, используемого для реализации класса обратного счета, именуемого CountDown.

```
// Объявление пространства имен для счетчиков.

namespace Counter {
  // Простой счетчик для счета в обратном направлении.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
   }

    public void reset(int n) {
      val = n;
    }

    public int count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}
```

Здесь класс CountDown объявляется внутри области видимости, определенной пространством имен Counter.

А теперь рассмотрим программу, которая демонстрирует использование пространства имен Counter.

```
// Демонстрация использования пространства имен.

using System;

// Объявляем пространство имен для счетчиков.
namespace Counter {
  // Простой счетчик для счета в обратном направлении.
  class CountDown {
    int val;

    public CountDown(int n) { val = n; }

    public void reset(int n) {
      val = n;
    }
    public int count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

class NSDemo {
  public static void Main() {
    Counter.CountDown cd1 = new Counter.CountDown(10);
    int i;

    do {
```

```
      i = cd1.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    Counter.CountDown cd2 = new Counter.CountDown(20);

    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.reset(4);
    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
  }
}
```

Вот результаты выполнения этой программы:

```
10 9 8 7 6 5 4 3 2 1 0
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
4 3 2 1 0
```

Здесь имеет смысл обратить ваше внимание вот на что. Во-первых, поскольку класс `CountDown` объявляется внутри пространства имен `Counter`, то при создании объекта класса `CountDown`, как показано в следующей инструкции, имя класса необходимо указывать вместе с именем пространства имен `Counter`.

```
Counter.CountDown cd1 = new Counter.CountDown(10);
```

Но если объект `Counter` уже создан, то в дальнейшем называть его (или любой из его членов) полностью (по "имени-отчеству") необязательно. Таким образом, метод `cd1.count()` можно вызывать без указания имени пространства имен, как показано в этой строке кода:

```
i = cd1.count();
```

Пространства имен предотвращают конфликты по совпадению имен

Основное преимущество использования пространств имен состоит в том, что имена, объявленные внутри одного из них, не конфликтуют с такими же именами, объявленными вне его. Например, в следующей программе создается еще один класс `CountDown`, но он находится в пространстве имен `Counter2`.

```
// Пространства имен предотвращают конфликты,
// связанные с совпадением имен.

using System;

// Объявляем пространство имен для счетчиков.
namespace Counter {
  // Простой счетчик для счета в обратном направлении.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
```

```
    }

    public void reset(int n) {
      val = n;
    }

    public int count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

// Объявляем еще одно пространство имен.
namespace Counter2 {
  /* Этот класс CountDown находится в пространстве
     имен Counter2 и не будет конфликтовать с одноименным
     классом, определенным в пространстве имен Counter. */
  class CountDown {
    public void count() {
      Console.WriteLine("Этот метод count() находится в" +
                        " пространстве имен Counter2.");
    }
  }
}

class NSDemo {
  public static void Main() {
    // Этот класс CountDown находится в
    // пространстве имен Counter.
    Counter.CountDown cd1 = new Counter.CountDown(10);

    // Этот класс CountDown находится в
    // пространстве имен Counter2.
    Counter2.CountDown cd2 = new Counter2.CountDown();

    int i;

    do {
      i = cd1.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.count();
  }
}
```

Результаты выполнения этой программы имеют такой вид:

```
10 9 8 7 6 5 4 3 2 1 0
Этот метод count() находится в пространстве имен Counter2.
```

Как подтверждают результаты выполнения этой программы, класс `CountDown` внутри пространства имен `Counter` отделен от класса `CountDown`, определенного в пространстве имен `Counter2`, и поэтому имена не конфликтуют. Хотя этот пример очень простой, он позволяет понять, как избежать конфликтов при совпадении имен между своим кодом и кодом, написанным другими, поместив собственные классы в определенное пространство имен.

Ключевое слово `using`

Как разъяснялось в главе 2, если программа включает часто встречающиеся ссылки на определенные члены пространства имен, то необходимость указывать имя этого пространства имен при каждом к ним обращении, очень скоро утомит вас. Эту проблему позволяет решить директива `using`. В примерах этой книги использовалась директива `using`, чтобы сделать текущим C#-пространство имен `System`, поэтому вы уже с ним знакомы. Нетрудно предположить, что директиву `using` можно также использовать для объявления действующими пространств имен, создаваемых программистом.

Существует две формы применения директивы `using`. Первая имеет такой вид:

```
using имя;
```

Здесь элемент *имя* означает имя пространства имен, к которому необходимо получить доступ. С этой формой директивы `using` вы уже знакомы. Все члены, определенные внутри заданного пространства имен, становятся частью этого (текущего) пространства имен, поэтому их можно использовать без дополнительного упоминания его имени. Директива `using` должна находиться в начале каждого программного файла, т.е. предшествовать всем остальным объявлениям.

В следующей программе переработан пример использования счетчиков из предыдущего раздела, чтобы показать, как с помощью директивы `using` можно чтобы сделать текущим создаваемое программистом пространство имен.

```
// Демонстрация использования пространства имен.

using System;

// Делаем текущим пространство имен Counter.
using Counter;

// Объявляем пространство имен для счетчиков.
namespace Counter {
  // Простой счетчик для счета в обратном направлении.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    public void reset(int n) {
      val = n;
    }

    public int count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

class NSDemo {
  public static void Main() {
    // Теперь класс CountDown можно использовать
    // без указания имени пространства имен.
    CountDown cd1 = new CountDown(10);
```

```
    int i;

    do {
      i = cd1.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    CountDown cd2 = new CountDown(20);

    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.reset(4);
    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
  }
}
```

Эта программа иллюстрирует еще один важный момент: использование одного пространства имен не аннулирует другое. При объявлении действующим некоторого пространства имен его имя просто добавляется к именам других, которые действуют в данный момент. Следовательно, в этой программе действуют пространства имен `System` и `Counter`.

Вторая форма использования директивы `using`

Директива `using` обладает еще одной формой применения:

```
using псевдоимя = имя;
```

Здесь элемент *псевдоимя* задает еще одно имя для класса или пространства имен, заданного элементом *имя*. Теперь программу счета в обратном порядке переделаем еще раз, чтобы показать, как создается *псевдоимя* `Count` для составного имени `Counter.CountDown`.

```
// Демонстрация использования псевдоимени.

using System;

// Создаем псевдоимя для класса Counter.CountDown.
using Count = Counter.CountDown;
// Объявляем пространство имен для счетчиков.
namespace Counter {
  // Простой счетчик для счета в обратном направлении.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    public void reset(int n) {
```

```
      val = n;
    }

    public int count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

class NSDemo {
  public static void Main() {
    // Здесь Count используется в качестве имени
    // вместо Counter.CountDown.
    Count cd1 = new Count(10);
    int i;

    do {
      i = cd1.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    Count cd2 = new Count(20);

    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.reset(4);
    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
  }
}
```

После того как имя `Count` было определено в качестве еще одного имени для составного имени `Counter.CountDown`, его можно использовать для объявления объектов класса `CountDown` без уточняющего указания пространства имен. Например, в нашей программе при выполнении строки

```
Count cd1 = new Count(10);
```

создается объект класса `CountDown`.

Аддитивность пространств имен

В одной программе одно и то же пространство имен можно объявить больше одного раза. Это позволяет распределить его по нескольким файлам или даже разделить его внутри одного файла. Например, в следующей программе определяется два пространства имен `Counter`. Одно содержит класс `CountDown`, второе — класс `CountUp`. При компиляции содержимое двух пространств имен `Counter` объединяется в одно.

```
// Демонстрация аддитивности пространств имен.

using System;
```

```
// Делаем "видимым" пространство имен Counter.
using Counter;

// Теперь действующим является первое пространство
// имен Counter.
namespace Counter {
  // Простой счетчик для счета в обратном направлении.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    public void reset(int n) {
      val = n;
    }

    public int count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

// Теперь действующим является второе пространство
// имен Counter.
namespace Counter {
  // Простой счетчик для счета в прямом направлении.
  class CountUp {
    int val;
    int target;

    public int Target {
      get{
        return target;
      }
    }

    public CountUp(int n) {
      target = n;
      val = 0;
    }

    public void reset(int n) {
      target = n;
      val = 0;
    }

    public int count() {
      if(val < target) return val++;
      else return target;
    }
  }
}

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

```
    CountDown cd = new CountDown(10);
    CountUp cu = new CountUp(8);
    int i;

    do {
      i = cd.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    do {
      i = cu.count();
      Console.Write(i + " ");
    } while(i < cu.Target);

  }
}
```

При выполнении этой программы получаем следующие результаты:

```
10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8
```

Хотелось бы обратить ваше внимание вот на что. Инструкция

```
using Counter;
```

делает "видимым" содержимое обоих пространств имен. Поэтому к методам `CountDown` и `CountUp` можно обращаться напрямую, без уточняющей информации о пространстве имен. И тот факт, что пространство имен `Counter` было разделено на две части, не имеет никакого значения.

Пространства имен могут быть вложенными

Одно пространство имен можно вложить в другое. Рассмотрим следующую программу:

```
// Демонстрация вложенных пространств имен.

using System;

namespace NS1 {
  class ClassA {
     public ClassA() {
       Console.WriteLine("Создание класса ClassA.");
     }
  }
  namespace NS2 { // Вложенное пространство имен.
    class ClassB {
       public ClassB() {
         Console.WriteLine("Создание класса ClassB.");
       }
    }
  }
}

class NestedNSDemo {
  public static void Main() {
    NS1.ClassA a= new NS1.ClassA();

  // NS2.ClassB b = new NS2.ClassB(); // Ошибка!!!
```

```
    // Пространство имен NS2 не находится в зоне видимости.

    NS1.NS2.ClassB b = new NS1.NS2.ClassB(); // Здесь все
                                             // правильно.
  }
}
```

Вот результаты выполнения этой программы:

```
Создание класса ClassA.
Создание класса ClassB.
```

В этой программе пространство имен `NS2` вложено в пространство имен `NS1`. Следовательно, при обращении к классу `ClassB` его имя необходимо уточнять, указывая оба пространства имен: как `NS1`, так и `NS2`. Одного лишь имени `NS2` недостаточно. Как видно в программе, имена пространств имен разделяются точкой.

Вложенные пространства имен можно задавать с помощью одной инструкции, но разделив их точками. Например, задание вложенных пространств имен

```
namespace OuterNS {
  namespace InnerNS {
    // ...
  }
}
```

можно записать в таком виде:

```
namespace OuterNS.InnerNS {
  // ...
}
```

Пространство имен по умолчанию

Если для программы не объявлено пространство имен, используется пространство имен, действующее по умолчанию. Вот почему необязательно было указывать его для программ, приведенных в предыдущих главах. Но если для коротких простых программ (подобных тем, что приведены в этой книге) такой стандартный подход вполне приемлем (и даже удобен), большинство реальных программ содержится внутри некоторого пространства имен. Главная причина инкапсуляции программного кода внутри пространства имен состоит в предотвращении конфликтов при совпадении имен. Пространства имен — это еще один инструмент, позволяющий программисту так организовать свои программы, чтобы они не теряли жизнеспособности в сложной среде с сетевой структурой.

Препроцессор

В C# определен ряд директив препроцессора, которые влияют на способ интерпретации исходного кода компилятором. Эти директивы обрабатывают текст исходного файла, в котором они находятся, еще до трансляции программы в объектный код. Директивы препроцессора — в основном "выходцы" из C++, поскольку препроцессор C# очень похож на тот, который определен в языке C++. Термин *директива препроцессора* своим происхождением обязан тому факту, что эти инструкции традиционно обрабатывались на отдельном этапе компиляции, именуемой *процессором предварительной обработки*, или *препроцессором* (preprocessor). Современная технология компиляторов больше не требует отдельного этапа для обработки директив препроцессором, но название осталось.

В C# определены следующие директивы препроцессора:

```
#define      #elif    #else      #endif
#endregion   #error   #if        #line
#region      #undef   #warning
```

Все директивы препроцессора начинаются со знака "#". Кроме того, каждая директива препроцессора должна занимать отдельную строку.

Откровенно говоря, поскольку в C# использована современная объектно-ориентированная архитектура, в директивах препроцессора программисты не испытывают острой необходимости, как это было в языках программирования более ранних поколений. Тем не менее время от времени они могут быть полезными, особенно для условной компиляции. Рассмотрим все перечисленные выше директивы.

#define

Директива `#define` определяет последовательность символов, именуемую *идентификатором*. С помощью директив `#if` или `#elif` можно определить наличие или отсутствие в программе идентификатора, а результат такой проверки используется для управления компиляцией. Общая форма записи директивы `#define` такова:

```
#define идентификатор
```

Обратите внимание на то, что в инструкции нет завершающей точки с запятой. Между самой директивой `#define` и идентификатором может стоять любое количество пробелов, но завершить идентификатор можно только символом новой строки. Например, чтобы определить идентификатор `EXPERIMENTAL`, используйте следующую директиву:

```
#define EXPERIMENTAL
```

На заметку

В C/C++ директиву `#define` можно использовать для выполнения текстовых подстановок, определяя для заданного значения осмысленное имя, а также для создания макросов, действующих подобно функциям. Такое использование директивы `#define` C# не поддерживает. В C# директива `#define` используется только для определения идентификатора.

#if и #endif

Директивы `#if` и `#endif` позволяют выполнить условную компиляцию последовательности инструкций программного кода в зависимости от того, истинно ли выражение, включающее одно или несколько идентификаторов. Истинным считается идентификатор, определенный в программе. В противном случае он считается ложным. Следовательно, если символ определен с помощью директивы `#define`, он оценивается как истинный.

Общая форма использования директивы `#if` такова:

```
#if символьное_выражение
  последовательность_инструкций
#endif
```

Если выражение, стоящее после директивы `#if` (*символьное_выражение*), истинно, код, расположенный между нею и директивой `#endif` (*последовательность_инструкций*), компилируется. В противном случае он опускается. Директива `#endif` означает конец `#if`-блока.

Символьное выражение может состоять из одного идентификатора. Более сложное выражение можно образовать с помощью следующих операторов: `!`, `==`, `!=`, `&&` и `||`. Разрешено также использовать круглые скобки.

Рассмотрим пример использования директив `#if`, `#endif` и `#define`.

```
// Демонстрация использования директив #if, #endif
// и #define.

#define EXPERIMENTAL

using System;

class Test {
  public static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine(
          "Компилируется для экспериментальной версии.");
    #endif

    Console.WriteLine(
         "Эта информация отображается во всех версиях.");
  }
}
```

При выполнении программы отображаются следующие результаты:

```
Компилируется для экспериментальной версии.
Эта информация отображается во всех версиях.
```

В этой программе с помощью директивы `#define` определяется идентификатор `EXPERIMENTAL`. Поэтому при использовании директивы `#if` символьное выражение `EXPERIMENTAL` оценивается как истинное, и компилируется первая (из двух) `WriteLine()`-инструкция. Если удалить определение идентификатора `EXPERIMENTAL` и перекомпилировать программу, первая `WriteLine()`-инструкция не скомпилируется, поскольку результат выполнения директивы `#if` будет оценен как ложный. Вторая `WriteLine()`-инструкция скомпилируется обязательно, поскольку она не является частью `#if`-блока. Как упоминалось выше, в директиве `#if` можно использовать символьное выражение. Вот пример:

```
// Использование символьного выражения.

#define EXPERIMENTAL
#define TRIAL

using System;

class Test {
  public static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine(
          "Компилируется для экспериментальной версии.");
    #endif

    #if EXPERIMENTAL && TRIAL
       Console.Error.WriteLine(
         "Тестирование экспериментальной пробной версии.");
    #endif
```

```
    Console.WriteLine(
         "Эта информация отображается во всех версиях.");
  }
}
```

Вот результаты выполнения этой программы:

```
Компилируется для экспериментальной версии.
Тестирование экспериментальной пробной версии.
Эта информация отображается во всех версиях.
```

В этом примере определяются два идентификатора, `EXPERIMENTAL` и `TRIAL`. Вторая `WriteLine()`-инструкция компилируется только в случае, если определены оба идентификатора.

#else и #elif

Директива `#else` работает подобно `else`-инструкции в языке C#, т.е. она предлагает альтернативу на случай, если директива `#if` выявит ложный результат. Следующий пример представляет собой расширенный вариант предыдущего.

```
// Демонстрация использования директивы #else.

#define EXPERIMENTAL

using System;

class Test {
  public static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine(
          "Компилируется для экспериментальной версии.");
    #else
      Console.WriteLine("Компилируется для бета-версии.");
    #endif

    #if EXPERIMENTAL && TRIAL
       Console.Error.WriteLine(
         "Тестирование экспериментальной пробной версии.");
    #else
       Console.Error.WriteLine(
              "Это не экспериментальная пробная версия.");
    #endif

    Console.WriteLine(
          "Эта информация отображается во всех версиях.");
  }
}
```

При выполнении этой программы получены такие результаты:

```
Компилируется для экспериментальной версии.
Это не экспериментальная пробная версия.
Эта информация отображается во всех версиях.
```

Поскольку идентификатор `TRIAL` не определен, компилируется `#else`-блок второй условной последовательности инструкций.

Обратите внимание на то, что директива `#else` отмечает одновременно как конец `#if`-блока, так и начало `#else`-блока, поскольку с любой директивой `#if` может быть связана только одна директива `#endif`.

Директива `#elif` означает "иначе если" и используется в `if-else-if`-цепочках многовариантной компиляции. С директивой `#elif` связано символьное выражение. Если оно истинно, следующий за ним блок кода (*последовательность_инструкций*) компилируется, и другие `#elif`-выражения не проверяются. В противном случае тестируется следующий `#elif`-блок. Общая форма цепочки `#elif`-блоков имеет следующий вид:

```
#if символьное_выражение
  последовательность_инструкций
#elif символьное_выражение
 последовательность_инструкций
#elif символьное_выражение
 последовательность_инструкций
#elif символьное_выражение
 последовательность_инструкций
#elif символьное_выражение
.
.
.
#endif
```

Рассмотрим пример:

```
// Демонстрация использования директивы #elif.

#define RELEASE

using System;

class Test {
  public static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine(
           "Компилируется для экспериментальной версии.");
    #elif RELEASE
      Console.WriteLine("Компилируется для бета-версии.");
    #else
      Console.WriteLine(
           "Компилируется для внутреннего тестирования.");
    #endif

    #if TRIAL && !RELEASE
       Console.WriteLine("Пробная версия.");
    #endif

    Console.WriteLine(
          "Эта информация отображается во всех версиях.");
  }
}
```

Результаты выполнения этой программы выглядят так:

```
Компилируется для бета-версии.
Эта информация отображается во всех версиях.
```

#undef

Директива `#undef` аннулирует приведенное выше определение идентификатора, который указан после директивы. Общая форма директивы `#undef` имеет следующий вид:

```
#undef идентификатор
```

Рассмотрим пример:

```
#define SMALL

#if SMALL
  // ...
#undef SMALL
// Здесь идентификатор SMALL уже не определен.
```

После выполнения директивы `#undef` идентификатор `SMALL` больше не считается определенным.

Директива `#undef` используется главным образом для того, чтобы разрешить локализацию идентификатора только в пределах нужных разделов кода.

#error

Директива `#error` вынуждает компилятор прекратить компиляцию. Она используется в целях отладки.

Общая форма директивы `#error` имеет следующий вид:

`#error` *сообщение_об_ошибке*

При использовании директивы `#error` отображается заданное *сообщение_об_ошибке*. Например, при обработке компилятором строки

```
#error Это тестовая ошибка!
```

процесс компиляции будет остановлен, а на экране появится сообщение "Это тестовая ошибка!".

#warning

Директива `#warning` подобна директиве `#error`, но она не извещает об ошибке, а содержит предупреждение. Процесс компиляции при этом не останавливается. Общая форма директивы `#warning` имеет следующий вид:

`#warning` *предупреждающее_сообщение*

#line

Директива `#line` устанавливает номер строки и имя файла, который содержит директиву `#line`. Номер строки и имя файла используются во время компиляции при выводе сообщений об ошибках или предупреждений. Общая форма записи директивы `#line` выглядит так:

`#line` *номер "имя_файла"*

Здесь элемент *номер* представляет собой любое положительное целое число, которое станет новым номером строки, а необязательный элемент *имя_файла* — любой допустимый идентификатор файла, который станет новым именем файла. Директива `#line` в основном используется при отладке и в специальных приложениях.

Чтобы вернуть нумерацию строк в исходное состояние, используйте ключевое слово `default`:

```
#line default
```

#region и #endregion

Директивы `#region` и `#endregion` позволяют определить область, которую можно будет разворачивать или сворачивать при использовании интегрированной среды разработки (IDE) Visual Studio. Вот общая форма использования этих директив:

```
#region имя_области
  // последовательность_инструкций
#endregion
```

Нетрудно догадаться, что элемент *имя_области* означает имя области.

Компоновочные файлы и модификатор доступа `internal`

Неотъемлемой частью C#-программирования является компоновочный файл (assembly), который содержит информацию о развертывании программы и ее версии. Компоновочные файлы имеют важное значение для .NET-среды. Согласно документации Microsoft, "компоновочные файлы являются строительными блоками среды .NET Framework." Компоновочные файлы поддерживают механизм безопасного взаимодействия компонентов, межъязыковой работоспособности и управления версиями. Компоновочный файл также определяет область видимости.

Компоновочный файл состоит из четырех разделов. Первый представляет собой *декларацию* (manifest). Декларация содержит информацию о компоновочном файле. Сюда относятся такие данные, как имя компоновочного файла, номер его версии, информация о соответствии типов и параметры "культурного уровня". Второй раздел включает *метаданные*, или информацию о типах данных, используемых в программе. В числе прочих достоинств метаданных — обеспечение взаимодействия программ, написанных на различных языках программирования. Третий раздел компоновочного файла содержит программный код, который хранится в формате Microsoft Intermediate Language (MSIL). Наконец, четвертый раздел представляет собой ресурсы, используемые программой.

К счастью, при использовании языка C# компоновочные файлы создаются автоматически, без дополнительных (или с минимальными) усилий со стороны программиста. Дело в том, что выполняемый файл, создаваемый в результате компиляции C#-программы, в действительности является компоновочным файлом, который содержит выполняемый код программы и другую информацию. Следовательно, при компиляции C#-программы автоматически создается компоновочный файл.

Подробное рассмотрение компоновочных файлов выходит за рамки этой книги. (Компоновочные файлы — неотъемлемая часть .NET-разработки, а не средство языка C#.) Но одна часть языка C# напрямую связана с компоновочным файлом: модификатор доступа `internal`. Вот о нем-то и пойдет речь в следующем разделе.

Модификатор доступа `internal`

Помимо модификаторов доступа `public`, `private` и `protected`, с которыми вы уже встречались в этой книге, в C# также определен модификатор `internal`. Его назначение — заявить о том, что некоторый член известен во всех файлах, входящих в состав компоновочного, но неизвестен вне его. Проще говоря, член, отмеченный модификатором `internal`, известен только программе, но не где-то еще. Модификатор доступа `internal` чрезвычайно полезен при создании программных компонентов.

Модификатор `internal` можно применить к классам и членам классов, а также к структурам и членам структур. Модификатор `internal` можно также применить к объявлениям интерфейсов и перечислений.

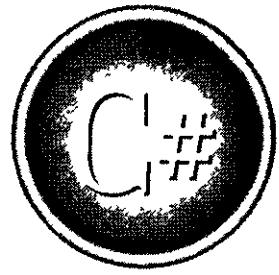
Совместно с модификатором `internal` можно использовать модификатор `protected`. В результате будет установлен уровень доступа `protected internal`, который можно применять только к членам класса. К члену, объявленному с использованием пары модификаторов `protected internal`, можно получить доступ внутри его компоновочного файла. Он также доступен для производных типов.

Рассмотрим пример использования модификатора доступа `internal`.

```
// Использование модификатора доступа internal.

using System;

class InternalTest {
  internal int x;
}

class InternalDemo {
  public static void Main() {
    InternalTest ob = new InternalTest();

    ob.x = 10; // Доступ возможен: x -- в том же файле.

    Console.WriteLine("Значение ob.x: " + ob.x);

  }
}
```

Внутри класса `InternalTest` поле `x` объявлено с использованием модификатора доступа `internal`. Это означает, что оно доступно в программе, как показано в коде класса `InternalDemo`, но недоступно вне ее.

Полный справочник по



Глава 17

Динамическая идентификация типов, отражение и атрибуты

Эта глава посвящена трем взаимосвязанным мощным средствам C#: динамической идентификации типов, отражению и атрибутам. *Динамическая идентификация типов* — это механизм, который позволяет распознать тип данных во время выполнения программы. *Отражение* представляет собой средство, с помощью которого можно получить информацию о типе. Используя эту информацию, во время выполнения программы можно создавать объекты, а затем работать с ними. Это средство обладает большой эффективностью, поскольку оно позволяет динамически расширять функции, выполняемые программой. *Атрибут* предназначен для описания элементов C#-программы. Например, можно определить атрибуты для классов, методов и полей. Информацию об атрибутах можно запрашивать и получать во время выполнения программы. Для поддержки атрибутов используются средства как динамической идентификации типов, так и отражения соответствующей информации.

Динамическая идентификация типов

Динамическая идентификация типов (runtime type identification — RTTI) позволяет определить тип объекта во время выполнения программы, что необходимо во многих ситуациях. Например, можно совершенно точно узнать, на объект какого типа в действительности указывает ссылка на базовый класс. Еще одно применение RTTI — заранее проверить, удачно ли будет выполнена операция приведения типа, не допустив возникновения исключения, связанного с некорректно заданной операцией приведения типа. Динамическая идентификация типов также является ключевым компонентом средства отражения (информации о типе).

В C# предусмотрено три ключевых слова, которые поддерживают динамическую идентификацию типов: `is`, `as` и `typeof`. Рассмотрим назначение каждого из них в отдельности.

Проверка типа с помощью ключевого слова `is`

С помощью оператора `is` можно определить, имеет ли рассматриваемый объект заданный тип. Общая форма его записи имеет следующий вид:

выражение `is` *тип*

Здесь тип элемента *выражение* сравнивается с элементом *тип*. Если тип элемента *выражение* совпадает (или совместим) с элементом *тип*, результат выполнения операции принимается равным значению ИСТИНА. В противном случае — значению ЛОЖЬ. Следовательно, если результат истинен, *выражение* можно привести к типу, заданному элементом *тип*.

Рассмотрим пример использования оператора `is`.

```
// Демонстрация выполнения оператора is.

using System;

class A {}
class B : A {}

class UseIs {
  public static void Main() {
    A a = new A();
    B b = new B();
```

```
    if(a is A) Console.WriteLine("Объект a имеет тип A.");
    if(b is A)
      Console.WriteLine("Объект b совместим с типом A, " +
                        "поскольку его тип выведен из типа A.");
    if(a is B)
      Console.WriteLine("Этот текст не будет отображен, " +
                "поскольку объект a не выведен из класса B.");

    if(b is B) Console.WriteLine("Объект b имеет тип B.");
    if(a is object) Console.WriteLine("a -- это объект.");
  }
}
```

Результаты выполнения этой программы таковы:

```
Объект a имеет тип A.
Объект b совместим с типом A, поскольку его тип выведен из типа A.
Объект b имеет тип B.
a -- это объект.
```

Несмотря на то что все сообщения в этой программе говорят сами за себя, некоторые из них все же требуют пояснений. Обратите внимание на следующую инструкцию:

```
if(b is A)
      Console.WriteLine("Объект b совместим с типом A, " +
                        "поскольку его тип выведен из типа A.");
```

В данном случае `if`-инструкция выполнилась успешно, поскольку переменная `b` является ссылкой типа `B`, который выведен из типа `A`. Следовательно, объект `b` совместим с типом `A`. Однако обратное утверждение не является справедливым. При выполнении строки кода

```
if(a is B)
      Console.WriteLine("Этот текст не будет отображен, " +
                "поскольку объект a не выведен из класса B.");
```

`if`-инструкция успехом не увенчается, поскольку объект `a` имеет тип `A`, который не выведен из типа `B`. Следовательно, объект `a` и класс `B` несовместимы по типу.

Использование оператора `as`

Иногда во время работы программы требуется выполнить операцию приведения типов, не генерируя исключение в случае, если попытка окажется неудачной. Для этого предусмотрен оператор `as`, формат которого таков:

выражение `as` *тип*

Нетрудно догадаться, что используемый здесь элемент *выражение* участвует в попытке приведения его к типу, заданному элементом *тип*. В случае успешного выполнения этой операции возвращается ссылка на *тип*. В противном случае возвращается нулевая ссылка.

Оператор `as` в некоторых случаях предлагает удобную альтернативу оператору `is`. Рассмотрим, например, следующую программу, в которой оператор `is` позволяет предотвратить неверное приведение типов:

```
// Использование оператора is для предотвращения
// неверной операции приведения типов.

using System;

class A {}
class B : A {}
```

```
class CheckCast {
  public static void Main() {
    A a = new A();
    B b = new B();

    // Проверяем, можно ли объект a привести к типу B.
    if(a is B) // При положительном результате выполняем
               // операцию приведения типов.
      b = (B) a;
    else // В противном случае операция приведения
         // типов опускается.
      b = null;

    if(b==null)
      Console.WriteLine(
       "Операция приведения типов b = (B) a НЕ РАЗРЕШЕНА.");
    else
      Console.WriteLine(
           "Операция приведения типов b = (B) a разрешена.");
  }
}
```

Результаты выполнения этой программы таковы:

```
Операция приведения b = (B) a НЕ РАЗРЕШЕНА.
```

Как подтверждают эти результаты, поскольку тип объекта `a` не совместим с типом `B`, операция приведения объекта `a` к типу `B` недопустима, и ее выполнение предотвращается с помощью инструкции `if`. Как видите, реализация такого подхода требует выполнения двух этапов. Первый состоит в подтверждении обоснованности операции приведения типов, а второй — в самом ее выполнении. С помощью оператора `as` эти два этапа можно объединить в один, как показано в следующей программе.

```
// Демонстрация использования оператора as.

using System;

class A {}
class B : A {}

class CheckCast {
  public static void Main() {
    A a = new A();
    B b = new B();

    b = a as B; // Выполняем операцию приведения типов,
                // если она возможна.

    if(b==null)
      Console.WriteLine("Операция приведения типов " +
                        "b = (B) a НЕ РАЗРЕШЕНА.");
    else
      Console.WriteLine(
       "Операция приведения типов b = (B) a разрешена.");
  }
}
```

Вот результаты выполнения этой программы:

```
Операция приведения типов b = (B) a НЕ РАЗРЕШЕНА.
```

В этой версии оператор as проверяет допустимость операции приведения типов, а затем, если она законна, выполняет ее, причем все это реализуется в одной инструкции.

Использование оператора typeof

Несмотря на полезность операторов as и is, они просто проверяют (причем каждый по-своему) совместимость двух типов. Программист же зачастую сталкивается с необходимостью получить информацию о типе данных. Для таких случаев в C# предусмотрен оператор typeof. Его назначение состоит в считывании объекта класса System.Type для заданного типа. Используя этот объект, можно определить характеристики конкретного типа данных.

Оператор typeof используется в следующем формате:

```
typeof(тип)
```

Здесь элемент *тип* означает тип, информацию о котором мы хотим получить. Объект типа Type, возвращаемый при выполнении оператора typeof, инкапсулирует информацию, связанную с заданным типом.

Получив Type-объект, можно обращаться к информации о заданном типе, используя различные свойства, поля и методы, определенные в классе Type. Класс Type содержит множество членов, но их обсуждение мы отложим до следующего раздела, посвященного отражению информации о типе. Однако, чтобы все же продемонстрировать одно из возможных применений класса Type, рассмотрим программу, в которой используются три его свойства: FullName, IsClass и IsAbstract. Свойство FullName позволяет получить полное имя типа. Свойство IsClass возвращает значение true, если типом объекта является класс. Свойство IsAbstract возвращает значение true, если рассматриваемый класс является абстрактным.

```
// Демонстрация использования оператора typeof.

using System;
using System.IO;

class UseTypeof {
  public static void Main() {
    Type t = typeof(StreamReader);

    Console.WriteLine(t.FullName);

    if(t.IsClass) Console.WriteLine("Это класс.");
    if(t.IsAbstract) Console.WriteLine(
                              "Это абстрактный класс.");
    else Console.WriteLine("Это конкретный класс.");

  }
}
```

При выполнении этой программы получены такие результаты:

```
System.IO.StreamReader
Это класс.
Это конкретный класс.
```

Эта программа получает объект типа Type с описанием типа StreamReader. Затем она отображает полное имя этого типа, определяет, класс ли это и является ли он абстрактным.

Отражение

Как упоминалось в начале этой главы, *отражение* (reflection) — это средство C#, которое позволяет получить информацию о типе. Термин *отражение* произошел от характера процесса: объект класса `Type` воспроизводит, или отражает, базовый тип, который он представляет. Для получения интересующей вас информации вы "задаете вопросы" объекту класса `Type`, а он возвращает (отражает) для вас информацию, связанную с этим типом. Отражение — мощный механизм, позволяющий узнать характеристики типа, точное имя которого становится известным только во время выполнения программы, и соответствующим образом использовать их.

Многие классы, которые поддерживают средство отражения, являются частью интерфейса .NET Reflection API, который определен в пространстве имен `System.Reflection`. Таким образом, в программы, которые используют средство отражения, обычно включается следующая инструкция:

```
using System.Reflection;
```

Ядро подсистемы отображения: класс `System.Type`

Класс `System.Type` — "сердце" подсистемы отображения, поскольку он инкапсулирует тип. Он содержит множество свойств и методов, которые можно использовать для получения информации о типе во время выполнения программы. Класс `Type` выведен из абстрактного класса `System.Reflection.MemberInfo`.

В классе `MemberInfo` определены следующие абстрактные свойства, которые предназначены только для чтения:

`Type DeclaringType`	Тип класса или интерфейса, в котором объявляется анализируемый член
`MemberTypes MemberType`	Тип члена
`string Name`	Имя типа
`Type ReflectedType`	Тип отражаемого объекта

Обратите внимание на то, что свойство `MemberType` имеет тип `MemberTypes`. Свойство `MemberTypes` представляет собой перечисление, которое определяет значения, соответствующие различным типам членов. Среди прочих они включают следующие:

```
MemberTypes.Constructor
MemberTypes.Method
MemberTypes.Field
MemberTypes.Event
MemberTypes.Property
```

Следовательно, тип члена можно определить, опросив свойство `MemberType`. Например, если свойство `MemberType` содержит значение `MemberTypes.Method`, значит, интересующий нас член является методом.

Класс `MemberInfo` включает два абстрактных метода: `GetCustomAttributes()` и `IsDefined()`. Оба они связаны с атрибутами.

К методам и свойствам, определенным в классе `MemberInfo`, класс `Type` добавляет немало "своих". Ниже перечислены наиболее часто используемые методы, определенные в классе `Type`.

Метод	Назначение
`ConstructorInfo[] GetConstructors()`	Получает список конструкторов для заданного типа
`EventInfo[] GetEvents()`	Получает список событий для заданного типа
`FieldInfo[] GetFields()`	Получает список полей для заданного типа
`MemberInfo[] GetMembers()`	Получает список членов для заданного типа
`MethodInfo[] GetMethods()`	Получает список методов для заданного типа
`PropertyInfo[] GetProperties()`	Получает список свойств для заданного типа

Теперь ознакомьтесь с наиболее часто используемыми свойствами, определенными в классе `Type`.

Свойство	Назначение
`Assembly Assembly`	Получает компоновочный файл для заданного типа
`TypeAttributes Attributes`	Получает атрибуты для заданного типа
`Type BaseType`	Получает непосредственный базовый тип для заданного типа
`string FullName`	Получает полное имя заданного типа
`bool IsAbstract`	Истинно, если заданный тип является абстрактным
`bool IsArray`	Истинно, если заданный тип является массивом
`bool IsClass`	Истинно, если заданный тип является классом
`bool IsEnum`	Истинно, если заданный тип является перечислением
`string Namespace`	Получает пространство имен для заданного типа

Использование отражения

Используя методы и свойства класса `Type`, во время выполнения программы можно получить подробную информацию о типе. Это чрезвычайно мощное средство, поскольку, получив информацию о типе, можно вызвать конструкторы этого типа, его методы и использовать его свойства. Таким образом, отражение позволяет использовать код, который недоступен в период компиляции программы.

Программный интерфейс Reflection API —довольно обширная тема, и здесь невозможно рассмотреть ее в полном объеме. (Полное описание Reflection API — это материал для целой книги!) Но интерфейс Reflection API столь логичен, что, поняв, как использовать одну его часть, нетрудно разобраться во всем остальном. В следующих разделах описаны ключевые способы применения средства отражения: получение информации о методах, вызов методов, создание объектов и загрузка типов из компоновочных файлов.

Получение информации о методах

С помощью `Type`-объекта можно получить список методов, поддерживаемых заданным типом. Для этого используется метод `GetMethods()`. Один из форматов его вызова таков:

```
MethodInfo[] GetMethods()
```

Метод GetMethods() возвращает массив объектов типа MethodInfo, которые описывают методы, поддерживаемые вызывающим типом. Класс MethodInfo определен в пространстве имен System.Reflection.

Класс MethodInfo — производный от абстрактного класса MethodBase, который, в свою очередь, выведен из класса MemberInfo. Таким образом, программисту доступны свойства и методы, определенные во всех трех классах. Например, чтобы получить имя метода, используйте свойство Name. Особого внимания заслуживают два члена класса MethodInfo: ReturnType и GetParameters().

Свойство ReturnType, которое имеет тип Type, позволяет получить тип значения, возвращаемого методом.

Метод GetParameters() возвращает список параметров, связанных с методом. Формат его вызова таков:

```
ParameterInfo[] GetParameters()
```

Информация о параметрах содержится в объекте класса ParameterInfo. В классе ParameterInfo определено множество свойств и методов, которые используются для описания параметров. Из них стоит обратить внимание на следующие два: свойство Name, которое представляет собой строку, содержащую имя параметра, и свойство ParameterType, которое описывает тип параметра. Тип параметра инкапсулирован в объекте класса Type.

Рассмотрим программу, в которой средство отражения используется для получения методов, поддерживаемых классом MyClass. Для каждого метода программа отображает его имя и тип возвращаемого им значения, а также имя и тип всех параметров, которые может иметь тот или иной метод.

```
// Анализ методов с помощью средства отражения.

using System;
using System.Reflection;

class MyClass {
  int x;
  int y;

  public MyClass(int i, int j) {
    x = i;
    y = j;
  }

  public int sum() {
    return x+y;
  }

  public bool isBetween(int i) {
    if(x < i && i < y) return true;
    else return false;
  }

  public void set(int a, int b) {
    x = a;
    y = b;
  }

  public void set(double a, double b) {
    x = (int) a;
    y = (int) b;
```

```
  }

  public void show() {
    Console.WriteLine(" x: {0}, y: {1}", x, y);
  }
}

class ReflectDemo {
  public static void Main() {
    Type t = typeof(MyClass); // Получаем Type-объект,
                              // представляющий MyClass.

    Console.WriteLine(
             "Анализ методов, определенных в " + t.Name);
    Console.WriteLine();

    Console.WriteLine("Поддерживаемые методы: ");

    MethodInfo[] mi = t.GetMethods();

    // Отображаем методы, поддерживаемые классом MyClass.
    foreach(MethodInfo m in mi) {
      // Отображаем тип значения, возвращаемого методом,
      // и имя метода.
      Console.Write("   " + m.ReturnType.Name +
                    " " + m.Name + "(");

      // Отображаем параметры.
      ParameterInfo[] pi = m.GetParameters();

      for(int i=0; i < pi.Length; i++) {
        Console.Write(pi[i].ParameterType.Name +
                      " " + pi[i].Name);
        if(i+1 < pi.Length) Console.Write(", ");
      }

      Console.WriteLine(")");

      Console.WriteLine();
    }
  }
}
```

Результаты выполнения этой программы такие:

```
Анализ методов, определенных в MyClass

Поддерживаемые методы:
   Int32 GetHashCode()

   Boolean Equals(Object obj)

   String ToString()

   Int32 sum()

   Boolean isBetween(Int32 i)

   Void set(Int32 a, Int32 b)
```

```
Void set(Double a, Double b)

Void show()

Type GetType()
```

Обратите внимание на то, что помимо методов, определенных в классе `MyClass`, здесь также отображаются методы, определенные в классе `object`. Дело в том, что все типы в C# выведены из класса `object`. Также стоит отметить, что для имен типов здесь используются имена .NET-структуры. Обратите внимание еще на то, что метод `set()` отображен дважды. Этому есть простое объяснение: метод `set()` перегружен. Одна его версия принимает аргументы типа `int`, а вторая — аргументы типа `double`.

Эта программа требует некоторых пояснений. Прежде всего, отметим, что в классе `MyClass` определяется `public`-конструктор и ряд `public`-методов, включая перегруженный метод `set()`. При выполнении строки кода из метода `Main()` получаем объект класса `Type`, представляющий класс `MyClass`:

```
Type t = typeof(MyClass); // Получаем Type-объект,
                          // представляющий MyClass.
```

Используя переменную `t` и интерфейс Reflection API, программа отображает информацию о методах, поддерживаемых классом `MyClass`. Сначала с помощью следующей инструкции получаем список методов:

```
MethodInfo[] mi = t.GetMethods();
```

Затем выполняется цикл `foreach`, на итерациях которого для каждого метода отображается тип возвращаемого им значения, имя метода и его параметры:

```
// Отображаем тип значения, возвращаемого методом,
// и имя метода.
Console.Write("   " + m.ReturnType.Name +
              " " + m.Name + "(");

// Отображаем параметры.
ParameterInfo[] pi = m.GetParameters();

for(int i=0; i < pi.Length; i++) {
  Console.Write(pi[i].ParameterType.Name +
                " " + pi[i].Name);
  if(i+1 < pi.Length) Console.Write(", ");
}
```

В этом фрагменте программы информация о параметрах каждого метода считывается посредством вызова метода `GetParameters()` и сохраняется в массиве `pi`. Затем в цикле `for` выполняется опрос массива `pi` и отображается тип каждого параметра и его имя. Главное здесь то, что эта информация считывается динамически во время выполнения программы, т.е. без предварительной информации о классе `MyClass`.

Второй формат вызова метода `GetMethods()`

Второй формат вызова метода `GetMethods()` позволяет задать различные флаги, которые фильтруют возвращаемые методы. Этот формат таков:

```
MethodInfo[] GetMethods(BindingFlags flags)
```

Эта версия получает только те методы, которые соответствуют заданному критерию. `BindingFlags` — это перечисление. Ниже описаны наиболее употребительные его значения:

Значение	Описание
DeclaredOnly	Считывание только тех методов, которые определены в заданном классе. Унаследованные методы в результат не включаются
Instance	Считывание методов экземпляров
NonPublic	Считывание не-public-методов
Public	Считывание public-методов
Static	Считывание static-методов

Два или больше задаваемых флагов можно объединять с помощью оператора ИЛИ. С флагами Public или NonPublic необходимо устанавливать флаги Instance или Static. В противном случае метод GetMethods() не возвратит ни одного метода.

Одно из основных применений BindingFlags-формата, используемого при вызове метода GetMethods(), — получение списка определенных в классе методов, но без учета унаследованных. Этот вариант особенно полезен в случае, когда нам не нужна информация о методах, определенных объектом. Попробуем, например, заменить вызов метода GetMethods() в предыдущей программе таким вариантом:

```
// Теперь получим только те методы, которые объявлены
// в классе MyClass.
MethodInfo[] mi = t.GetMethods(BindingFlags.DeclaredOnly |
                               BindingFlags.Instance |
                               BindingFlags.Public) ;
```

После внесения в программу этого изменения получаем следующие результаты:

```
Анализ методов, определенных в MyClass

Поддерживаемые методы:
   Int32 sum()

   Boolean isBetween(Int32 i)

   Void set(Int32 a, Int32 b)

   Void set(Double a, Double b)

   Void show()
```

Как видите, теперь отображены только те методы, которые явно определены в классе MyClass.

Вызов методов с помощью средства отражения

Зная, какие методы поддерживает тип, можно вызвать любой из них. Для этого используется метод Invoke(), который определен в классе MethodInfo. Формат его вызова таков:

```
object Invoke(object ob, object[] args)
```

Здесь параметр *ob* — это ссылка на объект, для которого вызывается нужный метод. Для static-методов параметр *ob* должен содержать значение null. Любые аргументы, которые необходимо передать вызываемому методу, указываются в массиве *args*. Если метод вызывается без аргументов, параметр *args* должен иметь null-значение. При этом длина массива *args* должна совпадать с количеством аргументов, передаваемых методу. Следовательно, если необходимо передать два аргумента, массив *args* должен состоять из двух элементов, а не, например, из трех или четырех.

Для вызова нужного метода достаточно вызвать метод `Invoke()` для экземпляра класса `MethodInfo`, полученного в результате вызова метода `GetMethods()`. Эта процедура демонстрируется следующей программой:

```
// Вызов методов с использованием средства отражения.

using System;
using System.Reflection;

class MyClass {
  int x;
  int y;

  public MyClass(int i, int j) {
    x = i;
    y = j;
  }

  public int sum() {
    return x+y;
  }

  public bool isBetween(int i) {
    if((x < i) && (i < y)) return true;
    else return false;
  }

  public void set(int a, int b) {
    Console.Write("Внутри метода set(int, int). ");
    x = a;
    y = b;
    show();
  }

  // Перегруженный метод set.
  public void set(double a, double b) {
    Console.Write("Внутри метода set(double, double). ");
    x = (int) a;
    y = (int) b;
    show();
  }

  public void show() {
    Console.WriteLine(
              "Значение x: {0}, значение y: {1}", x, y);
  }
}

class InvokeMethDemo {
  public static void Main() {
    Type t = typeof(MyClass);
    MyClass reflectOb = new MyClass(10, 20);
    int val;

    Console.WriteLine("Вызов методов, определенных в " +
                      t.Name);
    Console.WriteLine();
    MethodInfo[] mi = t.GetMethods();
```

```
    // Вызываем каждый метод.
    foreach(MethodInfo m in mi) {
      // Получаем параметры.
      ParameterInfo[] pi = m.GetParameters();

      if(m.Name.CompareTo("set")==0 &&
         pi[0].ParameterType == typeof(int)) {
        object[] args = new object[2];
        args[0] = 9;
        args[1] = 18;
        m.Invoke(reflectOb, args);
      }
      else if(m.Name.CompareTo("set")==0 &&
         pi[0].ParameterType == typeof(double)) {
        object[] args = new object[2];
        args[0] = 1.12;
        args[1] = 23.4;
        m.Invoke(reflectOb, args);
      }
      else if(m.Name.CompareTo("sum")==0) {
        val = (int) m.Invoke(reflectOb, null);
        Console.WriteLine(
                "Результат вызова метода sum равен " + val);
      }
      else if(m.Name.CompareTo("isBetween")==0) {
        object[] args = new object[1];
        args[0] = 14;
        if((bool) m.Invoke(reflectOb, args))
          Console.WriteLine("14 находится между x и y.");
      }
      else if(m.Name.CompareTo("show")==0) {
        m.Invoke(reflectOb, null);
      }
    }
  }
}
```

Результаты выполнения этой программы таковы:

```
Вызов методов, определенных в MyClass

Результат вызова метода sum равен 30
14 находится между x и y.
Внутри метода set(int, int). Значение x: 9, значение y: 18
Внутри метода set(double, double). Значение x: 1, значение y: 23
Значение x: 1, значение y: 23
```

Обратите внимание на то, как организуется вызов методов. Сначала получаем список методов. Затем в цикле `foreach` извлекаем информацию о параметрах каждого метода. После этого, используя последовательность `if/else`-инструкций, вызываем каждый метод с соответствующим количеством параметров определенного типа. Особое внимание обратите на способ вызова перегруженного метода `set()`:

```
if(m.Name.CompareTo("set")==0 &&
   pi[0].ParameterType == typeof(int)) {
  object[] args = new object[2];
  args[0] = 9;
  args[1] = 18;
  m.Invoke(reflectOb, args);
}
```

```
else if(m.Name.CompareTo("set")==0 &&
        pi[0].ParameterType == typeof(double)) {
  object[] args = new object[2];
  args[0] = 1.12;
  args[1] = 23.4;
  m.Invoke(reflectOb, args);
}
```

Если имя метода совпадает со строкой `set`, проверяется тип первого параметра, чтобы определить версию метода `set()`. Если окажется, что рассматривается версия `set(int, int)`, в массив `args` загружаются `int`-аргументы и вызывается метод `set()`. В противном случае для вызова метода `set()` используются аргументы типа `double`.

Получение конструкторов типа

В предыдущем примере продемонстрирована возможность вызова методов с использованием средств отражения, однако такой подход не имеет преимуществ по *сравнению* с непосредственным вызовом методов (в данном случае класса `MyClass`), поскольку объект типа `MyClass` создается явным образом. Другими словами, проще вызывать эти методы обычным способом. Однако мощь отражения начинает проявляться в тех случаях, когда объект создается динамически во время работы программы. Для этого нужно сначала получить список конструкторов. Затем создать экземпляр типа, вызвав один из конструкторов. Этот механизм позволяет реализовать объект любого типа во время работы программы, не называя его в инструкции объявления.

Чтобы получить конструкторы типа, вызовите метод `GetConstructors()` для объекта класса `Type`. Один из наиболее употребительных форматов его вызова таков:

```
ConstructorInfo[] GetConstructors()
```

Он возвращает массив объектов типа `ConstructorInfo`, которые описывают эти конструкторы.

Класс `ConstructorInfo` выведен из абстрактного класса `MethodBase`, который является производным от класса `MemberInfo`. Класс `ConstructorInfo` определяет также собственные члены. Из них нас интересует прежде всего метод `GetParameters()`, который возвращает список параметров, связанных с конструктором. Он работает подобно методу `GetParameters()`, определенному в описанном выше классе `MethodInfo`.

Получив информацию о конструкторе, можно с его помощью создать объект, вызвав метод `Invoke()`, определенный в классе `ConstructorInfo`. Формат вызова метода `Invoke()` в этом случае таков:

```
object Invoke(object[] args)
```

Любые аргументы, которые необходимо передать конструктору, задаются с помощью массива *args*. Если конструктор вызывается без аргументов, параметр *args* должен иметь `null`-значение. При этом длина массива *args* должна в точности совпадать с количеством аргументов. Метод `Invoke()` возвращает ссылку на создаваемый объект.

В следующей программе демонстрируется использование средства отражения для создания экземпляра класса `MyClass`:

```
// Создание объекта с помощью средства отражения.

using System;
using System.Reflection;
```

```
class MyClass {
  int x;
  int y;

  public MyClass(int i) {
    Console.WriteLine(
       "Создание объекта по формату MyClass(int). ");
    x = y = i;
  }

  public MyClass(int i, int j) {
    Console.WriteLine(
        "Создание объекта по формату MyClass(int, int). ");
    x = i;
    y = j;
    show();
  }

  public int sum() {
    return x+y;
  }

  public bool isBetween(int i) {
    if((x < i) && (i < y)) return true;
    else return false;
  }

  public void set(int a, int b) {
    Console.Write("Внутри метода set(int, int). ");
    x = a;
    y = b;
    show();
  }

  // Перегруженный метод set().
  public void set(double a, double b) {
    Console.Write("Внутри метода set(double, double). ");
    x = (int) a;
    y = (int) b;
    show();
  }

  public void show() {
    Console.WriteLine(
              "Значение x: {0}, значение y: {1}", x, y);
  }

}

class InvokeConsDemo {
  public static void Main() {
    Type t = typeof(MyClass);
    int val;

    // Получаем информацию о конструкторах.
    ConstructorInfo[] ci = t.GetConstructors();

    Console.WriteLine("Имеются следующие конструкторы: ");
```

```
  foreach(ConstructorInfo c in ci) {
    // Отображаем тип возвращаемого значения и имя.
    Console.Write("    " + t.Name + "(");

    // Отображаем параметры.
    ParameterInfo[] pi = c.GetParameters();

    for(int i=0; i < pi.Length; i++) {
      Console.Write(pi[i].ParameterType.Name +
                    " " + pi[i].Name);
      if(i+1 < pi.Length) Console.Write(", ");
    }

    Console.WriteLine(")");
  }
  Console.WriteLine();

  // Находим подходящий конструктор.
  int x;

  for(x=0; x < ci.Length; x++) {
    ParameterInfo[] pi =  ci[x].GetParameters();
    if(pi.Length == 2) break;
  }

  if(x == ci.Length) {
    Console.WriteLine(
              "Подходящий конструктор не найден.");
    return;
  }
  else
    Console.WriteLine(
        "Найден конструктор с двумя параметрами.\n");

  // Создаем объект.
  object[] consargs = new object[2];
  consargs[0] = 10;
  consargs[1] = 20;
  object reflectOb = ci[x].Invoke(consargs);

  Console.WriteLine(
           "\nВызов методов для объекта reflectOb.");
  Console.WriteLine();
  MethodInfo[] mi = t.GetMethods();

  // Вызываем каждый метод.
  foreach(MethodInfo m in mi) {
    // Получаем параметры.
    ParameterInfo[] pi = m.GetParameters();

    if(m.Name.CompareTo("set")==0 &&
       pi[0].ParameterType == typeof(int)) {
      // Это метод set(int, int).
      object[] args = new object[2];
      args[0] = 9;
      args[1] = 18;
      m.Invoke(reflectOb, args);
    }
```

```
      else if(m.Name.CompareTo("set")==0 &&
              pi[0].ParameterType == typeof(double)) {
        // Это метод set(double, double).
        object[] args = new object[2];
        args[0] = 1.12;
        args[1] = 23.4;
        m.Invoke(reflectOb, args);
      }
      else if(m.Name.CompareTo("sum")==0) {
        val = (int) m.Invoke(reflectOb, null);
        Console.WriteLine(
          "Результат выполнения метода sum() равен " + val);
      }
      else if(m.Name.CompareTo("isBetween")==0) {
        object[] args = new object[1];
        args[0] = 14;
        if((bool) m.Invoke(reflectOb, args))
          Console.WriteLine("14 находится между x и y.");
      }
      else if(m.Name.CompareTo("show")==0) {
        m.Invoke(reflectOb, null);
      }
    }
  }
}
```

Результаты выполнения этой программы таковы:

```
Имеются следующие конструкторы:
   MyClass(Int32 i)
   MyClass(Int32 i, Int32 j)

Найден конструктор с двумя параметрами.

Создание объекта по формату MyClass(int, int).
Значение x: 10, значение y: 20

Вызов методов для объекта reflectOb.

Результат выполнения метода sum() равен 30
14 находится между x и y.
Внутри метода set(int, int). Значение x: 9, значение y: 18
Внутри метода set(double, double). Значение x: 1, значение y: 23
Значение x: 1, значение y: 23
```

Теперь разберемся, как используется средство отражения для создания объекта класса `MyClass`. Сначала с помощью следующей инструкции получаем список открытых конструкторов:

```
ConstructorInfo[] ci = t.GetConstructors();
```

Затем в целях иллюстрации отображаем конструкторы, определенные в этом классе. После этого с помощью следующего кода просматриваем полученный список, чтобы найти конструктор, который принимает два аргумента:

```
for(x=0; x < ci.Length; x++) {
  ParameterInfo[] pi = ci[x].GetParameters();
  if(pi.Length == 2) break;
}
```

Если нужный конструктор обнаружен (как в данном случае), создаем объект, выполнив такую последовательность инструкций:

```
// Создаем объект.
object[] consargs = new object[2];
consargs[0] = 10;
consargs[1] = 20;
object reflectOb = ci[x].Invoke(consargs);
```

После обращения к методу `Invoke()` объект `reflectOb` будет ссылаться на объект класса `MyClass`.

В этом примере в целях упрощения предполагалось, что конструктор, который принимает два `int`-аргумента, — единственный среди всех конструкторов, определенных в классе `MyClass`. В реальных приложениях необходимо проверять тип каждого аргумента.

Получение типов из компоновочных файлов

В предыдущем примере с помощью средства отражения мы многое узнали о классе `MyClass`, но не все: мы не получили данные о самом типе `MyClass`. Несмотря на то что мы динамически извлекли из соответствующих объектов информацию о типе `MyClass`, мы исходили из того, что нам заранее было известно имя типа `MyClass`, и использовали инструкцию `typeof` для получения объекта класса `Type`, для которого вызывались все методы средства отражения (напрямую или опосредованно). И хотя в некоторых ситуациях такой подход себя вполне оправдывает, возможности средства отражения проявляются в полной мере тогда, когда программа в состоянии определить необходимые типы посредством анализа содержимого других компоновчных файлов.

Как было описано в главе 16, компоновочный файл включает информацию о классах, структурах и пр., которые он содержит. Интерфейс Reflection API позволяет загрузить компоновочный файл, извлечь информацию о нем и создать экземпляры любого из содержащихся в нем типов. Используя этот механизм, программа может проанализировать среду выполнения и заставить ее поработать в нужном направлении, не определяя явным образом "точки приложения" во время компиляции. Это чрезвычайно эффективное средство. Например, представьте себе программу, которая действует как броузер типов, отображая доступные в системе типы. Или представьте другое приложение, которое выполняло бы роль средства проектирования, позволяющего визуально связывать отдельные части программы, состоящей из различных типов, поддерживаемых системой. Если все данные о типе поддаются обнаружению, то не существует ограничений на применение средства отражения.

Для получения информации о компоновочном файле сначала необходимо создать объект класса `Assembly`. Класс `Assembly` не определяет ни одного `public`-конструктора. Но объект класса `Assembly` можно создать, вызвав один из его методов. Например, воспользуемся методом `LoadFrom()`. Вот формат его использования:

```
static Assembly LoadFrom(string имя_файла)
```

Здесь элемент *имя_файла* означает имя компоновочного файла.

Создав объект класса `Assembly`, можно получить содержащуюся в нем информацию о типах с помощью метода `GetTypes()`. Формат его вызова таков:

```
Type[] GetTypes()
```

Этот метод возвращает массив типов, содержащихся в компоновочном файле.

Чтобы продемонстрировать получение информации о типах из компоновочного файла, нужно иметь два файла. Первый должен включать набор классов. Поэтому создадим файл `MyClasses.cs` с таким содержимым:

```
// Этот файл содержит три класса.
// Назовите его MyClasses.cs.
```

```
using System;

class MyClass {
  int x;
  int y;

  public MyClass(int i) {
    Console.WriteLine(
         "Создание объекта по формату MyClass(int). ");
    x = y = i;
    show();
  }

  public MyClass(int i, int j) {
    Console.WriteLine(
      "Создание объекта по формату MyClass(int, int). ");
    x = i;
    y = j;
    show();
  }

  public int sum() {
    return x+y;
  }

  public bool isBetween(int i) {
    if((x < i) && (i < y)) return true;
    else return false;
  }

  public void set(int a, int b) {
    Console.Write("Внутри метода set(int, int). ");
    x = a;
    y = b;
    show();
  }

  // Перегруженный метод set.
  public void set(double a, double b) {
    Console.Write("Внутри метода set(double, double). ");
    x = (int) a;
    y = (int) b;
    show();
  }

  public void show() {
    Console.WriteLine(
              "Значение x: {0}, значение y: {1}", x, y);
  }

}

class AnotherClass {
  string remark;

  public AnotherClass(string str) {
    remark = str;
  }
```

```
  public void show() {
    Console.WriteLine(remark);
  }
}

class Demo {
  public static void Main() {
    Console.WriteLine("Это заглушка.");
  }
}
```

Этот файл содержит класс `MyClass`, который мы использовали в предыдущих примерах. Кроме того, сюда входит класс `AnotherClass` и еще один класс — `Demo`. Таким образом, компоновочный файл, генерируемый программой, должен содержать три класса. Теперь скомпилируем этот файл, чтобы получить файл `MyClasses.exe`. Это и есть компоновочный файл, который мы будем опрашивать.

Теперь рассмотрим программу, которая извлекает информацию о файле `MyClasses.exe`.

```
/* Находим компоновочный файл, определяем типы и
   создаем объект, используя средство отражения. */

using System;
using System.Reflection;

class ReflectAssemblyDemo {
  public static void Main() {
    int val;

    // Загружаем компоновочный файл MyClasses.exe.
    Assembly asm = Assembly.LoadFrom("MyClasses.exe");

    // Узнаем, какие типы содержит файл MyClasses.exe.
    Type[] alltypes = asm.GetTypes();
    foreach(Type temp in alltypes)
      Console.WriteLine("Обнаружено: " + temp.Name);

    Console.WriteLine();

    // Используем первый тип,
    // которым в данном случае является MyClass.
    Type t = alltypes[0]; // Анализируем первый
                          // обнаруженный класс.
    Console.WriteLine("Используем: " + t.Name);

    // Получаем информацию о конструкторах.
    ConstructorInfo[] ci = t.GetConstructors();

    Console.WriteLine("Имеются следующие конструкторы: ");
    foreach(ConstructorInfo c in ci) {
      // Отображаем тип возвращаемого значения и имя.
      Console.Write("    " + t.Name + "(");

      // Отображаем параметры.
      ParameterInfo[] pi = c.GetParameters();

      for(int i=0; i < pi.Length; i++) {
        Console.Write(pi[i].ParameterType.Name +
```

```
                    " " + pi[i].Name);
      if(i+1 < pi.Length) Console.Write(", ");
    }

    Console.WriteLine(")");
  }
  Console.WriteLine();

  // Находим подходящий конструктор.
  int x;

  for(x=0; x < ci.Length; x++) {
    ParameterInfo[] pi =  ci[x].GetParameters();
    if(pi.Length == 2) break;
  }

  if(x == ci.Length) {
    Console.WriteLine(
                  "Подходящий конструктор не найден.");
    return;
  }
  else
    Console.WriteLine(
            "Найден конструктор с двумя параметрами.\n");

  // Создаем объект.
  object[] consargs = new object[2];
  consargs[0] = 10;
  consargs[1] = 20;
  object reflectOb = ci[x].Invoke(consargs);

  Console.WriteLine(
              "\nВызов методов для объекта reflectOb.");
  Console.WriteLine();
  MethodInfo[] mi = t.GetMethods();

  // Вызываем каждый метод.
  foreach(MethodInfo m in mi) {
    // Получаем параметры.
    ParameterInfo[] pi = m.GetParameters();

    if(m.Name.CompareTo("set")==0 &&
       pi[0].ParameterType == typeof(int)) {
      // Это метод set(int, int).
      object[] args = new object[2];
      args[0] = 9;
      args[1] = 18;
      m.Invoke(reflectOb, args);
    }
    else if(m.Name.CompareTo("set")==0 &&
       pi[0].ParameterType == typeof(double)) {
      // Это метод set(double, double).
      object[] args = new object[2];
      args[0] = 1.12;
      args[1] = 23.4;
      m.Invoke(reflectOb, args);
    }
    else if(m.Name.CompareTo("sum")==0) {
```

```
        val = (int) m.Invoke(reflectOb, null);
        Console.WriteLine(
               "Результат выполнения метода sum() равен " +
               val);
      }
      else if(m.Name.CompareTo("isBetween")==0) {
        object[] args = new object[1];
        args[0] = 14;
        if((bool) m.Invoke(reflectOb, args))
          Console.WriteLine("14 находится между x и y.");
      }
      else if(m.Name.CompareTo("show")==0) {
        m.Invoke(reflectOb, null);
      }
    }

  }
}
```

Результаты выполнения этой программы таковы:

```
Обнаружено: MyClass
Обнаружено: AnotherClass
Обнаружено: Demo

Используем: MyClass
Имеются следующие конструкторы:
   MyClass(Int32 i)
   MyClass(Int32 i, Int32 j)

Найден конструктор с двумя параметрами.

Создание объекта по формату MyClass(int, int).
Значение x: 10, значение y: 20

Вызов методов для объекта reflectOb.

Результат выполнения метода sum() равен 30
14 находится между x и y.
Внутри метода set(int, int). Значение x: 9, значение y: 18
Внутри метода set(double, double). Значение x: 1, значение y: 23
Значение x: 1, значение y: 23
```

Как видно из результатов выполнения программы, были обнаружены все три класса, содержащиеся в файле `MyClasses.exe`. Первый из выявленных классов, в данном случае это `MyClass`, был использован для реализации объекта и выполнения его методов. Причем все это было проделано без использования информации о содержимом файла `MyClasses.exe`.

Информация о типах, содержащихся в файле `MyClasses.exe`, извлекается с помощью следующей последовательности инструкций, которыми открывается метод `Main()`:

```
    // Загружаем компоновочный файл MyClasses.exe.
    Assembly asm = Assembly.LoadFrom("MyClasses.exe");

    // Узнаем, какие типы содержит файл MyClasses.exe.
    Type[] alltypes = asm.GetTypes();
    foreach(Type temp in alltypes)
      Console.WriteLine("Обнаружено: " + temp.Name);
```

Эту последовательность инструкций можно использовать в случае, когда нужно динамически загрузить и опросить компоновочный файл.

Кстати, компоновочный файл необязательно должен быть exe-файлом. Компоновочные файлы также можно найти среди файлов динамически подключаемой библиотеки (dynamic link library — DLL), которые имеют расширение dll. Например, файл MyClasses.cs можно скомпилировать с помощью такой командной строки:

```
csc /t:library MyClasses.cs
```

В результате выполнения этой команды мы получили бы выходной файл MyClasses.dll. При внесении программного кода в DLL-библиотеку не требуется создавать метод Main(). Для всех exe-файлов наличие такой точки входа, как метод Main(), обязательно. Поэтому класс Demo содержит заглушку для метода Main(). Для DLL-библиотек точки входа могут отсутствовать. Если вы захотите превратить MyClass в DLL-файл, вам придется изменить обращение к методу LoadFrom() следующим образом:

```
Assembly asm = Assembly.LoadFrom("MyClasses.dll");
```

Полная автоматизация получения информации о типах

Прежде чем завершить изучение темы отражения информации о типах, стоит рассмотреть еще один пример. Несмотря на то что в предыдущей программе нам удалось использовать класс MyClass без явного указания его имени в программе, все же мы предварительно знали содержимое класса MyClass. Например, нам были заранее известны имена его методов (set() и sum()). Однако с помощью средства отражения можно использовать тип, о котором нам предварительно ничего не известно. Для этого необходимо получить информацию, необходимую для создания объекта, и сгенерировать вызовы методов. Такой подход эффективен, например, в случае визуального средства проектирования, поскольку в нем используются типы, имеющиеся в системе.

Чтобы понять, как динамически извлечь информацию о типе, рассмотрим следующий пример, в котором загружается компоновочный файл MyClasses.exe, создается объект класса MyClass, а затем вызываются все объявленные методы без каких бы то ни было предварительных сведений.

```
// Использование класса MyClass без опоры на
// предварительные данные о нем.

using System;
using System.Reflection;

class ReflectAssemblyDemo {
  public static void Main() {
    int val;
    Assembly asm = Assembly.LoadFrom("MyClasses.exe");

    Type[] alltypes = asm.GetTypes();

    Type t = alltypes[0]; // Используем первый
                          // обнаруженный класс.

    Console.WriteLine("Используем: " + t.Name);

    ConstructorInfo[] ci = t.GetConstructors();

    // Используем первый обнаруженный конструктор.
```

```
ParameterInfo[] cpi = ci[0].GetParameters();
object reflectOb;

if(cpi.Length > 0) {
  object[] consargs = new object[cpi.Length];

  // Инициализируем аргументы.
  for(int n=0; n < cpi.Length; n++)
    consargs[n] = 10 + n * 20;

  // Создаем объект.
  reflectOb = ci[0].Invoke(consargs);
} else
  reflectOb = ci[0].Invoke(null);


Console.WriteLine(
          "\nВызываем методы для объекта reflectOb.");
Console.WriteLine();

// Игнорируем унаследованные методы.
MethodInfo[] mi = t.GetMethods(
                          BindingFlags.DeclaredOnly |
                          BindingFlags.Instance |
                          BindingFlags.Public) ;

// Вызываем каждый метод.
foreach(MethodInfo m in mi) {
  Console.WriteLine("Вызов метода {0} ", m.Name);

  // Получаем параметры.
  ParameterInfo[] pi = m.GetParameters();

  // Выполняем методы.
  switch(pi.Length) {
    case 0: // без аргументов
      if(m.ReturnType == typeof(int)) {
        val = (int) m.Invoke(reflectOb, null);
        Console.WriteLine("Результат равен " + val);
      }
      else if(m.ReturnType == typeof(void)) {
        m.Invoke(reflectOb, null);
      }
      break;
    case 1: // один аргумент
      if(pi[0].ParameterType == typeof(int)) {
        object[] args = new object[1];
        args[0] = 14;
        if((bool) m.Invoke(reflectOb, args))
          Console.WriteLine(
                    "14 находится между x и y.");
        else
          Console.WriteLine(
                    "14 не находится между x и y.");
      }
      break;
    case 2: // два аргумента
      if((pi[0].ParameterType == typeof(int)) &&
```

```
            (pi[1].ParameterType == typeof(int))) {
          object[] args = new object[2];
          args[0] = 9;
          args[1] = 18;
          m.Invoke(reflectOb, args);
        }
        else if((pi[0].ParameterType == typeof(double)) &&
                (pi[1].ParameterType == typeof(double))) {
          object[] args = new object[2];
          args[0] = 1.12;
          args[1] = 23.4;
          m.Invoke(reflectOb, args);
        }
        break;
      }
      Console.WriteLine();
    }

  }
}
```

Результаты выполнения этой программы таковы:

```
Используем: MyClass
Создание объекта по формату MyClass(int).
Значение x: 10, значение y: 10

Вызываем методы для объекта reflectOb.

Вызов метода sum
Результат равен 20

Вызов метода isBetween
14 не находится между x и y.

Вызов метода set
Внутри метода set(int, int). Значение x: 9, значение y: 18

Вызов метода set
Внутри метода set(double, double). Значение x: 1, значение y: 23

Вызов метода show
Значение x: 1, значение y: 23
```

Обратите внимание вот на что. Во-первых, программа получает (и использует) информацию только о тех методах, которые явно объявлены в классе `MyClass`. Эта фильтрация достигается благодаря использованию `BindingFlags`-формата вызова метода `GetMethods()`. Тем самым становится возможным отсев унаследованных методов. *Во-вторых*, отметьте, каким образом программа динамически получает количество параметров и тип значений, возвращаемых каждым методом. Количество параметров определяется с помощью `switch`-инструкции. В каждой `case`-ветви этой инструкции проверяется тип (типы) параметра (параметров) и тип возвращаемого методом значения. Затем на основе этой информации и организуется соответствующий вызов метода.

Атрибуты

В C# предусмотрена возможность вносить в программу информацию описательного характера в формате *атрибута*. Атрибут содержит дополнительные сведения о классе, структуре, методе и т.д. Например, можно создать атрибут, определяющий тип кнопки, для отображения которой предназначен класс. Атрибуты указываются внутри квадратных скобок, предваряя элемент, к которому они применяются. Таким образом, атрибут не является членом класса. Он просто содержит дополнительную информацию об элементе.

Основы применения атрибутов

Атрибут поддерживается классом, производным от класса `System.Attribute`. Таким образом, все классы атрибутов являются подклассами класса `Attribute`. Хотя класс класс `Attribute` определяет фундаментальную функциональность, она не всегда востребована при работе с атрибутами. Для классов атрибутов принято использовать суффикс `Attribute`. Например, для класса атрибута, предназначенного для описания ошибок, вполне подошло бы имя `ErrorAttribute`.

Объявление класса атрибута предваряется атрибутом `AttributeUsage`. Этот встроенный атрибут задает типы элементов, к которым может применяться объявляемый атрибут.

Создание атрибута

В классе атрибута определяются члены, которые поддерживают данный атрибут. Обычно классы атрибутов очень просты и содержат лишь небольшое количество полей или свойств. Например, в атрибуте может содержаться комментарий, описывающий элемент, к которому относится атрибут. Такой атрибут может иметь следующий вид:

```
[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
  string pri_remark; // Базовое поле для свойства remark.

  public RemarkAttribute(string comment) {
    pri_remark = comment;
  }

  public string remark {
    get {
      return pri_remark;
    }
  }
}
```

Рассмотрим представленный класс построчно.

Имя этого атрибута `RemarkAttribute`. Его объявление предваряется атрибутом `AttributeUsage`, который означает, что атрибут `RemarkAttribute` можно применить к элементам всех типов. С помощью атрибута `AttributeUsage` можно сократить список элементов, к которым будет относиться атрибут; эту возможность мы рассмотрим в следующей главе.

Затем следует объявление класса атрибута `RemarkAttribute`, производного от `Attribute`. Класс `RemarkAttribute` содержит одно закрытое поле `pri_remark`, которое служит основой для единственного открытого свойства `remark`, предназначенного только для чтения. В этом свойстве хранится описание, связанное с атрибутом. В

классе RemarkAttribute определен один открытый конструктор, который принимает строковый аргумент и присваивает его значение полю pri_remark. Этим, собственно, и ограничиваются функции атрибута RemarkAttribute, который совершенно готов к применению.

Присоединение атрибута

Определив класс атрибута, можно присоединить его к соответствующему элементу. Атрибут предшествует элементу, к которому он присоединяется, и задается путем заключения его конструктора в квадратные скобки. Например, вот как атрибут RemarkAttribute можно связать с классом:

```
[RemarkAttribute("Этот класс использует атрибут.")]
class UseAttrib {
  // ...
}
```

При выполнении этого фрагмента кода создается объект класса RemarkAttribute, который содержит комментарий "Этот класс использует атрибут.". Затем атрибут связывается с классом UseAttrib.

При связывании атрибута необязательно указывать суффикс Attribute. Например, предыдущее объявление класса можно было бы записать так:

```
[Remark("Этот класс использует атрибут.")]
class UseAttrib {
  // ...
}
```

Здесь используется только имя Remark. Несмотря на корректность использования этой короткой формы, при связывании атрибута все же безопаснее использовать его полное имя, поскольку это позволяет избежать возможной путаницы и неоднозначности.

Получение атрибутов объекта

После того как атрибут присоединен к элементу, другие части программы могут его извлечь. Для этого обычно используют один из двух методов. Первый — метод GetCustomAttributes(), который определен в классе MemberInfo и унаследован классом Type. Он считывает список всех атрибутов, связанных с элементом, а формат его вызова таков:

```
object[] GetCustomAttributes(bool searchBases)
```

Если аргумент *searchBases* имеет значение true, в результирующий список включаются атрибуты всех базовых классов по цепочке наследования. В противном случае будут включены только атрибуты, определенные заданным типом.

Второй — метод GetCustomAttribute(), который определен в классе Attribute. Вот один из форматов его вызова:

```
static Attribute GetCustomAttribute(MemberInfo mi,
                                    Type attribtype)
```

Здесь аргумент *mi* означает объект класса MemberInfo, описывающий элемент, для которого извлекается атрибут. Нужный атрибут указывается аргументом *attribtype*. Этот метод используется в том случае, если известно имя атрибута, который нужно получить. Например, чтобы получить ссылку на атрибут RemarkAttribute, можно использовать такую последовательность:

```
// Считываем RemarkAttribute.
Type tRemAtt = typeof(RemarkAttribute);
RemarkAttribute ra = (RemarkAttribute)
      Attribute.GetCustomAttribute(t, tRemAtt);
```

Имея ссылку на атрибут, можно получить доступ к его членам. Другими словами, информация, связанная с атрибутом, доступна программе, использующей элемент, к которому присоединен атрибут. Например, при выполнении следующей инструкции отображается значение поля remark:

```
Console.WriteLine(ra.remark);
```

Использование атрибута RemarkAttribute демонстрируется в приведенной ниже программе.

```
// Простой пример атрибута.

using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
  string pri_remark; // Базовое поле для свойства remark.

  public RemarkAttribute(string comment) {
    pri_remark = comment;
  }

  public string remark {
    get {
      return pri_remark;
    }
  }
}

[RemarkAttribute("Этот класс использует атрибут.")]
class UseAttrib {
  // ...
}

class AttribDemo {
  public static void Main() {
    Type t = typeof(UseAttrib);

    Console.Write("Атрибуты в " + t.Name + ": ");

    object[] attribs = t.GetCustomAttributes(false);
    foreach(object o in attribs) {
      Console.WriteLine(o);
    }

    Console.Write("Remark: ");

    // Считываем атрибут RemarkAttribute.
    Type tRemAtt = typeof(RemarkAttribute);
    RemarkAttribute ra = (RemarkAttribute)
          Attribute.GetCustomAttribute(t, tRemAtt);


    Console.WriteLine(ra.remark);
  }
}
```

Результаты выполнения этой программы таковы:

```
Атрибуты в UseAttrib: RemarkAttribute
Remark: Этот класс использует атрибут.
```

Сравнение позиционных и именованных параметров

В предыдущем примере атрибут `RemarkAttribute` был инициализирован посредством передачи конструктору строки описания. В этом случае, т.е. при использовании обычного синтаксиса конструктора, параметр `comment`, принимаемый конструктором `RemarkAttribute()`, называется *позиционным параметром*. Возникновение этого термина объясняется тем, что аргумент метода связывается с параметром посредством своей позиции. Так работают в C# все методы и конструкторы. Но для атрибутов можно создавать *именованные параметры* и присваивать им начальные значения, используя их имена.

Именованный параметр поддерживается либо открытым полем, либо свойством, которое не должно быть предназначено только для чтения. Такое поле или свойство автоматически можно использовать в качестве именованного параметра. При определении атрибута для элемента именованный параметр получает значение с помощью инструкции присваивания, которая содержится в конструкторе атрибута. Формат спецификации атрибута, включающей именованные параметры, таков:

```
[attrib(список_позиционных_параметров,
        именованный_параметр_1 = value,
        именованный_параметр_2 = value, ...)]
```

Позиционные параметры (если они имеются) должны стоять в начале списка параметров. Затем каждому именованному параметру присваивается значение. Порядок следования именованных параметров не имеет значения. Именованным параметрам присваивать значения необязательно. Но в данном случае значения инициализации использованы по назначению.

Чтобы понять, как используются именованные параметры, лучше всего рассмотреть пример. Перед вами версия определения класса `RemarkAttribute`, в которую добавлено поле `supplement`, позволяющее хранить дополнительный комментарий.

```
[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
  string pri_remark; // Базовое поле для свойства remark.

  public string supplement; // Это именованный параметр.

  public RemarkAttribute(string comment) {
    pri_remark = comment;
    supplement = "Данные отсутствуют";
  }

  public string remark {
    get {
      return pri_remark;
    }
  }
}
```

Как видите, поле `supplement` инициализируется строкой "Данные отсутствуют" в конструкторе класса. С помощью конструктора невозможно присваивоить этому полю другое начальное значение. Однако, как показано в следующем фрагменте кода, поле `supplement` можно использовать в качестве именованного параметра:

```
[RemarkAttribute("Этот класс использует атрибут.",
    supplement = "Это дополнительная информация.")]
class UseAttrib {
  // ...
}
```

Обратите особое внимание на то, как вызывается конструктор класса RemarkAttribute. Сначала задается позиционный аргумент. За ним, после запятой, следует именованный параметр supplement, которому присваивается значение. Обращение к конструктору завершает закрывающая круглая скобка. Таким образом, именованный параметр инициализируется внутри вызова конструктора. Этот синтаксис можно обощить. Итак, позиционные параметры необходимо задавать в порядке, который определен конструктором. Именованные параметры задаются посредством присваивания значений их именам.

Рассмотрим программу, которая демонстрирует использование поля supplement:

```
// Использование именованного параметра атрибута.

using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
  string pri_remark; // Базовое поле для свойства remark.

  public string supplement; // Это именованный параметр.

  public RemarkAttribute(string comment) {
    pri_remark = comment;
    supplement = "Данные отсутствуют";
  }

  public string remark {
    get {
      return pri_remark;
    }
  }
}

[RemarkAttribute("Этот класс использует атрибут.",
    supplement = "Это дополнительная информация.")]
class UseAttrib {
  // ...
}

class NamedParamDemo {
  public static void Main() {
    Type t = typeof(UseAttrib);

    Console.Write("Атрибуты в " + t.Name + ": ");

    object[] attribs = t.GetCustomAttributes(false);
    foreach(object o in attribs) {
      Console.WriteLine(o);
    }

    // Считывание атрибута RemarkAttribute.
    Type tRemAtt = typeof(RemarkAttribute);
    RemarkAttribute ra = (RemarkAttribute)
          Attribute.GetCustomAttribute(t, tRemAtt);

    Console.Write("Remark: ");
    Console.WriteLine(ra.remark);
```

```
    Console.Write("Supplement: ");
    Console.WriteLine(ra.supplement);
  }
}
```

Вот результаты выполнения этой программы:

```
Атрибуты в UseAttrib: RemarkAttribute
Remark: Этот класс использует атрибут.
Supplement: Это дополнительная информация.
```

Как разъяснялось выше, в качестве именованного параметра можно также использовать свойство. Например, в следующей программе в класс атрибута `RemarkAttribute` добавляется `int`-свойство с именем `priority`.

```
// Использование свойства в качестве именованного
// параметра атрибута.

using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
  string pri_remark; // Базовое поле для свойства remark.

  int pri_priority; // Базовое поле для свойства priority.

  public string supplement; // Это именованный параметр.

  public RemarkAttribute(string comment) {
    pri_remark = comment;
    supplement = "Данные отсутствуют";
  }

  public string remark {
    get {
      return pri_remark;
    }
  }

  // Используем свойство в качестве именованного параметра.
  public int priority {
    get {
      return pri_priority;
    }
    set {
      pri_priority = value;
    }
  }
}

[RemarkAttribute(
        "Этот класс использует атрибут.",
        supplement = "Это дополнительная информация.",
        priority = 10)]
class UseAttrib {
  // ...
}

class NamedParamDemo {
```

```
  public static void Main() {
    Type t = typeof(UseAttrib);

    Console.Write("Атрибуты в " + t.Name + ": ");

    object[] attribs = t.GetCustomAttributes(false);
    foreach(object o in attribs) {
      Console.WriteLine(o);
    }

    // Считываем атрибут RemarkAttribute.
    Type tRemAtt = typeof(RemarkAttribute);
    RemarkAttribute ra = (RemarkAttribute)
          Attribute.GetCustomAttribute(t, tRemAtt);

    Console.Write("Remark: ");
    Console.WriteLine(ra.remark);

    Console.Write("Supplement: ");
    Console.WriteLine(ra.supplement);

    Console.WriteLine("Priority: " + ra.priority);
  }
}
```

Вот результаты выполнения этой программы:

```
Атрибуты в UseAttrib: RemarkAttribute
Remark: Этот класс использует атрибут.
Supplement: Это дополнительная информация.
Priority: 10
```

Обратите внимание на определение атрибута (перед определением класса `UseAttrib`):

```
[RemarkAttribute(
         "Этот класс использует атрибут.",
         supplement = "Это дополнительная информация.",
         priority = 10)]
```

Задание именованных атрибутов `supplement` и `priority` не подчинено определенному порядку. Эти два присваивания можно поменять местами, и это никак не отразится на атрибуте в целом.

Использование встроенных атрибутов

В C# определено три встроенных атрибута: `AttributeUsage`, `Conditional` и `Obsolete`. Рассмотрим их по порядку.

Атрибут `AttributeUsage`

Как упоминалось выше, атрибут `AttributeUsage` определяет типы элементов, к которым можно применить атрибут. `AttributeUsage` — это еще одно имя для класса `System.AttributeUsageAttribute`. В классе `AttributeUsage` определен следующий конструктор:

`AttributeUsage(AttributeTargets` *item*`)`

Здесь параметр *item* означает элемент или элементы, к которым может быть применен этот атрибут. Тип `AttributeTargets` — это перечисление, которое определяет следующие значения:

`All`	`Assembly`	`Class`	`Constructor`
`Delegate`	`Enum`	`Event`	`Field`
`Interface`	`Method`	`Module`	`Parameter`
`Property`	`ReturnValue`	`Struct`	

Два или больше из этих значений можно объединить с помощью операции ИЛИ. Например, чтобы определить атрибут, применяемый только к полям и свойствам, используйте следующий вариант объединения значений перечисления `AttributeTargets`:

```
AttributeTargets.Field | AttributeTargets.Property
```

Конструктор класса `AttributeUsage` поддерживает два именованных параметра. Первый — это параметр `AllowMultiple`, который принимает значение типа `bool`. Если оно истинно, этот атрибут можно применить к одному элементу более одного раза. Второй — параметр `Inherited`, который также принимает значение типа `bool`. Если оно истинно, этот атрибут наследуется производными классами. В противном случае — не наследуется. По умолчанию оба параметра `AllowMultiple` и `Inherited` устанавливаются равными значению `false`.

Атрибут `Conditional`

Атрибут `Conditional`, пожалуй, самый интересный из всех встроенных C#-атрибутов. Он позволяет создавать *условные методы*. Условный метод вызывается только в том случае, если соответствующий идентификатор определен с помощью директивы `#define`. В противном случае вызов метода опускается. Таким образом, условный метод предлагает альтернативу условной компиляции на основе директивы `#if`.

`Conditional` — еще одно имя для класса `System.Diagnostics.ConditionalAttribute`. Чтобы использовать атрибут `Conditional`, необходимо включить в программу объявление пространства имен `System.Diagnostics`.

Как всегда, лучше начать с примера.

```
// Демонстрация использования атрибута Conditional.

#define TRIAL

using System;
using System.Diagnostics;

class Test {

  [Conditional("TRIAL")]
  void trial() {
    Console.WriteLine(
          "Пробная версия, не для распространения.");
  }

  [Conditional("RELEASE")]
  void release() {
    Console.WriteLine("Окончательная версия.");
  }

  public static void Main() {
    Test t = new Test();
```

```
    t.trial(); // Вызывается только в случае, если
               // идентификатор TRIAL определен.
    t.release(); // Вызывается только в случае, если
                 // идентификатор RELEASE определен.
  }
}
```

Вот результаты выполнения этой программы:

```
Пробная версия, не для распространения.
```

Рассмотрим внимательно код этой программы, чтобы понять, почему получены такие результаты. Прежде всего следует отметить, что в программе определяется идентификатор `TRIAL`, и обратить ваше внимание на определение методов `trial()` и `release()`. В обоих случаях им предшествует атрибут `Conditional`, который используется в таком формате:

```
[Conditional "symbol"]
```

Здесь элемент *symbol* означает идентификатор, который определяет, будет ли выполнен этот метод. Этот атрибут можно использовать только для методов. Если соответствующий идентификатор определен, вызываемый метод выполняется. В противном случае метод не выполняется.

Внутри метода `Main()` вызывается как метод `trial()`, так и метод `release()`. Однако в программе определен только идентификатор `TRIAL`. Поэтому выполняется один метод `trial()`. Вызов же метода `release()` игнорируется. Если определить также и идентификатор `RELEASE`, выполнится и метод `release()`. Если при этом удалить определение идентификатора `TRIAL`, метод `trial()` вызван не будет.

На условные методы налагается ряд ограничений. Они должны возвращать `void`-значение. Они должны быть членами класса, а не интерфейса. Их определение не может предварять ключевое слово `override`.

Атрибут `Obsolete`

Имя атрибута `Obsolete` представляет собой сокращение от имени класса `System.ObsoleteAttribute`. Этот атрибут позволяет отметить какой-либо элемент программы как *устаревший*. Формат его применения таков:

```
[Obsolete("message")]
```

Здесь параметр *message* содержит сообщение, которое будет отображено в случае компиляции соответствующего элемента программы. Рассмотрим короткий пример.

```
// Демонстрация использования атрибута Obsolete.

using System;

class Test {

  [Obsolete("Лучше использовать метод myMeth2.")]
  static int myMeth(int a, int b) {
    return a / b;
  }

  // Улучшенная версия метода myMeth().
  static int myMeth2(int a, int b) {
    return b == 0 ? 0 : a /b;
  }

  public static void Main() {
```

```
    // Предупреждение, отображаемое при выполнении
    // этой инструкции.
     Console.WriteLine("4 / 3 is " + Test.myMeth(4, 3));

    // Здесь не будет никакого предупреждения.
     Console.WriteLine("4 / 3 is " + Test.myMeth2(4, 3));
  }
}
```

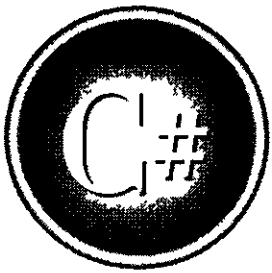
Если при компиляции этой программы в методе `Main()` встретится вызов метода `myMeth()`, сгенерируется предупреждение, в котором пользователю будет предложено использовать вместо метода `myMeth()` метод `myMeth2()`.

Второй формат применения атрибута `Obsolete` выглядит так:

```
[Obsolete("message", error)]
```

Здесь параметр *error* имеет тип `Boolean`. Если его значение равно `true`, то при использовании устаревшего элемента программы будет сгенерировано не предупреждение, а сообщение об ошибке. Нетрудно догадаться, что разница между этими двумя форматами состоит в том, что программа с ошибкой не может быть скомпилирована в выполняемую программу.

Полный справочник по



Глава 18

Опасный код, указатели и другие темы

Такое название темы обычно вызывает у программистов удивление. Опасный код зачастую включает использование указателей. Код, отмеченный как "опасный", и собственно указатели позволяют использовать средства языка C# для создания приложений, которые обычно ассоциируются с применением C++, т.е. приложений, которые отличаются высокой производительностью и претендуют на звание системных. Более того, включение "опасного кода" и указателей дает C# такие возможности, которых не достает языку Java.

В этой главе рассматриваются ключевые слова, которые в предыдущих главах не употреблялись.

Опасный код

C# позволяет программистам писать то, что называется "опасный кодом" (unsafe code). Опасный код — это код, который не плохо написан, а код, который не выполняется под полным управлением системы Common Language Runtime (CLR). Как разъяснялось в главе 1, язык C# обычно используется для создания управляемого кода. Однако можно написать и "неуправляемый" код, который не подчиняется тем же средствам управления и ограничениям, налагаемым на управляемый код. Такой код называется "опасным", поскольку невозможно проконтролировать невыполнение им опасных действий. Таким образом, термин *опасный* не означает, что коду присуща некорректность. Он просто означает возможность выполнения действий, которые не являются предметом управления системы CLR.

Если опасный код способен вызвать проблемы, то зачем, спрашивается, вообще создавать такой код? Дело в том, что управляемый код не допускает использование *указателей*. Если вы знакомы с языками C или C++, то вам должно быть известно, что указатели — это переменные, которые хранят адреса других объектов. Следовательно, указатели в некотором роде подобны ссылкам в C#. Основное различие между ними заключается в том, что указатель может указывать на что угодно в памяти, а ссылка всегда указывает на объект "своего" типа. Но если указатель может указывать на что угодно, возможно неправильное его использование. Кроме того, работая с указателями, можно легко внести в код ошибку, которую будет трудно отыскать. Вот почему C# не поддерживает указатели при создании управляемого кода. Теме не менее указатели существуют, причем для некоторых типов программ (например, системных утилит) они не просто полезны, они — необходимы, и C# позволяет (что поделаешь) программистам создавать их и использовать. Однако все операции с указателями должны быть отмечены как "опасные", поскольку они выполняются вне управляемого контекста.

Объявление и использование указателей в C# происходит аналогично тому, как это делается в C/C++ (если вы знаете, как использовать указатели в C/C++, можете так же работать с ними и в C#). Но помните: особенность C# — создание управляемого кода. Его способность поддерживать неуправляемый код позволяет применять C#-программы к задачам специальной категории. Но такое C#-программирование уже не попадает под определение стандартного. И в самом деле, чтобы скомпилировать неуправляемый код, необходимо использовать опцию компилятора `/unsafe`.

Поскольку указатели составляют сердцевину опасного кода, пожалуй, стоит познакомиться с ними поближе.

Основы использования указателей

Указатели — это переменные, которые хранят адреса других переменных. Например, если x содержит адрес переменной y, то о переменной x говорят, что она "указывает" на y. Поскольку *указатель указывает* на некоторую переменную, значение этой переменной можно получить или изменить посредством указателя. Операции, выполняемые с помощью указателей, часто называют *операциями непрямого доступа.*

Объявление указателя

Переменные-указатели (или переменные типа указатель) должны быть объявлены таковыми. Формат объявления переменной-указателя таков:

тип имя_переменной;*

Здесь элемент *тип* означает базовый тип указателя, причем он не должен быть ссылочным. Следовательно, нельзя объявлять указатель на объект класса. Обратите внимание на расположение оператора "звездочка" (*). Он стоит после имени типа. Элемент *имя_переменной* представляет собой имя переменной-указателя.

Рассмотрим пример. Чтобы объявить переменную ip указателем на int-значение, используйте следующую инструкцию:

```
int* ip;
```

Для объявления указателя на float-значение используйте такую инструкцию:

```
float* fp;
```

В общем случае использование символа "звездочка" (*) в инструкции объявления после имени типа создает тип указателя.

Тип данных, на которые будет указывать указатель, определяется его базовым типом. Следовательно, в предыдущих примерах переменную ip можно использовать для указания на int-значение, а переменную fp — на float-значение. Однако помните: не существует реального средства, которое могло бы помешать указателю указывать на "бог-знает-что". Вот потому-то указатели потенциально опасны.

Если к C# вы пришли от C/C++, то должны понять важное различие между способами объявления указателей в C# и C/C++. При объявлении типа указателя в C/C++ оператор "*" *не* распространяется на весь список переменных, участвующих в объявлении. Поэтому в C/C++ при выполнении инструкции

```
int* p, q;
```

объявляется указатель p на int-значение и int-переменная с именем q. Эта инструкция эквивалентна следующим двум объявлениям:

```
int* p;
int q;
```

Однако в C# оператор "*" распространяется на все объявление, и поэтому при выполнении инструкции

```
int* p, q;
```

создаются два указателя p и q на int-значения. Таким образом, в C# предыдущая инструкция эквивалентна таким двум объявлениям:

```
int* p;
int* q;
```

Это важное различие обязательно следует иметь в виду при переводе C/C++-кода на "рельсы" C#.

Операторы "*" и "&"

С указателями используются два оператора: "*" и "&". Оператор "&" — унарный. Он возвращает адрес памяти, по которому расположен его операнд. (Вспомните: унарный оператор требует только одного операнда.) Например, при выполнении следующего фрагмента кода

```
int* ip;
int num = 10;
ip = &num;
```

в переменную ip помещается адрес переменной num. Этот адрес соответствует области во внутренней памяти компьютера, которая принадлежит переменной num. Выполнение последней инструкции никак не повлияло на значение переменной num. Итак, переменная ip содержит не значение 10 (начальное значение переменной num), а адрес, по которому оно хранится. Назначение оператора "&" можно "перевести" на русский язык как "адрес переменной", перед которой он стоит. Следовательно, последнюю инструкцию присваивания можно выразить так: "переменная ip получает адрес переменной num".

Второй оператор работы с указателями (*) служит дополнением к первому (&). Это также унарный оператор, но он обращается к значению переменной, расположенной по адресу, заданному его операндом. Другими словами, он указывает на значение переменной, адресуемой заданным указателем. Если (продолжая работу с предыдущим фрагментом кода) переменная ip содержит адрес переменной num, то при выполнении инструкции

```
int val = *ip;
```

переменной val будет присвоено значение 10, являющееся значением переменной num, на которую указывает переменная ip. Назначение оператора "*" можно выразить словосочетанием "по адресу". В данном случае предыдущую инструкцию можно прочитать так: "переменная val получает значение (расположенное) по адресу ip".

Оператор "*" также можно использовать с левой стороны от оператора присваивания. В этом случае он устанавливает значение, адресуемое заданным указателем. Например, при выполнении инструкции

```
*ip = 100;
```

число 100 присваивается переменной, адресуемой указателем ip (в данном случае имеется в виду переменная num). Таким образом, эту инструкцию можно прочитать так: "по адресу ip помещаем значение 100".

Использование ключевого слова unsafe

Код, в котором используются указатели, должен быть отмечен как "опасный" с помощью ключевого слова unsafe. Так можно отмечать отдельные инструкции и методы целиком. Например, рассмотрим программу, в методе Main() которой используются указатели, и поэтому весь метод отмечен словом unsafe.

```
// Демонстрация использования указателей и
// ключевого слова unsafe.

using System;

class UnsafeCode {
  // Отмечаем метод Main() как "опасный".
  unsafe public static void Main() {
    int count = 99;
```

```
    int* p; // Создаем указатель на int-значение.

    p = &count; // Помещаем адрес переменной count
                // в указатель p.

    Console.WriteLine(
           "Начальное значение переменной count равно " +
           *p);

    *p = 10; // Присваиваем значение 10 переменной count
             // через указатель p.

    Console.WriteLine(
          "Новое значение переменной count равно " + *p);
  }
}
```

Вот результаты выполнения этой программы:

```
Начальное значение переменной count равно 99
Новое значение переменной count равно 10
```

Использование модификатора `fixed`

При работе с указателями зачастую используется модификатор `fixed`. Он предотвращает удаление управляемых переменных системой сбора мусора. Это необходимо в том случае, если, например, указатель ссылается на какое-нибудь поле в объекте класса. Поскольку указатель "ничего не знает" о действиях "сборщика мусора", то в случае удаления такого объекта этот указатель будет указывать на неверный объект. Формат применения модификатора `fixed` таков:

```
fixed (type* p = &var) {
  // Использование зафиксированного объекта.
}
```

Здесь элемент *p* — указатель, которому присваивается адрес переменной. Объект будет оставаться в текущей области памяти до тех пор, пока не выполнится соответствующий блок кода. Инструкция `fixed` может включать вместо блока кода единственную инструкцию. Ключевое слово `fixed` можно использовать только в контексте "опасного кода". Используя список элементов, разделенных запятыми, можно объявить сразу несколько фиксированных указателей.

Рассмотрим пример использования модификатора `fixed`:

```
// Демонстрация использования модификатора fixed.

using System;

class Test {
  public int num;
  public Test(int i) { num = i; }
}

class FixedCode {
  // Отмечаем метод Main() как опасный.
  unsafe public static void Main() {
    Test o = new Test(19);

    fixed (int* p = &o.num) { // Используем модификатор
                              // fixed, чтобы поместить
                              // адрес поля o.num в p.
```

```
        Console.WriteLine(
             "Начальное значение поля o.num равно " + *p);

        *p = 10; // Присваиваем число 10 переменной count
                 // через указатель p.

        Console.WriteLine(
                 "Новое значение поля o.num равно " + *p);
    }
  }
}
```

При выполнении этой программы получены такие результаты:

```
Начальное значение поля o.num равно 19
Новое значение поля o.num равно 10
```

Здесь модификатор `fixed` защищает объект `o` от удаления. Поскольку `p` указывает на поле `o.num`, то в случае удаления объекта `o` указатель `p` некорректно ссылался бы на область памяти.

Доступ к членам структур с помощью указателей

Указатель может ссылаться на объект структурного типа, если он не содержит ссылочных типов. При доступе к члену структуры посредством указателя необходимо использовать оператор "стрелка" (->), а не оператор "точка" (.). Рассмотрим, например, следующую структуру:

```
struct MyStruct {
  public int x;
  public int y;
  public int sum() { return x + y; }
}
```

Теперь покажем, как получить доступ к ее членам с помощью указателя:

```
MyStruct o = new MyStruct();
MyStruct* p; // Объявляем указатель.

p = &o;
p->x = 10;
p->y = 20;

Console.WriteLine("Сумма равна " + p->sum());
```

Арифметические операции над указателями

С указателями можно использовать только четыре арифметических оператора: ++, --, + и -. Чтобы лучше понять, что происходит при выполнении арифметических действий с указателями, начнем с примера. Пусть `p1` — указатель на `int`-переменную с текущим значением 2 000 (т.е. `p1` содержит адрес 2 000). После выполнения выражения

```
p1++;
```

содержимое указателя `p1` станет равным 2 004, а не 2 001! Дело в том, что при каждом инкрементировании указатель `p1` будет указывать на следующее `int`-значение. Поскольку в C# `int`-значения занимают четыре байта, то при инкрементировании `p1` его значение увеличивается на 4. Для операции декрементирования справедливо обратное утверждение, т.е. при каждом декрементировании значение `p1` будет уменьшаться на 4. Например, после выполнения инструкции

```
p1--;
```

указатель p1 будет иметь значение 1 996, если до этого оно было равно 2 000.

Итак, каждый раз, когда указатель инкрементируется, он будет указывать на область памяти, содержащую следующий элемент базового типа этого указателя. А при каждом декрементировании он будет указывать на область памяти, содержащую предыдущий элемент базового типа этого указателя.

Арифметические операции над указателями не ограничиваются использованием операторов инкремента и декремента. Со значениями указателей можно выполнять операции сложения и вычитания, используя в качестве второго операнда целочисленные значения. Выражение

```
p1 = p1 + 9;
```

заставляет p1 указывать на девятый элемент базового типа указателя p1 относительно элемента, на который p1 указывал до выполнения этой инструкции.

Несмотря на то что складывать указатели нельзя, один указатель можно вычесть из другого (в предположении, что они оба имеют один и тот же базовый тип). Разность покажет количество элементов базового типа, которые разделяют эти два указателя.

Помимо сложения (и вычитания) указателя и (из) целочисленного значения, а также вычитания двух указателей, над указателями никакие другие арифметические операции не выполняются. Например, с указателями нельзя складывать float- или double-значения.

Чтобы понять, как формируется результат выполнения арифметических операций над указателями, выполним следующую короткую программу. Она выводит реальные физические адреса, которые содержат указатель на int-значение (ip) и указатель на float-значение (fp). Обратите внимание на каждое изменение адреса (зависящее от базового типа указателя), которое происходит при повторении цикла.

```
// Демонстрируем результаты выполнения арифметических
// операций над указателями.

using System;

class PtrArithDemo {
  unsafe public static void Main() {
    int x;
    int i;
    double d;

    int* ip = &i;
    double* fp = &d;

    Console.WriteLine("int     double\n");

    for(x=0; x < 10; x++) {
      Console.WriteLine((uint) (ip) + " " +
                        (uint) (fp));
      ip++;
      fp++;
    }
  }
}
```

Ниже показаны возможные результаты выполнения этой программы. Ваши результаты могут отличаться от приведенных, но интервалы между значениями должны быть такими же.

```
int     double

1243324 1243328
1243328 1243336
1243332 1243344
1243336 1243352
1243340 1243360
1243344 1243368
1243348 1243376
1243352 1243384
1243356 1243392
1243360 1243400
```

Как подтверждают результаты выполнения этой программы, арифметические операции над указателями выполняются в зависимости от базового типа каждого указателя. Поскольку любое `int`-значение занимает четыре байта, а `double`-значение — восемь, то и сами адреса изменяются с учетом этих значений.

Сравнение указателей

Указатели можно сравнивать, используя операторы отношения ==, < и >. Однако для того чтобы результат сравнения указателей поддавался интерпретации, сравниваемые указатели должны быть каким-то образом связаны. Например, если `p1` и `p2` — указатели, которые указывают на две отдельные и никак не связанные переменные, то любое сравнение `p1` и `p2` в общем случае не имеет смысла. Но если `p1` и `p2` указывают на переменные, между которыми существует некоторая связь (как, например, между элементами одного и того же массива), то результат сравнения указателей `p1` и `p2` может иметь определенный смысл.

Рассмотрим пример, в котором сравнение указателей используется для отыскания среднего элемента массива.

```
// Демонстрация возможности сравнения указателей.

using System;

class PtrCompDemo {
  unsafe public static void Main() {

    int[] nums = new int[11];
    int x;

    // Находим средний элемент массива.
    fixed (int* start = &nums[0]) {
      fixed(int* end = &nums[nums.Length-1]) {
        for(x=0; start+x <= end-x; x++) ;
      }
    }
    Console.WriteLine(
              "Средний элемент массива имеет номер " + x);
  }
}
```

Вот как выглядят результаты выполнения этой программы:

```
Средний элемент массива имеет номер 6
```

Эта программа находит средний элемент, первоначально установив указатель `start` равным адресу первого элемента, а указатель `end` — адресу последнего элемента массива. Затем, используя возможности выполнения арифметических операций над указателями, мы увеличиваем указатель `start` на целочисленное значение `x`, а указа-

тель end — уменьшаем на то же значение x до тех пор, пока результат суммирования start и x не станет меньше или равным результату вычитания end и x.

Одно уточнение: указатели start и end должны быть созданы внутри fixed-инструкции, поскольку они указывают на элементы массива, который представляет собой ссылочный тип данных. Не забывайте, что в C# массивы реализованы как объекты и могут быть удалены сборщиком мусора.

Указатели и массивы

В C# указатели и массивы связаны между собой. Например, имя массива без индекса образует указатель на начало этого массива. Рассмотрим следующую программу:

```
/* Имя массива без индекса образует указатель на начало
   этого массива. */

using System;

class PtrArray {
  unsafe public static void Main() {
    int[] nums = new int[10];

    fixed(int* p = &nums[0], p2 = nums) {
      if(p == p2)
        Console.WriteLine(
          "Указатели p и p2 содержат один и тот же адрес.");
    }
  }
}
```

Вот какие результаты получены при выполнении этой программы:

```
Указатели p и p2 содержат один и тот же адрес.
```

Как подтверждают результаты выполнения этой программы, выражение

```
&nums[0]
```

эквивалентно

```
nums
```

Поскольку вторая форма короче, большинство программистов используют именно ее в случае, когда нужен указатель на начало массива.

Индексация указателя

Указатель, который ссылается на массив, можно индексировать так, как если бы это было имя массива. Этот синтаксис обеспечивает альтернативу арифметическим операциям над указателями, поскольку он более удобен в некоторых ситуациях. Рассмотрим пример.

```
// Индексирование указателя подобно массиву.

using System;

class PtrIndexDemo {
  unsafe public static void Main() {
    int[] nums = new int[10];

    // Индексируем указатель.
    Console.WriteLine(
             "Индексируем указатель подобно массиву.");
```

```
    fixed (int* p = nums) {
      for(int i=0; i < 10; i++)
        p[i] = i; // Индексируем указатель подобно массиву.

      for(int i=0; i < 10; i++)
        Console.WriteLine("p[{0}]: {1} ", i, p[i]);
    }

    // Используем арифметические операции над указателями.
    Console.WriteLine(
   "\nИспользуем арифметические операции над указателями.");
    fixed (int* p = nums) {
      for(int i=0; i < 10; i++)
        *(p+i) = i; // Используем арифметические
                    // операции над указателями.

      for(int i=0; i < 10; i++)
        Console.WriteLine("*(p+{0}): {1} ", i, *(p+i));
    }
  }
}
```

Вот результаты выполнения этой программы:

```
Индексируем указатель подобно массиву.
p[0]: 0
p[1]: 1
p[2]: 2
p[3]: 3
p[4]: 4
p[5]: 5
p[6]: 6
p[7]: 7
p[8]: 8
p[9]: 9

Используем арифметические операции над указателями.
*(p+0): 0
*(p+1): 1
*(p+2): 2
*(p+3): 3
*(p+4): 4
*(p+5): 5
*(p+6): 6
*(p+7): 7
*(p+8): 8
*(p+9): 9
```

Как видно по результатам выполнения этой программы, выражение (в котором участвует указатель) в формате

```
*(ptr + i)
```

можно переписать с использованием синтаксиса, применяемого при индексировании массивов:

```
ptr[i]
```

При индексировании указателя необходимо помнить следующее. Во-первых, при этом нарушение границ массива никак не контролируется. Следовательно, существует возможность получить доступ к "элементу" за концом массива, если на него ссылается указатель. Во-вторых, указатель не имеет свойства `Length`. Поэтому при использовании указателя невозможно узнать длину массива.

Указатели и строки

Несмотря на то что в C# строки реализованы как объекты, к отдельным их символам можно получить доступ с помощью указателя. Для этого достаточно присвоить `char*`-указателю адрес начала этой строки, используя `fixed`-инструкцию:

```
fixed(char* p = str) { // ...
```

После выполнения такой `fixed`-инструкции `p` будет указывать на начало символьного массива, который составляет эту строку. Этот символьный массив заканчивается символом конца строки, т.е. нулевым символом. Этот факт можно использовать для проверки достижения конца массива. В C/C++ символьные строки реализованы в виде символьных массивов, завершающихся нулевым символом. Таким образом, получив `char*`-указатель на строку, можно обрабатывать строки практически так же, как это делается в C/C++.

Рассмотрим программу, которая демонстрирует доступ к строке с помощью `char*`-указателя:

```
// Использование fixed-инструкций для получения
// указателя на начало строки.

using System;

class FixedString {
  unsafe public static void Main() {
    string str = "Это простой тест.";

    // Направляем указатель p на начало строки str.
    fixed(char* p = str) {

      // Отображаем содержимое строки str
      // с помощью указателя p.
      for(int i=0; p[i] != 0; i++)
        Console.Write(p[i]);
    }

    Console.WriteLine();

  }
}
```

Вот результаты выполнения этой программы:

```
Это простой тест.
```

Использование многоуровневой непрямой адресации

Можно создать указатель, который будет ссылаться на другой указатель, а тот — на конечное значение. Эту ситуацию называют *многоуровневой непрямой адресацией* (multiple indirection), или использованием указателя на указатель. Идея многоуровневой непрямой адресации схематично проиллюстрирована на рис. 18.1. Как видите, значение обычного указателя (при одноуровневой непрямой адресации) представляет собой адрес переменной, которая содержит некоторое значение. В случае применения указателя на указатель первый содержит адрес второго, а тот указывает на переменную, содержащую определенное значение.

При использовании непрямой адресации можно организовать любое желаемое количество уровней, но, как правило, ограничиваются лишь двумя, поскольку увеличение числа уровней часто ведет к возникновению концептуальных ошибок.

Переменную, которая является указателем на указатель, нужно объявить соответствующим образом. Для этого достаточно поставить дополнительный символ "звездочка"(*) после имени типа. Например, следующее объявление сообщает компилятору о том, что q — это указатель на указатель на значение типа int:

```
int** q;
```

Необходимо помнить, что переменная q здесь — не указатель на целочисленное значение, а указатель на указатель на int-значение.

Чтобы получить доступ к значению, адресуемому указателем на указатель, необходимо дважды применить оператор "*", как показано в следующем примере:

```
using System;

class MultipleIndirect {
  unsafe public static void Main() {
    int x;   // Переменная содержит значение.
    int* p;  // Переменная содержит указатель
             // на int-значение.
    int** q; // Переменная содержит указатель на указатель
             // на int-значение.

    x = 10;
    p = &x; // Помещаем адрес x в p.
    q = &p; // Помещаем адрес p в q.

    Console.WriteLine(**q); // Отображаем значение x.
  }
}
```

При выполнении этой программы мы получили бы значение переменной x, т.е. число 10. Здесь переменная p объявлена как указатель на int-значение, а переменная q — как указатель на указатель на int-значение.

И еще: не следует путать многоуровневую непрямую адресацию с такими высокоуровневыми структурами данных, как связные списки, которые используют указатели. Это — две принципиально различные концепции.

Рис. 18.1. Одноуровневая и многоуровневая непрямая адресация

Массивы указателей

Указатели, подобно данным других типов, могут храниться в массивах. Вот, например, как выглядит объявление трехэлементного массива указателей на int-значения:

```
int * [] ptrs = new int * [3];
```

Чтобы присвоить адрес int-переменной с именем var третьему элементу этого массива указателей, запишите следующее:

```
ptrs[2] = &var;
```

Чтобы получить значение переменной var, используйте такой синтаксис:

```
*ptrs[2]
```

Ключевые слова смешанного типа

В заключение части I рассмотрим определенные в C# ключевые слова, которые еще не были здесь описаны.

sizeof

Не исключено, что вам понадобится узнать размер (в байтах) одного из C#-типов значений. Для получения этой информации используйте оператор sizeof. Формат его применения таков:

```
sizeof(тип)
```

Здесь элемент *тип* — это тип, размер которого мы хотим получить. Оператор sizeof можно использовать только в контексте опасного (unsafe) кода. Таким образом, он предназначен в основном для специальных ситуаций, особенно при работе со смешанным (управляемым и неуправляемым) кодом.

lock

Ключевое слово lock используется при работе с несколькими потоками. В C# программа может содержать два или больше потоков выполнения. В этом случае программы работают в многозадачном режиме. Другими словами, отдельные фрагменты программ выполняются не только независимо один от другого, или, можно сказать, одновременно. Это приводит к возникновению определенной проблемы: а что если два потока попытаются использовать ресурс, который одновременно может использовать только один поток? Чтобы решить эту проблему, можно создать *критический раздел кода*, который в данный момент может выполняться только одним потоком. Такой подход реализуется с помощью ключевого слова lock. Его формат таков:

```
lock(obj) {
  // Критический раздел.
}
```

Здесь элемент *obj* представляет объект, который стремится получить блокировку. Если один поток уже вошел в критический раздел, то второй должен ожидать до тех пор, пока не выполнится первый. Критический раздел может быть выполнен при получении разрешения на установку блокировки. (Подробнее см. главу 21.)

readonly

В классе можно создать поле, предназначенное только для чтения, если объявить его с помощью ключевого слова readonly, причем readonly-поле можно инициализировать, но после этого изменить его содержимое уже нельзя. Следовательно, использование readonly-полей — удобный способ создать константы (например, обозначающие размерность массивов), которые применяются на протяжении всей про-

граммы. Предназначенные только для чтения поля могут быть как статическими, так и нестатическими.

Рассмотрим пример создания readonly-поля:

```
// Демонстрация использования readonly-поля.

using System;

class MyClass {
  public static readonly int SIZE = 10;
}

class DemoReadOnly {
  public static void Main() {
    int[]nums = new int[MyClass.SIZE];

    for(int i=0; i<MyClass.SIZE; i++)
      nums[i] = i;

    foreach(int i in nums)
      Console.Write(i + " ");

    // MyClass.SIZE = 100; // Ошибка!!! readonly-поле
                           // изменять нельзя!
  }
}
```

Здесь поле MyClass.SIZE инициализируется значением 10. После этого его можно использовать, но не изменять. Чтобы убедиться в этом, попытайтесь удалить символ комментария, стоящий в начале последней строки, и скомпилировать программу. Вы получите сообщение об ошибке.

stackalloc

С помощью ключевого слова stackalloc можно использовать память стека. Это можно делать только при инициализации локальных переменных. Распределение памяти стека имеет следующий формат:

тип *p = stackalloc *тип*[*размер*]

Здесь *p* — указатель, который получает адрес области памяти достаточно большого размера, чтобы хранить *размер* объектов типа *тип*. Ключевое слово stackalloc можно использовать только в контексте опасного (unsafe) кода.

Обычно память для объектов выделяется из кучи, которая представляет собой область свободной памяти. Выделение памяти из области стека — своего рода исключение. Переменные, размещенные в области стека, не обрабатываются сборщиком мусора. Но они существуют только при выполнении блока, где были объявлены. По завершении выполнения блока эта память освобождается. Единственное достоинство использования ключевого слова stackalloc — можно не беспокоиться о том, что соответствующая область памяти попадет под "метлу" сборщика мусора.

Рассмотрим пример использования ключевого слова stackalloc:

```
// Демонстрация использования ключевого слова stackalloc.

using System;

class UseStackAlloc {
  unsafe public static void Main() {
    int* ptrs = stackalloc int[3];
```

```
    ptrs[0] = 1;
    ptrs[1] = 2;
    ptrs[2] = 3;

    for(int i=0; i < 3; i++)
      Console.WriteLine(ptrs[i]);
  }
}
```

Результаты выполнения этой программы таковы:

```
1
2
3
```

Инструкция `using`

Ключевое слово `using`, применение которого в качестве директивы описано выше, имеет и второй вариант использования, а именно в качестве инструкции `using`. В этом случае возможны следующие две формы:

```
using (obj) {
  // Использование объекта obj.
}

using (type obj = инициализатор) {
  // Использование объекта obj.
}
```

Здесь элемент *obj* представляет объект, используемый внутри блока `using`. В первой форме этот объект объявляется вне `using`-инструкции, а во второй — внутри. При завершении блока для объекта *obj* вызывается метод `Dispose()` (определенный в интерфейсе `System.IDisposable`). Инструкция `using` применяется только к объектам, которые реализованы в интерфейсе `System.IDisposable`.

Рассмотрим пример использования каждой формы инструкция `using`:

```
// Демонстрация использования инструкции using.

using System;
using System.IO;

class UsingDemo {
  public static void Main() {
    StreamReader sr = new StreamReader("test.txt");

    // Используем объект внутри инструкции using.
    using(sr) {
      Console.WriteLine(sr.ReadLine());
      sr.Close();
    }

    // Создаем StreamReader-объект внутри инструкции using.
    using(StreamReader sr2 = new StreamReader("test.txt")) {
      Console.WriteLine(sr2.ReadLine());
      sr2.Close();
    }
  }
}
```

Класс `StreamReader` реализует интерфейс `IDisposable` (через свой базовый класс `TextReader`). Следовательно, его можно использовать в `using`-инструкции. (Описание интерфейса `IDisposable` см. в главе 24.)

Модификаторы `const` и `volatile`

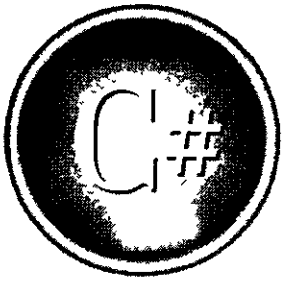
Модификатор `const` используется для объявления полей или локальных переменных, которые не должны изменяться. Этим переменным необходимо присвоить начальные значения при объявлении. Таким образом, `const`-переменная является по сути константой. Например, при выполнении инструкции

```
const int i = 10;
```

создается `const`-переменная `i`, которая имеет значение 10.

Модификатор `volatile` сообщает компилятору о том, что значение соответствующего поля может быть изменено в программе неявным образом. Например, поле, содержащее текущее значение системного времени, может обновляться операционной системой автоматически. В этой ситуации содержимое такого поля изменяется без явного выполнения инструкции присваивания. Причина внешнего изменения поля может быть очень важной. Дело в том, что C#-компилятору разрешается оптимизировать определенные выражения автоматически в предположении, что содержимое поля остается неизменным, если оно не находится в левой части инструкции присваивания. Но если некоторые факторы, не относящиеся к текущему коду (например, связанные с обработкой второго потока выполнения), изменят значение этого поля, такое предположение окажется неверным. Использование модификатора `volatile` позволяет сообщить компилятору о том, что он должен получать значение этого поля при каждом обращении к нему.

Полный справочник по



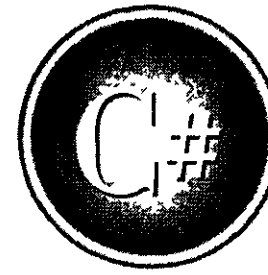
Часть II

Библиотека C#

Часть II посвящена описанию библиотеки C#. Как упоминалось в части I, используемая в C# библиотека классов является по сути библиотекой .NET Framework. Таким образом, материал этого раздела применим не только к языку C#, но и в целом к среде .NET Framework.

Библиотека C# организована с использованием пространств имен. Для работы с какой-либо ее частью с помощью директивы using импортируется соответствующее пространство имен. Конечно, можно также указывать полное имя элемента, т.е. сопровождать его названием пространства имен, но легче импортировать (так чаще всего и поступают) само пространство имен.

Полный справочник по



Глава 19

Пространство имен System

Эта глава посвящена пространству имен System, которое занимает наивысший уровень в библиотеке C#. Оно содержит классы, структуры, интерфейсы и перечисления, которые наиболее часто употребляются в C#-программах или считаются важной составляющей среды .NET Framework. Таким образом, пространство имен System определяет ядро библиотеки C#.

Пространство имен System также содержит множество вложенных пространств имен, предназначенных для поддержки таких подсистем, как System.Net. Некоторые из них описаны ниже в этой книге. Однако в этой главе рассматриваются только члены самого пространства имен System.

Члены пространства имен System

Помимо большого количества классов исключений, пространство имен System содержит следующие классы:

Activator	AppDomain	AppDomainSetup
Array	AssemblyLoadEventArgs	Attribute
AttributeUsageAttribute	BitConverter	Buffer
CharEnumerator	CLSCompliantAttribute	Console
ContextBoundObject	ContextStaticAttribute	Convert
DBNull	Delegate	Enum
Environment	EventArgs	Exception
FlagsAttribute	GC	LoaderOptimizationAttribute
LocalDataStoreSlot	MarshalByRefObject	Math
MTAThreadAttribute	MulticastDelegate	NonSerializedAttribute
Object	ObsoleteAttribute	OperatingSystem
ParamArrayAttribute	Random	ResolveEventArgs
SerializableAttribute	STAThreadAttribute	String
ThreadStaticAttribute	TimeZone	Type
UnhandledExceptionEventArgs	Uri	UriBuilder
ValueType	Version	WeakReference

В пространстве имен System определены такие структуры:

ArgIterator	Boolean	Byte
Char	DateTime	Decimal
Double	Guid	Int16
Int32	Int64	IntPtr
RuntimeArgumentHandle	RuntimeFieldHandle	RuntimeMethodHandle
RuntimeTypeHandle	SByte	Single
TimeSpan	TypedReference	UInt16
UInt32	UInt64	UIntPtr
Void		

В пространстве имен System определены следующие интерфейсы:

IAppDomainSetup	IAsyncResult	ICloneable
IComparable	IConvertible	ICustomFormatter
IDisposable	IFormatProvider	IFormattable
IServiceProvider		

В пространстве имен `System` определены такие делегаты:

`AssemblyLoadEventHandler`	`AsyncCallback`	`CrossAppDomainDelegate`
`EventHandler`	`ResolveEventHandler`	`UnhandledExceptionEventHandler`

В пространстве имен `System` определены следующие перечисления:

`AttributeTargets`	`DayOfWeek`	`Environment.SpecialFolder`
`LoaderOptimization`	`PlatformID`	`TypeCode`
`UriHostNameType`	`UriPartial`	

Как видно по приведенным выше таблицам, пространство имен `System` отличается довольно большим объемом, и все его составляющие невозможно детально рассмотреть в одной главе. Более того, хотя некоторые члены `System` в общем случае применимы к среде .NET Framework, но C#-программистами они обычно не используются. Следует также отметить, что некоторые классы пространства имен `System` (например, `Type`, `Exception` и `Attribute`) рассмотрены в части I или в других разделах настоящей книги. А поскольку класс `System.String`, в котором определяется C#-тип `string`, представляет собой очень большую и важную тему, его описание приводится в главе 20 (как тесно связанное с темой форматирования). Поэтому в настоящей главе описаны только те члены пространства имен `System`, которые пользуются повышенным вниманием у C#-программистов и не упоминаются в других разделах книги.

Класс `Math`

В классе `Math` определены такие стандартные математические операции, как извлечение квадратного корня, вычисление синуса, косинуса и логарифмов. Методы, определенные в классе `Math`, перечислены в табл. 19.1. Все углы задаются в радианах. Обратите внимание на то, что все методы, определенные в классе `Math`, являются `static`-методами. Поэтому для их использования не нужно создавать объект класса `Math`, а значит, нет необходимости и в конструкторах класса `Math`.

В классе `Math` также определены следующие два поля:

```
public const double E
public const double PI
```

где `E` — значение основания натурального логарифма, известное как *e*, а `PI` — значение иррационального числа pi.

`Math` является `sealed`-классом, т.е. он не может иметь производных классов.

Таблица 19.1. Методы, определенные в классе `Math`

Метод	*Описание*
`public static double Abs(double v)`	Возвращает абсолютную величину параметра `v`
`public static float Abs(float v)`	Возвращает абсолютную величину параметра `v`
`public static decimal Abs(decimal v)`	Возвращает абсолютную величину параметра `v`
`public static int Abs(int v)`	Возвращает абсолютную величину параметра `v`
`public static short Abs(short v)`	Возвращает абсолютную величину параметра `v`
`public static long Abs(long v)`	Возвращает абсолютную величину параметра `v`
`public static sbyte Abs(sbyte v)`	Возвращает абсолютную величину параметра `v`

Метод	Описание
`public static double Acos(double v)`	Возвращает арккосинус параметра *v*. Значение *v* должно находиться в диапазоне между -1 и 1
`public static double Asin(double v)`	Возвращает арксинус параметра *v*. Значение *v* должно находиться в диапазоне между -1 и 1
`public static double Atan(double v)`	Возвращает арктангенс параметра *v*
`public static double Atan2(double y, double x)`	Возвращает арктангенс частного *y*/*x*
`public static double Ceiling(double v)`	Возвращает наименьшее целое (представленное в виде значения с плавающей точкой), которое не меньше параметра *v*. Например, при *v*, равном 1.02, метод `Ceiling()` возвратит 2.0. А при *v*, равном -1.02, метод `Ceiling()` возвратит -1
`public static double Cos(double v)`	Возвращает косинус параметра *v*
`public static double Cosh(double v)`	Возвращает гиперболический косинус параметра *v*
`public static double Exp(double v)`	Возвращает основание натурального логарифма e, возведенное в степень *v*
`public static double Floor(double v)`	Возвращает наибольшее целое (представленное в виде значения с плавающей точкой), которое не больше параметра *v*. Например, при *v*, равном `1.02`, метод `Floor()` возвратит `1.0`. А при *v*, равном `-1.02`, метод `Floor()` возвратит `-2`
`public static double IEEERemainder(double dividend, double divisor)`	Возвращает остаток от деления *dividend*/*divisor*
`public static double Log(double v)`	Возвращает натуральный логарифм для параметра *v*
`public static double Log(double v, double base)`	Возвращает логарифм для параметра *v* по основанию *base*
`public static double Log10(double v)`	Возвращает логарифм для параметра *v* по основанию 10
`public static double Max(double v1, double v2)`	Возвращает большее из значений *v1* и *v2*
`public static float Max(float v1, float v2)`	Возвращает большее из значений *v1* и *v2*
`public static decimal Max(decimal v1, decimal v2)`	Возвращает большее из значений *v1* и *v2*
`public static int Max(int v1, int v2)`	Возвращает большее из значений *v1* и *v2*
`public static short Max(short v1, short v2)`	Возвращает большее из значений *v1* и *v2*
`public static long Max(long v1, long v2)`	Возвращает большее из значений *v1* и *v2*

Метод	Описание
`public static uint Max(uint v1, uint v2)`	Возвращает большее из значений `v1` и `v2`
`public static ushort Max(ushort v1, ushort v2)`	Возвращает большее из значений `v1` и `v2`
`public static ulong Max(ulong v1, ulong v2)`	Возвращает большее из значений `v1` и `v2`
`public static byte Max(byte v1, byte v2)`	Возвращает большее из значений `v1` и `v2`
`public static sbyte Max(sbyte v1, sbyte v2)`	Возвращает большее из значений `v1` и `v2`
`public static double Min(double v1, double v2)`	Возвращает меньшее из значений `v1` и `v2`
`public static float Min(float v1, float v2)`	Возвращает меньшее из значений `v1` и `v2`
`public static decimal Min(decimal v1, decimal v2)`	Возвращает меньшее из значений `v1` и `v2`
`public static int Min(int v1, int v2)`	Возвращает меньшее из значений `v1` и `v2`
`public static short Min(short v1, short v2)`	Возвращает меньшее из значений `v1` и `v2`
`public static long Min(long v1, long v2)`	Возвращает меньшее из значений `v1` и `v2`
`public static uint Min(uint v1, uint v2)`	Возвращает меньшее из значений `v1` и `v2`
`public static ushort Min(ushort v1, ushort v2)`	Возвращает меньшее из значений `v1` и `v2`
`public static ulong Min(ulong v1, ulong v2)`	Возвращает меньшее из значений `v1` и `v2`
`public static byte Min(byte v1, byte v2)`	Возвращает меньшее из значений `v1` и `v2`
`public static sbyte Min(sbyte v1, sbyte v2)`	Возвращает меньшее из значений `v1` и `v2`
`public static double Pow(double base, double exp)`	Возвращает значение `base`, возведенное в степень `exp` ($base^{exp}$)
`public static double Round(double v)`	Возвращает значение `v`, округленное до ближайшего целого числа
`public static decimal Round(decimal v)`	Возвращает значение `v`, округленное до ближайшего целого числа

Метод	Описание
`public static double Round(double v, int frac)`	Возвращает значение `v`, округленное до числа, количество цифр дробной части которого равно значению параметра `frac`
`public static decimal Round(decimal v, int frac)`	Возвращает значение `v`, округленное до числа, количество цифр дробной части которого равно значению параметра `frac`
`public static int Sign(double v)`	Возвращает -1, если значение `v` меньше нуля, 0, если `v` равно нулю, и 1, если `v` больше нуля
`public static int Sign(float v)`	Возвращает -1, если значение `v` меньше нуля, 0, если `v` равно нулю, и 1, если `v` больше нуля
`public static int Sign(decimal v)`	Возвращает -1, если значение `v` меньше нуля, 0, если `v` равно нулю, и 1, если `v` больше нуля
`public static int Sign(int v)`	Возвращает -1, если значение `v` меньше нуля, 0, если `v` равно нулю, и 1, если `v` больше нуля
`public static int Sign(short v)`	Возвращает -1, если значение `v` меньше нуля, 0, если `v` равно нулю, и 1, если `v` больше нуля
`public static int Sign(long v)`	Возвращает -1, если значение `v` меньше нуля, 0, если `v` равно нулю, и 1, если `v` больше нуля
`public static int Sign(sbyte v)`	Возвращает -1, если значение `v` меньше нуля, 0, если `v` равно нулю, и 1, если `v` больше нуля
`public static double Sin(double v)`	Возвращает синус параметра `v`
`public static double Sinh(double v)`	Возвращает гиперболический синус параметра `v`
`public static double Sqrt(double v)`	Возвращает квадратный корень параметра `v`
`public static double Tan(double v)`	Возвращает тангенс параметра `v`
`public static double Tanh(double v)`	Возвращает гиперболический тангенс параметра `v`

Рассмотрим пример программы, в которой используется метод `Sqrt()`, позволяющий применить теорему Пифагора. Здесь вычисляется длина гипотенузы по заданным длинам двух остальных сторон (катетов) прямоугольного треугольника.

```
// Реализация теоремы Пифагора.

using System;

class Pythagorean {
  public static void Main() {
    double s1;
    double s2;
    double hypot;
    string str;

    Console.WriteLine("Введите длину первого катета: ");
    str = Console.ReadLine();
    s1 = Double.Parse(str);
```

```
    Console.WriteLine("Введите длину второго катета: ");
    str = Console.ReadLine();
    s2 = Double.Parse(str);

    hypot = Math.Sqrt(s1*s1 + s2*s2);

    Console.WriteLine("Гипотенуза равна " + hypot);
  }
}
```

Результаты выполнения этой программы таковы:

```
Введите длину первого катета:
3
Введите длину второго катета:
4
Гипотенуза равна 5
```

Теперь рассмотрим пример программы, в которой используется метод `Pow()` для вычисления объема начального капиталовложения, необходимого для достижения желаемой будущей стоимости при заданных годовом показателе ожидаемого дохода и количестве лет. Формула вычисления объема начального капиталовложения имеет следующий вид:

$$\text{InitialInvestment} = \text{FutureValue} / (1 + \text{InterestRate})^{\text{Years}}$$

Поскольку метод `Pow()` принимает аргументы типа `double`, процентная ставка и количество лет представляются в виде `double`-значений. Для значений будущей стоимости и объема начального капиталовложения используется тип `decimal`.

```
/* Вычисление объема начального капиталовложения,
   необходимого для достижения известной будущей
   стоимости при заданных годовом показателе ожидаемого
   дохода и количестве лет. */

using System;

class IntialInvestment {
  public static void Main() {
    decimal InitInvest; // начальное капиталовложение
    decimal FutVal;     // будущая стоимость

    double NumYears;    // количество лет
    double IntRate;     // годовой показатель
                        // ожидаемого дохода

    string str;

    Console.Write("Введите значение будущей стоимости: ");
    str = Console.ReadLine();
    try {
      FutVal = Decimal.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    Console.Write(
        "Введите процентную ставку (например, 0.085): ");
    str = Console.ReadLine();
    try {
```

```
      IntRate = Double.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    Console.Write("Введите количество лет: ");
    str = Console.ReadLine();
    try {
      NumYears = Double.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    InitInvest = FutVal / (decimal) Math.Pow(IntRate+1.0,
                                             NumYears);

    Console.WriteLine(
       "Требуемый объем начального капиталовложения: {0:C}",
       InitInvest);
  }
}
```

Результаты выполнения этой программы таковы:

```
Введите значение будущей стоимости: 10000
Введите процентную ставку (например, 0.085): 0.07
Введите количество лет: 10
Требуемый объем начального капиталовложения: $5,083.49
```

Структуры типов значений

Структуры типов значений были представлены в главе 14 в связи с их использованием для преобразования строк, которые содержат числовые значения, удобные для восприятия человеком, в эквивалентные двоичные величины. Здесь же они рассматриваются более подробно.

Структуры типов значений лежат в основе C#-типов значений. Используя члены, определенные этими структурами, можно выполнять операции, разрешенные для определенных типов значений. Ниже показаны .NET-имена структур и их эквиваленты в виде ключевых слов C#:

.NET-имя структуры	*C#-имя*	*.NET-имя структуры*	*C#-имя*
Boolean	bool	Char	char
Decimal	decimal	Double	double
Single	float	Int16	short
Int32	int	Int64	long
UInt16	ushort	UInt32	uint
UInt64	ulong	Byte	byte
SByte	sbyte		

Все эти структуры рассматриваются в следующих разделах.

На заметку

Некоторые методы, определенные в структурах типов значений, принимают параметры типа `IFormatProvider` или `NumberStyles`. Тип `IFormatProvider` кратко описан ниже в этой главе. Тип `NumberStyles` представляет собой перечисление, принадлежащее пространству имен `System.Globalization`. Тема форматирования раскрыта в главе 20.

Структуры целочисленных типов

К структурам целочисленных типов относятся следующие:

```
Byte        SByte       Int16       UInt16
Int32       UInt32      Int64       UInt64
```

Все эти структуры содержат одни и те же методы (см. табл. 19.2), которые отличаются лишь типом значения, возвращаемого методом `Parse()`. Метод `Parse()` возвращает значение типа, представленного соответствующей структурой. Например, для структуры `Int32` метод `Parse()` возвращает значение типа `int`, а для структуры `UInt16` — значение типа `ushort`. (Использование метода `Parse()` продемонстрировано в главе 14.)

Кроме методов, перечисленных в табл. 19.2, структуры целочисленных типов также определяют следующие `const`-поля:

```
MaxValue
MinValue
```

В каждой структуре эти поля содержат наибольшее и наименьшее значения, которые можно хранить с помощью типа, представленного конкретной структурой.

Все структуры целочисленных типов реализуют следующие интерфейсы: `IComparable`, `IConvertible` и `IFormattable`.

Таблица 19.2. Методы, поддерживаемые структурами целочисленных типов

Метод	*Описание*
`public int CompareTo(object v)`	Сравнивает числовое значение вызывающего объекта со значением параметра `v`. Возвращает нуль, если сравниваемые значения равны. Возвращает отрицательное число, если вызывающий объект имеет меньшее значение, и — положительное, если вызывающий объект имеет большее значение
`public override bool Equals( object v)`	Возвращает значение ИСТИНА, если значение вызывающего объекта равно значению параметра `v`
`public override int GetHashCode()`	Возвращает хеш-код для вызывающего объекта
`public TypeCode GetTypeCode()`	Возвращает значение перечисления `TypeCode` для эквивалентного типа. Например, для структуры `Int32` возвращает значение `TypeCode.Int32`
`public static` *тип_возврата* `Parse(string str)`	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре `str`. Если содержимое строки не представляет числовое значение в соответствии с определением типа структуры, генерируется исключение
`public static` *тип_возврата* `Parse(string str, IFormatProvider fmtpvdr)`	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре `str` с использованием форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра `fmtpvdr`. Если содержимое строки не представляет числовое значение в соответствии с определением типа структуры, генерируется исключение

Метод	Описание
`public static тип_возврата Parse(string str, NumberStyles styles)`	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре `str`, с использованием информации стилевого характера, заданной в параметре `styles`. Если содержимое строки не представляет числовое значение в соответствии с определением типа структуры, генерируется исключение
`public static тип_возврата Parse(string str, NumberStyles styles, IFormatProvider fmtpvdr)`	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре `str`, с использованием информации стилевого характера, заданной в параметре `styles`, а также форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра `fmtpvdr`. Если содержимое строки не представляет числовое значение в соответствии с определением типа структуры, генерируется исключение
`public override string ToString()`	Возвращает строковое представление значения вызывающего объекта
`public string ToString(string format)`	Возвращает строковое представление значения вызывающего объекта в соответствии с требованиями форматирующей строки, переданной в параметре `format`
`public string ToString(IFormatProvider fmtpvdr)`	Возвращает строковое представление значения вызывающего объекта с использованием форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра `fmtpvdr`
`public string ToString(string format, IFormatProvider fmtpvdr)`	Возвращает строковое представление значения вызывающего объекта с использованием форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра `fmtpvdr`, а также форматирующей строки, переданной в параметре `format`

Структуры типов данных с плавающей точкой

Определены только две структуры типов данных с плавающей точкой: `Double` и `Single`. Структура `Single` представляет тип `float`. Ее методы перечислены в табл. 19.3, а поля — в табл. 19.4. Структура `Double` представляет тип `double`. Ее методы перечислены в табл. 19.5, а поля — в табл. 19.6. Подобно структурам целочисленного типа, в вызовах методов `Parse()` или `ToString()` можно задавать форматы данных (присущие конкретному естественному языку, диалекту или территориальному образованию), а также форматирующую строку.

Структуры типов данных с плавающей точкой реализуют следующие интерфейсы: `IComparable`, `IConvertible` и `IFormattable`.

Таблица 19.3. Методы, поддерживаемые структурой `Single`

Метод	Описание
`public int CompareTo(object v)`	Сравнивает числовое значение вызывающего объекта со значением параметра `v`. Возвращает нуль, если сравниваемые значения равны. Возвращает отрицательное число, если вызывающий объект имеет меньшее значение, и — положительное, если вызывающий объект имеет большее значение
`public override bool Equals(object v)`	Возвращает значение ИСТИНА, если значение вызывающего объекта равно значению параметра `v`

Метод	Описание
`public override int GetHashCode()`	Возвращает хеш-код для вызывающего объекта
`public TypeCode GetTypeCode()`	Возвращает значение перечисления `TypeCode` для структуры `Single`, т.е. `TypeCode.Single`
`public static bool IsInfinity(` `  float v)`	Возвращает значение ИСТИНА, если значение *v* представляет бесконечность (со знаком "плюс" либо со знаком "минус"). В противном случае возвращает значение ЛОЖЬ
`public static bool IsNaN(float v)`	Возвращает значение ИСТИНА, если значение *v* — не число. В противном случае возвращает значение ЛОЖЬ
`public static bool` `  IsPositiveInfinity(float v)`	Возвращает значение ИСТИНА, если значение *v* представляет бесконечность со знаком "плюс". В противном случае возвращает значение ЛОЖЬ
`public static bool` `  IsNegativeInfinity(float v)`	Возвращает значение ИСТИНА, если значение *v* представляет бесконечность со знаком "минус". В противном случае возвращает значение ЛОЖЬ
`public static float Parse(` `  string str)`	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре *str*. Если содержимое строки не представляет значение типа `float`, генерируется исключение

`public static float Parse(`
`  string str,`
`  IFormatProvider fmtpvdr)`
	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре *str* с использованием форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра *fmtpvdr*. Если содержимое строки не представляет значение типа `float`, генерируется исключение

`public static float Parse(`
`  string str,`
`  NumberStyles styles)`
	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре *str*, с использованием информации стилевого характера, заданной в параметре *styles*. Если содержимое строки не представляет значение типа `float`, генерируется исключение

`public static float Parse(`
`  string str,`
`  NumberStyles styles,`
`  IFormatProvider fmtpvdr)`
	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре *str*, с использованием информации стилевого характера, заданной в параметре *styles*, а также форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра *fmtpvdr*. Если содержимое строки не представляет значение типа `float`, генерируется исключение

`public override string ToString()`	Возвращает строковое представление значения вызывающего объекта
`public string ToString(` `  string format)`	Возвращает строковое представление значения вызывающего объекта в соответствии с требованиями форматирующей строки, переданной в параметре *format*
`public string ToString(` `  IFormatProvider fmtpvdr)`	Возвращает строковое представление значения вызывающего объекта с использованием форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра *fmtpvdr*

`public string ToString(`
`  string format,`
`  IFormatProvider fmtpvdr)`
	Возвращает строковое представление значения вызывающего объекта с использованием форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра *fmtpvdr*, а также форматирующей строки, переданной в параметре *format*

Таблица 19.4. Поля, поддерживаемые структурой `Single`

Поле	*Описание*
`public const float Epsilon`	Наименьшее ненулевое положительное значение
`public const float MaxValue`	Наибольшее значение, которое можно хранить с помощью типа `float`
`public const float MinValue`	Наименьшее значение, которое можно хранить с помощью типа `float`
`public const float NaN`	Значение, которое не является числом
`public const float NegativeInfinity`	Значение, представляющее минус бесконечность
`public const float PositiveInfinity`	Значение, представляющее плюс бесконечность

Таблица 19.5. Методы, поддерживаемые структурой `Double`

Метод	*Описание*
`public int CompareTo(object v)`	Сравнивает числовое значение вызывающего объекта со значением параметра `v`. Возвращает нуль, если сравниваемые значения равны. Возвращает отрицательное число, если вызывающий объект имеет меньшее значение, и — положительное, если вызывающий объект имеет большее значение
`public override bool Equals( object v)`	Возвращает значение ИСТИНА, если значение вызывающего объекта равно значению параметра `v`
`public override int GetHashCode()`	Возвращает хеш-код для вызывающего объекта
`public TypeCode GetTypeCode()`	Возвращает значение перечисления `TypeCode` для структуры `Double`, т.е., `TypeCode.Double`
`public static bool IsInfinity( double v)`	Возвращает значение ИСТИНА, если значение `v` представляет бесконечность (со знаком "плюс" либо со знаком "минус"). В противном случае возвращает значение ЛОЖЬ
`public static bool IsNaN( double v)`	Возвращает значение ИСТИНА, если значение `v` — не число. В противном случае возвращает значение ЛОЖЬ
`public static bool IsPositiveInfinity(double v)`	Возвращает значение ИСТИНА, если значение `v` представляет бесконечность со знаком "плюс". В противном случае возвращает значение ЛОЖЬ
`public static bool IsNegativeInfinity(double v)`	Возвращает значение ИСТИНА, если значение `v` представляет бесконечность со знаком "минус". В противном случае возвращает значение ЛОЖЬ
`public static double Parse( string str)`	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре `str`. Если содержимое строки не представляет значение типа `double`, генерируется исключение
`public static double Parse( string str, IFormatProvider fmtpvdr)`	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре `str` с использованием форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра `fmtpvdr`. Если содержимое строки не представляет значение типа `double`, генерируется исключение
`public static double Parse( string str, NumberStyles styles)`	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре `str`, с использованием информации стилевого характера, заданной в параметре `styles`. Если содержимое строки не представляет значение типа `double`, генерируется исключение

Метод	Описание
`public static double Parse( string str, NumberStyles styles, IFormatProvider fmtpvdr)`	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре `str`, с использованием информации стилевого характера, заданной в параметре `styles`, а также форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра `fmtpvdr`. Если содержимое строки не представляет значение типа `double`, генерируется исключение
`public override string ToString()`	Возвращает строковое представление значения вызывающего объекта
`public string ToString( string format)`	Возвращает строковое представление значения вызывающего объекта в соответствии с требованиями форматирующей строки, переданной в параметре `format`
`public string ToString( IFormatProvider fmtpvdr)`	Возвращает строковое представление значения вызывающего объекта с использованием форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра `fmtpvdr`
`public string ToString( string format, IFormatProvider fmtpvdr)`	Возвращает строковое представление значения вызывающего объекта с использованием форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра `fmtpvdr`, а также форматирующей строки, переданной в параметре `format`

Таблица 19.6. Поля, поддерживаемые структурой `Double`

Поле	Описание
`public const double Epsilon`	Наименьшее ненулевое положительное значение
`public const double MaxValue`	Наибольшее значение, которое можно хранить с помощью типа `double`
`public const double MinValue`	Наименьшее значение, которое можно хранить с помощью типа `double`
`public const double NaN`	Значение, которое не является числом
`public const double NegativeInfinity`	Значение, представляющее минус бесконечность
`public const double PositiveInfinity`	Значение, представляющее плюс бесконечность

Структура `Decimal`

Структура `Decimal` несколько сложнее, чем описанные выше. Она содержит множество конструкторов, полей, методов и операторов, которые способствуют совместному использованию типа `decimal` и других числовых C#-типов. Например, ряд методов обеспечивает преобразование значений типа `decimal` в значения других числовых типов.

В структуре `Decimal` определено восемь открытых конструкторов. Наиболее часто используются следующие шесть:

```
public Decimal(int v)
public Decimal(uint v)
public Decimal(long v)
```

```
public Decimal(ulong v)
public Decimal(float v)
public Decimal(double v)
```

Каждый из перечисленных конструкторов создает Decimal-объект на основе заданного значения.

Decimal-объект также можно создать, указав его составляющие при вызове следующего конструктора:

```
public Decimal(int low, int middle, int high,
               bool signFlag, byte scaleFactor)
```

Значение типа decimal состоит из трех частей. Первая представляет собой 96-разрядное целое число, вторая — флаг знака и третья — коэффициент масштабирования. 96-разрядное целое число передается в 32-разрядные участки памяти с помощью параметров *low*, *middle* и *high*. Знак передается через параметр *signFlag*, который устанавливается равным значению false для положительного числа и значению true — для отрицательного. Коэффициент масштабирования передается посредством параметра scaleFactor, который должен иметь значение в диапазоне от 0 до 28. Этот коэффициент задает степень числа 10 (т.е. $10^{scaleFactor}$), на которую делится число для получения его дробной части.

Вместо того чтобы передавать каждый компонент в отдельности, можно задать составные части Decimal-объекта в массиве целых чисел. В этом случае используйте следующий конструктор:

```
public Decimal(int[] parts)
```

Первые три int-значения в параметре *parts* содержат 96-разрядное целое число. 31-й разряд элемента parts[3] содержит флаг числа (нуль — для положительного и 1 — для отрицательного), а в разрядах 16–23 хранится масштабный коэффициент.

В структуре Decimal реализованы следующие интерфейсы: IComparable, IConvertible и IFormattable.

Рассмотрим пример создания значения типа decimal "вручную".

```
// Создание decimal-значения "вручную".

using System;

class CreateDec {
  public static void Main() {
    decimal d = new decimal(12345, 0, 0, false, 2);

    Console.WriteLine(d);
  }
}
```

Результат выполнения этой программы таков:

```
123.45
```

В этом примере значение 96-разрядного целого числа равно 12345. Это число — положительное и имеет два десятичных разряда.

Методы, определенные в структуре Decimal, приведены в табл. 19.7, а поля — в табл. 19.8. В структуре Decimal также определено множество операторов и преобразований, позволяющих совместно использовать decimal-значения в выражениях с другими числовыми типами. Правила использования значений типа decimal в выражениях и инструкциях присваивания описаны в главе 3.

Таблица 19.7. Методы, определенные в структуре `Decimal`

Метод	Описание
`public static decimal Add(decimal v1, decimal v2)`	Возвращает значение *v1* + *v2*
`Public static int CompareTo(decimal v1, decimal v2)`	Сравнивает числовые значения параметров *v1* и *v2*. Возвращает нуль, если сравниваемые значения равны. Возвращает отрицательное число, если *v1* меньше *v2*, и — положительное, если *v1* больше *v2*
`public int CompareTo(object v)`	Сравнивает числовое значение вызывающего объекта со значением параметра *v*. Возвращает нуль, если сравниваемые значения равны. Возвращает отрицательное число, если вызывающий объект имеет меньшее значение, и положительное, если вызывающий объект имеет большее значение
`public static decimal Divide(decimal v1, decimal v2)`	Возвращает значение *v1* / *v2*
`Public override bool Equals(object v)`	Возвращает значение ИСТИНА, если значение вызывающего объекта равно значению параметра *v*
`public static bool Equals(decimal v1, decimal v2)`	Возвращает значение ИСТИНА, если *v1* равно *v2*
`public static decimal Floor(decimal v)`	Возвращает наибольшее целое число (представленное в виде значения типа `decimal`), которое не больше параметра *v*. Например, при *v*, равном `1.02`, метод `Floor()` возвратит `1.0`. А при *v*, равном `-1.02`, метод `Floor()` возвратит `-2`
`public static decimal FromOACurrency(long v)`	Преобразует значение, предоставленное приложением OLE Automation и переданное в параметре *v*, в его `decimal`-эквивалент и возвращает результат
`public static int[] GetBits(decimal v)`	Возвращает двоичное представление значения параметра *v* и возвращает его в массиве `int`-элементов. Организация этого массива описана в тексте этого раздела
`public override int GetHashCode()`	Возвращает хеш-код для вызывающего объекта
`public TypeCode GetTypeCode()`	Возвращает значение перечисления `TypeCode` для структуры `Decimal`, т.е. `TypeCode.Decimal`
`public static decimal Multiply(decimal v1, decimal v2)`	Возвращает значение *v1* * *v2*
`public static decimal Negate(decimal v)`	Возвращает значение `-v`
`public static decimal Parse(string str)`	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре *str*. Если содержимое строки не представляет значение типа `decimal`, генерируется исключение
`public static decimal Parse(string str, IFormatProvider fmtpvdr)`	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре *str* с использованием форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра *fmtpvdr*. Если содержимое строки не представляет значение типа `decimal`, генерируется исключение

Метод	Описание
`public static decimal Parse( string str, NumberStyles styles)`	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре `str`, с использованием информации стилевого характера, заданной в параметре `styles`. Если содержимое строки не представляет значение типа `decimal`, генерируется исключение
`public static decimal Parse( string str, NumberStyles styles, IFormatProvider fmtpvdr)`	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре `str`, с использованием информации стилевого характера, заданной в параметре `styles`, а также форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра `fmtpvdr`. Если содержимое строки не представляет значение типа `decimal`, генерируется исключение
`public static decimal Remainder(decimal v1, decimal v2)`	Возвращает остаток от целочисленного деления `v1 / v2`
`public static decimal Round(decimal v, int decPlaces)`	Возвращает значение `v`, округленное до числа, количество цифр дробной части которого равно значению параметра `decPlaces`, которое должно находиться в диапазоне 0-28
`public static decimal Subtract(decimal v1, decimal v2)`	Возвращает значение `v1 - v2`
`public static byte ToByte(decimal v)`	Возвращает `byte`-эквивалент параметра `v`. Дробная часть отбрасывается. Если значение параметра `v` не попадает в диапазон представления чисел, соответствующий типу `byte`, генерируется исключение типа `OverflowException`
`public static double ToDouble(decimal v)`	Возвращает `double`-эквивалент параметра `v`. При этом возможна потеря точности, поскольку тип `double` имеет меньше значащих цифр, чем тип `decimal`
`public static short ToInt16(decimal v)`	Возвращает `short`-эквивалент параметра `v`. Дробная часть отбрасывается. Если значение параметра `v` не попадает в диапазон представления чисел, соответствующий типу `short`, генерируется исключение типа `OverflowException`
`public static int ToInt32(decimal v)`	Возвращает `int`-эквивалент параметра `v`. Дробная часть отбрасывается. Если значение параметра `v` не попадает в диапазон представления чисел, соответствующий типу `int`, генерируется исключение типа `OverflowException`
`public static long ToInt64(decimal v)`	Возвращает `long`-эквивалент параметра `v`. Дробная часть отбрасывается. Если значение параметра `v` не попадает в диапазон представления чисел, соответствующий типу `long`, генерируется исключение типа `OverflowException`
`public static long ToOACurrency(decimal v)`	Преобразует значение параметра `v` в эквивалентное значение OLE Automation и возвращает результат
`public static sbyte ToSByte(decimal v)`	Возвращает `sbyte`-эквивалент параметра `v`. Дробная часть отбрасывается. Если значение параметра `v` не попадает в диапазон представления чисел, соответствующий типу `sbyte`, генерируется исключение типа `OverflowException`
`public static float ToSingle(decimal v)`	Возвращает `float`-эквивалент параметра `v`. При этом возможна потеря точности, поскольку тип `float` имеет меньше значащих цифр, чем тип `decimal`

Метод	*Описание*
`public override string ToString()`	Возвращает строковое представление значения вызывающего объекта
`public string ToString(string format)`	Возвращает строковое представление значения вызывающего объекта в соответствии с требованиями форматирующей строки, переданной в параметре `format`
`public string ToString(IFormatProvider fmtpvdr)`	Возвращает строковое представление значения вызывающего объекта с использованием форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра `fmtpvdr`
`public string ToString(string format, IFormatProvider fmtpvdr)`	Возвращает строковое представление значения вызывающего объекта с использованием форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра `fmtpvdr`, а также форматирующей строки, переданной в параметре `format`
`public static ushort ToUInt16(decimal v)`	Возвращает `ushort`-эквивалент параметра `v`. Дробная часть отбрасывается. Если значение параметра `v` не попадает в диапазон представления чисел, соответствующий типу `ushort`, генерируется исключение типа `OverflowException`
`public static uint ToUInt32(decimal v)`	Возвращает `uint`-эквивалент параметра `v`. Дробная часть отбрасывается. Если значение параметра `v` не попадает в диапазон представления чисел, соответствующий типу `uint`, генерируется исключение типа `OverflowException`
`public static ulong ToUInt64(decimal v)`	Возвращает `ulong`-эквивалент параметра `v`. Дробная часть отбрасывается. Если значение параметра `v` не попадает в диапазон представления чисел, соответствующий типу `ulong`, генерируется исключение типа `OverflowException`
`public static decimal Truncate(decimal v)`	Возвращает целую часть числа, заданного параметром `v`. Соответственно, любая дробная часть при этом отбрасывается

Таблица 19.8. Поля, поддерживаемые структурой `Decimal`

Поле	*Описание*
`public static readonly decimal MaxValue`	Наибольшее значение, которое позволяет хранить тип `decimal`
`public static readonly decimal MinusOne`	Представление числа -1 в формате `decimal`-значения
`public static readonly decimal MinValue`	Наименьшее значение, которое позволяет хранить тип `decimal`
`public static readonly decimal One`	Представление числа 1 в формате `decimal`-значения
`public static readonly decimal Zero`	Представление числа 0 в формате `decimal`-значения.

Структура `Char`

Пожалуй, наиболее используемой (можно сказать, структурой "каждодневного применения") является структура `Char`. Она предоставляет большое количество методов, которые позволяют обрабатывать символы и определять, к какой категории они относятся. Например, вызвав метод `ToUpper()`, можно преобразовать строчный сим-

вол в его прописной эквивалент. А с помощью метода `IsDigit()` можно определить, является ли анализируемый символ цифрой.

Методы, определяемые в структуре `Char`, перечислены в табл. 19.9. В структуре `Char` также определены следующие поля:

```
public const char MaxValue
public const char MinValue
```

Они представляют наибольшее и наименьшее значения, которые может хранить переменная типа `char`. Структура `Char` реализует интерфейсы `IComparable` и `IConvertible`.

Таблица 19.9. Методы, определенные в структуре `Char`

Метод	*Описание*
`public int CompareTo(object v)`	Сравнивает символ в вызывающем объекте с символом параметра *v*. Возвращает нуль, если сравниваемые символы равны. Возвращает отрицательное число, если вызывающий объект имеет меньшее значение, и — положительное, если вызывающий объект имеет большее значение
`public override bool Equals( object v)`	Возвращает значение ИСТИНА, если значение вызывающего объекта равно значению параметра *v*
`public override int GetHashCode()`	Возвращает хеш-код для вызывающего объекта
`public static double GetNumericValue(char ch)`	Возвращает числовое значение параметра *ch*, если *ch* — цифра. В противном случае возвращает -1
`public static double GetNumericValue(string str, int idx)`	Возвращает числовое значение символа *str[idx]*, если он является цифрой. В противном случае возвращает -1
`public TypeCode GetTypeCode()`	Возвращает значение перечисления `TypeCode` для структуры `Char`, т.е. `TypeCode.Char`
`public static UnicodeCategory GetUnicodeCategory(char ch)`	Возвращает значение перечисления `UnicodeCategory` для параметра *ch*. `UnicodeCategory` — это перечисление, определенное в пространстве имен `System.Globalization`, в котором символы Unicode разделены по категориям
`public static UnicodeCategory GetUnicodeCategory(string str, int idx)`	Возвращает значение перечисления `UnicodeCategory` для символа *str[idx]*. `UnicodeCategory` — это перечисление, определенное в пространстве имен `System.Globalization`, в котором символы Unicode разделены по категориям
`public static bool IsControl(char ch)`	Возвращает значение ИСТИНА, если параметр *ch* является управляющим символом. В противном случае возвращает значение ЛОЖЬ
`public static bool IsControl(string str, int idx)`	Возвращает значение ИСТИНА, если символ *str[idx]* является управляющим символволом. В противном случае возвращает значение ЛОЖЬ
`public static bool IsDigit( char ch)`	Возвращает значение ИСТИНА, если параметр *ch* является цифрой. В противном случае возвращает значение ЛОЖЬ
`public static bool IsDigit(string str, int idx)`	Возвращает значение ИСТИНА, если символ *str[idx]* является цифрой. В противном случае возвращает значение ЛОЖЬ
`public static bool IsLetter(char ch)`	Возвращает значение ИСТИНА, если параметр *ch* является буквой алфавита. В противном случае возвращает значение ЛОЖЬ

Метод	Описание
`public static bool IsLetter(string str, int idx)`	Возвращает значение ИСТИНА, если символ `str[idx]` является буквой алфавита. В противном случае возвращает значение ЛОЖЬ
`public static bool IsLetterOrDigit(char ch)`	Возвращает значение ИСТИНА, если параметр `ch` является буквой алфавита или цифрой. В противном случае возвращает значение ЛОЖЬ
`public static bool IsLetterOrDigit(string str, int idx)`	Возвращает значение ИСТИНА, если символ `str[idx]` является буквой алфавита или цифрой. В противном случае возвращает значение ЛОЖЬ
`public static bool IsLower(char ch)`	Возвращает значение ИСТИНА, если параметр `ch` является строчной буквой алфавита. В противном случае возвращает значение ЛОЖЬ
`public static bool IsLower(string str, int idx)`	Возвращает значение ИСТИНА, если символ `str[idx]` является строчной буквой алфавита. В противном случае возвращает значение ЛОЖЬ
`public static bool IsNumber(char ch)`	Возвращает значение ИСТИНА, если параметр `ch` является шестнадцатиричной цифрой (0–9 или A–F). В противном случае возвращает значение ЛОЖЬ
`public static bool IsNumber(string str, int idx)`	Возвращает значение ИСТИНА, если символ `str[idx]` является шестнадцатиричной цифрой (0–9 или A–F). В противном случае возвращает значение ЛОЖЬ
`public static bool IsPunctuation(char ch)`	Возвращает значение ИСТИНА, если параметр `ch` является знаком пунктуации. В противном случае возвращает значение ЛОЖЬ
`public static bool IsPunctuation(string str, int idx)`	Возвращает значение ИСТИНА, если символ `str[idx]` является знаком пунктуации. В противном случае возвращает значение ЛОЖЬ
`public static bool IsSeparator(char ch)`	Возвращает значение ИСТИНА, если параметр `ch` является разделительным знаком, например пробелом. В противном случае возвращает значение ЛОЖЬ
`public static bool IsSeparator(string str, int idx)`	Возвращает значение ИСТИНА, если символ `str[idx]` является разделительным знаком, например пробелом. В противном случае возвращает значение ЛОЖЬ
`public static bool IsSurrogate(char ch)`	Возвращает значение ИСТИНА, если параметр `ch` является псевдосимволом Unicode. В противном случае возвращает значение ЛОЖЬ
`public static bool IsSurrogate(string str, int idx)`	Возвращает значение ИСТИНА, если символ `str[idx]` является псевдосимволом Unicode. В противном случае возвращает значение ЛОЖЬ
`public static bool IsSymbol(char ch)`	Возвращает значение ИСТИНА, если параметр `ch` является символическим знаком, например валютным символом. В противном случае возвращает значение ЛОЖЬ
`public static bool IsSymbol(string str, int idx)`	Возвращает значение ИСТИНА, если символ `str[idx]` является символическим знаком, например валютным символом. В противном случае возвращает значение ЛОЖЬ
`public static bool IsUpper(char ch)`	Возвращает значение ИСТИНА, если параметр `ch` является прописной буквой алфавита. В противном случае возвращает значение ЛОЖЬ
`public static bool IsUpper(string str, int idx)`	Возвращает значение ИСТИНА, если символ `str[idx]` является прописной буквой алфавита. В противном случае возвращает значение ЛОЖЬ

Метод	Описание
`public static bool IsWhiteSpace(char ch)`	Возвращает значение ИСТИНА, если параметр `ch` является пробелом, символом табуляции или пустой строки. В противном случае возвращает значение ЛОЖЬ
`public static bool IsWhiteSpace(string str, int idx)`	Возвращает значение ИСТИНА, если символ `str[idx]` является пробелом, символом табуляции или пустой строки. В противном случае возвращает значение ЛОЖЬ
`public static char Parse(string str)`	Возвращает `char`-эквивалент символа в параметре `str`. Если строка `str` содержит более одного символа, генерируется исключение типа `FormatException`
`public static char ToLower(char ch)`	Возвращает строчный эквивалент параметра `ch`, если `ch` — прописная буква. В противном случае возвращает значение `ch` неизменным
`public static char ToLower(char ch, CultureInfo c)`	Возвращает строчный эквивалент параметра `ch`, если `ch` — прописная буква. В противном случае возвращает значение `ch` неизменным. Преобразование выполняется в соответствии с заданной в параметре `c` информацией о представлении данных, соответствующем конкретному естественному языку, диалекту или территориальному образованию. `CultureInfo` — это класс, определенный в пространстве имен `System.Globalization`
`public static char ToUpper(char ch)`	Возвращает прописной эквивалент параметра `ch`, если `ch` — строчная буква. В противном случае возвращает значение `ch` неизменным
`public static char ToUpper(char ch, CultureInfo c)`	Возвращает прописной эквивалент параметра `ch`, если `ch` — строчная буква. В противном случае возвращает значение `ch` неизменным. Преобразование выполняется в соответствии с заданной в параметре `c` информацией о представлении данных, соответствующем конкретному естественному языку, диалекту или территориальному образованию. `CultureInfo` — это класс, определенный в пространстве имен `System.Globalization`
`public override string ToString()`	Возвращает строковое представление значения вызывающего `Char`-объекта
`public static string ToString(char ch)`	Возвращает строковое представление значения параметра `ch`
`public string ToString(IFormatProvider fmtpvdr)`	Возвращает строковое представление значения вызывающего `Char`-объекта с использованием заданной в параметре `fmtpvdr` информации о представлении данных, соответствующем конкретному естественному языку, диалекту или территориальному образованию

Рассмотрим программу, которая демонстрирует использование методов, определенных в структуре `Char`:

```
// Демонстрация использования нескольких методов,
// определенных в структуре Char.

using System;

class CharDemo {
  public static void Main() {
    string str = "Это простой тест. $23";
    int i;
```

```
    for(i=0; i < str.Length; i++) {
      Console.Write(str[i] + " является");
      if(Char.IsDigit(str[i]))
        Console.Write(" цифрой");
      if(Char.IsLetter(str[i]))
        Console.Write(" буквой");
      if(Char.IsLower(str[i]))
        Console.Write(" строчной");
      if(Char.IsUpper(str[i]))
        Console.Write(" прописной ");
      if(Char.IsSymbol(str[i]))
        Console.Write(" символическим знаком");
      if(Char.IsSeparator(str[i]))
        Console.Write(" разделителем");
      if(Char.IsWhiteSpace(str[i]))
        Console.Write(" пробелом");
      if(Char.IsPunctuation(str[i]))
        Console.Write(" знаком пунктуации");

      Console.WriteLine();
    }

    Console.WriteLine("Исходная строка: " + str);

    // Преобразуем в буквы в прописные.
    string newstr = "";
    for(i=0; i < str.Length; i++)
      newstr += Char.ToUpper(str[i]);

    Console.WriteLine("После преобразования: " + newstr);

  }
}
```

Результаты выполнения этой программы таковы:

```
Э является буквой прописной
т является буквой строчной
о является буквой строчной
  является разделителем пробелом
п является буквой строчной
р является буквой строчной
о является буквой строчной
с является буквой строчной
т является буквой строчной
о является буквой строчной
й является буквой строчной
  является разделителем пробелом
т является буквой строчной
е является буквой строчной
с является буквой строчной
т является буквой строчной
. является знаком пунктуации
  является разделителем пробелом
$ является символическим знаком
2 является цифрой
3 является цифрой
Исходная строка: Это простой тест. $23
После преобразования: ЭТО ПРОСТОЙ ТЕСТ. $23
```

Структура Boolean

Структура Boolean предназначена для поддержки типа данных bool. Методы, определенные в структуре Boolean, перечислены в табл. 19.10. В ней также определены следующие поля:

```
public static readonly string FalseString
public static readonly string TrueString
```

Они содержат удобные для восприятия человеком формы логических констант true и false. Например, если вывести значение FalseString с помощью метода WriteLine(), будет отображена строка "False".

Структура Boolean реализует интерфейсы IComparable и IConvertible.

Таблица 19.10. Методы, определенные в структуре Boolean

Метод	*Описание*
`public int CompareTo(object v)`	Сравнивает значение вызывающего объекта со значением параметра *v*. Возвращает нуль, если сравниваемые значения равны. Возвращает отрицательное число, если вызывающий объект имеет значение false, а параметр *v* — true. Возвращает положительное число, если вызывающий объект имеет значение true, а параметр *v* — false
`public override bool Equals( object v)`	Возвращает значение true, если значение вызывающего объекта равно значению параметра *v*
`public override int GetHashCode()`	Возвращает хеш-код для вызывающего объекта
`public TypeCode GetTypeCode()`	Возвращает значение перечисления TypeCode для структуры Boolean, т.е. TypeCode.Boolean
`public static bool Parse( string str)`	Возвращает bool-эквивалент строки, содержащейся в параметре *str*. Если эта строка не содержит ни вариант "True", ни "False", генерируется исключение. При этом не важно, какие буквы здесь используются: прописные или строчные
`public override string ToString()`	Возвращает строковое представление значения вызывающего объекта
`string ToString( IFormatProvider fmtpvdr)`	Возвращает строковое представление значения вызывающего объекта с использованием заданной в параметре *fmtpvdr* информации о представлении данных, соответствующем конкретному естественному языку, диалекту или территориальному образованию

Класс Array

Один из самых полезных классов в пространстве имен System — Array. Array — это базовый класс для всех массивов в C#. Следовательно, его методы можно применять для массивов любого из встроенных типов, а также массивов, тип которых вы создадите сами. Свойства, определенные в классе Array, перечислены в табл. 19.11, а методы — в табл. 19.12.

Класс Array реализует следующие интерфейсы: ICloneable, ICollection, IEnumerable и IList. Интерфейсы ICollection, IEnumerable и IList определены в пространстве имен System.Collections и описаны в главе 22.

В ряде методов класса `Array` используется параметр интерфейсного типа `IComparer`. В этом интерфейсе, принадлежащем пространству имен `System.Collections`, определен метод `Compare()`, который сравнивает значения двух объектов.

```
int Compare(object v1, object v2)
```

Этот метод возвращает положительное число, если значение *v1* больше значения *v2*, отрицательное число, если *v1* меньше *v2*, и нуль, если сравниваемые значения равны.

В следующих разделах демонстрируется выполнение наиболее употребимых операций над массивами.

Сортировка массивов и поиск заданного элемента

Одной из наиболее употребимых операций, выполняемых над массивами, является сортировка. Поэтому класс `Array` поддерживает множество методов сортировки элементов массива. Используя метод `Sort()`, можно отсортировать весь массив, его часть или два массива, которые содержат соответствующие пары "ключ/значение". В отсортированном массиве можно организовать эффективный поиск заданных элементов с помощью метода `BinarySearch()`. Рассмотрим программу, в которой на основе `int`-массива показано использование методов `Sort()` и `BinarySearch()`:

```
// Сортировка массива и поиск в нем заданного значения.

using System;

class SortDemo {
  public static void Main() {
    int[] nums = { 5, 4, 6, 3, 14, 9, 8, 17, 1, 24, -1, 0 };

    // Отображаем исходный порядок следования
    // элементов в массиве.
    Console.Write("Исходный порядок:          ");
    foreach(int i in nums)
      Console.Write(i + " ");
    Console.WriteLine();

    // Сортируем массив.
    Array.Sort(nums);

    // Отображаем отсортированный массив.
    Console.Write("Порядок после сортировки: ");
    foreach(int i in nums)
      Console.Write(i + " ");
    Console.WriteLine();


    // Выполняем поиск числа 14.
    int idx = Array.BinarySearch(nums, 14);

    Console.WriteLine("Индекс значения 14 равен " + idx);
  }
}
```

Результаты выполнения этой программы таковы:

```
Исходный порядок:          5 4 6 3 14 9 8 17 1 24 -1 0
Порядок после сортировки: -1 0 1 3 4 5 6 8 9 14 17 24
Индекс значения 14 равен 9
```

В предыдущем примере массив имел базовый тип `int`, т.е. нессылочный тип. Все методы, определенные в классе `Array`, автоматически применимы для всех встроенных нессылочных типов. Но совсем иначе обстоит дело с массивами объектных ссылок. Чтобы выполнить сортировку или поиск значения в массиве объектных ссылок, тип класса этих объектов должен реализовать интерфейс `IComparable`. Если класс не реализует этот интерфейс, при попытке выполнить сортировку или поиск значения в массиве будет динамически сгенерировано исключение. К счастью, интерфейс `IComparable` реализовать нетрудно, поскольку он состоит только из одного метода:

```
int CompareTo(object v)
```

Этот метод сравнивает вызывающий объект со значением в параметре *v*. Он возвращает положительное число, если вызывающий объект больше значения *v*, нуль, если два сравниваемых объекта равны, и отрицательное число, если вызывающий объект меньше значения *v*. Рассмотрим пример программы, которая иллюстрирует сортировку и поиск в массиве, состоящем из объектов класса, определенного пользователем:

```
// Сортировка и поиск в массиве объектов.

using System;

class MyClass : IComparable {
  public int i;

  public MyClass(int x) { i = x; }

  // Реализуем интерфейс IComparable.
  public int CompareTo(object v) {
    return i - ((MyClass)v).i;
  }
}

class SortDemo {
  public static void Main() {
    MyClass[] nums = new MyClass[5];

    nums[0] = new MyClass(5);
    nums[1] = new MyClass(2);
    nums[2] = new MyClass(3);
    nums[3] = new MyClass(4);
    nums[4] = new MyClass(1);

    // Отображаем исходный порядок следования элементов
    // в массиве.
    Console.Write("Исходный порядок:          ");
    foreach(MyClass o in nums)
      Console.Write(o.i + " ");
    Console.WriteLine();

    // Сортируем массив.
    Array.Sort(nums);

    // Отображаем отсортированный массив.
    Console.Write("Порядок после сортировки: ");
    foreach(MyClass o in nums)
      Console.Write(o.i + " ");
    Console.WriteLine();
```

```
    // Поиск объекта MyClass(2).
    MyClass x = new MyClass(2);
    int idx = Array.BinarySearch(nums, x);

    Console.WriteLine("Индекс объекта MyClass(2) равен " +
                      idx);
  }
}
```

Результаты выполнения этой программы таковы:

```
Исходный порядок:          5 2 3 4 1
Порядок после сортировки: 1 2 3 4 5
Индекс объекта MyClass(2) равен 1
```

Реверсирование массива

Иногда полезно реверсировать содержимое массива. Например, может потребоваться заменить массив, который был отсортирован в возрастающем порядке, массивом, отсортированном в убывающем порядке. Выполнить реверсирование нетрудно: достаточно вызвать метод `Reverse()`. С помощью этого метода можно реверсировать весь массив или только некоторую его часть. Этот процесс демонстрируется в следующей программе:

```
// Реверсирование массива.

using System;

class ReverseDemo {
  public static void Main() {
    int[] nums = { 1, 2, 3, 4, 5 };

    // Отображаем исходный порядок.
    Console.Write("Исходный порядок:                ");
    foreach(int i in nums)
      Console.Write(i + " ");
    Console.WriteLine();

    // Реверсируем весь массив.
    Array.Reverse(nums);

    // Отображаем обратный порядок.
    Console.Write("Обратный порядок:                ");
    foreach(int i in nums)
      Console.Write(i + " ");
    Console.WriteLine();

    // Реверсируем часть массива.
    Array.Reverse(nums, 1, 3);

    // Отображаем порядок при частичном
    // реверсировании массива.
    Console.Write("После частичного реверсирования: ");
    foreach(int i in nums)
      Console.Write(i + " ");
    Console.WriteLine();
  }
}
```

Результаты выполнения этой программы таковы:

```
Исходный порядок:                  1 2 3 4 5
Обратный порядок:                  5 4 3 2 1
После частичного реверсирования: 5 2 3 4 1
```

Копирование массивов

Копирование всего массива или некоторой его части — еще одна часто используемая операция. Чтобы скопировать массив, используйте метод `Copy()`. Этот метод может помещать элементы в начало массива-приемника или в середину, в зависимости от того, какую версию метода `Copy()` вы используете. Использование метода `Copy()` демонстрируется в следующей программе:

```
// Копирование массива.

using System;

class CopyDemo {
  public static void Main() {
    int[] source = { 1, 2, 3, 4, 5 };
    int[] target = { 11, 12, 13, 14, 15 };
    int[] source2 = { -1, -2, -3, -4, -5 };

    // Отображаем массив-источник копирования.
    Console.Write("Массив-источник: ");
    foreach(int i in source)
      Console.Write(i + " ");
    Console.WriteLine();

    // Отображаем исходное содержимое массива-приемника.
    Console.Write(
           "Исходное содержимое массива-приемника: ");
    foreach(int i in target)
      Console.Write(i + " ");
    Console.WriteLine();

    // Копируем весь массив.
    Array.Copy(source, target, source.Length);

    // Отображаем копию массива.
    Console.Write("Приемник после копирования:  ");
    foreach(int i in target)
      Console.Write(i + " ");
    Console.WriteLine();

    // Копируем в середину массива target.
    Array.Copy(source2, 2, target, 3, 2);

    // Отображаем результат частичного копирования.
    Console.Write(
           "Приемник после частичного копирования:  ");
    foreach(int i in target)
      Console.Write(i + " ");
    Console.WriteLine();
  }
}
```

Результаты выполнения этой программы таковы:

```
Массив-источник: 1 2 3 4 5
Исходное содержимое массива-приемника: 11 12 13 14 15
Приемник после копирования:  1 2 3 4 5
Приемник после частичного копирования:  1 2 3 -3 -4
```

Таблица 19.11. Свойства, определенные в классе `Array`

Свойство	*Описание*
`public virtual bool IsFixedSize { get; }`	Предназначено только для чтения. Принимает значение `true`, если массив имеет фиксированный размер, и `false`, если массив может динамически его изменять
`public virtual bool IsReadOnly { get; }`	Предназначено только для чтения. Принимает значение `true`, если объект класса `Array` предназначен только для чтения, и `false` в противном случае
`public virtual bool IsSynchronized { get; }`	Предназначено только для чтения. Принимает значение `true`, если массив можно безопасно использовать в многопоточной среде, и `false` в противном случае
`public int Length { get; }`	Предназначено только для чтения. Содержит количество элементов в массиве
`public int Rank { get; }`	Предназначено только для чтения. Содержит размерность массива
`public virtual object SyncRoot { get; }`	Предназначено только для чтения. Содержит объект, который должен быть использован для синхронизации доступа к массиву

Таблица 19.12. Методы, определенные в классе `Array`

Метод	*Описание*
`public static int BinarySearch(Array a, object v)`	В массиве, заданном параметром *a*, выполняет поиск значения, заданного параметром *v*. Возвращает индекс первого вхождения искомого значения. Если оно не найдено, возвращает отрицательное число. Массив *a* должен быть отсортированным и одномерным
`public static int BinarySearch(Array a, object v, IComparer comp)`	В массиве, заданном параметром *a*, выполняет поиск значения, заданного параметром *v*, с использованием метода сравнения, заданного параметром *comp*. Возвращает индекс первого вхождения искомого значения. Если оно не найдено, возвращает отрицательное число. Массив *a* должен быть отсортированным и одномерным
`public static int BinarySearch(Array a, int start, int count, object v)`	В части массива, заданного параметром *a*, выполняет поиск значения, заданного параметром *v*. Поиск начинается с индекса, заданного параметром *start*, и охватывает *count* элементов. Возвращает индекс первого вхождения искомого значения. Если оно не найдено, возвращает отрицательное число. Массив *a* должен быть отсортированным и одномерным
`public static int BinarySearch(Array a, int start, int count, object v, IComparer comp)`	В части массива, заданного параметром *a*, выполняет поиск значения, заданного параметром *v*, с использованием метода сравнения, заданного параметром *comp*. Поиск начинается с индекса, заданного параметром *start*, и охватывает *count* элементов. Возвращает индекс первого вхождения искомого значения. Если оно не найдено, возвращает отрицательное число. Массив *a* должен быть отсортированным и одномерным

Метод	Описание
`public static void Clear(Array a, int start, int count)`	Устанавливает заданные элементы равными нулю. Диапазон элементов, подлежащих обнулению, начинается с индекса, заданного параметром `start`, и включает `count` элементов
`public virtual object Clone()`	Возвращает копию вызывающего массива. Эта копия ссылается на те же элементы, что и оригинал, за что получила название "поверхностной". Это означает, что изменения, вносимые в элементы, влияют на оба массива, поскольку они оба используют одни и те же элементы
`public static void Copy(Array source, Array dest, int count)`	Копирует `count` элементов из массива `source` в массив `dest`. Копирование начинается с начальных элементов каждого массива. Если оба массива имеют одинаковый ссылочный тип, метод `Copy()` создает "поверхностную копию", в результате чего оба массива будут ссылаться на одни и те же элементы
`public static void Copy(Array source, int srcStart, Array dest, int destStart, int count)`	Копирует `count` элементов из массива `source` (начиная с элемента с индексом `srcStart`) в массив `dest` (начиная с элемента с индексом `destStart`). Если оба массива имеют одинаковый ссылочный тип, метод `Copy()` создает "поверхностную копию", в результате чего оба массива будут ссылаться на одни и те же элементы
`public virtual void CopyTo(Array dest, int start)`	Копирует элементы вызывающего массива в массив `dest`, начиная с элемента `dest[start]`
`public static Array CreateInstance(Type t, int size)`	Возвращает ссылку на одномерный массив, который содержит `size` элементов типа `t`
`public static Array CreateInstance(Type t, int size1, int size2)`	Возвращает ссылку на двумерный массив размером `size1*size2`. Каждый элемент этого массива имеет тип `t`
`public static Array CreateInstance(Type t, int size1, int size2, int size3)`	Возвращает ссылку на трехмерный массив размером `size1*size2*size3`. Каждый элемент этого массива имеет тип `t`
`public static Array CreateInstance(Type t, int[] sizes)`	Возвращает ссылку на многомерный массив, размерности которого заданы в массиве `sizes`. Каждый элемент этого массива имеет тип `t`
`public static Array CreateInstance( Type t, int[] sizes, int[] startIndexes)`	Возвращает ссылку на многомерный массив, размерности которого заданы в массиве `sizes`. Каждый элемент этого массива имеет тип `t`. Начальный индекс каждой размерности задан в массиве `startIndexes`. Таким образом, этот метод позволяет создавать массивы, которые начинаются с некоторого индекса, отличного от нуля
`public override bool Equals(object v)`	Возвращает значение `true`, если значение вызывающего объекта равно значению параметра `v`
`public virtual IEnumerator GetEnumerator()`	Возвращает нумераторный объект для массива. Нумератор позволяет опрашивать массив в цикле. Нумераторы описаны в главе 22, "Коллекции"

Метод	Описание
`public int GetLength(int dim)`	Возвращает длину заданной размерности. Отсчет размерностей начинается с нуля; следовательно, для получения длины первой размерности необходимо передать методу значение 0, а для получения длины второй — значение 1
`public int GetLowerBound(int dim)`	Возвращает начальный индекс заданной размерности, который обычно равен нулю. Параметр `dim` ориентирован на то, что отсчет размерностей начинается с нуля; следовательно, для получения начального индекса первой размерности необходимо передать методу значение 0, а для получения начального индекса второй — значение 1
`public override int GetHashCode()`	Возвращает хеш-код для вызывающего объекта
`public TypeCode GetTypeCode()`	Возвращает значение перечисления `TypeCode` для класса `Array`, т.е. `TypeCode.Array`
`public int GetUpperBound(int dim)`	Возвращает конечный индекс заданной размерности, который обычно равен нулю. Параметр `dim` ориентирован на то, что отсчет размерностей начинается с нуля; следовательно, для получения конечного индекса первой размерности необходимо передать методу значение 0, а для получения конечного индекса второй — значение 1
`public object GetValue(int idx)`	Возвращает значение элемента вызывающего массива с индексом `idx`. Массив должен быть одномерным
`public object GetValue(int idx1, int idx2)`	Возвращает значение элемента вызывающего массива с индексами `[idx1, idx2]`. Массив должен быть двумерным
`public object GetValue(int idx1, int idx2, int idx3)`	Возвращает значение элемента вызывающего массива с индексами `[idx1, idx2, idx3]`. Массив должен быть трехмерным
`public object GetValue(int[] idxs)`	Возвращает значение элемента вызывающего массива с индексами, заданными с помощью параметра `idxs`. Вызывающий массив должен иметь столько размерностей, сколько элементов в массиве `idxs`
`public static int IndexOf(Array a, object v)`	Возвращает индекс первого элемента одномерного массива `a`, который имеет значение, заданное параметром `v`. Возвращает -1, если искомое значение не найдено. (Если массив имеет нижнюю границу, отличную от 0, признак неудачного выполнения метода будет равен значению нижней границы, уменьшенному на 1)
`public static int IndexOf(Array a, object v, int start)`	Возвращает индекс первого элемента одномерного массива `a`, который имеет значение, заданное параметром `v`. Поиск начинается с элемента `a[start]`. Возвращает -1, если искомое значение не найдено. (Если массив имеет нижнюю границу, отличную от 0, признак неудачного выполнения метода будет равен значению нижней границы, уменьшенному на 1)
`public static int IndexOf(Array a, object v, int start, int count)`	Возвращает индекс первого элемента одномерного массива `a`, который имеет значение, заданное параметром `v`. Поиск начинается с элемента `a[start]` и охватывает `count` элементов. Метод возвращает -1, если внутри заданного диапазона искомое значение не найдено. (Если массив имеет нижнюю границу, отличную от 0, признак неудачного выполнения метода будет равен значению нижней границы, уменьшенному на 1)

Продолжение табл. 19.12

Метод	Описание
`public void Initialize()`	Инициализирует каждый элемент вызывающего массива посредством вызова конструктора по умолчанию, соответствующего конкретному элементу. Этот метод можно использовать только для массивов нессылочных типов
`public static int LastIndexOf(Array a, object v)`	Возвращает индекс последнего элемента одномерного массива *a*, который имеет значение, заданное параметром *v*. Возвращает -1, если искомое значение не найдено. (Если массив имеет нижнюю границу, отличную от 0, признак неудачного выполнения метода будет равен значению нижней границы, уменьшенному на 1)
`public static int LastIndexOf(Array a, object v, int start)`	Возвращает индекс последнего элемента заданного диапазона одномерного массива *a*, который имеет значение, заданное параметром *v*. Поиск осуществляется в обратном порядке, начиная с элемента `a[start]` и заканчивая элементом `a[0]`. Метод возвращает число -1, если искомое значение не найдено. (Если массив имеет нижнюю границу, отличную от 0, признак неудачного выполнения метода будет равен значению нижней границы, уменьшенному на 1)
`public static int LastIndexOf(Array a, object v, int start, int count)`	Возвращает индекс последнего элемента заданного диапазона одномерного массива *a*, который имеет значение, заданное параметром *v*. Поиск осуществляется в обратном порядке, начиная с элемента `a[start]`, и охватывает `count` элементов. Метод возвращает -1, если внутри заданного диапазона искомое значение не найдено. (Если массив имеет нижнюю границу, отличную от 0, признак неудачного выполнения метода будет равен значению нижней границы, уменьшенному на 1)
`public static void Reverse(Array a)`	Меняет на обратный порядок следования элементов в массиве *a*
`public static void Reverse(Array a, int start, int count)`	Меняет на обратный порядок следования элементов в заданном диапазоне массива *a*. Упомянутый диапазон начинается с элемента `a[start]` и включает `count` элементов
`public void SetValue(object v, int idx)`	Устанавливает в вызывающем массиве элемент с индексом *idx* равным значению *v*. Массив должен быть одномерным
`public void SetValue(object v, int idx1, int idx2)`	Устанавливает в вызывающем массиве элемент с индексами `[idx1, idx2]` равным значению *v*. Массив должен быть двумерным
`public void SetValue(object v, int idx1, int idx2, int idx3)`	Устанавливает в вызывающем массиве элемент с индексами `[idx1, idx2, idx3]` равным значению *v*. Массив должен быть трехмерным
`public void SetValue(object v, int[] idxs)`	Устанавливает в вызывающем массиве элемент с индексами, заданными параметром *idxs*, равным значению *v*. Вызывающий массив должен столько размерностей, сколько элементов в массиве *idxs*
`public static void Sort(Array a)`	Сортирует массив *a* в порядке возрастания. Массив должен быть одномерным
`public static void Sort(Array a, IComparer comp)`	Сортирует массив *a* в порядке возрастания с использованием метода сравнения, заданного параметром *comp*. Массив должен быть одномерным

Окончание табл. 19.12

Метод	Описание
`public static void Sort(Array k, Array v)`	Сортирует в порядке возрастания два заданных одномерных массива. Массив `k` содержит ключи сортировки, а массив `v` — значения, связанные с этими ключами. Следовательно, эти два массива должны содержать пары ключ/значение. После сортировки элементы обоих массивов расположены в порядке возрастания ключей
`public static void Sort(Array k, Array v, IComparer comp)`	Сортирует в порядке возрастания два заданных одномерных массива с использованием метода сравнения, заданного параметром `comp`. Массив `k` содержит ключи сортировки, а массив `v` — значения, связанные с этими ключами. Следовательно, эти два массива должны содержать пары ключ/значение. После сортировки элементы обоих массивов расположены в порядке возрастания ключей
`public static void Sort(Array a, int start, int count)`	Сортирует заданный диапазон массива в порядке возрастания. Упомянутый диапазон начинается с элемента `a[start]` и включает `count` элементов. Массив должен быть одномерным
`public static void Sort(Array a, int start int count, IComparer comp)`	Сортирует заданный диапазон массива в порядке возрастания с использованием метода сравнения, заданного параметром `comp`. Упомянутый диапазон начинается с элемента `a[start]` и включает `count` элементов. Массив должен быть одномерным
`public static void Sort(Array k, Array v, int start, int count)`	Сортирует заданный диапазон двух одномерных массивов в порядке возрастания. В обоих массивах диапазон сортировки начинается с индекса, переданного в параметре `start`, и включает `count` элементов. Массив `k` содержит ключи сортировки, а массив `v` — значения, связанные с этими ключами. Следовательно, эти два массива должны содержать пары ключ/значение. После сортировки элементы диапазонов обоих массивов расположены в порядке возрастания ключей
`public static void Sort(Array k, Array v, int start, int count, IComparer comp)`	Сортирует заданный диапазон двух одномерных массивов в порядке возрастания с использованием метода сравнения, заданного параметром `comp`. В обоих массивах диапазон сортировки начинается с индекса, переданного в параметре `start`, и включает `count` элементов. Массив `k` содержит ключи сортировки, а массив `v` — значения, связанные с этими ключами. Следовательно, эти два массива должны содержать пары ключ/значение. После сортировки элементы диапазонов обоих массивов расположены в порядке возрастания ключей

Класс `BitConverter`

При написании программ часто приходится преобразовывать данные встроенных типов в массив байтов. Например, некоторые устройства могут принимать целочисленные значения, но эти значения должны посылаться побайтно. Не менее часто встречается и обратная ситуация. Иногда данные должны быть организованы для приема в виде упорядоченной последовательности байтов, которые нужно преобразовать в значение одного из встроенных типов. Например, некоторое устройство может выводить целые числа, посылаемые как поток байтов. Для решения подобных проблем преобразования данных в C# и предусмотрен класс `BitConverter`.

Класс `BitConverter` содержит методы, представленные в табл. 19.13. В нем также определено следующее поле:

```
public static readonly bool IsLittleEndian
```

Это поле принимает значение `true`, если текущая операционная среда обрабатывает сначала слово с младшим (наименее значимым), а затем со старшим (наиболее значимым) байтом. Такой формат обработки (хранения и передачи) называется *прямым*, т.е. прямым порядком байтов, ("little-endian" format). Поле `IsLittleEndian` принимает значение `false`, если текущая операционная среда обрабатывает сначала слово со старшим (наиболее значимым), а затем с младшим (наименее значимым) байтом. Такой формат обработки называется *обратным* ("big-endian" format). Компьютеры, собранные на базе микропроцессора Pentium фирмы Intel, используют формат с прямым порядком байтов.

Класс `BitConverter` является `sealed`-классом, т.е. не может иметь производных классов.

Таблица 19.13. Методы, определенные в классе `BitConverter`

Метод	*Описание*
`public static long DoubleToInt64Bits(double v)`	Преобразует значение параметра `v` в целочисленное значение типа `long` и возвращает результат
`public static byte[] GetBytes(bool v)`	Преобразует значение параметра `v` в однобайтовый массив и возвращает результат
`public static byte[] GetBytes(char v)`	Преобразует значение параметра `v` в двубайтовый массив и возвращает результат
`public static byte[] GetBytes(double v)`	Преобразует значение параметра `v` в восьмибайтовый массив и возвращает результат
`public static byte[] GetBytes(float v)`	Преобразует значение параметра `v` в четырехбайтовый массив и возвращает результат
`public static byte[] GetBytes(int v)`	Преобразует значение параметра `v` в четырехбайтовый массив и возвращает результат
`public static byte[] GetBytes(long v)`	Преобразует значение параметра `v` в восьмибайтовый массив и возвращает результат
`public static byte[] GetBytes(short v)`	Преобразует значение параметра `v` в двубайтовый массив и возвращает результат
`public static byte[] GetBytes(uint v)`	Преобразует значение параметра `v` в четырехбайтовый массив и возвращает результат
`public static byte[] GetBytes(ulong v)`	Преобразует значение параметра `v` в восьмибайтовый массив и возвращает результат
`public static byte[] GetBytes(ushort v)`	Преобразует значение параметра `v` в двубайтовый массив и возвращает результат
`public static double Int64BitsToDouble(long v)`	Преобразует значение параметра `v` в значение с плавающей точкой типа `double` и возвращает результат
`public static bool ToBoolean(byte[] a, int idx)`	Преобразует элемент `a[idx]` байтового массива `a` в его `bool`-эквивалент и возвращает результат. Ненулевое значение преобразуется в значение `true`, а нулевое — в `false`
`public static char ToChar(byte[] a, int start)`	Преобразует два байта, начиная с элемента `a[start]`, в соответствующий `char`-эквивалент и возвращает результат

Метод	Описание
`public static double ToDouble(byte[] a, int start)`	Преобразует восемь байтов, начиная с элемента `a[start]`, в соответствующий `double`-эквивалент и возвращает результат
`public static short ToInt16(byte[] a, int start)`	Преобразует два байта, начиная с элемента `a[start]`, в соответствующий `short`-эквивалент и возвращает результат
`public static int ToInt32(byte[] a, int start)`	Преобразует четыре байта, начиная с элемента `a[start]`, в соответствующий `int`-эквивалент и возвращает результат
`public static long ToInt64(byte[] a, int start)`	Преобразует восемь байтов, начиная с элемента `a[start]`, в соответствующий `long`-эквивалент и возвращает результат
`public static float ToSingle(byte[] a, int start)`	Преобразует четыре байта, начиная с элемента `a[start]`, в соответствующий `float`-эквивалент и возвращает результат
`public static string ToString(byte[] a)`	Преобразует байты массива `a` в строку. Строка содержит шестнадцатеричные значения (связанные с этими байтами), разделенные дефисами
`public static string ToString(byte[] a, int start)`	Преобразует байты массива `a`, начиная с элемента `a[start]`, в строку. Строка содержит шестнадцатеричные значения (связанные с этими байтами), разделенные дефисами
`public static string ToString(byte[] a, int start, int count)`	Преобразует `count` байт массива `a`, начиная с элемента `a[start]`, в строку. Строка содержит шестнадцатеричные значения (связанные с этими байтами), разделенные дефисами
`public static ushort ToUInt16(byte[] a, int start)`	Преобразует два байта, начиная с элемента `a[start]`, в соответствующий `ushort`-эквивалент и возвращает результат
`public static uint ToUInt32(byte[] a, int start)`	Преобразует четыре байта, начиная с элемента `a[start]`, в соответствующий `uint`-эквивалент и возвращает результат
`public static ulong ToUInt64(byte[] a, int start)`	Преобразует восемь байтов, начиная с элемента `a[start]`, в соответствующий `ulong`-эквивалент и возвращает результат

Генерирование случайных чисел с помощью класса Random

Чтобы сгенерировать последовательность псевдослучайных чисел, используйте класс `Random`. Последовательности случайных чисел используются во многих ситуациях, в том числе при моделировании и проведении имитационных экспериментов. Начало такой последовательности определяется некоторым начальным числом, которое автоматически предоставляется классом `Random` или задается явным образом.

В классе `Random` определены следующие два конструктора:

```
public Random()
public Random(int seed)
```

С помощью первой версии конструктора создается объект класса Random, который для вычисления начального числа последовательности случайных чисел использует системное время. При использовании второй версии конструктора начальное число задается в параметре *seed*.

Методы, определенные в классе Random, перечислены в табл. 19.14.

Таблица 19.14. Методы, определенные в классе Random

Метод	*Описание*
`public virtual int Next()`	Возвращает следующее случайное число типа `int`, которое будет находиться в диапазоне 0-`Int32.MaxValue`-1, включительно
`public virtual int Next(int upperBound)`	Возвращает следующее случайное число типа `int`, которое будет находиться в диапазоне 0- *upperBound*-1, включительно
`public virtual int Next(int lowerBound, int upperBound)`	Возвращает следующее случайное число типа `int`, которое будет находиться в диапазоне *lowerBound*-*upperBound*-1, включительно
`public virtual void NextBytes(byte[] buf)`	Заполняет буфер *buf* последовательностью случайных целых чисел. Каждый байт в массиве будет находиться в диапазоне 0-`Byte.MaxValue`-1, включительно
`public virtual double NextDouble()`	Возвращает следующее случайное число из последовательности (представленное в форме с плавающей точкой), которое будет больше или равно числу 0,0 и меньше 1,0
`protected virtual double Sample()`	Возвращает следующее случайное число из последовательности (представленное в форме с плавающей точкой), которое будет больше или равно числу 0,0 и меньше 1,0. Чтобы создать несимметричное или специализированное распределение, этот метод необходимо переопределить в производном классе

Рассмотрим программу, в которой демонстрируется использование класса Random посредством создания парных результатов игры в кости:

```
// Автоматизированная игра в кости.

using System;

class RandDice {
  public static void Main() {
    Random ran = new Random();

    Console.Write(ran.Next(1, 7) + " ");
    Console.WriteLine(ran.Next(1, 7));
  }
}
```

Вот как выглядят результаты нашей "игры" в три хода:

```
5 2
4 4
1 6
```

Работа программы начинается с создания объекта класса Random. Затем она запрашивает два случайных числа, которые находятся в диапазоне 1–6.

Управление памятью и класс GC

В классе GC инкапсулировано C#-средство сбора мусора. Методы, определенные в этом классе, представлены в табл. 19.15. В нем также определено следующее свойство, предназначенное только для чтения:

```
public static int MaxGeneration { get; }
```

Свойство MaxGeneration содержит номер поколения самой старой области выделенной памяти. При каждом выделении памяти (т.е при использовании оператора new) новой выделяемой области присваивается номер поколения, равный нулю Номера, соответствующие ранее выделенным областям памяти, при этом увеличиваются. Следовательно, свойство MaxGeneration означает номер области памяти, которая была выделена раньше других Наличие номеров поколений способствует повышению эффективности процесса сбора мусора.

В большинстве приложений программисты не пользуются возможностями класса GC. Но в отдельных случаях они могут оказаться весьма полезными Например, метод Collect() позволяет выполнять сбор мусора в удобное для вас время. Обычно это происходит в моменты, неизвестные вашей программе. Но поскольку процесс сбора мусора занимает некоторое время, у вас могут быть вполне обоснованные причины для того, чтобы это не происходило при выполнении критических с точки зрения времени задач, либо вы хотели бы для сбора мусора и других вспомогательных операций использовать вынужденные периоды ожидания (простоя).

GC — это sealed-класс, т е он не может иметь потомков

Таблица 19.15. Методы, определенные в классе GC

Метод	*Описание*
`public static void Collect()`	Инициализирует процесс сбора мусора
`public static void Collect(int MaxGen)`	Инициализирует процесс сбора мусора для областей памяти с номерами поколений от 0 до *MaxGen*
`public static int GetGeneration(object o)`	Возвращает номер поколения для области памяти, на которую ссылается параметр *o*
`public static int GetGeneration(WeakReference o)`	Возвращает номер поколения для области памяти, адресуемой "слабой" ссылкой, заданной параметром *o* Наличие "слабой" ссылки не защищает объект от угрозы подвергнуться процессу сбора мусора
`public static long GetTotalMemory(bool collect)`	Возвращает общий объем выделенной памяти (в байтах) на данный момент Если параметр *collect* равен значению true, до выдачи результата выполняется сбор мусора
`public static void KeepAlive(object o)`	Создает ссылку на объект *o*, тем самым защищая его от угрозы подвергнуться операции сбора мусора
`public static void ReRegisterForFinalize( object o)`	Вызывает выполнение деструктора Этот метод аннулирует действие метода SuppressFinalize()
`public static void SuppressFinalize(object o)`	Препятствует выполнению деструктора
`public static void WaitForPendingFinalizers()`	Прекращает выполнение вызывающего потока до тех пор, пока не будут выполнены все незаконченные деструкторы

Класс Object

`Object` — это класс, который лежит в основе C#-типа `object`. Члены класса `Object` рассматривались в главе 11, но ввиду его центральной роли в C# (и ради удобства читателя) его методы снова приводятся в табл. 19.16. В классе `Object` определен один конструктор:

```
Object()
```

Этот конструктор создает пустой объект.

Таблица 19.16. Методы, определенные в классе `Object`

Метод	*Назначение*
`public virtual bool Equals(object ob)`	Возвращает значение `true`, если вызывающий объект является таким же, как объект, адресуемый параметром *ob*. В противном случае возвращает значение `false`
`public static bool Equals(object ob1, object ob2)`	Возвращает значение `true`, если объект *ob1* является таким же, как объект *ob2*. В противном случае возвращает значение `false`
`protected Finalize()`	Выполняет завершающие действия перед процессом сбора мусора. В C# метод `Finalize()` доступен через деструктор
`public virtual int GetHashCode()`	Возвращает хеш-код, связанный с вызывающим объектом
`public Type GetType()`	Получает тип объекта во время выполнения программы
`protected object MemberwiseClone()`	Выполняет "поверхностное копирование" объекта, т.е. копируются члены, но не объекты, на которые ссылаются эти члены
`public static bool ReferenceEquals(object ob1, object ob2)`	Возвращает значение `true`, если объекты *ob1* и *ob2* ссылаются на один и тот же объект. В противном случае возвращает значение `false`
`public virtual string ToString()`	Возвращает строку, которая описывает объект

Интерфейс IComparable

Во многих классах необходимо реализовать интерфейс `IComparable`, поскольку он с помощью методов, определенных в C#-библиотеке, позволяет сравнить два объекта. Интерфейс `IComparable` легко реализовать, поскольку он состоит только из одного метода:

```
int CompareTo(object v)
```

Этот метод сравнивает вызывающий объект со значением параметра *v*. Метод возвращает положительное число, если вызывающий объект больше объекта *v*, нуль, если два сравниваемых объекта равны, и отрицательное число, если вызывающий объект меньше объекта *v*.

Интерфейс IConvertible

Интерфейс IConvertible реализован всеми структурами нессылочных типов. Он определяет преобразование типов. Как правило, в создаваемых программистами классах этот интерфейс реализовать не нужно.

Интерфейс ICloneable

Реализуя интерфейс ICloneable, вы позволяете создавать копию объекта. В интерфейсе ICloneable определен только один метод:

```
object Clone()
```

Этот метод создает копию вызывающего объекта. От того, как реализован метод Clone(), зависит вид создаваемой копии. Существует два вида копий: детальная и поверхностная. При создании копии первого вида копия и оригинал совершенно независимы. Следовательно, если исходный объект содержит ссылку на другой объект *O*, то в результате детального копирования будет также создана копия объекта *O*. В поверхностной копии копируются члены, но не объекты, на которые ссылаются эти члены. Если объект ссылается на другой объект *O*, то по окончании поверхностного копирования как копия, так и исходный объект будут ссылаться на один и тот же объект *O*, и любые изменения, вносимые в объект *O*, отразятся и на копии, и на оригинале. Обычно метод Clone() реализуется так, чтобы выполнялось детальное копирование. Поверхностные копии можно создавать с помощью метода MemberwiseClone(), который определен в классе Object.

Рассмотрим пример, который иллюстрирует использование интерфейса ICloneable. В следующей программе создается класс Test, который содержит ссылку на класс X. В классе Test для создания детальной копии используется метод Clone().

```
// Демонстрация использования интерфейса ICloneable.

using System;

class X {
  public int a;

  public X(int x) { a = x; }
}

class Test : ICloneable {
  public X o;
  public int b;

  public Test(int x, int y) {
    o = new X(x);
    b = y;
  }

  public void show(string name) {
    Console.Write("Значения объекта " + name + " : ");
    Console.WriteLine("o.a: {0}, b: {1}", o.a, b);
  }
```

```
  // Создаем детальную копию вызывающего объекта.
  public object Clone() {
    Test temp = new Test(o.a, b);
    return temp;
  }

}

class CloneDemo {
  public static void Main() {
    Test ob1 = new Test(10, 20);

    ob1.show("ob1");

    Console.WriteLine(
       "Создаем объект ob2 как клон объекта ob1.");
    Test ob2 = (Test) ob1.Clone();

    ob2.show("ob2");

    Console.WriteLine("Заменяем член ob1.o.a числом 99, "
                    + "а член ob1.b числом 88.");
    ob1.o.a = 99;
    ob1.b = 88;

    ob1.show("ob1");
    ob2.show("ob2");
  }
}
```

Результаты выполнения этой программы таковы:

```
Значения объекта ob1 : o.a: 10, b: 20
Создаем объект ob2 как клон объекта ob1.
Значения объекта ob2 : o.a: 10, b: 20
Заменяем член ob1.o.a числом 99, а член ob1.b числом 88.
Значения объекта ob1 : o.a: 99, b: 88
Значения объекта ob2 : o.a: 10, b: 20
```

Судя по приведенным результатам, объект `ob2` является копией объекта `ob1`, но `ob1` и `ob2` — отдельные объекты. Изменение одного никак не отражается на другом. Это достигается за счет того, что для копии создается новый объект `X`, которому присваивается то же значение, которое имеет объект `X` в оригинале.

Для реализации поверхностного копирования достаточно организовать внутри метода `Clone()` вызов метода `MemberwiseClone()`, определенного в классе `Object`. Попробуйте, например, изменить определение метода `Clone()` из предыдущей программы таким:

```
// Создаем поверхностную копию вызывающего объекта.
public object Clone() {
  Test temp = (Test) MemberwiseClone();
  return temp;
}
```

После внесения указанных изменений результаты выполнения той же программы будут другими:

```
Значения объекта ob1 : o.a: 10, b: 20
Создаем объект ob2 как клон объекта ob1.
Значения объекта ob2 : o.a: 10, b: 20
Заменяем член ob1.o.a числом 99, а член ob1.b числом 88.
```

```
Значения объекта ob1 : o.a: 99, b: 88
Значения объекта ob2 : o.a: 99, b: 20
```

Обратите внимание на то, что член `o` в объекте `ob1` и член `o` в объекте `ob2` ссылаются на один и тот же объект `X`. Теперь изменение одного объекта отражается на другом. Но `int`-поля `b` в каждом объекте по-прежнему независимы, поскольку имеют нессылочный тип и доступ к ним осуществляется не через ссылки.

Интерфейсы `IFormatProvider` и `IFormattable`

Интерфейс `IFormatProvider` определяет один метод `GetFormat()`, который возвращает объект, управляющий форматированием данных строки, удобной для восприятия человеком. Общий формат метода `GetFormat()` таков:

```
object GetFormat(Type fmt)
```

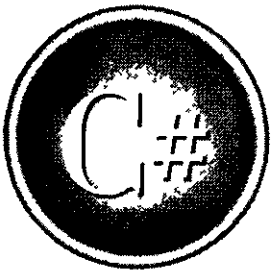
Здесь параметр *fmt* задает формат объекта. Форматирование описано в главе 20.

Интерфейс `IFormattable` поддерживает форматирование выводимого результата в удобной для восприятия человеком форме. В интерфейсе `IFormattable` определен следующий метод:

```
string ToString(string fmt, IFormatProvider fmtpvdr)
```

Здесь параметр *fmt* задает инструкции форматирования, а параметр *fmtpvdr* — источник (поставщик) формата. Подробно форматирование описано в главе 20.

Полный справочник по



Глава 20

Строки и форматирование

Эта глава посвящена классу `String`. Как известно каждому программисту, без обработки строк не обходится практически ни одна программа. Поэтому в классе `String` определено множество методов, свойств и полей, которые предлагают программисту богатую палитру инструментов, позволяющих создавать строки и манипулировать ими. С темой обработки строк тесно связана тема форматирования данных с целью приведения их к форме, удобной для восприятия человеком. Используя возможности соответствующей подсистемы, можно нужным образом форматировать числовые типы языка C#, дату и время, а также перечисления.

Строки в C#

Обзор средств обработки строк в C# был представлен в главе 7, и повторения "пройденного" здесь не предполагается. Но прежде чем перейти к рассмотрению класса `String`, имеет смысл поговорить об особенностях реализации строк в языке C#.

Во всех языках программирования *строка* (string) представляет собой последовательность символов, но точная реализация такой последовательности меняется при переходе от одного языка к другому. В языке C++ строки реализованы в виде массивов символов, но в C# все обстоит иначе. Строки в C# — это объекты встроенного типа данных `string`. Поэтому `string` является ссылочным типом данных. Более того, `string` — это C#-имя стандартного строкового типа .NET-среды `System.String`. Таким образом, C#-строка имеет доступ ко всем методам, свойствам, полям и операторам, определенным в классе `String`.

В созданной строке последовательность составляющих ее символов изменить нельзя. Благодаря этому ограничению строки в C# (по сравнению с другими языками) реализованы более эффективно. И хотя это ограничение может показаться серьезным недостатком, на самом деле оно таковым не является. Если вам понадобится строка, которая представляет собой вариацию "на тему" другой, уже существующей строки, просто создайте новую строку, которая будет содержать желаемые изменения, а затем удалите исходный вариант, если он вам больше не нужен. Поскольку неиспользуемые строковые объекты автоматически удаляются подсистемой сбора мусора, можно не беспокоиться об исключенных "из обращения" строках. Однако вы должны четко понимать, что ссылочные переменные типа `string`, конечно же, могут менять объекты, на которые они ссылаются. А вот последовательность символов конкретного `string`-объекта после его создания изменить уже нельзя.

Чтобы создать строку, которую можно изменять, в C# предусмотрен класс `StringBuilder`, определенный в пространстве имен `System.Text`. И все же в большинстве случаев лучше использовать тип `string`, а не класс `StringBuilder`.

Класс `String`

Класс `String` определен в пространстве имен `System`. Он реализует интерфейсы `IComparable`, `ICloneable`, `IConvertible` и `IEnumerable`. `String` — это `sealed`-класс, т.е. из него нельзя создать производный класс. Класс `String` содержит C#-средства обработки строк. Он лежит в основе встроенного C#-типа `string` и является частью среды .NET Framework. Следующие разделы посвящены детальному рассмотрению класса `String`.

Конструкторы класса `String`

В классе `String` определено несколько конструкторов, которые позволяют создавать строки различными способами. Чтобы создать строку из символьного массива, используйте один из следующих конструкторов:

```
public String(char[] chrs)
public String(char[] chrs, int start, int count)
```

Первый формат предназначен для построения строки, которая будет состоять из символов, содержащихся в массиве *chrs*. Строка, создаваемая с помощью второго формата, будет состоять из *count* символов, взятых из массива *chrs*, начиная с символа, индекс которого задан параметром *start*.

Существует также возможность создать строку, содержащую заданный символ, повторенный нужное количество раз. Для этого используйте этот конструктор:

```
public String(char ch, int count)
```

Здесь параметр *ch* задает символ, который будет повторен *count* раз.

Используя один из следующих конструкторов, можно создать строку, заданную указателем на символьный массив:

```
unsafe public String(char* chrs)
unsafe public String(char* chrs, int start, int count)
```

Конструктор первого формата предназначен для построения строки, содержащей символы, на которые указывает параметр *chrs*. При этом предполагается, что параметр *chrs* указывает на массив с завершающим нулем (символом конца строки). Строка, создаваемая с помощью конструктора второго формата, будет состоять из *count* символов, взятых из массива, адресуемого указателем *chrs*, начиная с символа, индекс которого задан параметром *start*.

Используя один из следующих конструкторов, можно создать строку, заданную указателем на массив байтов:

```
unsafe public String(sbyte* chrs)
unsafe public String(sbyte* chrs, int start, int count)
unsafe public String(sbyte* chrs, int start, int count, Encoding en)
```

Конструктор первого формата предназначен для построения строки, содержащей байты, на которые указывает параметр *chrs*. При этом предполагается, что параметр *chrs* указывает на массив с завершающим нулем (символом конца строки). Строка, создаваемая с помощью конструктора второго формата, будет состоять из *count* байтов, взятых из массива, адресуемого указателем *chrs*, начиная с байта, индекс которого задан параметром *start*. Третий формат конструктора позволяет указать тип кодирования байтов. По умолчанию используется тип `ASCIIEncoding`. Класс `Encoding` определен в пространстве имен `System.Text`.

Строковый литерал создает строковый объект автоматически. Поэтому строковый объект часто инициализируется присваиванием ему строкового литерала. Вот пример:

```
string str = "новая строка";
```

Поле, индексатор и свойство класса `String`

В классе `String` определено только одно поле:

```
public static readonly string Empty
```

Поле `Empty` определяет пустую строку, т.е. строку, которая не содержит символов. Не следует путать ее с нулевой (пустой) ссылкой типа `String`, которая просто ссылается на несуществующий объект.

В классе `String` определен единственный индексатор, предназначенный только для чтения:

```
public char this[int idx] { get; }
```

Этот индексатор позволяет получить символ по заданному индексу. Подобно массивам, индексация в строках начинается с нуля. Поскольку объекты `String` не подлежат изменению, в том, что класс `String` поддерживает индексатор, предназначенный *только для чтения*, есть здравый смысл.

В классе `String` определено единственное свойство, предназначенное только для чтения:

```
public int Length { get; }
```

Свойство `Length` возвращает количество символов, содержащихся в строке.

Операторы класса `String`

В классе `String` реализована перегрузка двух операторов: "==" и "`!=`". Чтобы узнать, равны ли две строки, используйте оператор "==". Если оператор "==" применяется к объектным ссылкам, то он определяет, ссылаются ли они обе на один и тот же объект. Но если оператор "==" применяется к двум ссылкам типа `String`, то сравнивается содержимое самих строк. То же справедливо и для оператора "`!=`": при сравнении `String`-объектов сравнивается содержимое строк. Но что касается других операторов отношений (например, "<" или ">="), то они сравнивают ссылки точно так же, как объекты любых других типов. Чтобы узнать, например, больше (меньше) ли одна строка другой, используйте метод `Compare()`, определенный в классе `String`.

Методы класса `String`

В классе `String` определено множество различных методов. При этом многие из них имеют два или больше перегруженных форматов. Поэтому вместо бессмысленного их перечисления, рассмотрим лишь наиболее употребимые методы и продемонстрируем их использование на конкретных примерах.

Сравнение строк

Из всех операций обработки строк, возможно, чаще всего используется операция сравнения одной строки с другой. Поэтому в классе `String` предусмотрен широкий выбор методов сравнения, которые перечислены в табл. 20.1. Самый универсальный из них — метод `Compare()`. Он может сравнивать две строки целиком или по частям, причем с учетом (или без) прописного или строчного варианта букв (т.е. регистра клавиатуры). В общем случае при сравнении строк, т.е. при определении того, больше ли одна строка другой, меньше или они равны, используется лексикографический порядок. При этом можно также задать специальную информацию (форматы данных, присущие естественному языку, диалекту или территориальному образованию), которая может повлиять на результат сравнения.

Использование нескольких версий метода `Compare()` демонстрируется в следующей программе:

```
// Сравнение строк.

using System;

class CompareDemo {
  public static void Main() {
    string str1 = "один";
    string str2 = "один";
```

```
    string str3 = "ОДИН";
    string str4 = "два";
    string str5 = "один, два";


    if(String.Compare(str1, str2) == 0)
      Console.WriteLine(str1 + " и " + str2 +
                        " равны.");
    else
      Console.WriteLine(str1 + " и " + str2 +
                        " не равны.");


    if(String.Compare(str1, str3) == 0)
      Console.WriteLine(str1 + " и " + str3 +
                        " равны.");
    else
      Console.WriteLine(str1 + " и " + str3 +
                        " не равны.");

    if(String.Compare(str1, str3, true) == 0)
      Console.WriteLine(str1 + " и " + str3 +
                        " равны без учета регистра.");
    else
      Console.WriteLine(str1 + " и " + str3 +
                        " не равны без учета регистра.");

     if(String.Compare(str1, str5) == 0)
      Console.WriteLine(str1 + " и " + str5 +
                        " равны.");
    else
      Console.WriteLine(str1 + " и " + str5 +
                        " не равны.");

    if(String.Compare(str1, 0, str5, 0, 3) == 0)
      Console.WriteLine("Первая часть строки " + str1 +
                        " и " +
                        str5 + " равны.");
    else
      Console.WriteLine("Первая часть строки " + str1 +
                        " и " +
                        str5 + " не равны.");

    int result = String.Compare(str1, str4);
    if(result < 0)
      Console.WriteLine(str1 + " меньше " + str4);
    else if(result > 0)
      Console.WriteLine(str1 + " больше " + str4);
    else
      Console.WriteLine(str1 + " равно " + str4);
  }
}
```

Результаты выполнения этой программы таковы:

```
один и один равны.
один и ОДИН не равны.
один и ОДИН равны без учета регистра.
один и один, два не равны.
Первая часть строки один и один, два равны.
один больше два
```

Таблица 20.1. Методы сравнения, определенные в классе `String`

Метод	Описание
`public static int Compare(string str1, string str2)`	Сравнивает строку, адресуемую параметром *str1*, со строкой, адресуемой параметром *str2*. Возвращает положительное число, если строка *str1* больше *str2*, отрицательное число, если *str1* меньше *str2*, и нуль, если строки *str1* и *str2* равны
`Public static int Compare(string str1, string str2, bool ignoreCase)`	Сравнивает строку, адресуемую параметром *str1*, со строкой, адресуемой параметром *str2*. Возвращает положительное число, если строка *str1* больше *str2*, отрицательное число, если *str1* меньше *str2*, и нуль, если строки *str1* и *str2* равны. Если параметр *ignoreCase* равен значению `true`, при сравнении не учитываются различия между прописным и строчным вариантами букв. В противном случае эти различия учитываются
`public static int Compare(string str1, string str2, bool ignoreCase, CultureInfo ci)`	Сравнивает строку, адресуемую параметром *str1*, со строкой, адресуемой параметром *str2*, с использованием специальной информации (связанной с конкретным естественным языком, диалектом или территориальным образованием), переданной в параметре *ci*. Возвращает положительное число, если строка *str1* больше *str2*, отрицательное число, если *str1* меньше *str2*, и нуль, если строки *str1* и *str2* равны. Если параметр *ignoreCase* равен значению `true`, при сравнении не учитываются различия между прописным и строчным вариантами букв. В противном случае эти различия учитываются. Класс `CultureInfo` определен в пространстве имен `System.Globalization`
`public static int Compare(string str1, int start1, string str2, int start2, int count)`	Сравнивает части строк, заданных параметрами *str1* и *str2*. Сравнение начинается со строковых элементов *str1*[*start1*] и *str2*[*start2*] и включает *count* символов. Метод возвращает положительное число, если часть строки *str1* больше части строки *str2*, отрицательное число, если *str1*-часть меньше *str2*-части, и нуль, если сравниваемые части строк *str1* и *str2* равны
`public static int Compare(string str1, int start1, string str2, int start2, int count, bool ignoreCase)`	Сравнивает части строк, заданных параметрами *str1* и *str2*. Сравнение начинается со строковых элементов *str1*[*start1*] и *str2*[*start2*] и включает *count* символов. Метод возвращает положительное число, если часть строки *str1* больше части строки *str2*, отрицательное число, если *str1*-часть меньше *str2*-части, и нуль, если сравниваемые части строк *str1* и *str2* равны. Если параметр *ignoreCase* равен значению `true`, при сравнении не учитываются различия между прописным и строчным вариантами букв. В противном случае эти различия учитываются
`public static int Compare(string str1, int start1, string str2, int start2, int count, bool ignoreCase, CultureInfo ci)`	Сравнивает части строк, заданных параметрами *str1* и *str2*, с использованием специальной информации (связанной с конкретным естественным языком, диалектом или территориальным образованием), переданной в параметре *ci*. Сравнение начинается со строковых элементов *str1*[*start1*] и *str2*[*start2*] и включает *count* символов. Метод возвращает положительное число, если часть строки *str1* больше части строки *str2*, отрицательное число, если *str1*-часть меньше *str2*-части, и нуль, если сравниваемые части строк *str1* и *str2* равны. Если параметр *ignoreCase* равен значению `true`, при сравнении не учитываются различия между прописным и строчным вариантами букв. В противном случае эти различия учитываются. Класс `CultureInfo` определен в пространстве имен `System.Globalization`

Окончание табл. 20.1

Метод	Описание
`public static int CompareOrdinal(string str1, string str2)`	Сравнивает строку, адресуемую параметром `str1`, со строкой, адресуемой параметром `str2`, независимо от языка, диалекта или территориального образования. Возвращает положительное число, если строка `str1` больше `str2`, отрицательное число, если `str1` меньше `str2`, и нуль, если строки `str1` и `str2` равны
`public static int CompareOrdinal(string str1, int start1, string str2, int start2, int count)`	Сравнивает части строк, заданных параметрами `str1` и `str2`, независимо от языка, диалекта или территориального образования. Сравнение начинается со строковых элементов `str1[start1]` и `str2[start2]` и включает `count` символов. Метод возвращает положительное число, если часть строки `str1` больше части строки `str2`, отрицательное число, если `str1`-часть меньше `str2`-части, и нуль, если сравниваемые части строк `str1` и `str2` равны
`Public int CompareTo(object str)`	Сравнивает вызывающую строку со строкой, заданной параметром `str`. Возвращает положительное число, если вызывающая строка больше строки `str`, отрицательное число, если вызывающая строка меньше строки `str`, и нуль, если сравниваемые строки равны
`Public int CompareTo(string str)`	Сравнивает вызывающую строку со строкой, заданной параметром `str`. Возвращает положительное число, если вызывающая строка больше строки `str`, отрицательное число, если вызывающая строка меньше строки `str`, и нуль, если сравниваемые строки равны

Конкатенация строк

Существует два способа конкатенации (объединения) двух или больше строк. Во-первых, как показано в главе 7, можно использовать для этого оператор "+". Во-вторых, можно применить один из методов конкатенации, определенных в классе `String`. Несмотря на то что оператор "+" — самое простое решение во многих случаях, методы конкатенации предоставляют дополнительные возможности.

Метод, который выполняет конкатенацию, именуется `Concat()`, а его простейший формат таков:

```
public static string Concat(string str1, string str2)
```

Метод возвращает строку, которая содержит строку `str2`, присоединенную к концу строки `str1`.

Еще один формат метода `Concat()` позволяет объединить три строки:

```
public static string Concat(string str1,
                            string str2,
                            string str3)
```

При вызове этой версии возвращается строка, которая содержит конкатенированные строки `str1`, `str2` и `str3`. По правде говоря, для выполнения описанных выше операций все же проще использовать оператор "+", а не метод `Concat()`.

А вот следующая версия метода `Concat()` объединяет произвольное число строк, что делает ее весьма полезной для программиста:

```
public static string Concat(params string[] strs)
```

Здесь метод Concat() принимает переменное число аргументов и возвращает результат их конкатенации. Использование этой версии метода Concat() демонстрируется в следующей программе:

```
// Демонстрация использования метода Concat().

using System;

class ConcatDemo {
  public static void Main() {

    string result = String.Concat(
                              "Мы ", "тестируем ",
                              "один ", "из ", "методов ",
                              "конкатенации ", "класса ",
                              "String.");

    Console.WriteLine("Результат: " + result);

  }
}
```

Результаты выполнения этой программы таковы:

```
Результат: Мы тестируем один из методов конкатенации класса String.
```

Некоторые версии метода Concat() принимают не string-, а object-ссылки. Они извлекают строковое представление из передаваемых им объектов и возвращают строку, содержащую конкатенированные строки. Форматы этих версий метода Concat() таковы:

```
public static string Concat(object v1, object v2)
public static string Concat(object v1,
                            object v2,
                            object v3)
public static string Concat(params object[] v)
```

Первая версия возвращает строку, которая содержит строковое представление объекта *v2*, присоединенное к концу строкового представления объекта *v1*. Вторая возвращает строку, которая содержит конкатенированные строковые представления объектов *v1*, *v2* и *v3*. Третья возвращает строку, содержащую конкатенированные строковые представления аргументов, переданных в виде "сборного" объекта *v*. Чтобы вы могли оценить потенциальную пользу этих методов, рассмотрим следующую программу:

```
// Еще один способ использования метода Concat().

using System;

class ConcatDemo {
  public static void Main() {

    string result = String.Concat("Привет ", 10, " ",
                                  20.0, " ",
                                  false, " ",
                                  23.45M);

    Console.WriteLine("Результат: " + result);
  }
}
```

Результаты выполнения этой программы таковы:

```
Результат: Привет 10 20 False 23.45
```

В этом примере метод `Concat()` конкатенирует строковые представления различных типов данных. Для получения строкового представления каждого аргумента здесь вызывается метод `ToString()`, связанный с соответствующим аргументом. Так, для значения 10 вызывается метод `Int32.ToString()`. Методу `Concat()` остается лишь объединить эти строки и возвратить результат. Этот формат метода `Concat()` очень удобен, поскольку позволяет программисту не получать вручную строковые представления до самой конкатенации.

Поиск строки

В классе `String` есть два набора методов, которые позволяют найти заданную строку (подстроку либо отдельный символ). При этом можно заказать поиск первого либо последнего вхождения искомого элемента. Чтобы отыскать первое вхождение символа или подстроки, используйте метод `IndexOf()`, который имеет два таких формата:

```
public int IndexOf(char ch)
public int IndexOf(string str)
```

Метод `IndexOf()`, используемый в первом формате, возвращает индекс первого вхождения символа *ch* в вызывающей строке. Второй формат позволяет найти первое вхождение строки *str*. В обоих случаях возвращается значение −1, если искомый элемент не найден.

Чтобы отыскать последнее вхождение символа или подстроки, используйте метод `LastIndexOf()`, который имеет два таких формата:

```
public int LastIndexOf(char ch)
public int LastIndexOf(string str)
```

Метод `LastIndexOf()`, используемый в первом формате, возвращает индекс последнего вхождения символа *ch* в вызывающей строке. Второй формат позволяет найти последнее вхождение строки *str*. В обоих случаях возвращается значение −1, если искомый элемент не найден.

В классе `String` определено два дополнительных метода поиска: `IndexOfAny()` и `LastIndexOfAny()`. Они выполняют поиск первого или последнего символа, который совпадает с любым элементом заданного набора символов. Вот их форматы:

```
public int IndexOfAny(char[] a)
public int LastIndexOfAny(char[] a)
```

Метод `IndexOfAny()` возвращает индекс первого вхождения любого символа из массива *a*, который обнаружится в вызывающей строке. Метод `LastIndexOfAny()` возвращает индекс последнего вхождения любого символа из массива *a*, который обнаружится в вызывающей строке. В обоих случаях возвращается значение −1, если совпадение не обнаружится.

При работе со строками часто возникает необходимость узнать, начинается ли строка с заданной подстроки либо оканчивается ею. Для таких ситуаций используются методы `StartsWith()` и `EndsWith()`:

```
public bool StartsWith(string str)
public bool EndsWith(string str)
```

Метод `StartsWith()` возвращает значение `true`, если вызывающая строка начинается с подстроки, переданной в параметре *str*. Метод `EndsWith()` возвращает значение `true`, если вызывающая строка оканчивается подстрокой, переданной в параметре *str*. В случае неудачного исхода оба метода возвращают значение `false`.

Использование методов поиска строк демонстрируется в следующей программе:

```
// Поиск строк.

using System;

class StringSearchDemo {
  public static void Main() {
    string str =
        "C# обладает мощными средствами обработки строк.";
    int idx;

    Console.WriteLine("str: " + str);

    idx = str.IndexOf('с');
    Console.WriteLine(
            "Индекс первого вхождения буквы 'с': " + idx);

    idx = str.LastIndexOf('с');
    Console.WriteLine(
          "Индекс последнего вхождения буквы 'с': " + idx);

    idx = str.IndexOf("ми");
    Console.WriteLine(
             "Индекс первого вхождения подстроки \"ми\": "
           + idx);

    idx = str.LastIndexOf("ми");
    Console.WriteLine(
           "Индекс последнего вхождения подстроки \"ми\": "
         + idx);

    char[] chrs = { 'а', 'б', 'в' };
    idx = str.IndexOfAny(chrs);
    Console.WriteLine(
         "Индекс первой из букв 'а', 'б' или 'в': " + idx);

    if(str.StartsWith("C# обладает"))
      Console.WriteLine(
              "str начинается с подстроки \"C# обладает\"");

    if(str.EndsWith("строк."))
      Console.WriteLine(
               "str оканчивается подстрокой \"строк.\"");
  }
}
```

Результаты выполнения этой программы таковы:

```
str: C# обладает мощными средствами обработки строк.
Индекс первого вхождения буквы 'с': 20
Индекс последнего вхождения буквы 'с': 41
Индекс первого вхождения подстроки "ми": 17
Индекс последнего вхождения подстроки "ми": 28
Индекс первой из букв 'а', 'б' или 'в': 4
str начинается с подстроки "C# обладает"
str оканчивается подстрокой "строк."
```

Некоторые методы поиска имеют дополнительные форматы, которые позволяют начать поиск с заданного индекса или указать диапазон поиска. Все версии методов

поиска символов в строке, которые определены в классе `String`, приведены в табл. 20.2.

Таблица 20.2. Методы поиска символов в строке, определенные в классе `String`

Метод	*Описание*
`public bool EndsWith(string str)`	Возвращает значение `true`, если вызывающая строка оканчивается подстрокой, переданной в параметре `str`. В противном случае метод возвращает значение `false`
`public int IndexOf(char ch)`	Возвращает индекс первого вхождения символа `ch` в вызывающей строке. Возвращает значение -1, если искомый символ не обнаружен
`public int IndexOf(string str)`	Возвращает индекс первого вхождения подстроки `str` в вызывающей строке. Возвращает значение -1, если искомая подстрока не обнаружена
`public int IndexOf(char ch, int start)`	Возвращает индекс первого вхождения символа `ch` в вызывающей строке. Поиск начинается с элемента, индекс которого задан параметром `start`. Возвращает значение -1, если искомый символ не обнаружен
`public int IndexOf(string str, int start)`	Возвращает индекс первого вхождения подстроки `str` в вызывающей строке. Поиск начинается с элемента, индекс которого задан параметром `start`. Возвращает значение -1, если искомая подстрока не обнаружена
`public int IndexOf(char ch, int start, int count)`	Возвращает индекс первого вхождения символа `ch` в вызывающей строке. Поиск начинается с элемента, индекс которого задан параметром `start`, и охватывает `count` элементов. Метод возвращает значение -1, если искомый символ не обнаружен
`public int IndexOf(string str, int start, int count)`	Возвращает индекс первого вхождения подстроки `str` в вызывающей строке. Поиск начинается с элемента, индекс которого задан параметром `start`, и охватывает `count` элементов. Метод возвращает значение -1, если искомая подстрока не обнаружена
`public int IndexOfAny(char[] a)`	Возвращает индекс первого вхождения любого символа из последовательности `a`, который будет обнаружен в вызывающей строке. Возвращает значение -1, если совпадения символов не обнаружено
`public int IndexOfAny(char[] a, int start)`	Возвращает индекс первого вхождения любого символа из последовательности `a`, который будет обнаружен в вызывающей строке. Поиск начинается с элемента, индекс которого задан параметром `start`. Метод возвращает значение -1, если совпадения символов не обнаружено
`public int IndexOfAny(char[] a, int start, int count)`	Возвращает индекс первого вхождения любого символа из последовательности `a`, который будет обнаружен в вызывающей строке. Поиск начинается с элемента, индекс которого задан параметром `start`, и охватывает `count` элементов. Метод возвращает значение -1, если совпадения символов не обнаружено
`public int LastIndexOf(char ch)`	Возвращает индекс последнего вхождения символа `ch` в вызывающей строке. Возвращает значение -1, если искомый символ не обнаружен
`public int LastIndexOf(string str)`	Возвращает индекс последнего вхождения подстроки `str` в вызывающей строке. Возвращает значение -1, если искомая подстрока не обнаружена
`public int LastIndexOf(char ch, int start)`	Возвращает индекс последнего вхождения символа `ch`, обнаруженного в пределах диапазона вызывающей строки. Поиск выполняется в обратном порядке, начиная с элемента, индекс которого задан параметром `start`, и заканчивая элементом с нулевым индексом. Метод возвращает значение -1, если искомый символ не обнаружен

Метод	Описание
`public int LastIndexOf(string str, int start)`	Возвращает индекс последнего вхождения подстроки `str`, обнаруженной в пределах диапазона вызывающей строки. Поиск выполняется в обратном порядке, начиная с элемента, индекс которого задан параметром `start`, и заканчивая элементом с нулевым индексом. Метод возвращает значение -1, если искомая подстрока не обнаружена
`public int LastIndexOf(char ch, int start, int count)`	Возвращает индекс последнего вхождения символа `ch`, обнаруженного в пределах диапазона вызывающей строки. Поиск выполняется в обратном порядке, начиная с элемента, индекс которого задан параметром `start`, и охватывая `count` элементов. Метод возвращает значение -1, если искомый символ не обнаружен
`public int LastIndexOf(string str, int start, int count)`	Возвращает индекс последнего вхождения подстроки `str`, обнаруженной в пределах диапазона вызывающей строки. Поиск выполняется в обратном порядке, начиная с элемента, индекс которого задан параметром `start`, и охватывая `count` элементов. Метод возвращает значение -1, если искомая подстрока не обнаружена
`public int LastIndexOfAny(char[] a)`	Возвращает индекс последнего вхождения любого символа из последовательности `a`, который будет обнаружен в вызывающей строке. Возвращает значение -1, если совпадения символов не обнаружено
`public int LastIndexOfAny(char[] a, int start)`	Возвращает индекс последнего вхождения любого символа из последовательности `a`, который будет обнаружен в вызывающей строке. Поиск выполняется в обратном порядке, начиная с элемента, индекс которого задан параметром `start`, и заканчивая элементом с нулевым индексом. Возвращает значение -1, если совпадения символов не обнаружено
`public int LastIndexOfAny(char[] a, int start, int count)`	Возвращает индекс последнего вхождения любого символа из последовательности `a`, который будет обнаружен в вызывающей строке. Поиск выполняется в обратном порядке, начиная с элемента, индекс которого задан параметром `start`, и охватывая `count` элементов. Возвращает значение -1, если совпадения символов не обнаружено
`public bool StartsWith(string str)`	Возвращает значение `true`, если вызывающая строка начинается строкой, переданной в параметре `str`. В противном случае метод возвращает значение `false`

Разбиение и сборка строк

Двумя важными операциями обработки строк являются разбиение (декомпозиция) и сборка. При *разбиении* строка делится на составные части. А при *сборке* строка "собирается" из отдельных частей. Для разбиения строк в классе `String` определен метод `Split()`, а для сборки — метод `Join()`.

Форматы использования метода `Split()` таковы:

```
public string[] split(params char[] seps)
public string[] split(params char[] seps, int count)
```

Метод первого формата предназначен для разбиения вызывающей строки на подстроки, которые возвращаются методом в виде строкового массива. Символы, которые отделяют подстроки одну от другой, передаются в массиве `seps`. Если параметр `seps` содержит `null`-значение, в качестве разделителя подстрок используется пробел. Метод второго формата отличается от первого тем, что ограничивает количество возвращаемых подстрок значением `count`.

Рассмотрим теперь форматы применения метода `Join()`:

```
public static string Join(string sep, string[] strs)
public static string Join(string sep, string[] strs,
                          int start, int count)
```

Метод `Join()` в первом формате возвращает строку, которая содержит конкатенированные строки, переданные в массиве *strs*. Второй формат позволяет собрать строку, которая будет содержать *count* конкатенированных строк, переданных в массиве *strs*, начиная со строки *strs*[*start*]. В обоих случаях каждая строка, составляющая результат, отделяется от следующей разделительной строкой, заданной параметром *sep*.

Использование методов `Split()` и `Join()` демонстрируется в следующей программе:

```
// Демонстрация разбиения и сборки строк.

using System;

class SplitAndJoinDemo {
  public static void Main() {
    string str =
      "Какое слово ты скажешь, такое в ответ и услышишь.";
    char[] seps = {' ', '.', ','};

    // Разбиваем строку на части.
    string[] parts = str.Split(seps);
    Console.WriteLine(
           "Результат разбиения строки на части: ");
    for(int i=0; i < parts.Length; i++)
      Console.WriteLine(parts[i]);

    // Теперь собираем эти части в одну строку.
    string whole = String.Join(" | ", parts);
    Console.WriteLine("Результат сборки: ");
    Console.WriteLine(whole);
  }
}
```

Вот какой результат получен:

```
Результат разбиения строки на части:
Какое
слово
ты
скажешь

такое
в
ответ
и
услышишь

Результат сборки:
Какое | слово | ты | скажешь |  | такое | в | ответ | и | услышишь |
```

Декомпозиция строки — важная операция обработки строк, поскольку ее часто используют для получения отдельных *лексем*, составляющих строку. Программа ведения базы данных, например, может использовать метод `Split()` для разложения такого запроса, как "отобразить все балансы, превышающие значение 100" на отдельные составляющие (например, "отобразить" и "100"). В этом процессе разделители

удаляются, в результате получаем подстроку "отобразить" (без начальных или концевых пробелов), а не " отобразить ". Эта концепция иллюстрируется следующей программой. Сначала она выделяет лексемы из строк, содержащих такие бинарные математические операции, как 10 + 5, а затем выполняет эти операции и возвращает результаты.

```
// Извлечение лексем из строк.

using System;

class TokenizeDemo {
  public static void Main() {
    string[] input = {
                      "100 + 19",
                      "100 / 3.3",
                      "-3 * 9",
                      "100 - 87"
                     };
    char[] seps = {' '};

    for(int i=0; i < input.Length; i++) {
      // Разбиваем строку на части.
      string[] parts = input[i].Split(seps);
      Console.Write("Команда: ");
      for(int j=0; j < parts.Length; j++)
        Console.Write(parts[j] + " ");

      Console.Write(", результат: ");
      double n = Double.Parse(parts[0]);
      double n2 = Double.Parse(parts[2]);

      switch(parts[1]) {
        case "+":
          Console.WriteLine(n + n2);
          break;
        case "-":
          Console.WriteLine(n - n2);
          break;
        case "*":
          Console.WriteLine(n * n2);
          break;
        case "/":
          Console.WriteLine(n / n2);
          break;
      }
    }
  }
}
```

Результаты выполнения этой программы таковы:

```
Команда: 100 + 19 , результат: 119
Команда: 100 / 3.3 , результат: 30.3030303030303
Команда: -3 * 9 , результат: -27
Команда: 100 - 87 , результат: 13
```

Удаление символов и дополнение ими строк

Иногда возникает необходимость удалить из строки начальные и концевые пробелы. Без этой операции, как правило, не обходится ни один командный процессор. Например, программа ведения базы данных способна распознать слово "print". Но пользователь мог ввести эту команду с одним или несколькими начальными либо концевыми пробелами. Поэтому, прежде чем такая команда будет передана на распознание базой данных, из нее должны быть удалены "лишние" пробелы. И наоборот, возможны ситуации, когда строку нужно дополнить пробелами, чтобы она "доросла" до определенной минимальной длины. Например, при подготовке данных для форматированного вывода необходимо позаботиться о том, чтобы каждая выводимая строка имела определенную длину. К счастью, в C# предусмотрены методы, которые позволяют легко выполнить эти операции.

Чтобы удалить из строки начальные и концевые пробелы, используйте один из следующих вариантов метода `Trim()`:

```
public string Trim()
public string Trim(params char[] chrs)
```

Первый формат метода предназначен для удаления начальных и концевых пробелов из вызывающей строки. Второй позволяет удалить начальные и концевые символы, заданные параметром *chrs*. В обоих случаях возвращается строка, содержащая результат этой операции.

В C# предусмотрена возможность дополнить строку заданными символами справа либо слева. Для реализации "левостороннего" дополнения строки используйте один из следующих методов:

```
public string PadLeft(int len)
public string PadLeft(int len, char ch)
```

Первый формат метода предназначен для дополнения строки с левой стороны пробелами в таком количестве, чтобы общая длина вызывающей строки стала равной заданному значению *len*. Второй формат отличается от первого тем, что для дополнения строки вместо пробела используется символ, заданный параметром *ch*. В обоих случаях возвращается строка, содержащая результат этой операции.

Для реализации "правостороннего" дополнения строки используйте один из следующих методов:

```
public string PadRight(int len)
public string PadRight(int len, char ch)
```

Первый формат метода дополняет строку с правой стороны пробелами в таком количестве, чтобы общая длина вызывающей строки стала равной заданному значению *len*. Второй формат отличается от первого тем, что для дополнения строки вместо пробела используется символ, заданный параметром *ch*. В обоих случаях возвращается строка, содержащая результат этой операции.

Выполнение операций удаления символов и дополнения ими строк демонстрируется в следующей программе:

```
// Демонстрация операций удаления символов
// и дополнения ими строк.

using System;

class TrimPadDemo {
  public static void Main() {
    string str = "тест";

    Console.WriteLine("Исходная строка: " + str);
```

```
    // Дополнение пробелами с левой стороны строки.
    str = str.PadLeft(10);
    Console.WriteLine("|" + str + "|");

    // Дополнение пробелами с правой стороны строки.
    str = str.PadRight(20);
    Console.WriteLine("|" + str + "|");

    // Удаление начальных и концевых пробелов.
    str = str.Trim();
    Console.WriteLine("|" + str + "|");

    // "Левостороннее" дополнение строки символами "#".
    str = str.PadLeft(10, '#');
    Console.WriteLine("|" + str + "|");

    // "Правостороннее" дополнение строки символами "#".
    str = str.PadRight(20, '#');
    Console.WriteLine("|" + str + "|");

    // Удаление начальных и концевых символов "#".
    str = str.Trim('#');
    Console.WriteLine("|" + str + "|");
  }
}
```

Результаты выполнения этой программы таковы:

```
Исходная строка: тест
|      тест|
|      тест          |
|тест|
|######тест|
|######тест##########|
|тест|
```

Вставка, удаление и замена

С помощью метода `Insert()` можно вставлять одну строку в другую:

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

Здесь параметром *str* задается строка, вставляемая в вызывающую. Позиция вставки (индекс) задается параметром *start*. Метод возвращает строку, содержащую результат этой операции.

С помощью метода `Remove()` можно удалить заданную часть строки:

```
public string Remove(int start, int count)
```

Количество удаляемых символов задается параметром *count*. Позиция (индекс), с которой начинается удаление, задается параметром *start*. Метод возвращает строку, содержащую результат этой операции.

С помощью метода `Replace()` можно заменить часть строки заданным символом либо строкой. Этот метод используется в двух форматах:

```
public string Replace(char ch1, char ch2)
public string Replace(string str1, string str2)
```

Первый формат метода позволяет заменить в вызывающей строке все вхождения символа *ch1* символом *ch2*. Второй служит для замены в вызывающей строке всех

вхождений строки *str1* строкой *str2*. В обоих случаях возвращается строка, содержащая результат этой операции.

Рассмотрим программу, в которой демонстрируется использование методов `Insert()`, `Remove()` и `Replace()`:

```
// Вставка, замена и удаление строк.

using System;

class InsRepRevDemo {
  public static void Main() {
    string str = "Это тест";

    Console.WriteLine("Исходная строка: " + str);

    // Вставляем строку.
    str = str.Insert(4, "простой ");
    Console.WriteLine(str);

    // Заменяем строку.
    str = str.Replace("простой", "сложный");
    Console.WriteLine(str);

    // Заменяем символы.
    str = str.Replace('т', 'X');
    Console.WriteLine(str);

    // Удаляем подстроку.
    str = str.Remove(4, 5);
    Console.WriteLine(str);
  }
}
```

Результаты выполнения этой программы таковы:

```
Исходная строка: Это тест
Это простой тест
Это сложный тест
ЭXo сложный XecX
ЭXo ый XecX
```

Изменение "регистра"

Класс `String` содержит два удобных метода, которые позволяют изменить "регистр", т.е. способ написания букв в строке. Эти методы называются `ToUpper()` и `ToLower()`:

```
public string ToLower()
public string ToUpper()
```

Метод `ToLower()` заменяет все буквы в вызывающей строке их строчными вариантами, а метод `ToUpper()` — прописными. Оба метода возвращают строку, содержащую результат операции. Существуют также версии этих методов, которые позволяют задавать форматы данных, присущие конкретному естественному языку, диалекту или территориальному образованию.

Использование метода Substring()

С помощью метода Substring() можно получить нужную часть строки. Возможны две формы использования этого метода:

```
public string Substring(int idx)
public string Substring(int idx, int count)
```

При использовании первой формы выделяемая подстрока начинается с элемента, заданного индексом *idx*, и продолжается до конца вызывающей строки. Вторая форма позволяет выделить подстроку, которая начинается с элемента, заданного индексом *idx*, и включает *count* символов. В обоих случаях возвращается выделенная подстрока.

Использование метода Substring() демонстрируется в следующей программе:

```
// Использование метода Substring().

using System;

class SubstringDemo {
  public static void Main() {
    string str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    Console.WriteLine("str: " + str);

    Console.Write("str.Substring(15): ");
    string substr = str.Substring(15);
    Console.WriteLine(substr);

    Console.Write("str.Substring(0, 15): ");
    substr = str.Substring(0, 15);
    Console.WriteLine(substr);
  }
}
```

Результаты выполнения этой программы таковы:

```
str: ABCDEFGHIJKLMNOPQRSTUVWXYZ
str.Substring(15): PQRSTUVWXYZ
str.Substring(0, 15): ABCDEFGHIJKLMNO
```

Форматирование

Если данные встроенного типа (например, int или double) требуется представить в форме, удобной для восприятия человеком, необходимо создать их строковое представление. Несмотря на то что C# автоматически поддерживает стандартный формат для такого представления, у программиста есть возможность задать собственный формат. Например, как было показано в части I, числовые данные можно вывести в формате с указанием суммы в долларах и центах. Для числовых типов предусмотрен ряд методов форматирования данных, в том числе и методы Console.WriteLine(), String.Format() и ToString(). Во всех этих трех методах используется один и тот же подход к форматированию, поэтому, научившись форматировать данные с помощью одного их них, вы сможете применить свои навыки и к остальным методам.

Общее представление о форматировании

Форматирование реализуется двумя компонентами: спецификаторами и поставщиками (провайдерами) формата. Форма, которую принимает строковое представление форматируемого значения, определяется применяемым *спецификатором формата*. Другими словами, внешний вид этой формы зависит именно от спецификатора формата. Например, чтобы вывести числовое значение с использованием экспоненциального представления чисел (в виде мантиссы и порядка), необходимо использовать спецификатор формата E.

Во многих случаях на точный формат значения может оказывать влияние территориальное образование или естественный язык, который используется в данной программе. Например, в США денежные суммы представляются в долларах, а в Европе — в евро. Для отображения языковых различий в C# используются поставщики формата. *Поставщик формата* определяет способ интерпретации спецификатора формата и создается посредством реализации интерфейса `IFormatProvider`. Поставщики формата заданы для встроенных числовых типов и многих других типов среды .NET Framework. Поскольку форматировать данные можно, не задавая поставщик формата, в этой книге они не рассматриваются.

Для форматирования данных необходимо включить в вызов соответствующего метода желаемый спецификатор формата. Об использовании спецификаторов формата уже шла речь в главе 3, но здесь имеет смысл уделить им больше внимания. Несмотря на то что ниже мы рассматриваем спецификаторы формата применительно к методу `WriteLine()`, аналогичный подход следует распространить и на другие методы, которые поддерживает форматирование данных.

Чтобы отформатировать данные с помощью метода `WriteLine()`, используйте следующую версию метода `WriteLine()`:

```
WriteLine("строка_форматирования",
          arg0, arg1, ... , argN);
```

В этой версии метода `WriteLine()` передаваемые ему аргументы разделяются запятыми, а не знаками "+". Элемент *строка_форматирования* содержит две составляющих: "постоянную" и "переменную". Постоянная составляющая представляет собой печатные символы, отображаемые "как есть", а переменная состоит из команд форматирования.

Общая форма записи команд форматирования имеет такой вид:

```
{номер_аргумента, ширина: формат}
```

Здесь элемент *номер_аргумента* определяет порядковый номер отображаемого аргумента (начиная с нулевого). С помощью элемента *ширина* указывается минимальная ширина поля, а спецификатор формата задается элементом *формат*. Элементы *ширина* и *формат* указывать необязательно. Таким образом, в простейшей форме команда форматирования лишь означает, какой аргумент нужно отобразить. Следовательно, команда `{0}` означает *arg0*, `{1}` означает *arg1* и т.д.

Если при выполнении метода `WriteLine()` в строке форматирования встречается команда форматирования, вместо нее подставляется (и отображается) аргумент, соответствующий заданному элементу *номер_аргумента*. Таким образом, элементы *номер_аргумента* указывают позицию спецификации в строке форматирования, которая определяет, где именно должны быть отображены соответствующие данные.

Если в строке форматирования присутствует элемент *fmt*, то соответствующие данные отображаются с использованием заданного формата. В противном случае используется стандартный формат. При наличии элемента *width* выводимые данные дополняются пробелами, которые гарантируют, что поле, занимаемое выводимым значением, будет иметь минимальную ширину. Если значение *width* положительно,

выводимые данные выравниваются по правому краю, а если отрицательно, — то по левому.

В следующих разделах принципы форматирования и спецификаторы формата рассматриваются более подробно.

Спецификаторы формата для числовых данных

Спецификаторы формата, определенные для числовых данных, описаны в табл. 20 3. Каждый спецификатор формата может включать необязательный спецификатор точности. Например, чтобы указать, что значение, представляемое в формате с плавающей точкой, должно иметь два десятичных разряда, используйте спецификатор F2.

Как упоминалось выше, точный результат применения конкретного спецификатора формата зависит от параметров локализации, связанных с конкретным естественным языком, диалектом или территориальным образованием. Например, спецификатор валюты, C, автоматически отображает значение в денежном формате, соответствующем выбранному параметру локализации. Для большинства пользователей стандартная информация локализации соответствует их местной специфике и языку общения. Таким образом, используя спецификаторы формата, можно не беспокоиться о контексте локализации, в котором будет выполняться та или иная программа.

Рассмотрим программу, которая демонстрирует использование ряда спецификаторов формата для представления числовых данных:

```
// Демонстрация использования различных
// спецификаторов формата.

using System;

class FormatDemo {
  public static void Main() {
    double v = 17688.65849;
    double v2 = 0.15;
    int x = 21;

    Console.WriteLine("{0:F2}", v);

    Console.WriteLine("{0:N5}", v);

    Console.WriteLine("{0:e}", v);

    Console.WriteLine("{0:r}", v);

    Console.WriteLine("{0:p}", v2);

    Console.WriteLine("{0:X}", x);

    Console.WriteLine("{0:D12}", x);

    Console.WriteLine("{0:C}", 189.99);
  }
}
```

Результаты выполнения этой программы таковы:

```
17688.66
17,688.65849
1.768866e+004
17688.65849
15.00 %
```

```
15
000000000021
$189.99
```

Обратите внимание на то, как в некоторых форматах спецификатор точности влияет на форматируемое значение.

Таблица 20.3. Спецификаторы формата

Спецификатор	*Формат*	*Значение спецификатора точности*
C	Денежный	Задает количество десятичных разрядов
c	Аналогично C	
D	Целочисленный (используется только с целыми числами)	Минимальное количество цифр. При необходимости результат дополняется начальными нулями
d	Аналогично D	
E	Экспоненциальное представление чисел (с использованием прописной буквы *E*)	Задает количество десятичных разрядов. По умолчанию используется шесть
e	Экспоненциальное представление чисел (с использованием строчной буквы *e*)	Задает количество десятичных разрядов. По умолчанию используется шесть
F	Представление чисел с фиксированной точкой	Задает количество десятичных разрядов
f	Аналогично F	
G	Используется более короткий из E- либо F-форматов	См. спецификаторы E и F
g	Используется более короткий из e- либо f-форматов	См. спецификаторы e и f
N	Представление чисел с фиксированной точкой (и запятой в качестве разделителя групп разрядов)	Задает количество десятичных разрядов
n	Аналогично N	
P	Процентный	Задает количество десятичных разрядов
p	Аналогично P	
R или r	Числовое значение, которое можно с помощью метода `Parse()` преобразовать в эквивалентную "внутреннюю" форму (Это так называемый формат "кругового преобразования".)	Не используется
X	Шестнадцатеричный (использует прописные буквы *A–F*)	Минимальное количество цифр. При необходимости результат дополняется начальными нулями
x	Шестнадцатеричный (использует строчные буквы *a–f*)	Минимальное количество цифр. При необходимости результат дополняется начальными нулями

Использование методов `String.Format()` и `ToString()` для форматирования данных

Несмотря на то что встраивание команд форматирования в метод `WriteLine()` — очень удобный способ вывода данных в сформатированном виде, иногда имеет смысл ограничиться созданием строки, которая будет содержать форматированные данные, но не отображать немедленно эту строку. Другими словами, в некоторых ситуациях

нужно сформатировать данные заранее, чтобы они были готовы к последующему выводу на любое заданное устройство. Эта возможность имеет особую ценность в такой GUI-среде, как Windows, в которой операции ввода-вывода редко выполняются в расчете на консоль.

Получить строковое представление значения в форматированном виде можно двумя способами: с помощью метода `String.Format()` или передачей спецификатора формата методу `ToString()`.

Использование метода `String.Format()` для форматирования значений

Форматированное значение можно получить в результате вызова одной из версий метода `Format()`, определенного в классе `String` (эти версии представлены в табл. 20.4). Работа метода `Format()` во многом подобна методу `WriteLine()`, за исключением того, что метод `Format()` возвращает форматированную строку, а не выводит ее на консоль.

Таблица 20.4. Методы `Format()`

Метод	*Описание*
`public static string Format(string str, object v)`	Форматирует объект *v* в соответствии с первой командой форматирования, которая содержится в строке *str*. Возвращает копию строки *str*, в которой команда форматирования заменена форматированными данными
`public static string Format(string str, object v1, object v2)`	Форматирует объект *v1* в соответствии с первой командой форматирования, содержащейся в строке *str*, а объект *v2* — в соответствии со второй. Возвращает копию строки *str*, в которой команды форматирования заменены форматированными данными
`public static string Format(string str, object v1, object v2, object v3)`	Форматирует объекты *v1*, *v2* и *v3* согласно соответствующим командам форматирования, содержащимся в строке *str*. Возвращает копию строки *str*, в которой команды форматирования заменены форматированными данными
`public static string Format(string str, params object[] v)`	Форматирует значения, переданные в параметре *v*, в соответствии с командами форматирования, содержащимися в строке *str*. Возвращает копию строки *str*, в которой команды форматирования заменены форматированными данными
`public static string Format( IFormatProvider fmtprvdr, string str, params object[] v)`	Форматирует значения, переданные в параметре *v*, в соответствии с командами форматирования, содержащимися в строке *str*, и с использованием провайдера формата, заданного параметром *fmtprvdr*. Возвращает копию строки *str*, в которой команды форматирования заменены форматированными данными

Ниже приведена рассмотренная ранее программа, демонстрирующая возможности форматирования, но переписанная с использованием метода `String.Format()`. Результаты выполнения новой версии совпадают с результатами предыдущей.

```
// Использование метода String.Format() для
// форматирования значений.

using System;

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

```
    double v = 17688.65849;
    double v2 = 0.15;
    int x = 21;

    string str = String.Format("{0:F2}", v);
    Console.WriteLine(str);

    str = String.Format("{0:N5}", v);
    Console.WriteLine(str);

    str = String.Format("{0:e}", v);
    Console.WriteLine(str);

    str = String.Format("{0:r}", v);
    Console.WriteLine(str);

    str = String.Format("{0:p}", v2);
    Console.WriteLine(str);

    str = String.Format("{0:X}", x);
    Console.WriteLine(str);

    str = String.Format("{0:D12}", x);
    Console.WriteLine(str);

    str = String.Format("{0:C}", 189.99);
    Console.WriteLine(str);
  }
}
```

Подобно методу `WriteLine()`, метод `String.Format()` позволяет встраивать в его вызов обычный текст вместе со спецификаторами формата и использовать сразу несколько спецификаторов формата и значений. Рассмотрим программу, которая отображает текущие значения суммы и произведения чисел от 1 до 10:

```
// Еще один пример использования метода Format().

using System;

class FormatDemo2 {
  public static void Main() {
    int i;
    int sum = 0;
    int prod = 1;
    string str;

    /* Отображаем текущие значения суммы и
       произведения чисел от 1 до 10. */
    for(i=1; i <= 10; i++) {
      sum += i;
      prod *= i;
      str = String.Format(
                  "Сумма:{0,3:D}  Произведение:{1,8:D}",
                  sum, prod);
      Console.WriteLine(str);
    }
  }
}
```

Результаты выполнения этой программы таковы:

```
Сумма:  1  Произведение:       1
Сумма:  3  Произведение:       2
Сумма:  6  Произведение:       6
Сумма: 10  Произведение:      24
Сумма: 15  Произведение:     120
Сумма: 21  Произведение:     720
Сумма: 28  Произведение:    5040
Сумма: 36  Произведение:   40320
Сумма: 45  Произведение:  362880
Сумма: 55  Произведение: 3628800
```

Обратите внимание на такую инструкцию программы:

```
str = String.Format("Сумма:{0,3:D} Произведение:{1,8:D}",
                    sum, prod);
```

Вызов метода `Format()` содержит два спецификатора формата: один для суммы (переменная `sum`), а другой для произведения (переменная `prod`). Следует отметить, что номера аргументов задаются здесь точно так же, как при использовании метода `WriteLine()`. Обратите также внимание на включение в строку форматирования обычного текста (`Сумма:` и `Произведение:`), который становится частью выводимой строки.

Использование метода `ToString()` для форматирования данных

Для получения форматированного нужным образом строкового представления, соответствующего значению встроенного числового типа (например, `Int32` или `Double`), можно использовать метод `ToString()`:

```
public string ToString(string fmt)
```

Метод `ToString()` возвращает строковое представление вызывающего объекта в соответствии с заданным спецификатором формата, переданным в параметре *fmt*. Например, следующая программа создает денежное представление значения 188.99, используя спецификатор формата `C`:

```
string str = 189.99.ToString("C");
```

Обратите внимание на то, что спецификатор формата непосредственно передается методу `ToString()`. В отличие от методов `WriteLine()` или `Format()`, которые используют встроенные команды форматирования (вместе с номером аргумента и значением ширины поля), метод `ToString()` принимает только спецификатор формата.

Вот как выглядит новая версия предыдущей программы форматирования, которая для форматирования строк использует метод `ToString()`. Результаты выполнения новой версии совпадают с результатами предыдущей.

```
// Использование метода ToString() для
// форматирования значений.

using System;

class ToStringDemo {
  public static void Main() {
    double v = 17688.65849;
    double v2 = 0.15;
    int x = 21;
```

```
    string str = v.ToString("F2");
    Console.WriteLine(str);

    str = v.ToString("N5");
    Console.WriteLine(str);

    str = v.ToString("e");
    Console.WriteLine(str);

    str = v.ToString("r");
    Console.WriteLine(str);

    str = v2.ToString("p");
    Console.WriteLine(str);

    str = x.ToString("X");
    Console.WriteLine(str);

    str = x.ToString("D12");
    Console.WriteLine(str);

    str = 189.99.ToString("C");
    Console.WriteLine(str);
  }
}
```

Создание пользовательского числового формата

Несмотря на то что встроенные спецификаторы формата весьма полезны, в C# программист может определить собственный формат, используя средство, называемое *форматом изображения* (picture format). Своим названием этот термин обязан тому факту, что программист создает формат на основе примера (т.е. изображения) того, как должен выглядеть выводимый результат. Этот способ форматирования упоминался в части I. Здесь же мы рассмотрим его более детально.

Использование символов-заполнителей

При создании пользовательского формата задается пример, или изображение того, как должны выглядеть данные. Для этого используются символы, приведенные в табл. 20.5.

Таблица 20.5. Символы-заполнители, используемые для создания пользовательского формата

Заполнитель	*Описание*
#	Цифра
.	Десятичная точка
,	Разделитель групп разрядов
%	Процент
0	Используется для дополнения начальными и концевыми нулями

Окончание табл. 20.5

Заполнитель	Описание
;	Отделяет разделы, которые описывают формат для положительных, отрицательных и нулевых значений
E0 E+0 E-0 e0 e+0 e-0	Экспоненциальное представление чисел

Символ "точка" указывает местоположение десятичной точки.

Символ "#" задает позицию цифры и может располагаться слева или справа от десятичной точки или без таковой. Если этот символ (один или несколько) находится справа от десятичной точки, он (они) означает количество отображаемых десятичных разрядов. При необходимости значение округляется. Если символ "#" находится слева от десятичной точки, он указывает позиции цифр, относящиеся к целой части числа. При необходимости добавляются начальные нули. Если целая часть числа содержит больше цифр, чем в формате представлено символов "#", такая целая часть числа отображается полностью и ни в коем случае не усекается. Если десятичная точка отсутствует, наличие символа "#" означает, что форматируемое значение будет округлено до целого числа. Нулевое значение (без указания дополнить его начальными или концевыми нулями) отображаться не будет. Это значит, что при использовании такого формата, как ##.##, ничего не будет отображено, если форматируемое значение равно нулю. Чтобы отобразить таки нулевое значение, используйте символ-заполнитель "0".

Благодаря символу-заполнителю "0" выводимое значение дополняется начальными или концевыми нулями, гарантирующими наличие минимального количества цифр, заданного строкой форматирования. Этот символ можно использовать как с правой, так и с левой стороны от десятичной точки. Например, при выполнении инструкции

```
Console.WriteLine("{0:00##.#00)", 21.3);
```

отображается такой результат:

```
0021.300
```

Если форматируемое значение содержит больше цифр, чем задано спецификатором форматирования, то его целая часть (расположенная слева от десятичной точки) будет отображена полностью, а дробная (справа от десятичной точки) округлена.

При форматировании больших чисел можно заказать вставку разделителей групп разрядов, задав шаблон отображаемого значения в виде последовательности символов "#" со вставленной в нее запятой. Например, при выполнении инструкции

```
Console.WriteLine("{0:#,###.#}", 3421.3);
```

отображается такой результат:

```
3,421.3.
```

Обратите здесь внимание на то, что нет необходимости вставлять символ "запятая" во все предполагаемые позиции. Задания в шаблоне лишь одной запятой уже вполне достаточно для вставки ее в значение после каждых трех цифр (начиная от десятичной точки) в левой части числа. Например, при выполнении инструкции

```
Console.WriteLine("{0:#,###.#}", 8763421.3);
```

отображается следующий результат:

```
8,763,421.3.
```

Символы "запятая" в строке форматирования имеют и еще одно значение. Если они стоят слева от десятичной точки (но рядом с ней), то действуют как масштабный

множитель. Каждая запятая обеспечивает деление форматируемого значения на 1 000. Например, при выполнении инструкции

```
Console.WriteLine(
        "Значение в тысячах: {0:#,###,.#}", 8763421.3);
```

отображается такой результат:

```
Значение в тысячах: 8,763.4
```

Как видно по результату, мы получили масштабированное значение, представленное в тысячах.

Помимо символов-заполнителей пользовательский спецификатор формата может содержать и другие символы. Любые символы, отличные от заполнителей, без изменений отображаются в отформатированной строке, причем в соответствующих шаблону позициях. Например, при выполнении инструкции

```
Console.WriteLine(
  "Топливная экономичность равна {0:##.# миль на галлон}",
  21.3);
```

отображается такой результат:

```
Топливная экономичность равна 21.3 миль на галлон
```

При необходимости здесь можно также использовать такие управляющие последовательности, как `\t` или `\n`.

Символы-заполнители `E` и `e` означают, что форматируемое значение должно быть отображено в экспоненциальном представлении (в виде мантиссы и порядка). После символа `E` и `e` должен стоять хотя бы один символ "`0`". Количество символов "`0`" означает число десятичных цифр, которые должны отображаться при выводе. Дробная часть округляется в соответствии с заданным форматом. Прописному написанию символа-заполнителя соответствует отображение прописной буквы `E`, а строчному — строчная (`e`). Для гарантированного отображения знака порядка используйте формат `E+` или `e+`, а для отображения знака только в случае отрицательного порядка используйте один из таких форматов: `E`, `e`, `E-` или `e-`.

Символ "`;`" представляет собой разделитель, который позволяет задать различные форматы для отображения положительных, отрицательных и нулевых значений. Общая форма записи спецификатора формата с использованием символа-заполнителя "`;`" такова:

плюс-формат;минус-формат;нуль-формат

Рассмотрим пример:

```
Console.WriteLine("{0:#.##;(#.##);0.00}", num);
```

Если значение `num` положительно, при выводе оно будет отображаться с двумя десятичными разрядами. Если `num` отрицательно, то при выводе оно будет отображено с двумя десятичными разрядами и заключено в круглые скобки. Если `num` равно нулю, отобразится строка `0.00`. При использовании символа-заполнителя "`;`" необязательно задавать все три части формата. Если важно указать, как должны выглядеть при выводе положительные и отрицательные значения, опустите элемент *нуль-формат*. Чтобы для отображения отрицательных значений использовать стандартный формат, опустите элемент *минус-формат*. В этом случае элементы *плюс-формат* и *нуль-формат* необходимо разделить двумя символами "точка с запятой".

В следующей программе демонстрируется использование лишь нескольких из многочисленных пользовательских форматов, которые может создать программист:

```
// Использование пользовательских форматов.

using System;
```

```
class PictureFormatDemo {
  public static void Main() {
    double num = 64354.2345;

    Console.WriteLine("Стандартный формат: " + num);

    // Отображаем значение с двумя десятичными разрядами.
    Console.WriteLine(
             "Значение с двумя десятичными разрядами: " +
             "{0:#.##}", num);

    // Отображаем значение с запятыми и
    // двумя десятичными разрядами.
    Console.WriteLine("Добавляем запятые: {0:#,###.##}",
                      num);

    // Отображаем значение в
    // экспоненциальном представлении.
    Console.WriteLine(
         "Используем экспоненциальное представление: " +
         "{0:#.###e+00}", num);

    // Отображаем значение в тысячах.
    Console.WriteLine("Значение в тысячах: " +
                      "{0:#0,}", num);

    /* Отображаем положительные, отрицательные и нулевые
       значения по-разному. */
    Console.WriteLine("Отображаем положительные, " +
                      "отрицательные и нулевые " +
                      "значения по-разному.");
    Console.WriteLine("{0:#.#;(#.##);0.00}", num);
    num = -num;
    Console.WriteLine("{0:#.##;(#.##);0.00}", num);
    num = 0.0;
    Console.WriteLine("{0:#.##;(#.##);0.00}", num);

    // Отображаем значение в процентах.
    num = 0.17;
    Console.WriteLine("Отображаем в процентах: {0:#%}",
                      num);
  }
}
```

Результаты выполнения этой программы таковы:

```
Стандартный формат: 64354.2345
Значение с двумя десятичными разрядами: 64354.23
Добавляем запятые: 64,354.23
Используем экспоненциальное представление: 6.435e+04
Значение в тысячах: 64
Отображаем положительные, отрицательные и нулевые значения по-
разному.
64354.2
(64354.23)
0.00
Отображаем в процентах: 17%
```

Форматирование даты и времени

Форматирование часто применяется также к значениям такого типа данных, как DateTime. Как разъяснялось в главе 19, тип DateTime представляет дату и время. Значения даты и времени можно отобразить различными способами. Например:

```
02/25/2002
Monday, February 25, 2002
12:59:00
12:59:00 PM
```

Неудивительно, что значения даты и времени имеют в разных странах различные представления. Поэтому в C# предусмотрена специальная подсистема форматирования значений этого типа.

Форматирование значений даты и времени осуществляется с помощью соответствующих спецификаторов формата, которые приведены в табл. 20.6. Поскольку значения даты и времени могут меняться при изменении страны и языка общения, их представление зависит от параметров локализации, установленных для конкретного компьютера.

Таблица 20.6. Спецификаторы формата для представления значений даты и времени

Спецификатор	*Формат*
D	Дата в длинной форме
d	Дата в краткой форме
T	Время в длинной форме
t	Время в краткой форме
F	Дата и время в длинной форме
f	Дата и время в краткой форме
G	Дата в краткой форме, а время — в длинной
g	Дата в краткой форме и время — в краткой
M	Месяц и день
m	Аналогично M
R	Дата и время в стандартной форме по Гринвичу
r	Аналогично R
s	Сортируемый формат представления даты и времени
U	Длинная форма универсального сортируемого формата представления даты и времени; время отображается как универсальное синхронизированное время (Universal Time Coordinated — UTC)
u	Краткая форма универсального сортируемого формата представления даты и времени
Y	Месяц и год
y	Аналогично Y

Рассмотрим программу, которая демонстрирует использование спецификаторов формата, предназначенных для отображения значений даты и времени:

```
// Форматирование значений даты и времени.

using System;

class TimeAndDateFormatDemo {
```

```
  public static void Main() {
    DateTime dt = DateTime.Now; // Получаем текущее время.

    Console.WriteLine("Формат d: {0:d}", dt);
    Console.WriteLine("Формат D: {0:D}", dt);

    Console.WriteLine("Формат t: {0:t}", dt);
    Console.WriteLine("Формат T: {0:T}", dt);

    Console.WriteLine("Формат f: {0:f}", dt);
    Console.WriteLine("Формат F: {0:F}", dt);

    Console.WriteLine("Формат g: {0:g}", dt);
    Console.WriteLine("Формат G: {0:G}", dt);

    Console.WriteLine("Формат m: {0:m}", dt);
    Console.WriteLine("Формат M: {0:M}", dt);

    Console.WriteLine("Формат r: {0:r}", dt);
    Console.WriteLine("Формат R: {0:R}", dt);

    Console.WriteLine("Формат s: {0:s}", dt);

    Console.WriteLine("Формат u: {0:u}", dt);
    Console.WriteLine("Формат U: {0:U}", dt);

    Console.WriteLine("Формат y: {0:y}", dt);
    Console.WriteLine("Формат Y: {0:Y}", dt);
  }
}
```

Результаты выполнения этой программы таковы:

```
Формат d: 10.09.2003
Формат D: 10 Сентябрь 2003 г.
Формат t: 14:53
Формат T: 14:53:11
Формат f: 10 Сентябрь 2003 г. 14:53
Формат F: 10 Сентябрь 2003 г. 14:53:11
Формат g: 10.09.2003 14:53
Формат G: 10.09.2003 14:53:11
Формат m: Сентябрь 10
Формат M: Сентябрь 10
Формат r: Wed, 10 Sep 2003 14:53:11 GMT
Формат R: Wed, 10 Sep 2003 14:53:11 GMT
Формат s: 2003-09-10T14:53:11
Формат u: 2003-09-10 14:53:11Z
Формат U: 10 Сентябрь 2003 г. 11:53:11
Формат y: Сентябрь 2003 г.
Формат Y: Сентябрь 2003 г.
```

Следующая программа создает очень простые часы. Время обновляется каждую секунду. Начало каждого часа компьютер отмечает звуковым сигналом. Для получения (и вывода на экран) форматированного времени используется метод `ToString()`, определенный в классе `DateTime`. По достижении ровного часа к концу отформатированного времени добавляется символ звукового сигнала (`\a`).

```
// Простые часы.

using System;
```

```
class SimpleClock {
  public static void Main() {
    string t;
    int seconds;

    DateTime dt = DateTime.Now;
    seconds = dt.Second;

    for(;;) {
      dt = DateTime.Now;

      // Обновляем время при изменении значения
      // переменной seconds.
      if(seconds != dt.Second) {
        seconds = dt.Second;

        t = dt.ToString("T");

        if(dt.Minute==0 && dt.Second==0)
          t = t + "\a"; // Обеспечиваем звуковой сигнал
                        // в начале каждого часа.

        Console.WriteLine(t);
      }
    }
  }
}
```

Создание пользовательского формата даты и времени

Несмотря на то что в большинстве случаев достаточно стандартных спецификаторов формата для отображения даты и времени, существует возможность создания собственных форматов. Этот процесс аналогичен созданию пользовательских форматов для числовых типов, которое было описано выше. По сути, программист просто создает пример (изображение) того, как должна выглядеть информация, содержащая дату и время. Для создания пользовательского формата отображения значений даты и времени используйте один или несколько символов-заполнителей, перечисленных в табл. 20.7.

Изучив эту таблицу, вы увидите, что символы `d`, `f`, `g`, `m`, `M`, `s` и `t` несут ту же функцию, что и спецификаторы формата для представления значений даты и времени, приведенные в табл. 20.6. Итак, если один из этих символов используется самостоятельно, он интерпретируется как спецификатор формата. В противном случае он рассматривается как заполнитель. Если же нужно использовать тот или иной символ самостоятельно, но без интерпретации его в качестве заполнителя, предварите его знаком процента (`%`).

Использование нескольких пользовательских форматов представления даты и времени демонстрируется в следующей программе:

```
// Демонстрация пользовательских форматов представления
// даты и времени.

using System;

class CustomTimeAndDateFormatsDemo {
  public static void Main() {
    DateTime dt = DateTime.Now;
```

```
    Console.WriteLine("Время: {0:hh:mm tt}", dt);
    Console.WriteLine(
         "Время в 24-часовом исчислении: {0:HH:mm}", dt);
    Console.WriteLine("Дата: {0:ddd MMM dd, yyyy}", dt);

    Console.WriteLine("Эра: {0:gg}", dt);

    Console.WriteLine("Время с секундами: " +
                      "{0:HH:mm:ss tt}", dt);

    Console.WriteLine("Используем m для дня месяца: {0:m}",
                      dt);
    Console.WriteLine("Используем m для минут: {0:%m}",
                      dt);
  }
}
```

Результаты выполнения программы таковы:

```
Время: 11:14
Время в 24-часовом исчислении: 11:14
Дата: Чт сен 11, 2003
Эра: A.D.
Время с секундами: 11:14:52
Используем m для дня месяца: Сентябрь 11
Используем m для минут: 14
```

Примечание: A.D. — сокр. от лат. *anno Domini*, т.е. нашей эры.

Таблица 20.7. Символы-заполнители, используемые при создании пользовательских форматов даты и времени

Заполнитель	*Значение*
d	День месяца как число, лежащее в диапазоне 1–31
dd	День месяца как число, лежащее в диапазоне 1–31. Значения из диапазона 1–9 дополняются начальным нулем
ddd	Сокращенное название дня недели
dddd	Полное название дня недели
f, ff, fff, ffff, fffff, ffffff, fffffff	Дробная часть значения секунд. Количество десятичных разрядов определяется числом заданных букв "f"
g	Эра
h	Часы в диапазоне 1–12
hh	Часы в диапазоне 1–12. Значения из диапазона 1–9 дополняются начальным нулем
H	Часы в диапазоне 0–23
HH	Часы в диапазоне 0–23. Значения из диапазона 1–9 дополняются начальным нулем
m	Минуты
mm	Минуты. Значения из диапазона 1–9 дополняются начальным нулем
M	Месяц в виде числа из диапазона 1–12
MM	Месяц в виде числа из диапазона 1–12. Значения из диапазона 1–9 дополняются начальным нулем
MMM	Сокращенное название месяца
MMMM	Полное название месяца
s	Секунды

Заполнитель	*Значение*
`ss`	Секунды. Значения из диапазона 1–9 дополняются начальным нулем
`t`	Символ "A" или "P", означающий A M. (до полудня) или P.M. (после полудня), соответственно
`tt`	A M. или P.M.
`y`	Год в виде двух цифр, если недостаточно одной
`yy`	Год в виде двух цифр. Значения из диапазона 1–9 дополняются начальным нулем
`yyyy`	Год в виде четырех цифр
`z`	Смещение часового пояса в часах
`zz`	Смещение часового пояса в часах. Значения из диапазона 1–9 дополняются начальным нулем
`zzz`	Смещение часового пояса в часах и минутах
`:`	Разделитель для компонентов значения времени
`/`	Разделитель для компонентов значения даты
`%`*`fmt`*	Стандартный формат, соответствующий спецификатору формата *`fmt`*

Форматирование перечислений

Язык C# позволяет форматировать значения, определенные в перечислении. В общем случае значения перечислений можно отображать с использованием их имен или чисел. Спецификаторы формата, предназначенные для перечислений, приведены в табл. 20.8. Обратите внимание на форматы `G` и `F`. Перечисления, используемые для представления битовых полей, могут предваряться атрибутом `Flags`. Обычно битовые поля содержат значения, которые представляют отдельные биты и располагаются по степеням двойки. При наличии атрибута `Flags` спецификатор `G` отображает имена всех битовых составляющих заданного значения. Спецификатор `F` отображает имена всех битовых составляющих значения, если это значение можно построить, применив операцию ИЛИ к двум или более полям, определенным перечислением.

Использование спецификаторов формата применительно к перечислениям демонстрируется в следующей программе:

```
// Форматирование перечисления.

using System;

class EnumFmtDemo {
  enum Direction { Север, Юг, Восток, Запад }
  [Flags] enum Status { Готов=0x1, Автономный_режим=0x2,
                        Ожидание=0x4, Данные_переданы=0x8,
                        Данные_получены=0x10,
                        Системный_режим=0x20 }

  public static void Main() {
    Direction d = Direction.Запад;

    Console.WriteLine("{0:G}", d);
    Console.WriteLine("{0:F}", d);
    Console.WriteLine("{0:D}", d);
    Console.WriteLine("{0:X}", d);
```

```
    Status s = Status.Готов | Status.Данные_переданы;

    Console.WriteLine("{0:G}", s);
    Console.WriteLine("{0:F}", s);
    Console.WriteLine("{0:D}", s);
    Console.WriteLine("{0:X}", s);
  }
}
```

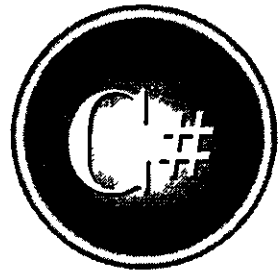
Результаты выполнения этой программы таковы:

```
Запад
Запад
3
00000003
Готов, Данные_переданы
Готов, Данные_переданы
9
00000009
```

Таблица 20.8. Спецификаторы формата для перечислений

Спецификатор	*Значение*
G	Отображает имя значения. Если форматируемое перечисление предваряется атрибутом `Flags`, спецификатор отображает имена всех битовых составляющих заданного значения (при условии, что оно допустимо)
g	Аналогично спецификатору `G`
F	Отображает имя значения. Если это значение можно создать, применив операцию ИЛИ к двум или более полям, определенным перечислением, спецификатор отображает имена всех битовых составляющих заданного значения, причем независимо от того, задан ли атрибут `Flags`
f	Аналогично спецификатору `F`
D	Отображает значение в виде десятичного целого числа
d	Аналогично спецификатору `D`
X	Отображает значение в виде шестнадцатеричного целого числа. Для гарантированного отображения восьми цифр значение при необходимости дополняется начальными нулями
x	Аналогично спецификатору `X`

Полный справочник по



Глава 21

Многопоточное программирование

Среди множества новых средств C# самым значительным, пожалуй, является встроенная поддержка *многопоточного программирования* (multithreaded programming). Многопоточная программа состоит из двух или больше частей, которые могут выполняться одновременно. Каждая часть такой программы называется *потоком* (thread), и каждый поток определяет собственный путь выполнения инструкций. Таким образом, многопоточность представляет собой специальную форму многозадачности.

Многопоточное программирование опирается на сочетание средств, предусмотренных языком C#, и классов, определенных в среде .NET Framework. Многие проблемы, связанные с многопоточностью, которые имеют место в других языках, в C# минимизированы или устранены совсем, поскольку поддержка многопоточности встроена в язык.

Основы многопоточности

Различают два вида многозадачности: с ориентацией на процессы и с ориентацией на потоки. Важно понимать различие между ними. *Процесс* по сути представляет собой выполняемую программу. Следовательно, многозадачность, ориентированная на процессы, — это средство, позволяющее компьютеру выполнять две или больше программ одновременно. Например, именно благодаря многозадачности, ориентированной на процессы, мы можем работать с текстовым редактором (или с электронными таблицами) и в то же самое время искать нужную информацию в Internet. В процессно-ориентированной многозадачности программа является наименьшим элементом кода, которым может манипулировать планировщик задач.

Поток — это управляемая единица выполняемого кода. В многозадачной среде, ориентированной на потоки, все процессы имеют по крайней мере один поток, но возможно и большее их количество. Это означает, что одна программа может выполнять сразу две и более задач. Например, текстовый редактор может форматировать текст и в то же время выводить что-либо на печать, поскольку эти два действия выполняются двумя отдельными потоками.

Итак, различие между процессно- и поточно-ориентированной многозадачностями можно определить следующим образом. Процессно-ориентированная многозадачность обеспечивает одновременное выполнение программ, а поточно-ориентированная — одновременное выполнение частей одной и той же программы.

Преимущество многопоточности состоит в том, что она позволяет писать очень эффективные программы, поскольку предоставляет возможность с толком использовать вынужденное время ожидания (простоя), которое имеет место во многих программах. Общеизвестно, что большинство устройств ввода-вывода (сетевые порты, дисководы или клавиатура) работают гораздо медленнее, чем центральный процессор (ЦП). Поэтому в программах львиная доля времени выполнения зачастую тратится на ожидание окончания отправки информации устройству (или получения от него). Используя многопоточность, можно построить программу так, чтобы она в такие периоды ожидания выполняла другую задачу. Например, пока одна часть программы будет отправлять файл по электронной почте, другая ее часть может считывать входные данные с клавиатуры, а еще одна — буферизировать следующий блок данных для отправки в Internet.

Поток может находиться в одном из нескольких возможных состояний. Он может *выполняться*. Он может быть готовым к выполнению (как только получит время ЦП). Выполняющийся поток может быть *приостановлен*, т.е. его выполнение временно прекращается. Позже оно может быть *возобновлено*. Поток может быть *заблокирован* в

ожидании необходимого ресурса. Наконец, поток может *завершиться*, и уж в этом случае его выполнение окончено и продолжению (возобновлению) не подлежит.

В среде .NET Framework определено два типа потоков: *высокоприоритетный* (foreground) и *низкоприоритетный*, или *фоновый* (background). По умолчанию поток создается высокоприоритетным, но его тип можно изменить, т.е. сделать его фоновым. Единственное различие между высоко- и низкоприоритетным потоками состоит в том, что последний будет автоматически завершен, если все высокоприоритетные потоки в его процессе остановились.

Поточно-ориентированная многозадачность не может обойтись без специального средства, именуемого *синхронизацией*, которое позволяет координировать выполнение потоков вполне определенными способами. В C# предусмотрена отдельная подсистема, посвященная синхронизации, ключевые средства которой здесь также рассматриваются.

Все процессы имеют по крайней мере один поток управления, который обычно называется *основным* (main thread), поскольку именно с этого потока начинается выполнение программы. Таким образом, все приведенные ранее в этой книге примеры программ использовали основной поток. Из основного можно создать и другие потоки.

Язык C# и среда .NET Framework поддерживают как процессно-, так и поточно-ориентированную многозадачность. Следовательно, используя C#, можно создавать как процессы, так и потоки, а затем ими эффективно управлять. При этом, чтобы создать новый процесс, потребуется написать небольшую программку, поскольку каждый процесс в значительной степени отделен от следующего. Здесь важно то, что C# обеспечивает поддержку многопоточности. Поскольку поддержка многопоточности является встроенной, C# значительно упрощает создание высокоэффективных многопоточных программ по сравнению с другими языками, например по сравнению с C++ (в который не встроена поддержка многопоточности).

Классы, которые поддерживают многопоточное программирование, определены в пространстве имен `System.Threading`. Поэтому в начало любой многопоточной программы необходимо включить следующую инструкцию:

```
using System.Threading;
```

Класс `Thread`

Многопоточная система C# встроена в класс `Thread`, который инкапсулирует поток управления. Класс `Thread` является `sealed`-классом, т.е. он не может иметь наследников. В классе `Thread` определен ряд методов и свойств для управления потоками. Наиболее употребляемые члены этого класса рассматриваются на протяжении этой главы.

Создание потока

Чтобы создать поток, необходимо создать объект типа `Thread`. В классе `Thread` определен следующий конструктор:

```
public Thread(ThreadStart entryPoint)
```

Здесь параметр *`entryPoint`* содержит имя метода, который будет вызван, чтобы начать выполнение потока. Тип `ThreadStart` — это делегат, определенный в среде .NET Framework:

```
public delegate void ThreadStart()
```

Итак, начальный метод должен иметь тип возвращаемого значения void и не принимать никаких аргументов.

Выполнение созданного потока не начнется до тех пор, пока не будет вызван метод Start(), который определяется в классе Thread. Его определение выглядит так:

```
public void Start()
```

Начавшись, выполнение потока будет продолжаться до тех пор, пока не завершится метод, заданный параметром *entryPoint*. Поэтому после выхода из *entryPoint*-метода выполнение потока автоматически завершается. Если попытаться вызвать метод Start() для потока, запущенного на выполнение, будет сгенерировано исключение типа ThreadStateException. Не забывайте, что класс Thread определен в пространстве имен System.Threading.

Рассмотрим пример создания нового потока и начала его выполнения:

```
// Создаем поток управления.

using System;
using System.Threading;

class MyThread {
  public int count;
  string thrdName;

  public MyThread(string name) {
    count = 0;
    thrdName = name;
  }

  // Начало (входная точка) потока.
  public void run() {
    Console.WriteLine(thrdName + " стартовал.");

    do {
      Thread.Sleep(500);
      Console.WriteLine("В потоке " + thrdName +
                        ", count = " + count);
      count++;
    } while(count < 10);

    Console.WriteLine(thrdName + " завершен.");
  }
}

class MultiThread {
  public static void Main() {
    Console.WriteLine("Основной поток стартовал.");

    // Сначала создаем объект класса MyThread.
    MyThread mt = new MyThread("Потомок #1");

    // Затем из этого объекта создаем поток.
    Thread newThrd = new Thread(new ThreadStart(mt.run));

    // Наконец, запускаем выполнение потока.
    newThrd.Start();

    do {
      Console.Write(".");
```

```
      Thread.Sleep(100);
    } while (mt.count != 10);

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Рассмотрим внимательно эту программу. Класс `MyThread` используется для создания второго потока управления. В его методе `run()` организован цикл, который "считает" от 0 до 9. Обратите внимание на вызов метода `Sleep()`, который является статическим и определен в классе `Thread`. Метод `Sleep()` заставляет поток, из которого он был вызван, приостановить выполнение на период времени, заданный в миллисекундах. В нашей программе проиллюстрирован следующий формат использования этого метода:

```
public static void Sleep(int milliseconds)
```

В параметре *`milliseconds`* задается время в миллисекундах, на которое будет приостановлено выполнение потока. Если параметр *`milliseconds`* равен нулю, вызывающий поток приостанавливается только для того, чтобы ожидающему потоку разрешить выполнение.

В методе `Main()` при выполнении следующих инструкций создается новый объект класса `Thread`:

```
// Сначала создаем объект класса MyThread.
    MyThread mt = new MyThread("Потомок #1");

    // Затем из этого объекта создаем поток.
    Thread newThrd = new Thread(new ThreadStart(mt.run));

    // Наконец, запускаем выполнение потока.
    newThrd.Start();
```

Как подсказывают комментарии, сначала создается объект класса `MyThread`. Этот объект затем используется для создания объекта класса `Thread` путем передачи значения `mt.run` в качестве имени стартового метода, или метода входной точки. Наконец, вызов метода `Start()` запускает новый поток на выполнение. Это приводит к вызову метода `run()` дочернего потока. После вызова метода `Start()` выполнение основного потока возвращается в метод `Main()`, а именно в цикл `do`. Теперь выполняются оба потока, разделяя время ЦП до тех пор, пока не закончатся их циклы. Программа генерирует такие результаты:

```
Основной поток стартовал.
.Потомок #1 стартовал.
....В потоке Потомок #1, count = 0
.....В потоке Потомок #1, count = 1
....В потоке Потомок #1, count = 2
.....В потоке Потомок #1, count = 3
....В потоке Потомок #1, count = 4
.....В потоке Потомок #1, count = 5
.....В потоке Потомок #1, count = 6
....В потоке Потомок #1, count = 7
.....В потоке Потомок #1, count = 8
....В потоке Потомок #1, count = 9
Потомок #1 завершен.
Основной поток завершен.
```

Часто в многопоточной программе нужно позаботиться о том, чтобы основной поток завершался последним. Формально программа продолжает выполняться до тех пор, пока не завершатся все высокоприоритетные потоки. Таким образом, совсем не

обязательно завершение основного потока последним. Однако добиваться этого — считается одним из признаков хорошего стиля программирования, поскольку в этом случае ясно определяется конечная точка программы. В предыдущем примере основной поток гарантированно завершается последним, поскольку цикл do останавливается, когда значение переменной count становится равным 10. Поскольку переменная count примет значение 10 только после того, как завершится выполнение потокового объекта newThrd, основной поток закончится последним. Ниже в этой главе будут показаны более удачные способы ожидания одним потоком завершения другого.

А если немного усовершенствовать

Несмотря на то что предыдущая программа вполне работоспособна, несколько простых усовершенствований сделают ее более эффективной. Во-первых, можно организовать начало выполнения потока сразу после его создания. Для потока класса MyThread это достижимо посредством создания объекта типа Thread внутри конструктора класса MyThread. Во-вторых, не обязательно хранить имя потока в классе MyThread, поскольку в классе Thread определено свойство Name, которое можно использовать с этой целью. Свойство Name определено таким образом:

```
public string Name { get; set; }
```

Так как свойство Name предназначено для чтения и записи, его можно использовать для запоминания имени потока или для его считывания.

Вот как выглядит усовершенствованная версия предыдущей программы:

```
// Альтернативный способ запуска потока.

using System;
using System.Threading;

class MyThread {
  public int count;
  public Thread thrd;

  public MyThread(string name) {
    count = 0;
    thrd = new Thread(new ThreadStart(this.run));
    thrd.Name = name; // Устанавливаем имя потока.
    thrd.Start(); // Запускаем поток на выполнение.
  }

  // Входная точка потока.
  void run() {
    Console.WriteLine(thrd.Name + " стартовал.");

    do {
      Thread.Sleep(500);
      Console.WriteLine("В потоке " + thrd.Name +
                        ", count = " + count);
      count++;
    } while(count < 10);

    Console.WriteLine(thrd.Name + " завершен.");
  }
}

class MultiThreadImproved {
```

```
  public static void Main() {
    Console.WriteLine("Основной поток стартовал.");

    // Сначала создаем объект класса MyThread.
    MyThread mt = new MyThread("Потомок #1");

    do {
      Console.Write(".");
      Thread.Sleep(100);
    } while (mt.count != 10);

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Эта версия программы генерирует те же результаты, что и предыдущая. Обратите внимание на то, что потоковый объект хранится теперь в переменной `thrd` внутри класса `MyThread`.

Создание нескольких потоков

В предыдущих примерах создавался только один дочерний поток. Однако программа способна порождать столько потоков, сколько потребуется в конкретной ситуации. Например, следующая программа создает три дочерних потока:

```
// Создание нескольких потоков управления.

using System;
using System.Threading;

class MyThread {
  public int count;
  public Thread thrd;

  public MyThread(string name) {
    count = 0;
    thrd = new Thread(new ThreadStart(this.run));
    thrd.Name = name;
    thrd.Start();
  }

  // Входная точка потока.
  void run() {
    Console.WriteLine(thrd.Name + " стартовал.");

    do {
      Thread.Sleep(500);
      Console.WriteLine("В потоке " + thrd.Name +
                        ", count = " + count);
      count++;
    } while(count < 10);

    Console.WriteLine(th d.Name + " завершен.");
  }
}

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

```
    Console.WriteLine("Основной поток стартовал.");

    // Создаем три потока.
    MyThread mt1 = new MyThread("Потомок #1");
    MyThread mt2 = new MyThread("Потомок #2");
    MyThread mt3 = new MyThread("Потомок #3");

    do {
      Console.Write(".");
      Thread.Sleep(100);
    } while (mt1.count < 10 ||
             mt2.count < 10 ||
             mt3.count < 10);

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Ниже показан возможный вариант результатов выполнения этой программы:

```
Основной поток стартовал.
.Потомок #1 стартовал.
Потомок #2 стартовал.
Потомок #3 стартовал.
....В потоке Потомок #1, count = 0
В потоке Потомок #2, count = 0
В потоке Потомок #3, count = 0
.....В потоке Потомок #1, count = 1
В потоке Потомок #2, count = 1
В потоке Потомок #3, count = 1
....В потоке Потомок #1, count = 2
В потоке Потомок #2, count = 2
В потоке Потомок #3, count = 2
.....В потоке Потомок #1, count = 3
В потоке Потомок #2, count = 3
В потоке Потомок #3, count = 3
....В потоке Потомок #1, count = 4
В потоке Потомок #2, count = 4
В потоке Потомок #3, count = 4
.....В потоке Потомок #1, count = 5
В потоке Потомок #2, count = 5
В потоке Потомок #3, count = 5
.....В потоке Потомок #1, count = 6
В потоке Потомок #2, count = 6
В потоке Потомок #3, count = 6
....В потоке Потомок #1, count = 7
В потоке Потомок #2, count = 7
В потоке Потомок #3, count = 7
.....В потоке Потомок #1, count = 8
В потоке Потомок #2, count = 8
В потоке Потомок #3, count = 8
....В потоке Потомок #1, count = 9
Потомок #1 завершен.
В потоке Потомок #2, count = 9
Потомок #2 завершен.
В потоке Потомок #3, count = 9
Потомок #3 завершен.
Основной поток завершен.
```

Как можно судить по приведенным результатам, сразу после старта все три потока разделяют время ЦП. Из-за различий в системных конфигурациях, операционных системах и других факторах среды результаты выполнения этой программы на вашем компьютере могут незначительно отличаться от представленных здесь.

Как определить, завершено ли выполнение потока

Иногда полезно знать, когда завершится выполнение потока. В предыдущих примерах это достигалось за счет проверки значения переменной `count`, но такое решение вряд ли можно считать удовлетворительным. К счастью, в классе `Thread` предусмотрено два средства, которые позволяют установить факт завершения выполнения потока. Одно из них — предназначенное только для чтения свойство `IsAlive`. Оно определяется так:

```
public bool IsAlive { get; }
```

Свойство `IsAlive` возвращает значение `true`, если поток, для которого оно опрашивается, еще выполняется. В противном случае оно возвращает значение `false`. Чтобы опробовать эту возможность, замените следующей версией класса `MoreThreads` ту, что представлена в предыдущей программе:

```
// Используем свойство IsAlive для установления факта
// завершения выполнения потока.
class MoreThreads {
  public static void Main() {
    Console.WriteLine("Основной поток стартовал.");

    // Создаем три потока.
    MyThread mt1 = new MyThread("Потомок #1");
    MyThread mt2 = new MyThread("Потомок #2");
    MyThread mt3 = new MyThread("Потомок #3");

    do {
      Console.Write(".");
      Thread.Sleep(100);
    } while (mt1.thrd.IsAlive &&
             mt2.thrd.IsAlive &&
             mt3.thrd.IsAlive);

    Console.WriteLine("Основной поток завершен.");
  }
}
```

С использованием этой версии класса `MoreThreads` результаты работы программы аналогичны предыдущим. Единственное различие между этими версиями состоит в том, что в последней для установления факта завершения выполнения дочерних потоков используется свойство `IsAlive`.

Второй способ, которые позволяет "дождаться" завершения выполнения потока, состоит в вызове метода `Join()`. Самый простой формат его использования имеет такой вид:

```
public void Join()
```

Метод `Join()` ожидает, пока поток, для которого он был вызван, не завершится. Имя этого метода ("join" в перев. с англ. — *присоединяться*) связано с идеей вызвать

указанный поток и подождать, когда тот присоединится к вызвавшему его методу. Если заданный поток даже не стартовал, будет сгенерировано исключение типа ThreadStateException. Другие форматы метода Join() позволяют указать максимальный объем времени, в течение которого вы собираетесь ожидать завершения выполнения потока.

Рассмотрим программу, которая для получения гарантии того, что основной поток завершится последним, использует метод Join():

```
// Использование метода Join().

using System;
using System.Threading;

class MyThread {
  public int count;
  public Thread thrd;

  public MyThread(string name) {
    count = 0;
    thrd = new Thread(new ThreadStart(this.run));
    thrd.Name = name;
    thrd.Start();
  }

  // Входная точка потока.
  void run() {
    Console.WriteLine(thrd.Name + " стартовал.");

    do {
      Thread.Sleep(500);
      Console.WriteLine("В потоке " + thrd.Name +
                        ", count = " + count);
      count++;
    } while(count < 10);

    Console.WriteLine(thrd.Name + " завершен.");
  }
}

// Используем метод Join() для ожидания завершения потоков.
class JoinThreads {
  public static void Main() {
    Console.WriteLine("Основной поток стартовал.");

    // Создаем три потока.
    MyThread mt1 = new MyThread("Потомок #1");
    MyThread mt2 = new MyThread("Потомок #2");
    MyThread mt3 = new MyThread("Потомок #3");

    mt1.thrd.Join();
    Console.WriteLine("Потомок #1 присоединен.");

    mt2.thrd.Join();
    Console.WriteLine("Потомок #2 присоединен.");

    mt3.thrd.Join();
    Console.WriteLine("Потомок #3 присоединен.");
```

```
    Console.WriteLine("Основной поток завершен.");
  }
}
```

Ниже приведены результаты выполнения этой программы. Не забывайте, что результаты, полученные вами, могут слегка отличаться от приведенных здесь.

```
Основной поток стартовал.
Потомок #1 стартовал.
Потомок #2 стартовал.
Потомок #3 стартовал.
В потоке Потомок #1, count = 0
В потоке Потомок #2, count = 0
В потоке Потомок #3, count = 0
В потоке Потомок #1, count = 1
В потоке Потомок #2, count = 1
В потоке Потомок #3, count = 1
В потоке Потомок #1, count = 2
В потоке Потомок #2, count = 2
В потоке Потомок #3, count = 2
В потоке Потомок #1, count = 3
В потоке Потомок #2, count = 3
В потоке Потомок #3, count = 3
В потоке Потомок #1, count = 4
В потоке Потомок #2, count = 4
В потоке Потомок #3, count = 4
В потоке Потомок #1, count = 5
В потоке Потомок #2, count = 5
В потоке Потомок #3, count = 5
В потоке Потомок #1, count = 6
В потоке Потомок #2, count = 6
В потоке Потомок #3, count = 6
В потоке Потомок #1, count = 7
В потоке Потомок #2, count = 7
В потоке Потомок #3, count = 7
В потоке Потомок #1, count = 8
В потоке Потомок #2, count = 8
В потоке Потомок #3, count = 8
В потоке Потомок #1, count = 9
Потомок #1 завершен.
В потоке Потомок #2, count = 9
Потомок #2 завершен.
В потоке Потомок #3, count = 9
Потомок #3 завершен.
Потомок #1 присоединен.
Потомок #2 присоединен.
Потомок #3 присоединен.
Основной поток завершен.
```

Нетрудно убедиться в том, что после выхода из методов `Join()` выполнение потоков завершено.

Свойство `IsBackground`

Как упоминалось выше, среда .NET Framework определяет два типа потоков: высокоприоритетные и фоновые. Единственное различие между ними состоит в том, что процесс не завершится до тех пор, пока не завершится выполнение всех его высоко-

приоритетных потоков, при этом фоновые потоки заканчиваются автоматически после завершения всех высокоприоритетных потоков. По умолчанию любой поток создается как высокоприоритетный. При необходимости его можно сделать фоновым с помощью свойства IsBackground, которое определено в классе Thread следующим образом:

```
public bool IsBackground { get; set; }
```

Чтобы перевести поток в категорию фоновых, достаточно присвоить свойству IsBackground значение true. Значение false означает, что соответствующий поток является высокоприоритетным.

Приоритеты потоков

Каждый поток имеет определенный приоритет. Приоритет потока, в частности, определяет, какой объем процессорного времени получает поток. В общем случае низкоприоритетным потокам выделяется немного времени ЦП, а высокоприоритетным — побольше. Нетрудно догадаться, что объем времени ЦП, выделяемый потоку, в значительной степени влияет на его выполнение и взаимодействие с другими потоками, выполняющимися в данный момент в системе.

Важно понимать, что существуют и другие факторы (помимо приоритета потока), которые влияют на объем времени ЦП, выделяемый потоку. Например, если высокоприоритетный поток находится в состоянии ожидания некоторого ресурса, например вводимых с клавиатуры данных, он будет заблокирован, и в это время сможет работать поток с более низким приоритетом. Поэтому в подобной ситуации низкоприоритетный поток может получить более продолжительный доступ к ЦП, чем высокоприоритетный.

После старта дочерний поток получает стандартное значение приоритета. Его можно изменить с помощью свойства Priority, которое является членом класса Thread. Общий формат его использования таков:

```
public ThreadPriority Priority{ get; set; }
```

Здесь ThreadPriority — перечисление, которое определяет следующие пять значений приоритета:

```
ThreadPriority.Highest
ThreadPriority.AboveNormal
ThreadPriority.Normal
ThreadPriority.BelowNormal
ThreadPriority.Lowest
```

По умолчанию потоку присваивается значение приоритета ThreadPriority .Normal.

Чтобы понять, как приоритеты влияют на выполнение потоков, воспользуемся примером, в котором выполняются два потока, причем с разными приоритетами. Эти потоки создаются как экземпляры класса MyThread. Метод run() содержит цикл, в котором подсчитывается количество итераций. Цикл останавливается либо на счете 1 000 000 000, либо в тот момент, когда статическая переменная stop примет значение true. До входа в этот цикл переменная stop установлена равной false. Первый поток, "досчитав" до 1 000 000 000, устанавливает переменную stop равной значению true. Это заставляет второй поток отказаться от следующего кванта времени (объем процессорного времени, выделяемого потоку). На каждом проходе цикла содержимое строковой переменной currentName сравнивается с именем выполняющегося потока. Если они не совпадают, значит, имело место переключение задач. При каждом пере-

ключении задач отображается имя нового потока, а переменной currentName присваивается имя текущего потока. Это позволяет наблюдать за тем, как часто каждый из потоков получает доступ к ЦП. По завершении выполнения обоих потоков отображается количество итераций, выполненных в каждом цикле.

```
// Демонстрация использования приоритетов потоков.

using System;
using System.Threading;

class MyThread {
  public int count;
  public Thread thrd;

  static bool stop = false;
  static string currentName;

  /* Создаем новый поток. Обратите внимание на то, что
     этот конструктор в действительности не запускает
     потоки на выполнение. */
  public MyThread(string name) {
    count = 0;
    thrd = new Thread(new ThreadStart(this.run));
    thrd.Name = name;
    currentName = name;
  }

  // Начинаем выполнение нового потока.
  void run() {
    Console.WriteLine("Поток " + thrd.Name + " стартовал.");
    do {
      count++;

      if(currentName != thrd.Name) {
        currentName = thrd.Name;
        Console.WriteLine("В потоке " + currentName);
      }

    } while(stop == false && count < 1000000000);
    stop = true;

    Console.WriteLine("Поток " + thrd.Name + " завершен.");
  }
}

class PriorityDemo {
  public static void Main() {
    MyThread mt1 = new MyThread("с высоким приоритетом");
    MyThread mt2 = new MyThread("с низким приоритетом");

    // Устанавливаем приоритеты.
    mt1.thrd.Priority = ThreadPriority.AboveNormal;
    mt2.thrd.Priority = ThreadPriority.BelowNormal;

    // Запускаем потоки на выполнение.
    mt1.thrd.Start();
    mt2.thrd.Start();
```

```
    mt1.thrd.Join();
    mt2.thrd.Join();

    Console.WriteLine();
    Console.WriteLine("Поток " + mt1.thrd.Name +
                      " досчитал до " +
                      mt1.count);
    Console.WriteLine("Поток " + mt2.thrd.Name +
                      " досчитал до " +
                      mt2.count);
  }
}
```

Вот как выглядят результаты выполнения этой программы на компьютере с 1-гигагерцевым процессором Pentium, на котором установлена ОС Windows 2000:

```
Поток с высоким приоритетом стартовал.
В потоке с высоким приоритетом
Поток с низким приоритетом стартовал.
В потоке с низким приоритетом
В потоке с высоким приоритетом
В потоке с низким приоритетом
В потоке с высоким приоритетом
В потоке с низким приоритетом
В потоке с высоким приоритетом
В потоке с низким приоритетом
В потоке с высоким приоритетом
В потоке с низким приоритетом
В потоке с высоким приоритетом
Поток с высоким приоритетом завершен.
Поток с низким приоритетом завершен.

Поток с высоким приоритетом досчитал до 1000000000
Поток с низким приоритетом досчитал до 25600064
```

Судя по полученным результатам, поток с высоким приоритетом получил приблизительно 98% процессорного времени. Конечно же, результаты, полученные на вашем компьютере, могут отличаться от приведенных, поскольку они зависят от быстродействия ЦП и количества их задач, выполняемых в системе. Результаты работы программы зависят также от версии Windows.

Поскольку многопоточный код в различных средах может вести себя по-разному, никогда не следует полагаться на характеристики выполнения программы, достигнутые в одной среде. Например, в предыдущем примере было бы ошибкой предполагать, что низкоприоритетный поток будет всегда выполняться в течение хотя бы небольшого отрезка времени перед тем, как завершится поток с высоким приоритетом. Например, в другой среде высокоприоритетный поток может завершиться еще до того, как поток с низким приоритетом хотя бы раз использует свой квант времени.

Синхронизация

При использовании в программе нескольких потоков иногда необходимо координировать их выполнение. Процесс координации потоков называется *синхронизацией*. К синхронизации прибегают в тех случаях, когда двум или большему числу потоков необходимо получить доступ к общему ресурсу, который в каждый момент времени может использовать только один поток. Например, когда один поток выводит данные в файл, в это самое время второй поток должен быть лишен возможности выполнять

аналогичные действия. Синхронизация необходима и в других ситуациях. Например, один поток ожидает, пока не произойдет событие, "судьба" которого зависит от другого потока. В этом случае необходимо иметь средство, которое бы удерживало первый поток в состоянии приостановки до тех пор, пока не произойдет ожидаемое им событие. После этого "спящий" поток должен возобновить выполнение.

В основе синхронизации лежит понятие *блокировки*, т.е. управление доступом к некоторому блоку кода в объекте. На то время, когда объект заблокирован одним потоком, никакой другой поток не может получить доступ к заблокированному блоку кода. Когда поток снимет блокировку, объект станет доступным для использования другим потоком.

Средство блокировки встроено в язык C#, поэтому доступ ко всем объектам может быть синхронизирован. Синхронизация поддерживается ключевым словом `lock`. Поскольку синхронизация заложена в C# с самого начала разработки языка, применять ее довольно легко.

Формат использования инструкции `lock` таков:

```
lock(object) {
  // Инструкции, подлежащие синхронизации.
}
```

Здесь параметр *object* представляет собой ссылку на синхронизируемый объект. Если нужно синхронизировать только один элемент, фигурные скобки можно опустить. Инструкция `lock` гарантирует, что указанный блок кода, защищенный блокировкой для данного объекта, может быть использован только потоком, который получает эту блокировку. Все другие потоки остаются заблокированными до тех пор, пока блокировка не будет снята. А снята она будет лишь при выходе из этого блока.

В следующей программе демонстрируется синхронизация посредством управления доступом к методу `sumIt()`, который суммирует элементы массива целочисленного типа:

```
// Использование инструкции lock для синхронизации
// доступа к объекту.

using System;
using System.Threading;

class SumArray {
  int sum;

  public int sumIt(int[] nums) {
    lock(this) { // Блокировка всего метода.
      sum = 0;   // Начальное значение суммы.

      for(int i=0; i < nums.Length; i++) {
        sum += nums[i];
        Console.WriteLine(
               "Промежуточная сумма для потока " +
               Thread.CurrentThread.Name +
               " равна " + sum);
        Thread.Sleep(10); // Разрешено переключение задач.
      }
      return sum;
    }
  }
}

class MyThread {
```

```
  public Thread thrd;
  int[] a;
  int answer;

  /* Создаем один объект класса SumArray для всех
     экземпляров класса MyThread. */
  static SumArray sa = new SumArray();

  // Создаем новый поток.
  public MyThread(string name, int[] nums) {
    a = nums;
    thrd = new Thread(new ThreadStart(this.run));
    thrd.Name = name;
    thrd.Start(); // Запускаем поток на выполнение.
  }

  // Начало выполнения нового потока.
  void run() {
    Console.WriteLine(thrd.Name + " стартовал.");

    answer = sa.sumIt(a);

    Console.WriteLine("Сумма для потока " + thrd.Name +
                      " равна " + answer);

    Console.WriteLine(thrd.Name + " завершен.");
  }
}

class Sync {
  public static void Main() {
    int[] a = {1, 2, 3, 4, 5};

    MyThread mt1 = new MyThread("Потомок #1", a);
    MyThread mt2 = new MyThread("Потомок #2", a);

    mt1.thrd.Join();
    mt2.thrd.Join();
  }
}
```

Результаты выполнения программы таковы:

```
Потомок #1 стартовал.
Потомок #2 стартовал.
Промежуточная сумма для потока Потомок #1 равна 1
Промежуточная сумма для потока Потомок #1 равна 3
Промежуточная сумма для потока Потомок #1 равна 6
Промежуточная сумма для потока Потомок #1 равна 10
Промежуточная сумма для потока Потомок #1 равна 15
Промежуточная сумма для потока Потомок #2 равна 1
Сумма для потока Потомок #1 равна 15
Потомок #1 завершен.
Промежуточная сумма для потока Потомок #2 равна 3
Промежуточная сумма для потока Потомок #2 равна 6
Промежуточная сумма для потока Потомок #2 равна 10
Промежуточная сумма для потока Потомок #2 равна 15
Сумма для потока Потомок #2 равна 15
Потомок #2 завершен.
```

Как видно по результатам, оба потока правильно подсчитали сумму чисел.

Эту программу стоит рассмотреть подробнее. В ней создается три класса. Первому присвоено имя SumArray. В нем определен метод sumIt(), который суммирует элементы целочисленного массива. Во втором классе, MyThread, используется static-объект sa типа SumArray. Таким образом, все объекты типа MyThread совместно используют только один объект типа SumArray. Этот объект служит для получения суммы целочисленных элементов массива. Обратите внимание на то, что промежуточная сумма хранится в поле sum класса SumArray. Следовательно, если два потока будут использовать метод sumIt() одновременно, каждый из них попытается хранить в поле sum "свою" промежуточную сумму. Поскольку это приведет к ошибочным результатам, доступ к методу sumIt() необходимо синхронизировать. Наконец, упомянутые два потока создаются в классе Sync, который дает им "путевку в жизнь", т.е. направляет на вычисление суммы элементов целочисленного массива.

В методе sumIt() инструкция lock предотвращает одновременное его использование различными потоками. Обратите внимание на то, что в этой инструкции блокировки в качестве синхронизируемого объекта служит ссылка this. Обычно именно так выполнятся инструкция lock, если блокируется вызывающий объект. Метод Sleep() вызывается специально для того, чтобы разрешить переключение задач (если оно возможно), но в данном случае это невозможно. Поскольку код метода sumIt() заблокирован, он доступен одновременно только для одного потока. Следовательно, после того, как начнет выполняться второй поток, он не войдет в метод sumIt() до тех пор, пока его (метод) полностью не отработает первый дочерний поток. Тем самым гарантируется получение корректного результата.

Чтобы до конца понять действие инструкции lock, попробуйте удалить ее из тела метода sumIt(). В этом случае метод sumIt() окажется без синхронизации, и любые потоки смогут использовать его одновременно для одного и того же объекта. Дело в том, что в поле sum хранится промежуточная сумма, и содержимое этого поля будет изменяться каждым потоком, который вызывает метод sumIt(). Таким образом, если два потока одновременно вызовут метод sumIt() для одного и того же объекта, будут получены некорректные результаты, поскольку без синхронизации переменная sum будет отражать смешанный результат суммирования обоих потоков. Вот, например, каким может быть результат выполнения этой программы после удаления инструкции lock из метода sumIt():

```
Потомок #1 стартовал.
Промежуточная сумма для потока Потомок #1 равна 1
Потомок #2 стартовал.
Промежуточная сумма для потока Потомок #2 равна 1
Промежуточная сумма для потока Потомок #1 равна 3
Промежуточная сумма для потока Потомок #2 равна 5
Промежуточная сумма для потока Потомок #1 равна 8
Промежуточная сумма для потока Потомок #2 равна 11
Промежуточная сумма для потока Потомок #1 равна 15
Промежуточная сумма для потока Потомок #2 равна 19
Промежуточная сумма для потока Потомок #1 равна 24
Промежуточная сумма для потока Потомок #2 равна 29
Сумма для потока Потомок #2 равна 29
Потомок #2 завершен.
Сумма для потока Потомок #1 равна 29
Потомок #1 завершен.
```

Судя по результатам, оба дочерних потока используют метод sumIt() одновременно для одного и того же объекта, поэтому значение поля sum неверно.

Итак, подытожим действие инструкции lock.

1. Если для заданного объекта инструкция `lock` размещена в некотором блоке кода, этот объект блокируется, и никакой другой поток не сможет запросить блокировку.
2. Другие потоки, пытающиеся запросить блокировку для того же объекта, перейдут в состояние ожидания и будут находиться в нем до тех пор, пока код не будет разблокирован.
3. Объект разблокируется, когда поток выходит из заблокированного блока.

Необходимо также отметить, что инструкция `lock` должна использоваться только для объектов, которые определены либо как `private`-, либо как `internal`-объекты. В противном случае внешние по отношению к вашей программе потоки смогут получать блокировку и не снимать ее.

Альтернативное решение

Несмотря на то что блокирование кода метода, как показано в предыдущей программе, не представляет большой сложности и является эффективным средством синхронизации, оно, к сожалению, работает не во всех случаях. Например, нужно получить синхронизированный доступ к методу класса, который создавали не вы лично, а в нем самом средств синхронизации изначально не предусмотрено. Такая ситуация возможна в том случае, если вы хотите использовать класс, который написан сторонней организацией, и невозможно получить доступ к его исходному коду. В этом случае, очевидно, вам не удастся добавить инструкцию `lock` в соответствующий метод класса. Как тогда синхронизировать доступ к объекту такого класса? К счастью, есть очень простое решение описанной проблемы: заблокировать доступ к объекту из внешнего (по отношению к объекту) кода, указав этот объект в инструкции `lock`. Например, рассмотрим альтернативную реализацию предыдущей программы. Обратите внимание на то, что код в методе `sumIt()` больше не блокируется. Теперь блокируются обращения к методу `sumIt()` внутри класса `MyThread`.

```
// Еще один способ использовать инструкцию lock
// для синхронизации доступа к объекту.

using System;
using System.Threading;

class SumArray {
  int sum;

  public int sumIt(int[] nums) {
    sum = 0; // Установка начального значения суммы.

    for(int i=0; i < nums.Length; i++) {
      sum += nums[i];
      Console.WriteLine(
             "Промежуточнаследования сумма " +
             Thread.CurrentThread.Name +
             " равна " + sum);
      Thread.Sleep(10); // Разрешение переключения задач.
    }
    return sum;
  }
}

class MyThread {
```

```
  public Thread thrd;
  int[] a;
  int answer;

  /* Создаем один объект класса SumArray для всех
     экземпляров класса MyThread. */
  static SumArray sa = new SumArray();

  // Создаем новый поток.
  public MyThread(string name, int[] nums) {
    a = nums;
    thrd = new Thread(new ThreadStart(this.run));
    thrd.Name = name;
    thrd.Start(); // Запускаем поток на выполнение.
  }

  // Начало выполнения нового потока.
  void run() {
    Console.WriteLine(thrd.Name + " стартовал.");

    // Инструкция lock содержит вызов метода sumIt().
    lock(sa) answer = sa.sumIt(a);

    Console.WriteLine("Сумма для потока " + thrd.Name +
                      " равна " + answer);

    Console.WriteLine(thrd.Name + " завершен.");
  }
}

class Sync {
  public static void Main() {
    int[] a = {1, 2, 3, 4, 5};

    MyThread mt1 = new MyThread("Потомок #1", a);
    MyThread mt2 = new MyThread("Потомок #2", a);

    mt1.thrd.Join();
    mt2.thrd.Join();
  }
}
```

Здесь блокируется обращение к методу `sa.sumIt()`, а не код в самом методе `sumIt()`. Это реализуется таким способом:

```
// Инструкция lock содержит вызов метода sumIt().
lock(sa) answer = sa.sumIt(a);
```

При выполнении этой программы получаем такие же корректные результаты, как и при использовании исходного решения.

Блокирование статического метода

Поскольку блокировка работает по отношению к объекту, то на первый взгляд может показаться, что невозможно заблокировать код `static`-метода, поскольку не существует объекта, для которого требуется выполнить блокировку. В действительности все обстоит иначе. Чтобы заблокировать `static`-метод, достаточно использовать инструкцию `lock` в следующем формате:

```
lock(typeof(class)) {
  // Блокируемый код.
}
```

Здесь *class* представляет собой имя класса, в котором содержится static-метод, подлежащий блокировке.

Класс Monitor и инструкция lock

Ключевое слово lock — это не что иное, как сокращенный вариант использования средств синхронизации, определенных в классе Monitor, принадлежащем пространству имен System.Threading. В классе Monitor определено несколько методов синхронизации. Например, чтобы получить возможность блокировки для некоторого объекта, вызовите метод Enter(), а чтобы снять блокировку — метод Exit(). Эти методы имеют следующий формат:

```
public static void Enter(object syncOb)
public static void Exit(object syncOb)
```

Здесь *syncOb* — синхронизируемый объект. Если при вызове метода Enter() заданный объект недоступен, вызывающий поток будет ожидать до тех пор, пока объект не станет доступным. Разработчики из компании Microsoft утверждают, что lock-блок "совершенно эквивалентен" вызову метода Enter() с последующим вызовом метода Exit(). Но поскольку lock — это встроенная инструкция языка C#, то для получения блокировки в C#-программировании предпочтительнее использовать именно ее.

Обратите внимание на метод TryEnter() из класса Monitor. Один из форматов его использования таков:

```
public static bool TryEnter(object syncOb)
```

Метод возвращает значение true, если вызывающий поток получает блокировку для объекта *syncOb*, и значение false в противном случае. Если заданный объект недоступен, вызывающий поток будет ожидать до тех пор, пока он не станет доступным.

В классе Monitor определены еще три метода: Wait(), Pulse() и PulseAll(). Они описаны в следующем разделе.

Взаимодействие потоков с помощью методов Wait(), Pulse() и PulseAll()

Рассмотрим следующую ситуацию. Поток (назовем его *T*) выполняет содержимое lock-блока и требует доступ к ресурсу (назовем его *R*), который временно недоступен. Что делать потоку *T*? Если поток *T* войдет в цикл опроса в ожидании доступности ресурса *R*, он свяжет объект, блокируя доступ к нему другим потокам. Это решение трудно назвать оптимальным, поскольку оно аннулирует преимущества программирования в многопоточной среде. Будет лучше, если поток *T* временно откажется от "претензий" на объект, позволив другому потоку выполнить свою работу. Когда же ресурс *R* станет доступным, поток *T* можно уведомить об этом, и он возобновит выполнение. Такой подход опирается на межпоточные средства общения, которые позволяют одному потоку уведомить другой о том, что он блокируется, а затем первого поставить в известность о том, что он может вобновить выполнение. C# поддерживает межпоточное взаимодействие с помощью методов Wait(), Pulse() и PulseAll().

Методы Wait(), Pulse() и PulseAll() определены в классе Monitor. Эти методы можно вызывать только внутри lock-блока кода. Когда выполнение потока вре-

менно блокируется, вызывается метод Wait(), т.е. он переходит в режим ожидания ("засыпает") и снимает блокировку с объекта, позволяя другому потоку использовать этот объект. Позже, когда другой поток входит в аналогичное состояние блокирования и вызывает метод Pulse() или PulseAll(), "спящий" поток "просыпается". Обращение к методу Pulse() возобновляет выполнение потока, стоящего первым в очереди потоков, пребывающих в режиме ожидания. Обращение к методу PulseAll() сообщает о снятии блокировки всем ожидающим потокам.

Вот два наиболее употребимых формата метода Wait():

```
public static bool Wait(object waitOb)
public static bool Wait(object waitOb, int milliseconds)
```

Первый формат означает ожидание до уведомления. Второй — подразумевает ожидание до уведомления или до истечения периода времени, заданного в миллисекундах. В обоих случаях параметр *waitOb* задает объект, к которому ожидается доступ.

Форматы использования методов Pulse() и PulseAll() таковы:

```
public static void Pulse(object waitOb)
public static void PulseAll(object waitOb)
```

Здесь параметр *waitOb* означает объект, освобождаемый от блокировки.

Если метод Wait(), Pulse() или PulseAll() вызывается из кода, который находится вне lock-блока, генерируется исключение типа SynchronizationLockException.

Пример использования методов Wait() и Pulse()

Чтобы понять необходимость применения методов Wait() и Pulse(), создадим программу, которая имитирует тиканье часов посредством отображения на экране слов "тик" и "так". Для этого создадим класс TickTock, который содержит два метода: tick() и tock(). Метод tick() отображает слово "тик", а метод tock() — слово "так". Для работы нашего "часового механизма" создаем два потока, причем один из них будет вызывать метод tick(), а другой — метод tock(). Наша задача — организовать выполнение этих потоков таким образом, чтобы программа последовательно отображала "тик-так", т.е. сначала слово "тик", а за ним — слово "так".

```
// Использование методов Wait() и Pulse() для создания
// тикающих часов.

using System;
using System.Threading;

class TickTock {

  public void tick(bool running) {
    lock(this) {
      if(!running) { // Останов часов.
        Monitor.Pulse(this); // Уведомление любых
                             // ожидающих потоков.
        return;
      }

      Console.Write("тик");
      Monitor.Pulse(this); // Разрешает выполнение
                           // метода tock().

      Monitor.Wait(this); // Ожидаем завершения
                          // метода tock().
    }
```

```
  }

  public void tock(bool running) {
    lock(this) {
      if(!running) { // Останов часов.
        Monitor.Pulse(this); // Уведомление любых
                             // ожидающих потоков.
        return;
      }

      Console.WriteLine("так");
      Monitor.Pulse(this); // Разрешает выполнение
                           // метода tick().

      Monitor.Wait(this); // Ожидаем завершения
                          // метода tick().
    }
  }
}

class MyThread {
  public Thread thrd;
  TickTock ttOb;

  // Создаем новый поток.
  public MyThread(string name, TickTock tt) {
    thrd = new Thread(new ThreadStart(this.run));
    ttOb = tt;
    thrd.Name = name;
    thrd.Start();
  }

  // Начинаем выполнение нового потока.
  void run() {
    if(thrd.Name == "тик") {
      for(int i=0; i<5; i++) ttOb.tick(true);
      ttOb.tick(false);
    }
    else {
      for(int i=0; i<5; i++) ttOb.tock(true);
      ttOb.tock(false);
    }
  }
}

class TickingClock {
  public static void Main() {
    TickTock tt = new TickTock();
    MyThread mt1 = new MyThread("тик", tt);
    MyThread mt2 = new MyThread("так", tt);

    mt1.thrd.Join();
    mt2.thrd.Join();
    Console.WriteLine("Часы остановлены");
  }
}
```

При выполнении эта программа сгенерировала следующие результаты:

```
тик так
тик так
тик так
тик так
тик так
Часы остановлены
```

Рассмотрим эту программу в деталях. В методе `Main()` создается объект класса `TickTock` с именем `tt`, который затем используется для запуска двух потоков. В методе `run()` класса `MyThread` выполняется сравнение имени текущего потока со словом "тик". Если сравниваемые значения совпали, вызывается метод `tick()`, в противном случае — метод `tock()`. При этом для каждого из пяти вызовов каждого метода в качестве аргумента передается значение `true`. Часы "тикают" до тех пор, пока передается именно значение `true`. Последний вызов каждого метода (с передачей в качестве аргумента значения `false`) останавливает часы.

Самая важная часть программы сосредоточена в методах `tick()` и `tock()`. Рассмотрим сначала метод `tick()`, который для удобства приведем ниже:

```
public void tick(bool running) {
  lock(this) {
    if(!running) { // Останов часов.
      Monitor.Pulse(this); // Уведомление любых
                           // ожидающих потоков.
      return;
    }

    Console.Write("тик ");
    Monitor.Pulse(this); // Разрешает выполнение
                         // метода tock().

    Monitor.Wait(this); // Ожидаем завершения
                        // метода tock().
  }
}
```

Прежде всего обратите внимание на то, что код метода `tick()` заключен в рамки `lock`-блока. Вспомните, методы `Wait()` и `Pulse()` можно использовать только внутри синхронизируемых блоков. Метод начинается с проверки значения параметра `running`. Этот параметр используется для отключения часов. Если он равен значению `false`, часы будут остановлены. В этом случае вызывается метод `Pulse()`, который позволяет заработать ожидающему потоку. Сюда мы скоро вернемся, а пока предположим, что часы никто (пока) не останавливает, и на экране отображается слово "тик", а затем происходит обращение к методу `Pulse()` с последующим вызовом метода `Wait()`. Обращение к методу `Pulse()` позволяет перейти в ожидание возможности доступа к тому же самому объекту. Вызов метода `Wait()` вынуждает метод `tick()` приостановиться до тех пор, пока другой поток не вызовет метод `Pulse()`. Таким образом, после вызова метода `tick()` отображается одно слово "тик", выдается разрешение на "оживление" другого потока, после чего первый приостанавливается.

Метод `tock()` — точная копия метода `tick()`, за исключением того, что он отображает слово "так". Итак, после его вызова отображается одно слово "так", вызывается метод `Pulse()`, а затем — и метод `Wait()`. Если рассматривать эти методы в паре, то за вызовом метода `tick()` может следовать только вызов метода `tock()`, за которым, в свою очередь, может следовать только вызов метода `tick()` и т.д.. Таким образом, эти два метода взаимно синхронизированы.

Для вызова метода `Pulse()` при останове часов есть все основания, поскольку он позволяет успешно завершить последнее обращение к методу `Wait()`. Не забывайте,

что методы tick() и tock() после отображения соответствующего сообщения вызывают метод Wait(). Следовательно, в момент останова часов один из этих методов находится в состоянии ожидания. А чтобы вывести его из этого состояния, необходимо вызвать метод Pulse() в последний раз. Ради эксперимента попробуйте удалить этот вызов метода Pulse() и посмотрите, как это скажется на результатах выполнения программы. Нетрудно убедиться, что программа в этом случае "зависнет" и для выхода из нее вам придется нажать клавиши <CTRL + C>. Дело в том, что для завершающего вызова метода Wait() из последнего выполнения метода tock() нет соответствующего обращения к методу Pulse(), который бы позволил бы методу tock() благополучно завершиться. Поэтому он (метод tock()) находится в бесконечном ожидании.

Если у вас есть какие-либо сомнения насчет необходимости вызовов методов Wait() и Pulse() для правильной работы наших "часов", то прежде чем переходить к следующему разделу, замените класс TickTock в предыдущей программе приведенной ниже версией. Здесь из класса удалены обращения к методам Wait() и Pulse().

```
// Нерабочая версия класса TickTock.
class TickTock {

  public void tick(bool running) {
    lock(this) {
      if(!running) { // Останов часов.
        return;
      }

      Console.Write("тик ");
    }
  }

  public void tock(bool running) {
    lock(this) {
      if(!running) { // Останов часов.
        return;
      }

      Console.WriteLine("так");
    }
  }
}
```

После замены класса TickTock результаты выполнения программы "часов" выглядят так:

```
тик тик тик тик тик так
так
так
так
так
Часы остановлены
```

В этом случае очевидно, что методы tick() и tock() больше не синхронизированы!

Взаимоблокировка

При разработке многопоточных программ необходимо позаботиться о том, чтобы во время их выполнения не создалась тупиковая ситуация, вызванная взаимоблокировкой. При *взаимоблокировке* (deadlock) один поток ожидает, пока другой не выпол-

нит некоторое действие, но в то же время второй поток ожидает действия первого. Таким образом, оба потока приостановлены, ожидая друг друга, и ни один из них не выполняется. Эта ситуация напоминает двух чрезмерно вежливых людей, каждый из которых настаивает на том, чтобы другой прошел в дверь первым!

Казалось бы, избежать взаимоблокировки нетрудно, но это не совсем так. Например, взаимоблокировка может возникнуть в ответвлениях программы. Рассмотрим класс `TickTock`. Как разъяснялось выше, если метод `Pulse()` не выполнится в последний раз из метода `tick()` или `tock()`, то другой метод (т.е. `tock()` или `tick()`) будет находиться в состоянии бесконечного ожидания, и программа зависнет. Зачастую причину взаимоблокировки нелегко понять, просто изучая исходный код программы, поскольку одновременно выполняющиеся потоки могут сложным образом взаимодействовать во время работы. Чтобы избежать взаимоблокировки, необходимо очень аккуратно подходить к программированию и тщательно тестировать написанный код. Как правило, если многопоточная программа вдруг "виснет", то наиболее вероятная причина этого — взаимоблокировка.

Использование атрибута `MethodImplAttribute`

Используя атрибут `MethodImplAttribute`, можно синхронизировать метод целиком. Этот подход можно рассматривать как альтернативу инструкции `lock` в случаях, когда необходимо заблокировать все содержимое метода. Атрибут `MethodImplAttribute` определен в пространстве имен `System.Runtime.CompilerServices`. Конструктор, используемый для синхронизации, имеет такой вид:

```
public MethodImplAttribute(MethodImplOptions opt)
```

Здесь параметр *opt* задает атрибут реализации. Для синхронизации метода задайте атрибут `MethodImplOptions.Synchronized`. Этот атрибут обеспечивает блокировку всего метода.

Рассмотрим версию класса `TickTock`, в которой для синхронизации используется атрибут `MethodImplAttribute`:

```
// Использование атрибута MethodImplAttribute
// для синхронизации метода.

using System;
using System.Threading;
using System.Runtime.CompilerServices;

class TickTock {

  /* Следующий атрибут синхронизирует метод
     tick() целиком. */
  [MethodImplAttribute(MethodImplOptions.Synchronized)]
  public void tick(bool running) {
    if(!running) { // Останов часов.
      Monitor.Pulse(this); // Уведомление для
                           // ожидающих потоков.
      return;
    }

    Console.Write("Тик ");
    Monitor.Pulse(this); // Разрешаем работать
```

```
                          // методу tock().

    Monitor.Wait(this); // Ожидаем, пока не завершится
                        // метод tock().
  }

  /* Следующий атрибут синхронизирует метод
     tock() целиком. */
  [MethodImplAttribute(MethodImplOptions.Synchronized)]
  public void tock(bool running) {
    if(!running) { // Останов часов.
      Monitor.Pulse(this); // Уведомление для
                           // ожидающих потоков.
      return;
    }

    Console.WriteLine("Так");
    Monitor.Pulse(this); // Разрешаем работать
                         // методу tick().

    Monitor.Wait(this); // Ожидаем, пока не завершится
                        // метод tick().
  }

}

class MyThread {
  public Thread thrd;
  TickTock ttOb;

  // Создаем новый поток.
  public MyThread(string name, TickTock tt) {
    thrd = new Thread(new ThreadStart(this.run));
    ttOb = tt;
    thrd.Name = name;
    thrd.Start();
  }

  // Начинаем выполнять новый поток.
  void run() {
    if(thrd.Name == "Тик") {
      for(int i=0; i<5; i++) ttOb.tick(true);
      ttOb.tick(false);
    }
    else {
      for(int i=0; i<5; i++) ttOb.tock(true);
      ttOb.tock(false);
    }
  }
}

class TickingClock {
  public static void Main() {
    TickTock tt = new TickTock();
    MyThread mt1 = new MyThread("Тик", tt);
    MyThread mt2 = new MyThread("Так", tt);
```

```
    mt1.thrd.Join();
    mt2.thrd.Join();
    Console.WriteLine("Часы остановлены.");
  }
}
```

Результаты выполнения программы с этим вариантом класса `TickTock` совпадают с приведенным выше (имеются в виду правильно работающие "часы").

При блокировании всего метода выбор между `lock`-инструкцией или атрибутом `MethodImplAttribute` за вами. Оба средства дают одинаковые результаты. Но поскольку `lock` — ключевое слово языка C#, то в примерах этой книги предпочтение отдано именно встроенному средству синхронизации.

Приостановка, возобновление и завершение выполнения потоков

Иногда выполнение потока необходимо приостановить. Например, поток можно использовать для отображения времени суток. Если пользователь желает убрать часы с экрана, соответствующий поток можно приостановить. Позже, когда необходимость в часах появится снова, выполнение приостановленного потока можно возобновить. В любом случае приостановить и возобновить поток — дело нехитрое. Иногда нужно и вовсе завершить выполнение потока. Завершение выполнения потока отличается от приостановки тем, что завершенный поток удаляется из системы, и его выполнение не может быть возобновлено в дальнейшем.

Для приостановки потока используйте метод `Thread.Suspend()`, а для его возобновления — метод `Thread.Resume()`. Форматы использования этих методов таковы:

```
public void Suspend()
public void Resume()
```

Если вызывающий поток находится не в подходящем для вызываемого метода состоянии, генерируется исключение типа `ThreadStateException`. Такие последствия может иметь, например, попытка возобновить поток, который не был приостановлен.

Чтобы завершить поток, используйте метод `Thread.Abort()`. Самый простой формат его использования выглядит так:

```
public void Abort()
```

Метод `Abort()` генерирует исключение типа `ThreadAbortException` для потока, из которого этот метод вызван. Это исключение и заставляет поток завершиться. Кроме того, то же самое исключение может быть перехвачено программным кодом (с автоматической его регенерацией для завершения потока). Однако следует учитывать, что метод `Abort()` не всегда способен немедленно остановить выполнение потока, поэтому, если важно, чтобы поток был завершен до продолжения вашей программы, необходимо сопроводить вызов метода `Abort()` вызовом метода `Join()`. В некоторых (довольно редких) случаях метод `Abort()` не в состоянии завершить поток. Это возможно в ситуации, когда `finally`-блок включен в бесконечный цикл.

В следующем примере демонстрируется приостановка, возобновление и завершение выполнения потока:

```
// Приостановка, возобновление и завершение потока.

using System;
using System.Threading;
```

```
class MyThread {
  public Thread thrd;

  public MyThread(string name) {
    thrd = new Thread(new ThreadStart(this.run));
    thrd.Name = name;
    thrd.Start();
  }

  // Это входная точка для потока.
  void run() {
    Console.WriteLine(thrd.Name + " стартовал.");

    for(int i = 1; i <= 1000; i++) {
      Console.Write(i + " ");
      if((i%10)==0) {
        Console.WriteLine();
        Thread.Sleep(250);
      }
    }
    Console.WriteLine(thrd.Name + " завершен.");
  }
}

class SuspendResumeStop {
  public static void Main() {
    MyThread mt1 = new MyThread("Мой поток");

    Thread.Sleep(1000); // Разрешаем стартовать
                        // дочернему потоку.

    mt1.thrd.Suspend();
    Console.WriteLine("Приостановка выполнения потока.");
    Thread.Sleep(1000);

    mt1.thrd.Resume();
    Console.WriteLine("Возобновление выполнения потока.");
    Thread.Sleep(1000);

    mt1.thrd.Suspend();
    Console.WriteLine("Приостановка выполнения потока.");
    Thread.Sleep(1000);

    mt1.thrd.Resume();
    Console.WriteLine("Возобновление выполнения потока.");
    Thread.Sleep(1000);

    Console.WriteLine("Завершение выполнения потока.");
    mt1.thrd.Abort();

    mt1.thrd.Join(); // Ожидаем завершения выполнения потока.

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Результаты выполнения этой программы таковы:

```
Мой поток стартовал.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Приостановка выполнения потока.
41 42 43 44 45 46 47 48 49 50
Возобновление выполнения потока.
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
Приостановка выполнения потока.
81 82 83 84 85 86 87 88 89 90
Возобновление выполнения потока.
91 92 93 94 95 96 97 98 99 100
101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120
Завершение выполнения потока.
Основной поток завершен.
```

Альтернативный формат использования метода `Abort()`

В некоторых случаях удобней использовать второй формат метода `Abort()`:

```
public void Abort(object info)
```

Здесь параметр *info* содержит информацию, которую необходимо передать в поток при его завершении. Эта информация доступна через свойство `ExceptionState` класса `ThreadAbortException`. Этот формат можно использовать для передачи потоку кода завершения, что и демонстрируется в следующем примере:

```
// Использование метода Abort(object).

using System;
using System.Threading;

class MyThread {
  public Thread thrd;

  public MyThread(string name) {
    thrd = new Thread(new ThreadStart(this.run));
    thrd.Name = name;
    thrd.Start();
  }

  // Это входная точка для потока.
  void run() {
    try {
      Console.WriteLine(thrd.Name + " стартовал.");

      for(int i = 1; i <= 1000; i++) {
        Console.Write(i + " ");
        if((i%10)==0) {
          Console.WriteLine();
          Thread.Sleep(250);
        }
      }
      Console.WriteLine(thrd.Name +
                        " завершился нормально.");
```

```
    } catch(ThreadAbortException exc) {
      Console.WriteLine(
         "Выполнение потока прервано, код завершения = " +
         exc.ExceptionState);
    }
  }
}

class UseAltAbort {
  public static void Main() {
    MyThread mt1 = new MyThread("Мой поток");

    Thread.Sleep(1000); // Разрешаем стартовать
                        // дочернему потоку.

    Console.WriteLine("Прерывание выполнения потока.");
    mt1.thrd.Abort(100);

    mt1.thrd.Join(); // Ожидаем завершения
                     // выполнения потока.

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Результаты выполнения этой таковы:

```
Мой поток стартовал.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Прерывание выполнения потока.
41 42 43 44 45 46 47 48 49 50
Выполнение потока прервано, код завершения = 100
Основной поток завершен.
```

Как подтверждают результаты выполнения этой программы, методу `Abort()` передано число 100. Это значение затем можно прочитать в свойстве `ExceptionState` класса исключения `ThreadAbortException`, перехватываемого потоком при завершении.

Отмена действия метода `Abort()`

Поток может переопределить запрос на прерывание выполнения. Для этого поток должен перехватить исключение типа `ThreadAbortException`, а затем вызвать метод `ResetAbort()`. Это защищает исключение от автоматической регенерирации по окончании его обработки потоком. Метод `ResetAbort()` объявляется следующим образом:

```
public static void ResetAbort()
```

Обращение к методу `ResetAbort()` может оказаться неудачным, если поток не имеет соответствующего уровня доступа, чтобы отменить прерывание выполнения.

Использование метода `ResetAbort()` демонстрируется в следующей программе:

```
// Использование метода ResetAbort().

using System;
using System.Threading;

class MyThread {
  public Thread thrd;
```

```
  public MyThread(string name) {
    thrd = new Thread(new ThreadStart(this.run));
    thrd.Name = name;
    thrd.Start();
  }

  // Это входная точкласса для потока.
  void run() {
    Console.WriteLine(thrd.Name + " стартовал.");

    for(int i = 1; i <= 1000; i++) {
      try {
        Console.Write(i + " ");
        if((i%10)==0) {
          Console.WriteLine();
          Thread.Sleep(250);
        }
      } catch(ThreadAbortException exc) {
        if((int)exc.ExceptionState == 0) {
          Console.WriteLine(
              "Прерывание отменено! Код завершения = " +
              exc.ExceptionState);
          Thread.ResetAbort();
        }
        else
          Console.WriteLine(
            "Выполнение потока прервано, код завершения = "
            + exc.ExceptionState);
      }
    }
    Console.WriteLine(thrd.Name + " завершился нормально.");
  }
}

class ResetAbort {
  public static void Main() {
    MyThread mt1 = new MyThread("Мой поток");

    Thread.Sleep(1000); // Разрешаем стартовать
                        // дочернему потоку.

    Console.WriteLine("Прерывание выполнения потока.");
    mt1.thrd.Abort(0); // Это не остановит выполнение
                       // потока.

    Thread.Sleep(1000); // Разрешаем дочернему потоку
                        // поработать немного дольше.

    Console.WriteLine("Прерывание выполнения потока.");
    mt1.thrd.Abort(100); // Эта инструкция в состоянии
                         // остановить выполнение потока.

    mt1.thrd.Join(); // Ожидаем завершения
                     // выполнения потока.

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Результаты выполнения этой программы таковы:

```
Мой поток стартовал.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Прерывание выполнения потока.
Прерывание отменено! Код завершения = 0
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
Прерывание выполнения потока.
Выполнение потока прервано, код завершения = 100
Основной поток завершен.
```

Вы убедились, что, если метод `Abort()` вызывается с аргументом, значение которого равно нулю, то посредством вызова метода `ResetAbort()` запрос на прерывание отменяется, и поток продолжает выполняться. Любое другое значение этого аргумента останавливает выполнение потока.

Определение состояния потока

Состояние потока можно получить из свойства `ThreadState`, определенного в классе `Thread`:

```
public Thread ThreadState{ get; }
```

Состояние потока возвращается как значение, определенное перечислением `ThreadState`. В нем определены такие значения:

`ThreadState.Aborted`	`ThreadState.AbortRequested`
`ThreadState.Background`	`ThreadState.Running`
`ThreadState.Stopped`	`ThreadState.StopRequested`
`ThreadState.Suspended`	`ThreadState.SuspendRequested`
`ThreadState.Unstarted`	`ThreadState.WaitSleepJoin`

Одно из перечисленных выше значений требует пояснения. В состояние, представленное значением `ThreadState.WaitSleepJoin`, поток входит, ожидая результатов вызова метода `Wait()`, `Sleep()` или `Join()`.

Использование основного потока

Как упоминалось в начале этой главы, все C#-программы имеют по крайней мере один поток управления, именуемый основным потоком, который автоматически создается в начале выполнения программы. Основной поток обрабатывается подобно всем остальным потокам.

Чтобы получить доступ к основному потоку, необходимо иметь объект класса `Thread`, который его представляет. Это реализуется с помощью свойства `CurrentThread`, которое является членом класса `Thread`. Формат его использования таков:

```
public static Thread CurrentThread{ get; }
```

Это свойство возвращает ссылку на поток, в котором оно опрашивается. Следовательно, если при выполнении основного потока обратиться к свойству

CurrentThread, мы получим ссылку на основной поток. Имея такую ссылку, можно управлять основным потоком, как и любым другим.

В следующей программе демонстрируется, как получить ссылку на основной поток, узнать его имя и приоритет, а также задать новое имя и приоритет:

```
// Управление основным потоком.

using System;
using System.Threading;

class UseMain {
  public static void Main() {
    Thread thrd;

    // Получаем ссылку на объект основного потока.
    thrd = Thread.CurrentThread;

    // Отображаем имя основного потока.
    if(thrd.Name == null)
      Console.WriteLine("Основной поток не имеет имени.");
    else
      Console.WriteLine("Имя основного потока: " +
                        thrd.Name);

    // Отображаем приоритет основного потока.
    Console.WriteLine("Приоритет: " +
                      thrd.Priority);

    Console.WriteLine();

    // Задаем имя и приоритет.
    Console.WriteLine("Установка имени и приоритета.\n");
    thrd.Name = "Основной поток";
    thrd.Priority = ThreadPriority.AboveNormal;

    Console.WriteLine(
             "У основного потока теперь есть имя: " +
             thrd.Name);

    Console.WriteLine("Приоритет теперь таков: " +
                      thrd.Priority);
  }
}
```

Результаты выполнения этой программы таковы:

```
Основной поток не имеет имени.
Приоритет: Normal

Установка имени и приоритета.

У основного потока теперь есть имя: Основной поток
Приоритет теперь таков: AboveNormal
```

Предупреждение: будьте осторожны при выполнении операций над основным потоком. Например, если обращение к методу Join()

```
thrd.Join();
```

добавить в конец метода Main(), программа никогда не завершится, поскольку она будет ожидать, пока не завершится основной поток!

Совет по созданию многопоточных программ

Ключ к эффективному использованию многопоточности лежит в "параллельном" мышлении (в противоположность последовательному). Например, если в вашей программе предполагается функционирование двух параллельно работающих подсистем, организуйте их в виде отдельных потоков. Но если создать слишком много потоков, реальное быстродействие программы может пострадать. Помните: каждое переключение контекста требует определенных расходов системных ресурсов. При большом количестве потоков на изменение контекста будет потрачено больше процессорного времени, чем на выполнение самой программы!

Запуск отдельной задачи

Несмотря на то что чаще всего в C#-программировании используется поточно-ориентированная многозадачность, в соответствующих случаях возможно применение и процессно-ориентированной многозадачности. В многозадачной среде, ориентированной на процессы, вместо запуска в той же программе еще одного потока, одна программа запускает на выполнение другую программу. В C# это реализуется с использованием класса `Process`, который определен в пространстве имен `System.Diagnostics`.

Самый простой способ запустить другую программу — использовать метод `Start()`, определенный в классе `Process`. Вот один из простейших его форматов:

```
public static Process Start(string name)
```

Здесь параметр *name* означает имя выполняемого файла или файла, связанного с выполняемым.

При завершении созданного процесса вызовите метод `Close()`, чтобы освободить память, занимаемую этим процессом. Метод `Close()` объявляется так:

```
public void Close()
```

Завершить процесс можно двумя способами. Если процесс представляет собой GUI-приложение, ориентированное на выполнение под управлением Windows, то для завершения такого процесса достаточно вызвать метод `CloseMainWindow()`, определяемый следующим образом:

```
public bool CloseMainWindow()
```

Этот метод посылает процессу сообщение, предписывающее ему остановиться. Метод возвращает значение `true`, если посланное сообщение получено. Метод возвращает значение `false`, если данное приложение не является GUI-программой или не имеет главного окна. Более того, метод `CloseMainWindow()` — лишь запрос на прекращениие работы. Если он игнорируется приложением, завершение не состоится.

Для безусловного звершения процесса необходимо вызвать метод `Kill()`:

```
public void Kill()
```

Однако использовать метод `Kill()` нужно с большой осторожностью. Он обеспечивает бесконтрольное завершение процесса. Любые несохраненные данные, связанные с этим процессом, будут, скорее всего, утеряны.

Ожидать завершения процесса можно с помощью метода `WaitForExit()`. Его два возможных формата таковы:

```
public void WaitForExit()
public bool WaitForExit(int milliseconds)
```

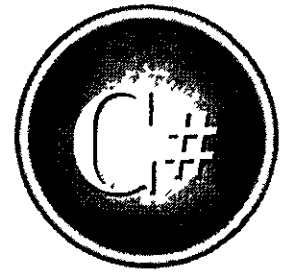
Первый формат позволяет ожидать до тех пор, пока процесс не завершится, а второй — лишь в течение заданного числа миллисекунд. Метод `WaitForExit()`, используемый во втором формате, возвращает значение `true`, если в течение заданного промежутка времени процесс завершился, и значение `false`, если он все еще выполняется.

Следующая программа демонстрирует создание процесса и ожидание его завершения. Программа запускает стандартную утилиту Windows `WordPad.exe`.

```
// Запуск нового процесса.

using System;
using System.Diagnostics;

class StartProcess {
  public static void Main() {
    Process newProc = Process.Start("wordpad.exe");

    Console.WriteLine("Новый процесс стартовал.");

    newProc.WaitForExit();

    newProc.Close(); // Освобождаем ресурсы системы.

    Console.WriteLine("Новый процесс завершился.");
  }
}
```

При выполнении этой программы запускается текстовый редактор WordPad, и на экране появится сообщение "Новый процесс стартовал". Затем программа будет ожидать, пока вы не закроете утилиту WordPad. После завершения работы текстового редактора WordPad отобразится последнее сообщение "Новый процесс завершился".

Полный справочник по



Глава 22

Работа с коллекциями

В C# под *коллекцией* понимается группа объектов. Пространство имен `System.Collections` содержит множество интерфейсов и классов, которые определяют и реализуют коллекции различных типов. Коллекции упрощают программирование, предлагая уже готовые решения для построения структур данных, разработка которых "с нуля" отличается большой трудоемкостью. Речь идет о встроенных коллекциях, которые поддерживают, например, функционирование стеков, очередей и хеш-таблиц. Коллекции пользуются большой популярностью у всех C#-программистов.

Обзор коллекций

Основное достоинство коллекций состоит в том, что они стандартизируют способ обработки групп объектов в прикладных программах. Все коллекции разработаны на основе набора четко определенных интерфейсов. Ряд встроенных реализаций таких интерфейсов, как `ArrayList`, `Hashtable`, `Stack` и `Queue`, вы можете использовать "как есть". У каждого программиста также есть возможность реализовать собственную коллекцию, но в большинстве случаев достаточно встроенных.

Среда .NET Framework поддерживает три основных типа коллекций: общего назначения, специализированные и ориентированные на побитовую организацию данных. Коллекции общего назначения реализуют ряд основных структур данных, включая динамический массив, стек и очередь. Сюда также относятся словари, предназначенные для хранения пар ключ/значение. Коллекции общего назначения работают с данными типа `object`, поэтому их можно использовать для хранения данных любого типа.

Коллекции специального назначения ориентированы на обработку данных конкретного типа или на обработку уникальным способом. Например, существуют специализированные коллекции, предназначенные только для обработки строк или однонаправленного списка.

Классы коллекций, ориентированных на побитовую организацию данных, служат для хранения групп битов. Коллекции этой категории поддерживают такой набор операций, который не характерен для коллекций других типов. Например, в известной многим бит-ориентированной коллекции `BitArray` определены такие побитовые операции, как И и исключающее ИЛИ.

Основополагающим для всех коллекций является реализация *перечислителя* (*нумератора*), который поддерживается интерфейсами `IEnumerator` и `IEnumerable`. Перечислитель обеспечивает стандартизованный способ поэлементного доступа к содержимому коллекции. Поскольку каждая коллекция должна реализовать интерфейс `IEnumerable`, к элементам любого класса коллекции можно получить доступ с помощью методов, определенных в интерфейсе `IEnumerator`. Следовательно, после внесения небольших изменений код, который позволяет циклически опрашивать коллекцию одного типа, можно успешно использовать для циклического опроса коллекции другого типа. Интересно отметить, что содержимое коллекции любого типа можно опросить с помощью нумератора, используемого в цикле `foreach`.

И еще. Если вы знакомы со средствами C++-программирования, то вам будет интересно узнать, что C#-классы коллекций по сути аналогичны классам стандартной библиотеки шаблонов (Standard Template Library — STL), определенной в C++. То, что в C++ называется контейнером, в C# именуется коллекцией. То же справедливо и для Java. Если вы знакомы с Java-средой Collections Framework, то очень легко освоите использование C#-коллекций.

Интерфейсы коллекций

В пространстве имен `System.Collections` определено множество интерфейсов. Но мы начнем именно с интерфейсов коллекций, поскольку они определяют функции, общие для всех классов коллекций. Интерфейсы, которые поддерживают коллекции, приведены в табл. 22.1. А в следующих разделах подробно рассматривается каждый интерфейс.

Интерфейс `ICollection`

Интерфейс `ICollection` можно назвать фундаментом, на котором построены все коллекции. В нем объявлены основные методы и свойства, без которых не может обойтись ни одна коллекция. Он наследует интерфейс `IEnumerable`. Не зная сути интерфейса `ICollection`, невозможно понять механизм действия коллекции.

В интерфейсе `ICollection` определены следующие свойства:

Свойство	*Описание*
`int Count { get; }`	Количество элементов коллекции в данный момент
`bool IsSynchronized { get; }`	Принимает значение `true`, если коллекция синхронизирована, и значение `false` в противном случае По умолчанию коллекции не синхронизированы Но для большинства коллекций можно получить синхронизированную версию
`object SyncRoot { get; }`	Объект, для которого коллекция может быть синхронизирована

Свойство `Count` — самое востребованное, поскольку содержит количество элементов, хранимых в коллекции в данный момент. Если свойство `Count` равно нулю, значит, коллекция пуста. В интерфейсе `ICollection` определен следующий метод:

```
void CopyTo(Array target, int startIdx)
```

Метод `CopyTo()` копирует содержимое коллекции в массив, заданный параметром *`target`*, начиная с индекса, заданного параметром *`startIdx`*. Можно сказать, что метод `CopyTo()` обеспечивает переход от коллекции к стандартному C#-массиву.

Поскольку интерфейс `ICollection` наследует интерфейс `IEnumerable`, он также включает его единственный метод `GetEnumerator()`:

```
IEnumerator GetEnumerator()
```

Этот метод возвращает нумератор коллекции.

Таблица 22.1. Интерфейсы коллекций

Интерфейс	*Описание*
`ICollection`	Определяет элементы, которые должны иметь все коллекции
`IEnumerable`	Определяет метод `GetEnumerator()`, который поддерживает нумератор для любого класса коллекции
`IEnumerator`	Содержит методы, которые позволяют поэлементно получать содержимое коллекции
`IList`	Определяет коллекцию, к которой можно получить доступ посредством индексатора
`IDictionary`	Определяет коллекцию, которая состоит из пар ключ/значение

Окончание табл. 22.1

Интерфейс	Описание
`IDictionaryEnumerator`	Определяет нумератор для коллекции, которая реализует интерфейс `IDictionary`
`IComparer`	Определяет метод `Compare()`, который выполняет сравнение объектов, хранимых в коллекции
`IHashCodeProvider`	Определяет хеш-функцию

Интерфейс `IList`

Интерфейс `IList` наследует интерфейс `ICollection` и определяет поведение коллекции, доступ к элементам которой разрешен посредством индекса с отсчетом от нуля. Помимо методов, определенных в интерфейсе `ICollection`, интерфейс `IList` определяет и собственные методы (они сведены в табл. 22.2). Некоторые из этих методов служат для модификации коллекции. Если же коллекция предназначена только для чтения или имеет фиксированный размер, вызов этих методов приведет к генерированию исключения типа `NotSupportedException`.

Таблица 22.2. Методы, определенные в интерфейсе `IList`

Метод	Описание
`int Add(object obj)`	Добавляет объект `obj` в вызывающую коллекцию. Возвращает индекс, по которому этот объект сохранен
`void Clear()`	Удаляет все элементы из вызывающей коллекции
`bool Contains(object obj)`	Возвращает значение `true`, если вызывающая коллекция содержит объект, переданный в параметре `obj`, и значение `false` в противном случае
`int IndexOf(object obj)`	Возвращает индекс объекта `obj`, если он (объект) содержится в вызывающей коллекции. Если объект `obj` не обнаружен, метод возвращает -1
`void Insert(int idx, object obj)`	Вставляет в вызывающую коллекцию объект `obj` по индексу, заданному параметром `idx`. Элементы, находившиеся до этого по индексу `idx` и далее, смещаются вперед, чтобы освободить место для вставляемого объекта `obj`
`void Remove(object obj)`	Удаляет из вызывающей коллекции объект, расположенный по индексу, заданному параметром `idx`. Элементы, находившиеся до этого за удаленным элементом, смещаются назад, чтобы ликвидировать образовавшуюся "брешь"
`void RemoveAt(int idx)`	Удаляет первое вхождение объекта `obj` из вызывающей коллекции. Элементы, находившиеся до этого за удаленным элементом, смещаются назад, чтобы ликвидировать образовавшуюся "брешь"

Объекты добавляются в коллекцию типа `IList` посредством вызова метода `Add()`. Обратите внимание на то, что метод `Add()` принимает аргументы типа `object`. Поскольку класс `object` является базовым для всех типов, в коллекции можно сохранить объект любого типа. Это справедливо и для типов значений (нессылочных типов), поскольку здесь автоматически выполняется приведение к объектному типу (boxing).

Удалить объект из коллекции можно с помощью метода `Remove()` или `RemoveAt()`. Метод `Remove()` удаляет заданный объект, а метод `RemoveAt()` — объект, расположенный по заданному индексу. Для полной очистки коллекции достаточно вызвать метод `Clear()`.

Чтобы узнать, содержит ли коллекция заданный объект, вызовите метод `Contains()`. Получить индекс заданного объекта поможет метод `IndexOf()`, а вставить элемент по заданному индексу — метод `Insert()`.

В классе `IList` определены следующие свойства:

```
bool IsFixedSize { get; }
bool IsReadOnly { get; }
```

Если коллекция имеет фиксированный размер, свойство `IsFixedSize` принимает значение `true`. Это означает, что в такую коллекцию нельзя вставлять элементы и удалять их из нее. Если коллекция предназначена только для чтения, свойство `IsReadOnly` имеет значение `true`. Это говорит о том, что содержимое коллекции изменению не подлежит.

В классе `IList` определен следующий индексатор:

```
object this[int idx] { get; set; }
```

Этот индексатор можно использовать для считывания или записи значения нужного элемента. Но его нельзя применить для добавления в коллекцию нового элемента. Для этой цели существует метод `Add()`. Но после того как элемент добавлен в коллекцию, к нему можно получить доступ с помощью индексатора.

Интерфейс `IDictionary`

Интерфейс `IDictionary` определяет поведение коллекции, которая устанавливает соответствие между уникальными ключами и значениями. Ключ — это объект, который используется для получения соответствующего ему значения. Следовательно, коллекция, которая реализует интерфейс `IDictionary`, служит для хранения пар ключ/значение. Сохраненную однажды пару можно затем извлечь по заданному ключу. Интерфейс `IDictionary` наследует интерфейс `ICollection`. Методы, объявленные в интерфейсе `IDictionary`, сведены в табл. 22.3. Некоторые из них при попытке задать нулевой ключ генерируют исключение типа `NotSupportedException`.

Таблица 22.3. Методы, определенные в интерфейсе `IDictionary`

Метод	*Описание*
`void Add(object k, object v)`	Добавляет в вызывающую коллекцию пару ключ/значение, заданную параметрами *k* и *v*. Ключ *k* не должен быть нулевым. Если окажется, что ключ *k* уже хранится в коллекции, генерируется исключение типа `ArgumentException`
`void Clear()`	Удаляет все пары ключ/значение из вызывающей коллекции
`bool Contains(object k)`	Возвращает значение `true`, если вызывающая коллекция содержит объект *k* в качестве ключа. В противном случае возвращает значение `false`
`IDictionaryEnumerator GetEnumerator()`	Возвращает нумератор для вызывающей коллекции
`void Remove(object k)`	Удаляет элемент, ключ которого равен значению *k*

Чтобы добавить пару ключ/значение в `IDictionary`-коллекцию, используйте метод `Add()`. Обратите внимание на то, что ключ и его значение задаются как отдельные параметры. Чтобы удалить из коллекции-словаря ненужный элемент, необходимо передать методу `Remove()` соответствующий ему ключ. Для очистки всей коллекции используется метод `Clear()`.

Чтобы определить, содержит ли коллекция заданный объект, достаточно вызвать метод `Contains()`, передав ему ключ интересующего вас объекта. Метод `GetEnumerator()` возвращает нумератор, совместимый с коллекцией `IDictionary`. Этот нумератор работает в "парном" (ключ/значение) режиме.

В интерфейсе `IDictionary` определены следующие свойства:

Свойство	*Описание*
`bool IsFixedSize { get; }`	Равно значению `true`, если словарь имеет фиксированный размер
`bool IsReadOnly { get; }`	Равно значению `true`, если словарь предназначен только для чтения
`ICollection Keys { get; }`	Получает коллекцию ключей
`ICollection Values { get; }`	Получает коллекцию значений

Обратите внимание на то, что с помощью свойств `Keys` и `Values` ключи и значения, хранимые в словарной коллекции, можно получить в виде отдельных списков.

В интерфейсе `IDictionary` определен следующий индексатор:

```
object this[object key] { get; set; }
```

Этот индексатор можно использовать для получения или установки значения элемента. Его можно также использовать для добавления в коллекцию нового элемента. Обратите внимание на то, что "индекс" в данном случае не является обычным индексом, а ключом элемента.

Интерфейсы `IEnumerable`, `IEnumerator` и `IDictionaryEnumerator`

Интерфейс `IEnumerable` должен быть реализован в любом классе, если в нем предполагается поддержка нумераторов. Как упоминалось выше, все классы коллекций реализуют интерфейс `IEnumerable`, поскольку он наследуется интерфейсом `ICollection`. В интерфейсе `IEnumerable` определен единственный метод с именем `GetEnumerator()`:

```
IEnumerator GetEnumerator()
```

Он возвращает нумератор для коллекции. Кроме того, реализация интерфейса `IEnumerable` позволяет получить доступ к содержимому коллекции с помощью цикла `foreach`.

Интерфейс `IEnumerator`, собственно, и определяет действие любого нумератора. Используя его методы, можно циклически опросить содержимое коллекции. Для коллекций, в которых хранятся пары ключ/значение (т.е. словари), метод `GetEnumerator()` возвращает объект типа `IDictionaryEnumerator`, а не типа `IEnumerator`. Класс `IDictionaryEnumerator` является производным от класса `IEnumerator` и распространяет свои функциональные возможности нумератора на область словарей.

Методы, определенные в интерфейсе `IEnumerator`, рассматриваются ниже в этой главе.

Интерфейс `IComparer`

В интерфейсе `IComparer` определен метод `Compare()`, который позволяет сравнивать два объекта:

```
int Compare(object v1, object v2)
```

Метод `Compare()` возвращает положительное число, если значение *v1* больше значения *v2*, отрицательное, если *v1* меньше *v2*, и нуль, если сравниваемые значения равны. Этот интерфейс можно использовать для задания способа сотировки элементов коллекции.

Интерфейс `IHashCodeProvider`

Интерфейс `IHashCodeProvider` должен быть реализован коллекцией, если программисту необходимо определить собственную версию метода `GetHashCode()`. Вспомните, что все объекты (для получения хеш-кода) наследуют метод `Object.GetHashCode()`, который используется по умолчанию. Посредством реализации интерфейса `IHashCodeProvider` можно определить альтернативный метод.

Структура `DictionaryEntry`

В пространстве имен `System.Collections` определен тип структуры с именем `DictionaryEntry`. Коллекции, в которых хранятся пары ключ/значение, используют для их хранения объект типа `DictionaryEntry`. В этой структуре определены следующие два свойства:

```
public object Key { get; set; }
public object Value { get; set; }
```

Эти свойства используются для получения доступа к ключу или к соответствующему ему значению. Объект типа `DictionaryEntry` можно создать с помощью следующего конструктора:

```
public DictionaryEntry(object k, object v)
```

Здесь параметр *k* принимает ключ, а параметр *v* — значение.

Классы коллекций общего назначения

Теперь, когда вы познакомились с интерфейсами коллекций, можно переходить к рассмотрению стандартных классов, которые их реализуют. Как упоминалось выше, классы коллекций делятся на три основных категории: общего назначения, специализированные и ориентированные на побитовую организацию данных. Классы общего назначения можно использовать для хранения объектов любого типа. Битовые предназначены для хранения битовой информации. Коллекции специального назначения разрабатываются для обработки данных конкретного типа. Этот раздел посвящен классам коллекций общего назначения.

Итак, перечислим классы коллекций общего назначения:

Класс	*Описание*
`ArrayList`	Динамический массив, т.е. массив который при необходимости может увеличивать свой размер
`Hashtable`	Хеш-таблица для пар ключ/значение
`Queue`	Очередь, или список, действующий по принципу: первым прибыл — первым обслужен
`SortedList`	Отсортированный список пар ключ/значение
`Stack`	Стек, или список, действующий по принципу: первым прибыл — последним обслужен

В следующих разделах эти классы коллекций и возможности их использования рассматриваются более подробно.

Класс ArrayList

Класс ArrayList предназначен для поддержки динамических массивов, которые при необходимости могут увеличиваться или сокращаться. В C# стандартные массивы имеют фиксированную длину, которая не может измениться во время выполнения программы. Это означает, что программист должен знать заранее, сколько элементов будет храниться в массиве. Но иногда до выполнения программы нельзя точно сказать, массив какого размера понадобится. В таких случаях и используется класс ArrayList. Объект класса ArrayList представляет собой массив переменной длины, элементами которого являются объектные ссылки. Любой объект класса ArrayList создается с некоторым начальным размером. При превышении этого размера коллекция автоматически его увеличивает. В случае удаления объектов массив можно сократить. Коллекция класса ArrayList, пожалуй, наиболее употребимая, поэтому ее стоит рассмотреть в деталях.

Класс ArrayList реализует интерфейсы ICollection, IList, IEnumerable и ICloneable. В классе ArrayList определены следующие конструкторы:

```
public ArrayList()
public ArrayList(ICollection c)
public ArrayList(int capacity)
```

Первый конструктор предназначен для создания пустого ArrayList-массива с начальной емкостью, равной 16 элементам. Второй служит для построения массива, который инициализируется элементами и емкостью коллекции, заданной параметром *c*. Третий конструктор создает список с заданной начальной емкостью. *Емкость* (или вместимость) — это размер массива для хранения элементов. При добавлении элементов в ArrayList-массив его емкость автоматически увеличивается, причем каждый раз, когда список должен расшириться, его емкость удваивается.

Помимо методов, определенных в интерфейсах, которые реализует класс ArrayList, в нем определены и собственные методы. Наиболее употребимые из них перечислены в табл. 22.4. Коллекцию класса ArrayList можно отсортировать с помощью метода Sort(). В отсортированной коллекции можно эффективно выполнять поиск элементов, используя метод BinarySearch(). При необходимости содержимое ArrayList-коллекции можно реверсировать, вызвав метод Reverse().

Класс ArrayList поддерживает ряд методов, которые действуют в некотором диапазоне элементов коллекции. Например, вызвав метод InsertRange(), можно вставить в ArrayList-массив другую коллекцию. С помощью метода RemoveRange() можно удалить из коллекции заданный диапазон элементов. А если нужно заменить элементы заданного диапазона одной коллекции элементами другой, используйте метод SetRange(). Сортировать можно не только всю коллекцию, но и заданный диапазон внутри нее. То же справедливо и для поиска.

По умолчанию коллекция класса ArrayList не синхронизирована. Чтобы поместить коллекцию в синхронизированную оболочку, вызовите метод Synchronized().

Таблица 22.4. Наиболее употребимые методы класса ArrayList

Метод	*Описание*
`public virtual void AddRange(ICollection c)`	Добавляет элементы из коллекции *c* в конец вызывающей коллекции
`public virtual int BinarySearch(object v)`	В вызывающей коллекции выполняет поиск значения, заданного параметром *v*. Возвращает индекс найденного элемента. Если искомое значение не обнаружено, возвращает отрицательное значение. Вызывающий список должен быть отсортирован

Метод	Описание
`public virtual int BinarySearch(object v, IComparer comp)`	В вызывающей коллекции выполняет поиск значения, заданного параметром `v`, на основе метода сравнения объектов, заданного параметром `comp`. Возвращает индекс найденного элемента. Если искомое значение не обнаружено, возвращает отрицательное значение. Вызывающий список должен быть отсортирован
`public virtual int BinarySearch(int startIdx, int count, object v, IComparer comp)`	В вызывающей коллекции выполняет поиск значения, заданного параметром `v`, на основе метода сравнения объектов, заданного параметром `comp`. Поиск начинается с элемента, индекс которого равен значению `startIdx`, и включает `count` элементов. Метод возвращает индекс найденного элемента. Если искомое значение не обнаружено, возвращает отрицательное значение. Вызывающий список должен быть отсортирован
`public virtual void CopyTo(Array ar, int startIdx)`	Копирует содержимое вызывающей коллекции, начиная с элемента, индекс которого равен значению `startIdx`, в массив, заданный параметром `ar`. Приемный массив должен быть одномерным и совместимым по типу с элементами коллекции
`public virtual void CopyTo(int srcIdx, Array ar, int destIdx, int count)`	Копирует `count` элементов вызывающей коллекции, начиная с элемента, индекс которого равен значению `srcIdx`, в массив, заданный параметром `ar`, начиная с элемента, индекс которого равен значению `destIdx`. Приемный массив должен быть одномерным и совместимым по типу с элементами коллекции
`public virtual ArrayList GetRange(int idx, int count)`	Возвращает часть вызывающей коллекции типа `ArrayList`. Диапазон возвращаемой коллекции начинается с индекса `idx` и включает `count` элементов. Возвращаемый объект ссылается на те же элементы, что и вызывающий объект
`public static ArrayList FixedSize(ArrayList ar)`	Превращает коллекцию `ar` в `ArrayList`-массив с фиксированным размером и возвращает результат
`public virtual void InsertRange(int startIdx, ICollection c)`	Вставляет элементы коллекции, заданной параметром `c`, в вызывающую коллекцию, начиная с индекса, заданного параметром `startIdx`
`public virtual int LastIndexOf(object v)`	Возвращает индекс последнего вхождения объекта `v` в вызывающей коллекции. Если искомый объект не обнаружен, возвращает значение -1
`public static ArrayList ReadOnly(ArrayList ar)`	Превращает коллекцию `ar` в `ArrayList`-массив, предназначенный только для чтения, и возвращает результат
`public virtual void RemoveRange(int idx, int count)`	Удаляет `count` элементов из вызывающей коллекции, начиная с элемента, индекс которого равен значению `idx`
`public virtual void Reverse()`	Располагает элементы вызывающей коллекции в обратном порядке
`public virtual void Reverse(int startIdx, int count)`	Располагает в обратном порядке `count` элементов вызывающей коллекции, начиная с индекса `startIdx`
`public virtual void SetRange(int startIdx, ICollection c)`	Заменяет элементы вызывающей коллекции, начиная с индекса `startIdx`, элементами коллекции, заданной параметром `c`
`public virtual void Sort()`	Сортирует коллекцию по возрастанию
`public virtual void Sort(IComparer comp)`	Сортирует вызывающую коллекцию на основе метода сравнения объектов, заданного параметром `comp`. Если параметр `comp` имеет нулевое значение, для каждого объекта используется стандартный метод сравнения

Окончание табл. 22.4

Метод	Описание
`public virtual void Sort(int startIdx, int endIdx, IComparer comp)`	Сортирует часть вызывающей коллекции на основе метода сравнения объектов, заданного параметром `comp`. Сортировка начинается с индекса `startIdx` и заканчивается индексом `endIdx`. Если параметр `comp` имеет нулевое значение, для каждого объекта используется стандартный метод сравнения
`public static ArrayList Synchronized(ArrayList list)`	Возвращает синхронизированную версию вызывающей коллекции
`public virtual object[] ToArray()`	Возвращает массив, который содержит копии элементов вызывающего объекта
`public virtual Array ToArray(Type type)`	Возвращает массив, который содержит копии элементов вызывающего объекта. Тип элементов в этом массиве задается параметром `type`
`public virtual void TrimToSize()`	Устанавливает свойство `Capacity` равным значению свойства `Count`

Помимо свойств, определенных в интерфейсах, реализуемых классом `ArrayList`, в нем также определено свойство `Capacity`:

```
public virtual int Capacity { get; set; }
```

Свойство `Capacity` позволяет узнать или установить емкость вызывающего динамического массива типа `ArrayList`. Емкость представляет собой количество элементов, которые можно сохранить в `ArrayList`-массиве без его увеличения. Как упоминалось выше, `ArrayList`-массив увеличивается автоматически, и поэтому нет необходимости устанавливать его емкость вручную. Но если вам заранее известно, сколько элементов должно содержаться в `ArrayList`-массиве, то из соображений эффективности в установке свойства `Capacity` есть резон. Тем самым вы сэкономите системные ресурсы, затрачиваемые на выделение дополнительной памяти.

И наоборот, если вы захотите уменьшить размер `ArrayList`-массива, то путем соответствующей установки свойства `Capacity` можно сделать его меньшим. Однако это значение не должно быть меньше значения свойства `Count`. Вспомните, что свойство `Count`, определенное в интерфейсе `ICollection`, содержит реальное количество объектов, хранимых в коллекции в данный момент. Попытка установить свойство `Capacity` равным значению, которое меньше значения свойства `Count`, приведет к генерированию исключения типа `ArgumentOutOfRangeException`. Чтобы сделать емкость `ArrayList`-массива равной действительному количеству элементов, хранимых в нем в данный момент, установите свойство `Capacity` равным свойству `Count`. Того же эффекта можно добиться, вызвав метод `TrimToSize()`.

Использование динамического массива типа `ArrayList` демонстрируется в следующей программе. Сначала она создает `ArrayList`-массив, а затем добавляет в него символы и отображает их. После удаления элементов список отображается снова. Последующее добавление новых элементов приводит к автоматическому увеличению емкости используемого списка. На "десерт" демонстрируется возможность изменения содержимого его элементов.

```
// Демонстрация использования ArrayList-массива.

using System;
using System.Collections;

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

```
// Создаем динамический массив.
ArrayList al = new ArrayList();

Console.WriteLine("Начальная емкость: " +
                  al.Capacity);
Console.WriteLine("Начальное количество элементов: " +
                  al.Count);

Console.WriteLine();

Console.WriteLine("Добавляем 6 элементов.");
// Добавляем элементы в динамический массив.
al.Add('C');
al.Add('A');
al.Add('E');
al.Add('B');
al.Add('D');
al.Add('F');

Console.WriteLine("Текущая емкость: " +
                  al.Capacity);
Console.WriteLine("Количество элементов: " +
                  al.Count);

// Отображаем массив, используя индексацию.
Console.Write("Текущее содержимое массива: ");
for(int i=0; i < al.Count; i++)
  Console.Write(al[i] + " ");
Console.WriteLine("\n");

Console.WriteLine("Удаляем 2 элемента.");
// Удаляем элементы из динамического массива.
al.Remove('F');
al.Remove('A');

Console.WriteLine("Текущая емкость: " +
                  al.Capacity);
Console.WriteLine("Количество элементов: " +
                  al.Count);

// Для отображения массива используем цикл foreach.
Console.Write("Содержимое: ");
foreach(char c in al)
  Console.Write(c + " ");
Console.WriteLine("\n");

Console.WriteLine("Добавляем еще 20 элементов.");
// Добавляем такое количество элементов в массив,
// которое заставит его увелить свой размер.
for(int i=0; i < 20; i++)
  al.Add((char)('a' + i));
Console.WriteLine("Текущая емкость: " +
                  al.Capacity);
Console.WriteLine(
  "Количество элементов после добавления 20 новых: " +
  al.Count);
Console.Write("Содержимое: ");
foreach(char c in al)
```

```
      Console.Write(c + " ");
    Console.WriteLine("\n");

    // Изменяем содержимое массива, используя индексацию.
    Console.WriteLine("Изменяем первые три элемента.");
    al[0] = 'X';
    al[1] = 'Y';
    al[2] = 'Z';
    Console.Write("Содержимое: ");
    foreach(char c in al)
      Console.Write(c + " ");
    Console.WriteLine();
  }
}
```

Результаты выполнения программы таковы:

```
Начальная емкость: 16
Начальное количество элементов: 0

Добавляем 6 элементов.
Текущая емкость: 16
Количество элементов: 6
Текущее содержимое массива: C A E B D F

Удаляем 2 элемента.
Текущая емкость: 16
Количество элементов: 4
Содержимое: C E B D

Добавляем еще 20 элементов.
Текущая емкость: 32
Количество элементов после добавления 20 новых: 24
Содержимое: C E B D a b c d e f g h i j k l m n o p q r s t

Изменяем первые три элемента.
Содержимое: X Y Z D a b c d e f g h i j k l m n o p q r s t
```

Обратите внимание на то, что коллекция с самого начала работы этой программы пуста, но ее начальная емкость равна 16. При необходимости коллекция увеличивается, причем каждый раз, когда возникает такая необходимость, емкость удваивается.

Сортировка `ArrayList`-массивов и выполнение поиска

Динамический массив типа `ArrayList` можно отсортировать с помощью метода `Sort()`. В отсортированном массиве можно эффективно выполнить поиск элемента, используя метод `BinarySearch()`. Применение упомянутых методов демонстрируется следующей программой:

```
// Сортировка ArrayList-массива и поиск в нем
// заданного элемента.

using System;
using System.Collections;

class SortSearchDemo {
  public static void Main() {
    // Создаем динамический массив.
    ArrayList al = new ArrayList();
```

```
    // Добавляем в него элементы.
    al.Add(55);
    al.Add(43);
    al.Add(-4);
    al.Add(88);
    al.Add(3);
    al.Add(19);

    Console.Write("Исходное содержимое: ");
    foreach(int i in al)
      Console.Write(i + " ");
    Console.WriteLine("\n");

    // Сортируем массив.
    al.Sort();

    // Используем цикл foreach для отображения
    // содержимого массива.
    Console.Write("Содержимое после сортировки: ");
    foreach(int i in al)
      Console.Write(i + " ");
    Console.WriteLine("\n");

    Console.WriteLine("Индекс элемента 43 равен " +
                      al.BinarySearch(43));
  }
}
```

Вот результаты выполнения программы:

```
Исходное содержимое: 55 43 -4 88 3 19

Содержимое после сортировки: -4 3 19 43 55 88

Индекс элемента 43 равен 3
```

Несмотря на то что `ArrayList`-массив может в одном списке хранить объекты любого типа, прежде чем выполнять сортировку или поиск объектов, необходимо обеспечить возможность сравнения объектов списка. Например, предыдущая программа сгенерировала бы исключение, если рассматриваемый список включал бы строку. (Конечно, программист может создавать собственные методы сравнения, например, строк и целочисленных значений. О пользовательских компараторах речь пойдет ниже в этой главе.)

Создание обычного массива из динамического

При работе с `ArrayList`-массивом иногда нужно получить обычный массив, который бы содержал элементы динамического. Это можно сделать с помощью метода `ToArray()`. Существует ряд причин для преобразования коллекции в массив. Во-первых, для определенных операций иногда важно сократить время обработки. Во-вторых, возможны ситуации, когда необходимо передать массив методу, который не перегружен для приема коллекции. В любом случае преобразование `ArrayList`-коллекции в массив не представляет сложности, и в этом легко убедиться, рассмотрев следующую программу:

```
// Преобразование ArrayList-коллекции в массив.

using System;
using System.Collections;
```

```
class ArrayListToArray {
  public static void Main() {
    ArrayList al = new ArrayList();

    // Добавляем элементы в динамический массив.
    al.Add(1);
    al.Add(2);
    al.Add(3);
    al.Add(4);

    Console.Write("Содержимое: ");
    foreach(int i in al)
      Console.Write(i + " ");
    Console.WriteLine();

    // Создаем обычный массив из динамического.
    int[] ia = (int[]) al.ToArray(typeof(int));
    int sum = 0;

    // Суммируем элементы массива.
    for(int i=0; i<ia.Length; i++)
      sum += ia[i];

    Console.WriteLine("Сумма равна: " + sum);
  }
}
```

Результаты выполнения этой программы таковы:

```
Содержимое: 1 2 3 4
Сумма равна: 10
```

Эта программа начинается с создания коллекции целочисленных значений. Затем вызывается метод `ToArray()`, который принимает в качестве параметра тип `int`. Это обеспечивает создание массива целочисленных значений. Поскольку в качестве типа значения, которое возвращает метод `ToArray()`, указан тип `Array`, содержимое создаваемого массива должно быть приведено к типу `int[]`. Наконец, суммирование значений обычного массива — вообще пара пустяков.

Класс `Hashtable`

Класс `Hashtable` предназначен для создания коллекции, в которой для хранения объектов используется *хеш-таблица*. Возможно, многим известно, что в хеш-таблице для хранения информации используется механизм, именуемый *хешированием* (hashing). Суть хеширования состоит в том, что для определения уникального значения, которое называется *хеш-кодом*, используется информационное содержимое соответствующего ему ключа. Хеш-код затем используется в качестве индекса, по которому в таблице отыскиваются данные, соответствующие этому ключу. Преобразование ключа в хеш-код выполняется автоматически, т.е. сам хеш-код вы даже не увидите. Но преимущество хеширования — в том, что оно позволяет сохранять постоянным время выполнения таких операций, как поиск, считывание и запись данных, даже для больших объемов информации. Класс `Hashtable` реализует интерфейсы `IDictionary`, `ICollection`, `IEnumerable`, `ISerializable`, `IDeserializationCallback` и `ICloneable`.

В классе `Hashtable` определено множество конструкторов, включая следующие (они используются чаще всего):

```
public Hashtable()
public Hashtable(IDictionary c)
```

```
public Hashtable(int capacity)
public Hashtable(int capacity, float fillRatio)
```

Первая форма позволяет создать стандартный объект класса `Hashtable`. Вторая для инициализации `Hashtable`-объекта использует элементы заданной коллекции *c*. Третья инициализирует емкость создаваемой хеш-таблицы значением *capacity*, а четвертая — как емкость (значением *capacity*), так и коэффициент заполнения (значением *fillRatio*). Значение *коэффициента заполнения* (также именуемого *коэффициентом нагрузки*), которое должно попадать в диапазон 0,1—1,0, определяет степень заполнения хеш-таблицы, после чего ее размер увеличивается. В частности, размер таблицы увеличится, когда количество элементов станет больше емкости таблицы, умноженной на ее коэффициент заполнения. Для конструкторов, которые в качестве параметра не принимают коэффициент заполнения, используется значение 1,0.

В классе `Hashtable` помимо методов, определенных в реализованных им интерфейсах, также определены собственные методы. Наиболее употребимые из них перечислены в табл. 22.5. Чтобы определить, содержит ли `Hashtable`-коллекция заданный ключ, достаточно вызвать метод `ContainsKey()`. А чтобы узнать, хранится ли в интересующей вас хеш-таблице заданное значение, вызовите метод `ContainsValue()`. Для опроса элементов `Hashtable`-коллекции необходимо получить нумератор типа `IDictionaryEnumerator`, вызвав метод `GetEnumerator()`. Вспомните, что для опроса содержимого коллекции, в которой хранятся пары ключ/значение, используется именно класс `IDictionaryEnumerator`.

Таблица 22.5. Наиболее употребимые методы класса `Hashtable`

Метод	Описание
`public virtual bool ContainsKey(object k)`	Возвращает значение `true`, если в вызывающей `Hashtable`-коллекции содержится ключ, заданный параметром *k*. В противном случае возвращает значение `false`
`public virtual bool ContainsValue(object v)`	Возвращает значение `true`, если в вызывающей `Hashtable`-коллекции содержится значение, заданное параметром *v*. В противном случае возвращает значение `false`
`public virtual IDictionaryEnumerator GetEnumerator()`	Возвращает для вызывающей `Hashtable`-коллекции нумератор типа `IDictionaryEnumerator`
`public static Hashtable Synchronized(Hashtable ht)`	Возвращает синхронизированную версию вызывающей `Hashtable`-коллекции, переданной в параметре *ht*

В классе `Hashtable`, помимо свойств, определенных в реализованных им интерфейсах, также определены два собственных `public`-свойства. Используя следующие свойства, можно из `Hashtable`-коллекции получить коллекцию ключей или значений:

```
public virtual ICollection Keys { get; }
public virtual ICollection Values { get; }
```

Поскольку в классе `Hashtable` не предусмотрена поддержка упорядоченных коллекций, при получении коллекции ключей или значений заданный порядок элементов не достигается. В классе `Hashtable` также определены два `protected`-свойства с именами `Hcp` и `Comparer`, которые доступны для производных классов.

В классе `Hashtable` пары ключ/значение хранятся в форме структуры типа `DictionaryEntry`, но по большей части вас это не будет касаться, поскольку свойства и методы обрабатывают ключи и значения отдельно. Например, при добавлении элемента в `Hashtable`-коллекцию необходимо вызвать метод `Add()`, который принимает два отдельных аргумента: ключ и значение.

Важно отметить, что Hashtable-коллекция не гарантирует сохранения порядка элементов. Дело в том, что хеширование обычно не применяется к отсортированным таблицам.

Рассмотрим пример, который демонстрирует использование Hashtable-коллекции:

```
// Демонстрация использования Hashtable-коллекции.

using System;
using System.Collections;

class HashtableDemo {
  public static void Main() {
    // Создаем хеш-таблицу.
    Hashtable ht = new Hashtable();

    // Добавляем элементы в хеш-таблицу.
    ht.Add("здание", "жилое помещение");
    ht.Add("автомобиль", "транспортное средство");
    ht.Add("книга", "набор печатных слов");
    ht.Add("яблоко", "съедобный фрукт");

    // Добавлять элементы можно также с помощью индексатора.
    ht["трактор"] = "сельскохозяйственная машина";

    // Получаем коллекцию ключей.
    ICollection c = ht.Keys;

    // Используем ключи для получения значений.
    foreach(string str in c)
      Console.WriteLine(str + ": " + ht[str]);
  }
}
```

Результаты выполнения этой программы таковы:

```
яблоко: съедобный фрукт
здание: жилое помещение
трактор: сельскохозяйственная машина
автомобиль: транспортное средство
книга: набор печатных слов
```

Как видно по приведенным результатам, пары ключ/значение хранятся отнюдь не в упорядоченном виде. Обратите внимание на то, как было получено и отображено содержимое хеш-таблицы ht. Во-первых, коллекция ключей считывается с помощью свойства Keys. Каждый ключ затем используется в качестве индекса хеш-таблицы ht, который позволяет найти и отобразить значение, соответствующее каждому ключу. Не забывайте, что индексатор, определенный в интерфейсе IDictionary и реализованный в классе Hashtable, использует ключ в роли индекса.

Класс SortedList

Класс SortedList предназначен для создания коллекции, которая хранит пары ключ/значение в упорядоченном виде, а именно отсортированы по ключу. Класс SortedList реализует интерфейсы IDictionary, ICollection, IEnumerable и ICloneable.

В классе SortedList определено несколько конструкторов, включая следующие:

```
public SortedList()
public SortedList(IDictionary c)
public SortedList(int capacity)
public SortedList(IComparer comp)
```

Первый конструктор позволяет создать пустую коллекцию с начальной емкостью, равной 16 элементам. Второй создает SortedList-коллекцию, которая инициализируется элементами и емкостью коллекции, заданной параметром *c*. Третий конструктор предназначен для построения пустого SortedList-списка, который инициализируется емкостью, заданной параметром *capacity*. Как вы помните, емкость — это размер базового массива, который используется для хранения элементов коллекции. Четвертая форма конструктора позволяет задать метод сравнения, который должен использоваться для сравнения объектов списка. С помощью этой формы создается пустая коллекция с начальной емкостью, равной 16 элементам.

Емкость SortedList-коллекции увеличивается автоматически, если в этом возникает необходимость, при добавлении элементов. Если окажется, что текущая емкость может быть превышена, она удваивается. Преимущество задания емкости при создании SortedList-списка состоит в минимизации затрат системных ресурсов, связанных с изменением размера коллекции. Конечно, задавать начальную емкость имеет смысл только в том случае, если вы знаете, какое количество элементов должно храниться в коллекции.

В классе SortedList помимо методов, определенных в реализованных им интерфейсах, также определены собственные методы. Наиболее употребимые приведены в табл. 22.6. Чтобы определить, содержит ли SortedList-коллекция заданный ключ, достаточно вызвать метод ContainsKey(). А чтобы узнать, хранится ли в списке заданное значение, вызовите метод ContainsValue(). Для опроса элементов SortedList-коллекции необходимо получить нумератор типа IDictionaryEnumerator, вызвав метод GetEnumerator(). Вспомните, что для опроса содержимого коллекции, в которой хранятся пары ключ/значение, используется именно класс IDictionaryEnumerator. Синхронизированную версию SortedList-коллекции можно получить с помощью метода Synchronized().

Таблица 22.6. Наиболее употребимые методы, определенные в классе SortedList

Метод	*Описание*
`public virtual bool ContainsKey(object k)`	Возвращает значение true, если в вызывающей SortedList-коллекции содержится ключ, заданный параметром *k*. В противном случае возвращает значение false
`public virtual bool ContainsValue(object v)`	Возвращает значение true, если в вызывающей SortedList-коллекции содержится значение, заданное параметром *k*. В противном случае возвращает значение false
`public virtual object GetByIndex(int idx)`	Возвращает значение, индекс которого задан параметром *idx*
`public virtual IDictionaryEnumerator GetEnumerator()`	Возвращает нумератор типа IDictionaryEnumerator для вызывающей SortedList-коллекции
`public virtual object GetKey(int idx)`	Возвращает ключ, индекс которого задан параметром *idx*
`public virtual IList GetKeyList()`	Возвращает IList-коллекцию ключей, хранимых в вызывающей SortedList-коллекции
`public virtual IList GetValueList()`	Возвращает IList-коллекцию значений, хранимых в вызывающей SortedList-коллекции

Окончание табл. 22.6

Метод	Описание
`public virtual int IndexOfKey(object k)`	Возвращает индекс ключа, заданного параметром *k*. Возвращает значение –1, если в списке нет заданного ключа
`public virtual int IndexOfValue(object v)`	Возвращает индекс первого вхождения значения, заданного параметром *v*. Возвращает –1, если в списке нет заданного ключа
`public virtual void SetByIndex(int idx, object v)`	Устанавливает значение по индексу, заданному параметром *idx*, равным значению, переданному в параметре *v*
`public static SortedList Synchronized(SortedList sl)`	Возвращает синхронизированную версию `SortedList`-коллекции, переданной в параметре *sl*
`public virtual void TrimToSize()`	Устанавливает свойство `Capacity` равным значению свойства `Count`

Существуют различные способы установки и считывания ключа либо значения. Чтобы получить значение, связанное с заданным индексом, вызовите метод `GetByIndex()`, а чтобы установить значение, заданное индексом, — метод `SetByIndex()`. Получить ключ, связанный с заданным индексом, можно с помощью метода `GetKey()`. Для получения списка всех ключей используйте метод `GetKeyList()`, а для получения списка всех значений — метод `GetValueList()`. Получить индекс ключа можно с помощью метода `IndexOfKey()`, а индекс значения — с помощью метода `IndexOfValue()`. Класс `SortedList` также поддерживает индексатор, определенный интерфейсом `IDictionary`, благодаря чему можно устанавливать или считывать значение, заданное соответствующим ключом.

В классе `SortedList` помимо свойств, определенных в реализованных им интерфейсах, определены два собственных свойства. Получить предназначенную только для чтения коллекцию ключей или значений, хранимых в `SortedList`-коллекции, можно с помощью таких свойств:

```
public virtual ICollection Keys { get; }
public virtual ICollection Values { get; }
```

Порядок следования ключей и значений в полученных коллекциях отражает порядок `SortedList`-коллекции.

Подобно `Hashtable`-коллекции, `SortedList`-список хранит пары ключ/значение в форме структуры типа `DictionaryEntry`, но с помощью методов и свойств, определенных в классе `SortedList`, программисты обычно получают отдельный доступ к ключам и значениям.

Использование отсотированного списка типа `SortedList` демонстрируется в программе, которая представляет собой переработанную и расширенную версию программы из предыдущего раздела, демонстрировавшей `Hashtable`-коллекцию. Изучая результаты выполнения этой программы, обратите внимание на то, что `SortedList`-коллекция отсортирована по ключу.

```
// Демонстрация SortedList-коллекции.

using System;
using System.Collections;

class SLDemo {
  public static void Main() {
    // Создаем упорядоченную коллекцию типа SortedList.
    SortedList sl = new SortedList();
```

```
    // Добавляем в список элементы.
    sl.Add("здание", "жилое помещение");
    sl.Add("автомобиль", "транспортное средство");
    sl.Add("книга", "набор печатных слов");
    sl.Add("яблоко", "съедобный фрукт");

    // Добавлять элементы можно также с помощью индексатора.
    sl["трактор"] = "сельскохозяйственная машина";

    // Получаем коллекцию ключей.
    ICollection c = sl.Keys;

    // Используем ключи для получения значений.
    Console.WriteLine(
          "Содержимое списка, полученное с помощью " +
          "индексатора.");
    foreach(string str in c)
      Console.WriteLine(str + ": " + sl[str]);

    Console.WriteLine();

    // Отображаем список, используя целочисленные индексы.
    Console.WriteLine(
          "Содержимое списка, полученное с помощью " +
          "целочисленных индексов.");
    for(int i=0; i<sl.Count; i++)
      Console.WriteLine(sl.GetByIndex(i));

    Console.WriteLine();

    // Отображаем целочисленные индексы элементов списка.
    Console.WriteLine(
           "Целочисленные индексы элементов списка.");
    foreach(string str in c)
      Console.WriteLine(str + ": " + sl.IndexOfKey(str));
  }
}
```

Результаты выполнения этой программы таковы:

```
Содержимое списка, полученное с помощью индексатора.
автомобиль: транспортное средство
здание: жилое помещение
книга: набор печатных слов
трактор: сельскохозяйственная машина
яблоко: съедобный фрукт

Содержимое списка, полученное с помощью целочисленных индексов.
транспортное средство
жилое помещение
набор печатных слов
сельскохозяйственная машина
съедобный фрукт

Целочисленные индексы элементов списка.
автомобиль: 0
здание: 1
книга: 2
трактор: 3
яблоко: 4
```

Класс Stack

Вероятно, большинству читателей известно, что *стек* представляет собой список, добавление и удаление элементов к которому осуществляется по принципу "последним пришел — первым обслужен" (*l*ast-*i*n, *f*irst-*o*ut — LIFO). Чтобы понять, как работает стек, представьте себе груду тарелок на столе. Тарелку, поставленную на стол первой, можно будет взять лишь последней, т.е. когда будут сняты все поставленные сверху тарелки. Стек — наиболее востребованная структура данных в программировании. Ее часто используют в системном программном обеспечении, компиляторах и программах из области создания искусственного интеллекта (в частности, в сфере программирования с обратным слежением).

Класс коллекции, предназначенный для поддержки стека, называется `Stack`. Он реализует интерфейсы `ICollection`, `IEnumerable` и `ICloneable`. Стек — это динамическая коллекция, которая при необходимости увеличивается, чтобы принять для хранения новые элементы, причем каждый раз, когда стек должен расшириться, его емкость удваивается.

В классе `Stack` определены следующие конструкторы:

```
public Stack()
public Stack(int capacity)
public Stack(ICollection c)
```

Первый конструктор предназначен для создания пустого стека с начальной емкостью, равной 10 элементам. Второй создает пустой стек с начальной емкостью, заданной параметром *capacity*. Третий конструктор служит для построения стека, который инициализируется элементами и емкостью коллекции, заданной параметром *c*.

Помимо методов, определенных в интерфейсах, которые реализует класс `Stack`, в нем определены также собственные методы, перечисленные в табл. 22.7. По описанию этих методов можно судить о том, как используется стек. Чтобы поместить объект в вершину стека, вызовите метод `Push()`. Чтобы извлечь верхний элемент и удалить его из стека, используйте метод `Pop()`. Если при вызове метода `Pop()` окажется, что вызывающий стек пуст, генерируется исключение типа `InvalidOperationException`. Метод `Peek()` позволяет вернуть верхний элемент, не удаляя его из стека.

Таблица 22.7. Методы, определенные в классе `Stack`

Метод	Описание
`public virtual bool Contains(object v)`	Возвращает значение `true`, если объект *v* содержится в вызывающем стеке. В противном случае возвращает значение `false`
`public virtual void Clear()`	Устанавливает свойство `Count` равным нулю, тем самым эффективно очищая стек
`public virtual object Peek()`	Возвращает элемент, расположенный в вершине стека, но не удаляет его
`public virtual object Pop()`	Возвращает элемент, расположенный в вершине стека, и удаляет его
`public virtual void Push(object v)`	Помещает объект *v* в стек
`public static Stack Synchronized(Stack stk)`	Возвращает синхронизированную версию `Stack`-списка, переданного в параметре *stk*
`public virtual object[] ToArray()`	Возвращает массив, который содержит копии элементов вызывающего стека

Рассмотрим пример создания стека и его использования: поместим в него несколько объектов класса `Integer`, а затем извлечем их.

```
// Демонстрация использования класса Stack.

using System;
using System.Collections;

class StackDemo {
  static void showPush(Stack st, int a) {
    st.Push(a);
    Console.WriteLine(
            "Помещаем в элемент стек: Push(" + a + ")");

    Console.Write("Содержимое стека: ");
    foreach(int i in st)
      Console.Write(i + " ");

    Console.WriteLine();
  }

  static void showPop(Stack st) {
    Console.Write("Извлекаем элемент из стека: Pop -> ");
    int a = (int) st.Pop();
    Console.WriteLine(a);

    Console.Write("Содержимое стека: ");
    foreach(int i in st)
      Console.Write(i + " ");

    Console.WriteLine();
  }

  public static void Main() {
    Stack st = new Stack();

    foreach(int i in st)
      Console.Write(i + " ");

    Console.WriteLine();

    showPush(st, 22);
    showPush(st, 65);
    showPush(st, 91);
    showPop(st);
    showPop(st);
    showPop(st);

    try {
      showPop(st);
    } catch (InvalidOperationException) {
      Console.WriteLine("Стек пуст.");
    }
  }
}
```

Ниже приведены результаты выполнения этой программы. Обратите внимание на то, как обрабатывается исключительная ситуация (исключение типа `InvalidOperationException`), возникающая при попытке извлечь элемент из пустого стека (эта ситуация называется *незагруженностью стека*).

```
Помещаем элемент в стек: Push(22)
Содержимое стека: 22
Помещаем элемент в стек: Push(65)
Содержимое стека: 65 22
Помещаем элемент в стек: Push(91)
Содержимое стека: 91 65 22
Извлекаем элемент из стека: Pop -> 91
Содержимое стека: 65 22
Извлекаем элемент из стека: Pop -> 65
Содержимое стека: 22
Извлекаем элемент из стека: Pop -> 22
Содержимое стека:
Извлекаем элемент из стека: Pop -> Стек пуст.
```

Класс Queue

Еще одной распространенной структурой данных является *очередь*. Добавление элементов в очередь и удаление их из нее осуществляется по принципу "первым пришел — первым обслужен" (first-in, first-out — FIFO). Другими словами, первый элемент, помещенный в очередь, первым же из нее и извлекается. Ну кто не сталкивался с очередями в реальной жизни? Например, каждому приходилось, вероятно, стоять в очереди к билетной кассе в кинотеатре или к кассе в супермаркете, чтобы оплатить покупку. В программировании очереди используются для организации таких механизмов, как выполнение нескольких процессов в системе и поддержка списка незаконченных транзакций (в системах ведения баз данных) или пакетов данных, полученных из Internet. Очереди также часто используются в области имитационного моделирования.

Класс коллекции, предназначенный для поддержки очереди, называется `Queue`. Он реализует интерфейсы `ICollection`, `IEnumerable` и `ICloneable`. Очередь — это динамическая коллекция, которая при необходимости увеличивается, чтобы принять для хранения новые элементы, причем каждый раз, когда такая необходимость возникает, текущий размер очереди умножается на коэффициент роста, который по умолчанию равен значению 2,0.

В классе `Queue` определены следующие конструкторы:

```
public Queue()
public Queue (int capacity)
public Queue (int capacity, float growFact)
public Queue (ICollection c)
```

Первый конструктор предназначен для создания пустой очереди с начальной емкостью, равной 32 элементам, и коэффициентом роста 2,0. Второй создает пустую очередь с начальной емкостью, заданной параметром *capacity*, и коэффициентом роста 2,0. Третий отличается от второго тем, что позволяет задать коэффициент роста посредством параметра *growFact*. Четвертый конструктор служит для создания очереди, которая инициализируется элементами и емкостью коллекции, заданной параметром *c*.

Помимо методов, определенных в интерфейсах, которые реализует класс `Queue`, в нем определены также собственные методы, перечисленные в табл. 22.8. О работе очереди можно получить представление по описанию этих методов. Чтобы поместить объект в очередь, вызовите метод `Enqueue()`. Чтобы извлечь верхний элемент и удалить его из очереди, используйте метод `Dequeue()`. Если при вызове метода `Dequeue()` окажется, что вызывающая очередь пуста, генерируется исключение типа `InvalidOperationException`. Метод `Peek()` позволяет вернуть следующий объект, не удаляя его из очереди.

Таблица 22.8. Методы, определенные в классе Queue

Метод	Описание
`public virtual bool Contains(object v)`	Возвращает значение `true`, если объект *v* содержится в вызывающей очереди. В противном случае возвращает значение `false`
`public virtual void Clear()`	Устанавливает свойство `Count` равным нулю, тем самым эффективно очищая очередь
`public virtual object Dequeue()`	Возвращает объект из начала вызывающей очереди. Возвращаемый объект из очереди удаляется
`public virtual void Enqueue(object v)`	Добавляет объект *v* в конец очереди
`public virtual object Peek()`	Возвращает объект из начала вызывающей очереди, но не удаляет его
`public static Queue Synchronized(Queue q)`	Возвращает синхронизированную версию очереди, заданной параметром *q*
`public virtual object[] ToArray()`	Возвращает массив, который содержит копии элементов из вызывающей очереди
`public virtual void TrimToSize()`	Устанавливает свойство `Capacity` равным значению свойства `Count`

Рассмотрим пример, в котором демонстрируется использование класса `Queue`:

```
// Демонстрация класса Queue.

using System;
using System.Collections;

class QueueDemo {
  static void showEnq(Queue q, int a) {
    q.Enqueue(a);
    Console.WriteLine(
         "Помещаем элемент в очередь: Enqueue(" + a + ")");

    Console.Write("Содержимое очереди: ");
    foreach(int i in q)
      Console.Write(i + " ");

    Console.WriteLine();
  }

  static void showDeq(Queue q) {
    Console.Write(
              "Извлекаем элемент из очереди: Dequeue -> ");
    int a = (int) q.Dequeue();
    Console.WriteLine(a);

    Console.Write("Содержимое очереди: ");
    foreach(int i in q)
      Console.Write(i + " ");

    Console.WriteLine();
  }

  public static void Main() {
    Queue q = new Queue();

    foreach(int i in q)
```

```
      Console.Write(i + " ");

    Console.WriteLine();

    showEnq(q, 22);
    showEnq(q, 65);
    showEnq(q, 91);
    showDeq(q);
    showDeq(q);
    showDeq(q);

    try {
      showDeq(q);
    } catch (InvalidOperationException) {
      Console.WriteLine("Очередь пуста.");
    }
  }
}
```

Результаты выполнения программы таковы:

```
Помещаем элемент в очередь: Enqueue(22)
Содержимое очереди: 22
Помещаем элемент в очередь: Enqueue(65)
Содержимое очереди: 22 65
Помещаем элемент в очередь: Enqueue(91)
Содержимое очереди: 22 65 91
Извлекаем элемент из очереди: Dequeue -> 22
Содержимое очереди: 65 91
Извлекаем элемент из очереди: Dequeue -> 65
Содержимое очереди: 91
Извлекаем элемент из очереди: Dequeue -> 91
Содержимое очереди:
Извлекаем элемент из очереди: Dequeue -> Очередь пуста.
```

Хранение битов с помощью класса `BitArray`

Класс `BitArray` предназначен для поддержки коллекции битов. Поскольку его назначение состоит в хранении битов, а не объектов, то и его возможности отличаются от возможностей других коллекций. Тем не менее класс `BitArray` поддерживает базовый набор средств коллекции посредством реализации интерфейсов `ICollection`, `IEnumerable` и `ICloneable`.

В классе `BitArray` определено множество конструкторов. Например, `BitArray`-коллекцию можно создать из массива булевых значений, используя этот конструктор:

```
public BitArray(bool[] bits)
```

В этом случае каждый элемент массива *bits* становится битом `BitArray`-коллекции. При этом каждый бит в коллекции соответствует элементу массива *bits*. Более того, порядок элементов массива *bits* аналогичен порядку битов в коллекции.

`BitArray`-коллекцию можно также создать из массива байтов. Для этого используйте следующий конструктор:

```
public BitArray(byte[] bits)
```

Здесь битами коллекции становится набор битов, содержащийся в массиве *bits*, причем элемент *bits*`[0]` определяет первые восемь битов, элемент *bits*`[1]` — вто-

рые восемь битов и т.д. Подобным образом можно создать BitArray-коллекцию из массива int-значений, используя следующий конструктор:

```
public BitArray(int[] bits)
```

В данном случае элемент *bits*[0] определяет первые 32 бита, элемент *bits*[1] — вторые 32 бита и т.д.

С помощью этого конструктора можно создать BitArray-коллекцию заданного размера:

```
public BitArray(int size)
```

Здесь параметр *size* задает количество битов в коллекции, причем все они инициализируются значением false.

Чтобы задать размер создаваемой коллекции и начальное значение ее битов, используйте следующий конструктор:

```
public BitArray(int size, bool v)
```

Здесь все биты в коллекции будут установлены равными значению, переданному в параметре *v*.

Наконец, новую BitArray-коллекцию можно создать из уже существующей. Для этого достаточно обратиться к конструктору

```
public BitArray(BitArray bits)
```

Новый объект будет содержать такую же коллекцию битов, как у представленой параметром *bits*, но эти две коллекции изолированы.

BitArray-коллекции могут быть индексированными. Каждый индекс соответствует определенному биту, причем нулевой индекс соответствует младшему биту.

В классе BitArray помимо методов, определенных в реализованных им интерфейсах, определены также собственные методы (см. табл. 22.9). Обратите внимание на то, что класс BitArray не поддерживает метод Synchronized(). Это означает, что для данного типа коллекции синхронизированную оболочку получить нельзя, и поэтому свойство IsSynchronized всегда равно значению false. Тем не менее доступом к BitArray-коллекции можно управлять посредством синхронизации объекта, возвращаемого свойством SyncRoot.

Таблица 22.9. Методы, определенные в классе BitArray

Можететод	*Описание*
`public BitArray And (BitArray ba)`	Выполняет операцию логического умножения (И) между соответствующими битами вызывающего объекта и битами коллекции, заданной параметром *ba*. Возвращает BitArray-коллекцию, содержащую результат
`public bool Get(int idx)`	Возвращает значение бита, индекс которого задан параметром *idx*
`public BitArray Not()`	Выполняет операцию поразрядного логического отрицания (НЕ) на битах вызывающей коллекции и возвращает BitArray-коллекцию, содержащую результат
`public BitArray Or(BitArray ba)`	Выполняет операцию логического сложения (ИЛИ) между соответствующими битами вызывающего объекта и битами коллекции, заданной параметром *ba*. Возвращает BitArray-коллекцию, содержащую результат
`Public void Set(int idx, bool v)`	Устанавливает бит, индекс которого задан параметром *idx*, равным значению параметра *v*
`public void SetAll(bool v)`	Устанавливает все биты равными значению *v*
`public BitArray Xor(BitArray ba)`	Выполняет операцию исключающего ИЛИ между соответствующими битами вызывающего объекта и битами коллекции, заданной параметром *ba*. Возвращает BitArray-коллекцию, содержащую результат

В классе BitArray помимо свойств, определенных в реализованных им интерфейсах, определено также собственное свойство Length:

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

Свойство Length устанавливает или возвращает текущее количество битов в коллекции. Следовательно, свойство Length содержит то же значение, что и стандартное свойство Count, которое определено для всех коллекций. Однако свойство Count предназначено только для чтения, чего не скажешь о свойстве Length. Таким образом, свойство Length можно использовать для изменения размера BitArray-коллекции. Если вы ее уменьшаете, "лишние" биты усекаются со стороны старших разрядов. Если же BitArray-коллекцию увеличивать, то со стороны старших разрядов добавляются биты, имеющие значение false.

В классе BitArray определен следующий индексатор:

```
object this[int idx] { get; set; }
```

Этот индексатор можно использовать для считывания или установки значения заданного элемента коллекции.

Рассмотрим программу, демонстрирующую использование BitArray-коллекции:

```
// Демонстрация использования класса BitArray.

using System;
using System.Collections;

class BADemo {
  public static void showbits(string rem,
                              BitArray bits) {
    Console.WriteLine(rem);
    for(int i=0; i < bits.Count; i++)
      Console.Write("{0, -6} ", bits[i]);
    Console.WriteLine("\n");
  }

  public static void Main() {
    BitArray ba = new BitArray(8);
    byte[] b = { 67 };
    BitArray ba2 = new BitArray(b);

    showbits("Исходное содержимое битовой коллекции ba:",
             ba);

    ba = ba.Not();

    showbits(
      "Содержимое коллекции ba после вызова метода Not():",
      ba);

    showbits("Содержимое коллекции ba2:", ba2);

    BitArray ba3 = ba.Xor(ba2);

    showbits("Результат операции ba XOR ba2:", ba3);
  }
}
```

Результаты выполнения этой программы таковы:

```
Исходное содержимое битовой коллекции ba:
False  False  False  False  False  False  False  False
```

```
Содержимое коллекции ba после вызова метода Not():
True   True   True   True   True   True   True   True

Содержимое коллекции ba2:
True   True   False  False  False  False  True   False

Результат операции ba XOR ba2:
False  False  True   True   True   True   False  True
```

Специализированные коллекции

В среде NET Framework предусмотрена возможность создания специализированных коллекций, которые оптимизированы для работы с конкретными типами данных или для особого вида обработки Эти классы коллекций (они определены в пространстве имен `System.Collections.Specialized`) перечислены в следующей таблице

Специализированная коллекция	*Описание*
`CollectionsUtil`	Коллекция, в которой игнорируются различия между строчным и прописным написанием символов в строках
`HybridDictionary`	Коллекция, в которой для хранения небольшого числа пар ключ/значение используется класс `ListDictionary` Но при превышении коллекцией определенного размера для хранения элементов автоматически используется класс `Hashtable`
`ListDictionary`	Коллекция, в которой для хранения пар ключ/значение используется связный список Такую коллекцию рекомендуется использовать лишь при небольшом количестве элементов
`NameValueCollection`	Отсортированная коллекция пар ключ/значение, в которой как ключ, так и значение имеют тип `string`
`StringCollection`	Коллекция, оптимизированная для хранения строк
`StringDictionary`	Хеш-таблица предназначенная для хранения пар ключ/значение, в которой как ключ, так и значение имеют тип `string`

В пространстве имен `System.Collections` также определены три абстрактных базовых класса, `CollectionBase`, `ReadOnlyCollectionBase` и `DictionaryBase`, которые предполагают создание производных классов и предназначены для использования в качестве отправной точки при разработке программистом собственных специализированных классов

Доступ к коллекциям с помощью нумератора

Часто при работе с коллекциями возникает необходимость в циклическом опросе ее элементов Например, нужно отобразить все элементы коллекции Один из способов достижения этого — использование цикла `foreach`, что и было продемонстрировано в предыдущем примере Еще один способ — использование нумератора Нумератор представляет собой объект, который реализует интерфейс `IEnumerator`

В интерфейсе `IEnumerator` определено единственное свойство `Current`

```
object Current { get; }
```

Свойство Current позволяет получить элемент, соответствующий текущему значению нумератора. Поскольку свойство Current предназначено только для чтения, нумератор можно использовать только для считывания значения объекта в коллекции, а не для его модификации.

В интерфейсе IEnumerator определены два метода. Первый, MoveNext(), объявляется так:

```
bool MoveNext()
```

При каждом обращении к методу MoveNext() текущая позиция нумератора перемещается к следующему элементу коллекции. Метод возвращает значение true, если к следующему элементу можно получить доступ, или значение false, если достигнут конец коллекции. До выполнения первого обращения к методу MoveNext() значение свойства Current неопределено.

Установить нумератор в начало коллекции можно с помощью метода Reset():

```
void Reset()
```

После вызова метода Reset() нумерация элементов начнется с начала коллекции, и для доступа к первому ее элементу необходимо вызвать метод MoveNext().

Использование нумератора

Чтобы получить доступ к коллекции с помощью нумератора, нужно сначала его реализовать. В каждом классе коллекции предусмотрен метод GetEnumerator(), который возвращает нумератор, устанавливая его в начало коллекции. Используя этот метод, можно поэлементо опросить всю коллекцию. Чтобы использовать нумератор для циклического обхода коллекции, выполните такие действия:

1. Получите нумератор (с установой его в начаю коллекции), вызвав метод GetEnumerator().
2. Организуйте цикл, в котором вызывается метод MoveNext(). Позаботьтесь о том, чтобы цикл работал до тех пор, пока метод MoveNext() возвращает значение true.
3. На каждой итерации цикла опросите соответствующий элемент коллекции с помощью свойства Current.

В представленной ниже программе используется класс ArrayList, но продемонстрированные здесь принципы работы с динамическим массивом применимы к любому типу коллекции.

```
// Демонстрация использования нумератора.

using System;
using System.Collections;

class EnumeratorDemo {
  public static void Main() {
    ArrayList list = new ArrayList(1);

    for(int i=0; i < 10; i++)
      list.Add(i);

    // Используем нумератор для доступа к списку.
    IEnumerator etr = list.GetEnumerator();
    while(etr.MoveNext())
      Console.Write(etr.Current + " ");
```

```
    Console.WriteLine();

    // Устанавливаем позицию нумератора в начало коллекции.
    etr.Reset();
    while(etr.MoveNext())
      Console.Write(etr.Current + " ");

    Console.WriteLine();
  }
}
```

Результаты выполнения программы выглядят так:

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

В общем случае, если нужно в цикле опросить содержимое коллекции, цикл `foreach` — более удобное средство, чем нумератор. Однако нумератор при желании позволяет "одним махом" вернуться в начало коллекции.

Использование интерфейса `IDictionaryEnumerator`

Класс коллекции, который реализует интерфейс `IDictionary`, предназначен для хранения пар ключ/значение. Для опроса элементов в такой коллекции используется интерфейс `IDictionaryEnumerator`, а не `IEnumerator`. Класс `IDictionaryEnumerator` является производным от класса `IEnumerator` и дополнительно определяет "свои" три свойства. Первое объявляется так:

```
DictionaryEntry Entry { get; }
```

Свойство `Entry` с помощью нумератора позволяет получить следующую пару ключ/значение в форме структуры типа `DictionaryEntry`. Вспомните, что в структуре `DictionaryEntry` определено два свойства, `Key` и `Value`, которые можно использовать для доступа к ключу или значению, относящемуся к соответствующему элементу. Вот как объявлены два остальные свойства интерфейса `IDictionaryEnumerator`:

```
object Key { get; }
object Value { get; }
```

Нетрудно догадаться, что эти свойства позволяют получить прямой доступ к ключу и значению.

Интерфейс `IDictionaryEnumerator` используется подобно обычному нумератору за исключением того, что текущие значения элементов здесь можно получить с помощью свойств `Entry`, `Key` или `Value`, а не свойства `Current`. Таким образом, реализовав `IDictionaryEnumerator`-нумератор, вы должны вызвать метод `MoveNext()`, чтобы получить первый элемент. Остальные элементы коллекции опрашиваются путем последующих вызовов метода `MoveNext()`. Когда все элементы коллекции будут исчерпаны, метод `MoveNext()` возвратит значение `false`.

Теперь самое время рассмотреть пример использования нумератора типа `IDictionaryEnumerator` для опроса элементов коллекции типа `Hashtable`:

```
// Демонстрация использования нумератора
// типа IDictionaryEnumerator.

using System;
using System.Collections;

class IDicEnumDemo {
  public static void Main() {
    // Создаем хеш-таблицу.
```

```
    Hashtable ht = new Hashtable();

    // Добавляем элементы в хеш-таблицу.
    ht.Add("Анатолий", "555-3456");
    ht.Add("Марина", "555-9876");
    ht.Add("Александр", "555-3452");
    ht.Add("Кирилл", "555-7756");

    // Демонстрируем работу нумератора.
    IDictionaryEnumerator etr = ht.GetEnumerator();
    Console.WriteLine(
       "Отображаем информацию с помощью свойства Entry.");
    while(etr.MoveNext())
     Console.WriteLine(etr.Entry.Key + ": " +
                       etr.Entry.Value);

    Console.WriteLine();

    Console.WriteLine("Отображаем информацию с помощью " +
                      "свойств Key и Value.");
    etr.Reset();
    while(etr.MoveNext())
     Console.WriteLine(etr.Key + ": " +
                       etr.Value);

  }
}
```

Вот результаты выполнения этой программы:

```
Отображаем информацию с помощью свойства Entry.
Анатолий: 555-3456
Марина: 555-9876
Александр: 555-3452
Кирилл: 555-7756

Отображаем информацию с помощью свойств Key и Value.
Анатолий: 555-3456
Марина: 555-9876
Александр: 555-3452
Кирилл: 555-7756
```

Хранение в коллекциях классов, определенных пользователем

Для простоты в предыдущих примерах демонстрировалось хранение в коллекциях значений таких встроенных типов, как `int`, `string` или `char`. Безусловно, на коллекции такое ограничение не налагается. А совсем даже наоборот. Мощь коллекций как раз и состоит в том, что они могут хранить объекты любого типа, включая объекты классов, создаваемых программистами. Рассмотрим, например, программу, в которой для хранения информации о запасах товаров на складе, инкапсулированной классом `Inventory`, используется класс `ArrayList`.

```
// Простой пример на тему инвентаризации.

using System;
```

```
using System.Collections;

class Inventory {
  string name;
  double cost;
  int onhand;

  public Inventory(string n, double c, int h) {
    name = n;
    cost = c;
    onhand = h;
  }

  public override string ToString() {
    return
      String.Format("{0,-16}Цена: {1,8:C}  В наличии: {2}",
                    name, cost, onhand);
  }
}

class InventoryList {
  public static void Main() {
    ArrayList inv = new ArrayList();

    // Добавляем элементы в список.
    inv.Add(new Inventory("Плоскогубцы", 5.95, 3));
    inv.Add(new Inventory("Гаечные ключи", 8.29, 2));
    inv.Add(new Inventory("Молотки", 3.50, 4));
    inv.Add(new Inventory("Сверла", 19.88, 8));

    Console.WriteLine("Информация о запасах на складе:");
    foreach(Inventory i in inv) {
      Console.WriteLine("   " + i);
    }
  }
}
```

Вот результаты выполнения программы:

```
Информация о запасах на складе:
   Плоскогубцы      Цена:     $5.95  В наличии: 3
   Гаечные ключи    Цена:     $8.29  В наличии: 2
   Молотки          Цена:     $3.50  В наличии: 4
   Сверла           Цена:    $19.88  В наличии: 8
```

В этой программе для хранения в коллекции объектов типа `Inventory` не нужно было предпринимать специальных мер. Поскольку все типы являются производными от класса `object`, в любой коллекции общего назначения можно сохранить объект любого типа.

Вероятно, вы заметили, что рассматриваемая программа довольно невелика. Если к тому же учесть, что в ней создается динамический массив, который способен хранить, считывать и обрабатывать информацию о содержимом склада, и на все это потребовалось меньше 40 строк кода, то могущество коллекций начинает не просто проявляться, а становится очевидным. Вероятно, многим читателям понятно, что если этот механизм ведения складского хозяйства закодировать вручную, то программа была бы в несколько раз больше. Поэтому подчеркнем еще раз, что коллекции предлагают готовые решения для широкого круга задач.

Все же справедливости ради отметим, что предыдущая программа имеет одно ограничение, которое, возможно, сразу и не бросается в глаза: эта коллекция не поддается сортировке. Дело в том, что наша ArrayList-коллекция не "знает", как сравнивать два объекта типа Inventory. Есть два способа исправить ситуацию. Во-первых, в классе Inventory можно реализовать интерфейс IComparable. Именно этот интерфейс определяет, как сравниваются два объекта класса. Во-вторых, когда потребуется выполнить операцию сравнения, можно определить объект типа IComparer. Оба упомянутых варианта иллюстрируются в следующих разделах.

Реализация интерфейса IComparable

Если необходимо отсортировать динамический массив (типа ArrayList) объектов, определенных пользователем (или если вам понадобится сохранить эти объекты в коллекции типа SortedList), то вы должны сообщить коллекции информацию о том, как сравнивать эти объекты. Один из способов — реализовать интерфейс IComparable. В этом интерфейсе определен только один метод CompareTo(), который позволяет определить, как должно выполняться сравнение объектов соответствующего типа. Общий формат использования метода CompareTo() таков:

```
int CompareTo(object obj)
```

Метод CompareTo() сравнивает вызывающий объект с объектом, заданным параметром *obj*. Чтобы отсортировать объекты коллекции в возрастающем порядке, этот метод (в вашей реализации) должен возвращать нуль, если сравниваемые объекты равны, положительное значение, если вызывающий объект больше объекта *obj*, и отрицательное число, если вызывающий объект меньше объекта *obj*. Для сортировки в убывающем порядке достаточно инвертировать результат описанного сравнения. Метод CompareTo() может сгенерировать исключение типа ArgumentException, если тип объекта *obj* несовместим с вызывающим объектом.

Рассмотрим пример реализации интерфейса IComparable. Здесь в класс Inventory, разработанный в предыдущем разделе, добавляется интерфейс IComparable, что позволяет эффективно отсортировать коллекцию объектов типа Inventory.

```
// Реализация интерфейса IComparable.

using System;
using System.Collections;

class Inventory : IComparable {
  string name;
  double cost;
  int onhand;

  public Inventory(string n, double c, int h) {
    name = n;
    cost = c;
    onhand = h;
  }

  public override string ToString() {
    return
      String.Format("{0,-16}Цена: {1,8:C}  В наличии: {2}",
                    name, cost, onhand);
  }
```

```
  // Реализуем интерфейс IComparable.
  public int CompareTo(object obj) {
    Inventory b;
    b = (Inventory) obj;
    return name.CompareTo(b.name);
  }
}

class IComparableDemo {
  public static void Main() {
    ArrayList inv = new ArrayList();

    // Добавляем элементы в список.
    inv.Add(new Inventory("Плоскогубцы", 5.95, 3));
    inv.Add(new Inventory("Гаечные ключи", 8.29, 2));
    inv.Add(new Inventory("Молотки", 3.50, 4));
    inv.Add(new Inventory("Сверла", 19.88, 8));

    Console.WriteLine(
        "Информация о запасах на складе до сортировки:");
    foreach(Inventory i in inv) {
      Console.WriteLine("   " + i);
    }
    Console.WriteLine();

    // Сртируем список.
    inv.Sort();

    Console.WriteLine(
      "Информация о запасах на складе после сортировки:");
    foreach(Inventory i in inv) {
      Console.WriteLine("   " + i);
    }
  }
}
```

А вот и результаты выполнения этого варианта программы (обратите внимание на то, что после вызова метода `Sort()` список товаров отсортирован по наименованию):

```
Информация о запасах на складе до сортировки:
   Плоскогубцы      Цена:     $5.95  В наличии: 3
   Гаечные ключи    Цена:     $8.29  В наличии: 2
   Молотки          Цена:     $3.50  В наличии: 4
   Сверла           Цена:    $19.88  В наличии: 8

Информация о запасах на складе после сортировки:
   Гаечные ключи    Цена:     $8.29  В наличии: 2
   Молотки          Цена:     $3.50  В наличии: 4
   Плоскогубцы      Цена:     $5.95  В наличии: 3
   Сверла           Цена:    $19.88  В наличии: 8
```

Использование интерфейса `IComparer`

Несмотря на то что реализация интерфейса `IComparable` для создаваемых программистом классов — самый простой способ решить проблему сортировки объектов (этих классов) в коллекции, возможен и другой подход к ее решению. Он состоит в использовании интерфейса `IComparer`. В этом случае необходимо сначала создать класс, который реализует интерфейс `IComparer`, а затем задать объект этого класса

именно тогда, когда требуется выполнить сравнение. В интерфейсе IComparer определен только один метод Compare():

```
int Compare(object obj1, object obj2)
```

Метод Compare() сравнивает объект *obj1* с объектом *obj2*. Для сортировки в возрастающем порядке метод Compare() должен возвратить нуль, если сравниваемые объекты равны, положительное значение, если объект *obj1* больше объекта *obj2*, и отрицательное число, если объект *obj1* меньше объекта *obj2*. Для сортировки в убывающем порядке достаточно инвертировать результат описанного сравнения. Метод CompareTo() может сгенерировать исключение типа ArgumentException, если типы сравниваемых объектов несовместимы.

Интерфейс IComparer можно определить при создании объекта коллекции типа SortedList, а именно при вызове метода ArrayList.Sort(IComparer), а также в других случаях определения классов коллекций. Основное достоинство использования интерфейса IComparer состоит в том, что он позволяет сортировать объекты классов, которые не реализуют интерфейс IComparable.

Следующая программа представляет собой еще одну вариацию на тему инвентаризации склада. На этот раз для сортировки списка товаров используется интерфейс IComparer. Сначала здесь создается класс CompInv, который реализует интерфейс IComparer и обеспечивает возможность сравнения двух объектов типа Inventory. Затем каждый объект этого класса используется в вызове метода Sort(), который, собственно, и сортирует список товаров.

```
// Использование интерфейса IComparer.

using System;
using System.Collections;

// Создаем класс CompInv.
// для объектов класса Inventory.
class CompInv : IComparer {

  // Реализуем интерфейс IComparer.
  public int Compare(object obj1, object obj2) {
    Inventory a, b;
    a = (Inventory) obj1;
    b = (Inventory) obj2;
    return a.name.CompareTo(b.name);
  }
}

class Inventory {
  public string name;
  double cost;
  int onhand;

  public Inventory(string n, double c, int h) {
    name = n;
    cost = c;
    onhand = h;
  }

  public override string ToString() {
    return
      String.Format("{0,-16}Цена: {1,8:C}  В наличии: {2}",
                    name, cost, onhand);
```

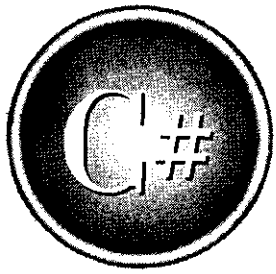
```
  }
}

class MailList {
  public static void Main() {
    CompInv comp = new CompInv();
    ArrayList inv = new ArrayList();

    // Добавляем элементы в список.
    inv.Add(new Inventory(Плоскогубцы", 5.95, 3));
    inv.Add(new Inventory("Гаечные ключи", 8.29, 2));
    inv.Add(new Inventory("Молотки", 3.50, 4));
    inv.Add(new Inventory("Сверла", 19.88, 8));

    Console.WriteLine(
        "Информация о запасах на складе до сортировки:");
    foreach(Inventory i in inv) {
      Console.WriteLine("   " + i);
    }
    Console.WriteLine();

    // Сортируем список, используя интерфейс IComparer.
    inv.Sort(comp);

    Console.WriteLine(
      "Информация о запасах на складе после сортировки:");
    foreach(Inventory i in inv) {
      Console.WriteLine("   " + i);
    }
  }
}
```

Результаты выполнения этой программы совпадают с предыдущими.

Резюме

Коллекции вооружают программиста мощным набором удачных решений, подходящих для множества распространенных задач программирования. Когда потребуется обеспечить эффективное хранение и считывание информации, непременно вспомните о коллекциях. Не следует оставлять коллекции лишь для таких "глобальных задач", как создание корпоративных баз данных, списков рассылки или систем управления складом. Они не менее эффективны в применении к более мелким проблемам. Например, коллекция типа `SortedList` прекрасно подошла бы для ведения списка ежедневных деловых встреч. Сферу применения коллекций могут ограничить лишь возможности вашего воображения.

Полный справочник по



Глава 23

Сетевые возможности и использование Internet

Язык C# ориентирован на использование в современной вычислительной среде, в которой Internet играет немаловажную роль. Поэтому неудивительно, что одним из основных критериев разработки C#-средств являлось обеспечение возможностей использования Internet. Несмотря на то что и такие языки программирования, как C и C++, позволяют подключаться к Internet, загружать файлы и получать желаемые ресурсы, этот процесс нельзя назвать простым и удобным, каким хотели бы его видеть многие программисты. Стандартные средства C# и библиотека .NET Library значительно улучшили эту ситуацию, существенно облегчив программистам создание Internet-приложений.

Поддержка сетевых возможностей реализована в двух пространствах имен. В первом, `System.Net`, определено множество высокоуровневых и удобных для использования классов, в которых предусмотрены различные операции для работы с Internet. Второе пространство имен, `System.Net.Sockets`, предназначено для поддержки сокетов, которые обеспечивают низкоуровневое управление сетевыми операциями. В этой главе мы используем классы пространства имен `System.Net`, которые более удобны в применении, и к тому же для большинства приложений их вполне достаточно.

На заметку

Поддержка серверных ASP.NET-ориентированных приложений реализована в пространстве имен `System.Web`.

Члены пространства имен `System.Net`

`System.Net` — обширное пространство имен, содержащее множество членов. И хотя здесь будут рассмотрены только некоторые из них, вы должны иметь представление о том, что вообще доступно программисту в этой области. Итак, вот классы, определенные в пространстве имен `System.Net`:

`AuthenticationManager`	`Authorization`	`Cookie`
`CookieCollection`	`CookieContainer`	`CookieException`
`CredentialCache`	`Dns`	`DnsPermission`
`DnsPermissionAttribute`	`EndPoint`	`EndpointPermission`
`FileWebRequest`	`FileWebResponse`	`GlobalProxySelection`
`HttpVersion`	`HttpWebRequest`	`HttpWebResponse`
`IPAddress`	`IPEndPoint`	`IPHostEntry`
`NetworkCredential`	`ProtocolViolationException`	`ServicePoint`
`ServicePointManager`	`SocketAddress`	`SocketPermission`
`SocketPermissionAttribute`	`WebClient`	`WebException`
`WebHeaderCollection`	`WebPermission`	`WebPermissionAttribute`
`WebProxy`	`WebRequest`	`WebResponse`

В пространстве имен `System.Net` определены также следующие интерфейсы:

`IAuthenticationModule`	`ICertificatePolicy`	`ICredentials`
`IWebProxy`	`IWebRequestCreate`	

и четыре перечисления:

`HttpStatusCode`	`NetworkAccess`
`TransportType`	`WebExceptionStatus`

Наконец, в пространстве имен `System.Net` определен один делегат: `HttpContinueDelegate`.

Несмотря на то что в пространстве имен `System.Net` определено так много членов, для решения большинства задач Internet-программирования достаточно ограничиться лишь некоторыми из них. Ядро сетевых программных средств составляют два абстрактных класса `WebRequest` и `WebResponse`. Эти классы наследуются классами, которые поддерживают конкретные сетевые протоколы. (*Протокол* определяет правила, используемые для передачи информации по сети.) Например, производные классы, предназначенные для поддержки стандартного протокола HTTP, имеют имена `HttpWebRequest` и `HttpWebResponse`.

Несмотря на то что классы `HttpWebRequest` и `HttpWebResponse` просты в применении, для некоторых задач можно воспользоваться даже еще более простым средством, а именно классом `WebClient`. Например, если вам нужно лишь загрузить в удаленный компьютер (или из него) некоторый файл, то в большинстве случаев лучше всего для этого использовать именно класс `WebClient`.

Универсальные идентификаторы ресурсов

В основе Internet-программирования лежит универсальный идентификатор ресурса (Uniform Resource Identifier — URI). URI-идентификатор описывает местоположение в сети некоторого ресурса. URI-идентификатор часто называют также унифицированным указателем информационного ресурса (Uniform Resource Locator — URL). Поскольку при описании членов пространства имен `System.Net` компания Microsoft использует термин URI, то в данной книге используется именно этот термин. Вероятно, читатель хорошо знаком с URI-идентификаторами, если ему приходилось вводить в Internet-броузер адрес интересующего его сайта.

URI-идентификатор записывается в следующей общей форме:

`Protocol://ServerID/FilePath?Query`

Элемент *`Protocol`* означает используемый протокол (например, HTTP). Элемент *`ServerID`* идентифицирует конкретный сервер (например, `Osborne.com` или `Weather.com`). Элемент *`FilePath`* задает маршрут к нужному файлу. Если элемент *`FilePath`* не задан, открывается страница, действующая по умолчанию для указанного значения *`ServerID`*. Наконец, элемент *`Query`* содержит информацию, которая должна быть послана серверу. Элемент *`Query`* необязателен. В языке C# URI-идентификаторы инкапсулированы в классе `Uri`, который рассматривается позже в этой главе.

Основы Internet-доступа

Классы, содержащиеся в пространстве имен `System.Net`, при взаимодействии с Internet поддерживают модель "запрос-ответ". Это означает, что программа-клиент запрашивает информацию с сервера, а затем ожидает ответ. Например, ваша программа может отправить серверу конкретный URI-идентификатор или адрес Web-сайта. В ответ вы получите гипертекст, связанный с указанным URI-идентификатором. Этот метод запроса-ответа удобен и прост для использования, поскольку большинство деталей обрабатываются автоматически.

Иерархии классов, в вершине которых стоят классы `WebRequest` и `WebResponse`, реализуют то, что Microsoft называет *сменными протоколами* (pluggable protocols). Как

известно большинству читателей, существуют различные типы протоколов сетевой связи. Наиболее распространенным для Internet-взаимодействия является *протокол передачи гипертекстовых файлов* (HyperText Transfer Protocol — HTTP). Часто используется и *протокол передачи файлов* (File Transfer Protocol — FTP). Напомню, что префикс URI-идентификатора задает именно протокол. Например, в таком URI-идентификаторе, как `HTTP://MyWebSite.com`, используется префикс `HTTP`, который означает протокол передачи гипертекстовых файлов.

Как упоминалось выше, `WebRequest` и `WebResponse` — абстрактные классы, в которых определяются операции запроса-ответа, общие для всех протоколов. Из этих абстрактных классов выведены классы, которые реализуют конкретные протоколы. Производные классы регистрируются самостоятельно с помощью статического метода `RegisterPrefix()`, который определен в классе `WebRequest`. При создании объекта класса `WebRequest` автоматически используется протокол (при условии его доступности), заданный префиксом URI. Достоинство использования настраиваемого адреса состоит в том, что большая часть программы остается неизменной при изменении типа протокола.

.NET-среда автоматически определяет протокол HTTP. Следовательно, если вы указываете URI-идентификатор с префиксом HTTP, то автоматически получите HTTP-совместимый класс, поддерживающий этот протокол. К классам, которые поддерживают протокол HTTP, относятся `HttpWebRequest` и `HttpWebResponse`. Эти классы — производные от классов `WebRequest` и `WebResponse`, и вполне естественно, что в них дополнительно определены собственные члены, которые применимы исключительно к протоколу HTTP.

В пространстве имен `System.Net` предусмотрена поддержка как синхронного, так и асинхронного взаимодействия. Для многих Internet-приложений предпочтительнее использовать синхронные транзакции, поскольку их проще реализовать. При синхронном взаимодействии программа посылает запрос, а затем ожидает ответа. Для некоторых типов высокоэффективных приложений лучше реализовать асинхронное взаимодействие. В этом случае ваша программа, ожидая информацию, может продолжать работу. Однако реализовать асинхронное взаимодействие гораздо труднее. Более того, не всем программам это "на руку". Например, нередки ситуации, когда без ожидаемой из Internet информации программе просто нечего дальше делать. В таких случаях потенциальный выигрыш, ожидаемый от применения асинхронного способа организации связи, не достигается. А поскольку синхронный доступ к Internet и проще реализуем, и более универсален, он и выбран для рассмотрения в этой главе.

Итак, начнем с самого главного: классов `WebRequest` и `WebResponse`, которые составляют ядро пространства имен `System.Net`.

Класс `WebRequest`

Класс `WebRequest` предназначен для управления сетевыми запросами. Этот класс абстрактный, поскольку не реализует конкретный протокол. Однако в нем определены методы и свойства, общие для всех запросов. Методы класса `WebRequest`, которые поддерживают синхронную передачу данных, представлены в табл. 23.1, а свойства — в табл. 23.2. Стандартные значения для свойств определяются производными классами. Класс `WebRequest` не определяет ни одного `public`-конструктора.

Чтобы отправить запрос по URI-адресу, необходимо сначала создать объект класса (выведенного из класса `WebRequest`), который реализует желаемый протокол. Это можно сделать с помощью статического метода `Create()`, определенного в классе `WebRequest`. Метод `Create()` возвращает объект класса, который является производным от класса `WebRequest` и реализует нужный протокол.

Таблица 23.1. Методы поддержки синхронной передачи данных, определенные в классе `WebRequest`

Метод	*Описание*
`public static WebRequest Create(string uri);`	Создает `WebRequest`-объект для URI-идентификатора, заданного строкой, переданной параметром *`uri`*. Возвращаемый объект реализует протокол, заданный префиксом URI-идентификатора. Следовательно, возвращаемый объект будет объектом класса, производного от `WebRequest`. Если затребованный протокол недоступен, генерируется исключение типа `NotSupportedException`. Если же недействителен указанный формат URI-идентификатора, генерируется исключение типа `UriFormatException`
`public static WebRequest Create(Uri uri);`	Создает `WebRequest`-объект для URI-идентификатора, заданного строкой, переданной параметром *`uri`*. Возвращаемый объект реализует протокол, заданный префиксом URI-идентификатора. Следовательно, возвращаемый объект будет объектом класса, производного от `WebRequest`. Если затребованный протокол недоступен, генерируется исключение типа `NotSupportedException`
`public virtual Stream GetRequestStream()`	Возвращает выходной поток, связанный с предварительно зарегистрированным URI-идентификатором
`public virtual WebResponse GetResponse()`	Отсылает предварительно созданный запрос и ожидает ответ. После получения ответа возвращает его в виде объекта класса `WebReponse`. Программа должна использовать этот объект для получения информации от заданного URI. При возникновении ошибки в процессе получения ответа генерируется исключение типа `WebException`

Таблица 23.2. Свойства, определенные в классе `WebRequest`

Свойство	*Описание*
`public virtual string ConnectionGroupName { get; set; }`	Получает или устанавливает групповое имя подключения. Группы подключения представляют собой способ создания набора запросов. Они не нужны для для простых Internet-транзакций
`public virtual long ContentLength { get; set; }`	Получает или устанавливает длину пакета переданных данных
`public virtual string ContentType { get; set; }`	Получает или устанавливает описание переданной информации
`public virtual ICredentials Credentials { get; set; }`	Получает или устанавливает уровень доступа. Уровни доступа необходимы для тех сайтов, которые требуют аутентификации (служба контроля доступа, проверяющая регистрационную информацию пользователя)
`public virtual WebHeaderCollection Headers{ get; set; }`	Получает или устанавливает коллекцию заголовков
`public virtual string Method { get; set; }`	Получает или устанавливает протокол
`public virtual bool PreAuthenticate { get; set; }`	Если равно значению `true`, в отправляемый запрос включается регистрационная информация. Если равно значению `false`, регистрационная информация предоставляется только по URI-требованию
`public virtual IWebProxy Proxy { get; set; }`	Получает или устанавливает путь к proxy-серверу. Применимо только в средах, в которых используется proxy-сервер
`public virtual Uri RequestUri { get; }`	Получает URI запроса
`public virtual int Timeout { get; set; }`	Получает или устанавливает количество миллисекунд, в течение которых будет ожидаться ответ на запрос. Для установки бесконечного ожидания используйте значение `Timeout.Infinite`

Класс WebResponse

Класс WebResponse инкапсулирует ответ, который принимается в качестве результата запроса. Класс WebResponse — абстрактный. Производные классы создают конкретные версии, ориентированные на поддержку того или иного протокола. Объект класса WebResponse обычно создается посредством вызова метода GetResponse(), определенного в классе WebRequest. Этот объект является экземпляром класса (выведенного из класса WebReponse), который реализует конкретный протокол. Методы, определенные в классе WebResponse, представлены в табл. 23.3, а свойства — в табл. 23.4. Значения этих свойств устанавливаются на основе каждого ответа. Класс WebResponse не определяет ни одного public-конструктора.

Таблица 23.3. Методы, определенные в классе WebResponse

Свойство	*Описание*
`public virtual void Close()`	Закрывает поток, содержащий ответ. Закрывает также поток ответа, который был возвращен методом GetResponseStream()
`public virtual Stream GetResponse Stream()`	Возвращает входной поток, связанный с затребованным URI. С помощью этого потока данные могут быть считаны с URI-источника

Таблица 23.4. Свойства, определенные в классе WebResponse

Свойство	*Описание*
`public virtual long ContentLength { get; set; }`	Получает или устанавливает длину пакета принятых данных. Устанавливается равным -1, если данные о длине недоступны
`public virtual string ContentType { get; set; }`	Получает описание принимаемой информации
`public virtual WebHeaderCollection Headers{ get; }`	Получает коллекцию заголовков, связанных с URI
`public virtual Uri ReponseUri { get; }`	Получает URI, который сгенерировал ответ. Этот адрес может отличаться от запрашиваемого, если ответ был перенаправлен на другой URI-адрес

Классы HttpWebRequest и HttpWebResponse

Классы HttpWebRequest и HttpWebResponse (производные от классов WebRequest и WebResponse, соответственно) реализуют протокол HTTP. В обоих определен ряд собственных свойств, которые позволяют детализировать информацию о HTTP-транзакциях. Использование некоторых из этих свойств демонстрируется позже в этой главе. Однако при выполнении простых Internet-операций вам вряд ли придется прибегать к этим дополнительным средствам.

Первый простой пример

Организация доступа к Internet сосредоточена в классах WebRequest и WebResponse. Прежде чем подробно проанализировать этот процесс, было бы полезно рассмотреть пример, который иллюстрирует реализацию модели "запрос-ответ" для Internet-доступа. Увидев эти классы в действии, легче понять, почему они организованы именно так.

Следующая программа выполняет простую и весьма распространенную Internet-операцию. Она получает гипертекст, содержащийся по заданному URI-адресу. В дан-

ном случае программа получает содержимое Web-сайта Osborne.com, который принадлежит Осборнскому отделению издательства McGraw-Hill, но вы можете использовать другой Web-сайт. Программа отображает гипертекст на экране порциями по 400 символов, что позволяет прочитать полученную информацию прежде, чем она сойдет с экрана.

```
// Доступ к Internet.

using System;
using System.Net;
using System.IO;

class NetDemo {
  public static void Main() {
    int ch;

    // Сначала создаем запрос по URI-адресу.
    HttpWebRequest req = (HttpWebRequest)
           WebRequest.Create("http://www.osborne.com");

    // Затем отправляем этот запрос и получаем ответ.
    HttpWebResponse resp = (HttpWebResponse)
           req.GetResponse();

    // Из ответа получаем входной поток.
    Stream istrm = resp.GetResponseStream();


    /* А теперь считываем и отображаем html-документ,
       полученный от заданного URI.  Текст "не улетит"
       с экрана, поскольку данные отображаются порциями
       объемом в 400 символов.  Просмотрев очередные
       400 символов, нажмите клавишу <ENTER>
       для получения следующей порции документа. */

    for(int i=1; ; i++) {
      ch =  istrm.ReadByte();
      if(ch == -1) break;
      Console.Write((char) ch);
      if((i%400)==0) {
        Console.Write("\nНажмите клавишу.");
        Console.Read();
      }
    }

    // Закрываем поток, содержащий ответ. При этом
    // автоматически закроется и входной поток istrm.
    resp.Close();
  }
}
```

Ниже приведена первая часть результатов выполнения этой программы. (Конечно же, текст, получаемый с этого Web-сайта, может со временем измениться.)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Find all the right computer books and learning tools at Osborne
McGraw-Hill</TITLE>
```

```
<META NAME="Title" CONTENT="Find all the right computer books and
learning tools at Osborne McGraw-Hill">
<META NAME="Keywords" CONTENT="osborne, mcgraw-hill, mcgraw hill, it
books, computer books, database books, programming
Нажмите клавишу.
books, networking books, certification books, computing books,
computer application books, hardware books, information technology
books, operating systems, web development, oracle press,
communications, complete reference, how to do everything, yellow
pages, book publisher, certification study guide, reference book,
security, network security, ebusiness, e-business,
a+, network+, i-net+, cisco ce
Нажмите клавишу.
.
.
.
```

Поскольку программа лишь посимвольно отображает принятое содержимое, она, в отличие от броузера, не форматирует его.

Каждая строка этой программы заслуживает рассмотрения. Прежде всего обратите внимание на использование пространства имен `System.Net`. Как упоминалось выше, именно это пространство имен содержит классы, обеспечивающие сетевые возможности C#. Здесь также используется пространство имен `System.IO`. Его присутствие обусловлено использованием объекта класса `Stream`, который позволяет считывать информацию с Web-сайта.

Программа начинается с создания `WebRequest`-объекта, который содержит нужный URI-адрес. Обратите внимание на то, что для этого используется метод `Create()`, а не конструктор. Метод `Create()` — статический член класса `WebRequest`. Несмотря на то что класс `WebRequest` — абстрактный, мы имеем возможность вызывать его статические методы. Метод `Create()` возвращает `WebRequest`-объект, который содержит протокол, "настраиваемый" на основе префикса URI-адреса. В данном случае используется протокол HTTP. Поэтому метод `Create()` возвращает `HttpWebRequest`-объект. Если значение, возвращаемое методом `Create()`, присваивается `HttpWebRequest`-ссылке с именем `req`, то оно обязательно должно быть приведено (в явной форме) к типу `HttpWebRequest`. После выполнения этой инструкции наш запрос создан, но еще не отправлен по заданному URI-адресу.

Чтобы отослать запрос, программа вызывает метод `GetResponse()` для объекта класса `WebRequest`. После отправки запроса метод `GetResponse()` ожидает ответа. По получении ответа метод `GetResponse()` возвращает `WebResponse`-объект, который инкапсулирует принятый ответ. Этот объект присваивается ссылке `resp`. Так как в данном случае в ответе используется протокол HTTP, результат вызова метода `GetResponse()` приводится к типу `HttpWebResponse`. Ответ содержит поток, который можно использовать для считывания данных с Web-сайта с заданным URI-адресом.

Затем посредством вызова метода `GetResponseStream()` для объекта `resp` считывается входной поток, который представляет собой стандартный объект класса `Stream`, имеющий все атрибуты любого другого входного потока. Ссылка на этот поток присваивается переменной `istrm`. Используя переменную `istrm`, данные, расположенные по заданному URI-адресу, можно прочитать так же, как считываются данные из файла.

Считанные данные программа отображает на экране. Поскольку объем данных может быть довольно большим, процесс отображения приостанавливается после вывода каждых 400 символов, и программа переходит в режим ожидания, который пре-

рвется нажатием клавиши <Enter>. Поэтому вы можете до бесконечности изучать первую порцию отображенной на экране информации. Обратите внимание на то, что символы считываются с помощью метода `ReadByte()`. Вспомните, что этот метод возвращает следующий символ из входного потока в виде `int`-значения, которое должно быть приведено к типу `char`. При достижении конца потока метод возвращает число −1.

Наконец, поток, содержащий ответ, закрывается посредством вызова метода `Close()` для объекта `resp`. При закрытии потока с ответом автоматически закрывается и входной поток. Важно закрывать поток, содержащий ответ, между запросами. В противном случае могут исчерпаться сетевые ресурсы, и следующее подключение к Internet может не состояться.

Здесь важно отметить еще одну деталь. Для отображения гипертекстовой информации с Web-сайта `Osborne.com` необязательно использовать объект класса `HttpWebRequest` или `HttpWebResponse`. Поскольку в предыдущей программе не использовались никакие специфические HTTP-средства, для решения этой задачи было бы вполне достаточно стандартных методов, определенных в классах `WebRequest` и `WebResponse`. Поэтому вызовы методов `Create()` и `GetResponse()` можно переписать так:

```
// Сначала создаем WebRequest-запрос по URI-адресу.
WebRequest req = WebRequest.Create(
                                 "http://www.osborne.com");

// Затем отправляем запрос и получаем ответ.
WebResponse resp = req.GetResponse();
```

Разработчики из компании Microsoft считают, что в случаях, когда необязательно использовать операцию приведения к типу реализации конкретного протокола, лучше обходиться средствами, предоставляемыми классами `WebRequest` и `WebResponse`. Это позволяет изменять протоколы, не внося изменений в программы. Но поскольку во всех примерах этой главы используется протокол HTTP (а в некоторых из них и специфические HTTP-средства), то в приведенных здесь программах задействованы классы `HttpWebRequest` и `HttpWebResponse`.

Обработка сетевых ошибок

Несмотря на то что программа, приведенная в предыдущем разделе, вполне корректна, она совершенно беззащитна перед потенциальными "ударами судьбы", которые могут внезапно оборвать ее "жизнь". В реальных приложениях необходимо предусмотреть все возможные неприятности и обеспечить полную обработку сетевых исключений, которые может сгенерировать программа. Для этого нужно проконтролировать обращения к методам `Create()`, `GetResponse()` и `GetResponseStream()`. Все типы потенциальных ошибок рассматриваются в следующих разделах.

Исключения, генерируемые методом `Create()`

Метод `Create()`, определенный в классе `WebRequest`, может сгенерировать три исключения. Если протокол, заданный URI-префиксом, не поддерживается, генерируется исключение типа `NotSupportedException`. Если неверно задан формат URI-идентификатора, генерируется исключение типа `UriFormatException`. Метод `Create()`, вызванный с использованием нулевой ссылки, может сгенерировать также исключение типа `ArgumentNullException`, хотя эта ошибка не относится к тем, которые генерируются сетевыми средствами.

Исключения, генерируемые методом GetReponse()

Если ошибка обнаружится в процессе получения ответа, т.е. при вызове метода GetReponse(), генерируется исключение типа WebException. Помимо членов, определенных для всех исключений, в классе WebException определены также два дополнительных свойства, которые связаны с сетевыми ошибками: Response и Status.

Внутри обработчика исключения с помощью свойства Response можно получить ссылку на объект класса WebResponse. Этот объект будет возвращен методом GetReponse(), если исключение не генерируется. Определение свойства Response выглядит так:

```
public WebResponse Response { get; }
```

Если ошибка все-таки возникла, то чтобы понять, что именно произошло, можно использовать свойство Status. Его определение имеет следующий вид:

```
public WebExceptionStatus Status {get; }
```

WebExceptionStatus — это перечисление, которое содержит следующие значения:

ConnectFailure	ConnectionClosed	KeepAliveFailure
NameResolutionFailure	Pending	PipelineFailure
ProtocolError	ProxyNameResolutionFailure	ReceiveFailure
RequestCanceled	SecureChannelFailure	SendFailure
ServerProtocolViolation	Success	Timeout
TrustFailure		

После выяснения причины ошибки программа может предпринять соответствующие действия.

Исключения, генерируемые методом GetResponseStream()

Метод GetResponseStream() класса GetResponse может генерировать исключение типа ProtocolViolationException, которое в общем случае означает, что произошла ошибка, связанная с протоколом. А поскольку ее возникновение имеет отношение к методу GetResponseStream(), значит нет ни одного потока, содержащего ответную информацию. В процессе чтения потока может также быть сгенерировано исключение типа IOException.

Обработка исключений

В следующую программу, которая основана на предыдущем примере, добавлены обработчики всех возможных сетевых исключений:

```
// Обработка сетевых исключений.

using System;
using System.Net;
using System.IO;

class NetExcDemo {
  public static void Main() {
    int ch;

    try {
```

```
      // Сначала создаем WebRequest-запрос по URI-адресу.
      HttpWebRequest req = (HttpWebRequest)
            WebRequest.Create("http://www.osborne.com");

      // Затем отправляем запрос и получаем ответ.
      HttpWebResponse resp = (HttpWebResponse)
            req.GetResponse();

      // Из ответа получаем входной поток.
      Stream istrm = resp.GetResponseStream();

    /* А теперь считываем и отображаем html-документ,
       полученный от заданного URI.  Текст "не улетит"
       с экрана, поскольку данные отображаются порциями
       объемом в 400 символов.  Просмотрев очередные
       400 символов, нажмите клавишу <ENTER>
       для получения следующей часть документа. */

      for(int i=1; ; i++) {
        ch =  istrm.ReadByte();
        if(ch == -1) break;
        Console.Write((char) ch);
        if((i%400)==0) {
          Console.Write("\nНажмите клавишу.");
          Console.Read();
        }
      }


      // Закрываем поток, содержащий ответ. При этом
      // автоматически закроется и входной поток istrm.
      resp.Close();

    } catch(WebException exc) {
      Console.WriteLine("Сетевая ошибка: " + exc.Message +
                        "\nКод состояния: " + exc.Status);
    } catch(ProtocolViolationException exc) {
      Console.WriteLine("Ошибка протокола: " +
                        exc.Message);
    } catch(UriFormatException exc) {
      Console.WriteLine("Ошибка формата URI: " +
                        exc.Message);
    } catch(NotSupportedException exc) {
      Console.WriteLine("Неизвестный протокол: " +
                        exc.Message);
    } catch(IOException exc) {
      Console.WriteLine("I/O Error: " + exc.Message);
    }
  }
}
```

В этом варианте программы будут перехвачены исключения, потенциально генерируемые Internet-методами. Например, если используемое в программе обращение к методу `Create()` заменить следующим

```
WebRequest.Create("http://www.osborne.com/moonrocket");
```

и перекомпилировать программу, а затем ее выполнить, вы непременно получите такое сообщение:

```
Сетевая ошибка: The remote server returned an error: (404) Not Found.
Код состояния: ProtocolError
```

Поскольку Web-сайт Osborne.com не имеет каталога с именем "moonrocket", вполне естественно, что этот URI-адрес не обнаружен.

Чтобы не загромождать примеры, большинство программ из этой главы не содержит полной обработки исключений. Но вы должны понимать, что в реальные приложения ее необходимо включать в полном объеме.

Класс URI

Возможно, вы обратили внимание на то, что в табл. 23.1 метод WebRequest.Create() представлен в двух различных версиях. Одна из них принимает URI в виде строки, и именно эта версия используется в предыдущих программах. Вторая принимает URI как экземпляр класса Uri. Класс Uri инкапсулирует URI-идентификатор. Используя класс Uri, можно создать такой URI-адрес, который будет принят второй версией метода Create(). При этом можно разбить URI-идентификатор на части. И хотя в случае простых Internet-операций можно обойтись без класса Uri, все же в более сложных случаях он окажется ценным подспорьем.

В классе Uri определено несколько конструкторов. Два наиболее употребимые из них определяются так:

```
public Uri(string uri)
public Uri(string base, string rel)
```

Первая форма позволяет создать Uri-объект на основе URI-адреса, заданного в виде строки. Вторая служит для создания Uri-объекта путем сложения относительного URI-идентификатора, заданного параметром *rel*, с базовым URI-идентификатором, заданным параметром *base*. Базовый URI-идентификатор определяет сам URI-адрес, а относительный — только сетевой маршрут.

В классе Uri определено множество полей, свойств и методов, которые позволяют управлять URI-идентификаторами или предоставляют доступ к различным его частям. Чаще всего используются следующие свойства.

Свойство	Описание
`public string Host { get; }`	Получает имя сервера
`public string LocalPath { get; }`	Получает маршрут, определяющий местоположение файла
`public string PathAndQuery { get; }`	Получает маршрут и строку запроса
`public int Port { get; }`	Получает номер порта для заданного протокола. Для HTTP номер порта равен 80
`public string Query { get; }`	Получает строку запроса
`public string Scheme { get; }`	Получает протокол

Эти свойства полезно применять при разбиении URI-идентификатора на составные части. Их использование демонстрирует следующая программа:

```
// Использование класса Uri.

using System;
using System.Net;
```

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

    Uri sample = new Uri(
           "http://MySite.com/somefile.txt?SomeQuery");

    Console.WriteLine("Хост: " + sample.Host);
    Console.WriteLine("Порт: " + sample.Port);
    Console.WriteLine("Протокол: " + sample.Scheme);
    Console.WriteLine("Локальный маршрут: " +
                      sample.LocalPath);
    Console.WriteLine("Запрос: " + sample.Query);
    Console.WriteLine("Маршрут и запрос: " +
                      sample.PathAndQuery);

  }
}
```

Вот результаты выполнения программы:

```
Хост: mysite.com
Порт: 80
Протокол: http
Локальный маршрут: /somefile.txt
Запрос: ?SomeQuery
Маршрут и запрос: /somefile.txt?SomeQuery
```

Доступ к дополнительной HTTP-информации

Используя класс `HttpWebResponse`, можно получить доступ не только к содержимому заданного ресурса, но и к информации, которая включает, например, время последней URI-модификации и имя сервера. Эту информацию можно получить с помощью различных свойств, перечисленных в табл. 23.5 (четыре из них определены в классе `WebResponse`). Об их использовании и пойдет речь в следующих разделах.

Таблица 23.5. Свойства, определенные в классе `HttpWebResponse`

Свойство	*Описание*
`public string CharacterSet { get; }`	Получает название используемого символьного набора
`public string ContentEncoding { get; }`	Получает название схемы кодирования.
`public long ContentLength { get; }`	Получает длину принимаемого содержимого. Если она недоступна, свойство содержит -1
`public string ContentType { get; }`	Получает описание содержимого
`public CookieCollection Cookies { get; set; }`	Получает или устанавливает список cookie-данных, присоединенных к ответу
`public WebHeaderCollection Headers{ get; }`	Получает коллекцию заголовков, присоединенных к ответу
`public DateTime LastModified { get; }`	Получает время последней URI-модификации
`public string Method { get; }`	Получает строку, которая задает способ ответа

Окончание табл. 23.5

Свойство	Описание
`public Version ProtocolVersion { get; }`	Получает объект класса `Version`, который описывает версию протокола HTTP, используемую в транзакции
`public Uri ReponseUri { get; }`	Получает URI-идентификатор, по которому сгенерирован ответ. Он может отличаться от запрошенного, если ответ был перенаправлен по другому URI-адресу
`public string Server { get; }`	Получает строку, которая представляет имя сервера
`public HttpStatusCode StatusCode { get; }`	Получает объект класса `HttpStatusCode`, который описывает состояние транзакции
`public string StatusDescription { get; }`	Получает строку, которая представляет состояние транзакции в форме, удобной для восприятия человеком

Доступ к заголовку

С помощью свойства `Headers`, которое определено в классе `HttpWebResponse`, можно получить доступ к заголовочной информации HTTP-ответа:

```
public WebHeaderCollection Headers{ get; }
```

HTTP-заголовок состоит из пар имя/значение, представленных в виде строк. Каждая такая пара хранится в в объекте класса `WebHeaderCollection`. Это — специализированная коллекция, предназначенная для хранения пар ключ/значение, которую можно использовать подобно любой другой коллекции (см. главу 22). Строковый массив имен можно получить из свойства `AllKeys`. Значение, связанное с конкретным именем, можно получить с помощью индексатора. Индексатор здесь перегружен на прием либо числового индекса, либо имени.

В следующей программе отображаются все заголовки, относящиеся к Web-сайту `Osborne.com`:

```
// Отображение заголовков Web-сайта.

using System;
using System.Net;

class HeaderDemo {
  public static void Main() {

    // Создаем WebRequest-запрос по URI-адресу.
    HttpWebRequest req = (HttpWebRequest)
           WebRequest.Create("http://www.osborne.com");

    // Отправляем этот запрос и получаем ответ.
    HttpWebResponse resp = (HttpWebResponse)
           req.GetResponse();

    // Получаем список имен.
    string[] names = resp.Headers.AllKeys;

    // Отображаем заголовок в виде пар имя/значение.
    Console.WriteLine("{0,-20}{1}\n", "Имя", "Значение");
    foreach(string n in names)
      Console.WriteLine("{0,-20}{1}", n, resp.Headers[n]);

    // Закрываем поток, содержащий ответ.
```

```
    resp.Close();
  }
}
```

Вот какие были получены результаты (не забывайте, что заголовочная информация Web-сайта `Osborne.com` со временем может измениться, поэтому результаты, полученные вами, могут несколько отличаться):

```
Имя                 Значение

Date                Mon, 14 Jan 2002 17:45:50 GMT
Server              Apache/1.3.9 (Unix) PHP/3.0.14
Keep-Alive          timeout=30, max=500
Connection          Keep-Alive
Transfer-Encoding   chunked
Content-Type        text/html
```

Доступ к cookie-данным

Доступ к cookie-данным, связанным с HTTP-ответом, можно получить с помощью свойства `Cookies`, которое определено в классе `HttpWebResponse`. Cookie-данные, которые хранятся броузером, состоят из пар имя/значение. Они способны упростить определенные типы операций Web-доступа. Вот как выглядит определение свойства `Cookies`:

```
public CookieCollection Cookies { get; set; }
```

Класс `CookieCollection` реализует интерфейсы `ICollection` и `IEnumerable`, и его можно использовать подобно любой другой коллекции (см. главу 22). Он имеет индексатор, который позволяет получить cookie-данные по заданному индексу или его имени.

Коллекция типа `CookieCollection` предназначена для хранения объектов типа `Cookie`. В классе `Cookie` определено несколько свойств, которые предоставляют доступ к различным частям cookie-данных. Нас интересуют свойства `Name` и `Value`:

```
public string Name { get; set; }
public string Value { get; set; }
```

Нетрудно догадаться, что cookie-имя содержится в свойстве `Name`, а значение — в свойстве `Value`.

Чтобы получить список cookie-составляющих, связанных с ответом, необходимо воспользоваться cookie-контейнером. Для этого в классе `HttpWebRequest` определено свойство `CookieContainer`:

```
public CookieContainer CookieContainer { get; set; }
```

В классе `CookieContainer` определены различные поля, свойства и методы, которые позволяют хранить cookie-данные. Однако при создании многих приложений нет необходимости в непосредственном использовании свойства `CookieContainer`. Обычно используют коллекцию типа `CookieCollection`, получаемую из ответа. Назначение класса `CookieContainer` — обеспечить механизм хранения cookie-данных.

Следующая программа отображает имена и значения cookie-разделов, связанных с URI-идентификатором, заданным в командной строке. При этом важно помнить, что не все Web-сайты используют cookie-данные, поэтому не стоит удивляться, если вам попадется именно такой.

```
/* Отображение cookie-данных.

   Чтобы узнать, какие cookie-данные использует
   интересующий вас Web-сайт, укажите его имя
```

```
   в командной строке.
   Например, если эту программу назвать
   Cookie, то после выполнения команды

     Cookie http://MSN.COM

   будут отображены cookie-данные,
   связанные с Web-сайтом MSN.COM.
*/

using System;
using System.Net;

class CookieDemo {
  public static void Main(string[] args) {

    if(args.Length != 1) {
      Console.WriteLine("Usage: CookieDemo <uri>");
      return ;
    }

    // Создаем WebRequest-запрос по заданному URI-адресу.
    HttpWebRequest req = (HttpWebRequest)
          WebRequest.Create(args[0]);

    // Получаем пустой cookie-контейнер.
    req.CookieContainer = new CookieContainer();

    // Отправляем запрос и получаем ответ.
    HttpWebResponse resp = (HttpWebResponse)
          req.GetResponse();

    // Отображаем cookie-данные.
    Console.WriteLine("Количество cookie-разделов: " +
                      resp.Cookies.Count);
    Console.WriteLine("{0,-20}{1}", "Имя", "Значение");

    for(int i=0; i < resp.Cookies.Count; i++)
      Console.WriteLine("{0, -20}{1}",
                        resp.Cookies[i].Name,
                        resp.Cookies[i].Value);

    // Закрываем поток, содержащий ответ.
    resp.Close();
  }
}
```

Использование свойства `LastModified`

Иногда нужно узнать, когда в последний раз обновлялся Web-сайт с заданным URI-адресом. С помощью класса `HttpWebResponse` это не составляет труда, поскольку в нем определено свойство `LastModified`:

```
public DateTime LastModified { get; }
```

Свойство `LastModified` получает время последней URI-модификации.

При выполнении следующей программы отображается время и дата последней модификации Web-сайта `Microsoft.com`:

```
// Использование свойства LastModified.

using System;
using System.Net;

class HeaderDemo {
  public static void Main() {

    HttpWebRequest req = (HttpWebRequest)
           WebRequest.Create("http://www.Microsoft.com");

    HttpWebResponse resp = (HttpWebResponse)
           req.GetResponse();

    Console.WriteLine(
                  "Время и дата последней модификации: " +
                  resp.LastModified);

    resp.Close();
  }
}
```

Учебный проект: программа MiniCrawler

Чтобы показать, насколько Internet-программирование упрощается благодаря использованию классов `WebRequest` и `WebReponse`, обратимся к разработке Web-такси. Web-такси — это программа (здесь она называется `MiniCrawler`), которая просто перемещается от одной Web-страницы к другой. Web-такси в различных средствах поиска используется для составления каталога содержимого. `MiniCrawler` — довольно простая программа. Она начинает работать с заданного вами URI-адреса, а затем считывает его содержимое в расчете найти ссылку на какой-нибудь Web-документ. Если такая ссылка найдется, "водитель" Web-такси поинтересуется вашим мнением о дальнейших действиях: перейти по этой ссылке, искать другую (на той же странице) или завершить работу.

Программа `MiniCrawler` имеет ряд ограничений. Во-первых, поиску подлежат только абсолютные ссылки, т.е. те, которые задаются с использованием гипертекстовой команды `href=`. Относительные ссылки игнорируются. Во-вторых, здесь не предусмотрена возможность вернуться назад, т.е. к предыдущей ссылке, по которой был сделан переход. В-третьих, эта программа отображает только одни гиперссылки без сопутствующего содержимого. Но, несмотря на эти ограничения, Web-такси вполне функционально. К тому же, при желании вы можете расширить его возможности. И в самом деле, почему бы вам не добавить в `MiniCrawler` какое-нибудь полезное средство? Ведь подобная попытка позволит лучше познакомиться с сетевыми классами и сетевыми C#-возможностями в целом.

Итак, рассмотрим рассмотрим код программы `MiniCrawler`:

```
// MiniCrawler: Web-такси.

using System;
using System.Net;
using System.IO;

class MiniCrawler {
```

```
// Находим гиперссылку в строке.
static string FindLink(string htmlstr,
                       ref int startloc) {
  int i;
  int start, end;
  string uri = null;
  string lowcasestr = htmlstr.ToLower();

  i = lowcasestr.IndexOf("href=\"http", startloc);
  if(i != -1) {
    start = htmlstr.IndexOf('"', i) + 1;
    end = htmlstr.IndexOf('"', start);
    uri = htmlstr.Substring(start, end-start);
    startloc = end;
  }

  return uri;
}

public static void Main(string[] args) {
  string link = null;
  string str;
  string answer;

  int curloc; // Содержит текущую позицию в ответе.

  if(args.Length != 1) {
    Console.WriteLine("Usage: MiniCrawler <uri>");
    return ;
  }

  string uristr = args[0]; // Содержит текущий URI-адрес.

  try {

    do {
      Console.WriteLine("Переход по адресу " + uristr);

      // Создаем WebRequest-запрос по заданному URI.
      HttpWebRequest req = (HttpWebRequest)
            WebRequest.Create(uristr);

      uristr = null; // Запрещаем дальнейшее
                     // использование этого URI-адреса.

      // Отсылаем этот запрос и получаем ответ.
      HttpWebResponse resp = (HttpWebResponse)
            req.GetResponse();

      // Из потока, содержащего ответ, получаем
      // входной поток.
      Stream istrm = resp.GetResponseStream();

      // Представляем входной поток
      // в виде StreamReader-объекта.
      StreamReader rdr = new StreamReader(istrm);

      // Считываем целую страницу.
```

```
        str = rdr.ReadToEnd();

        curloc = 0;

        do {
          // Находим следующий URI-адрес для перехода
          // по гиперссылке.
          link = FindLink(str, ref curloc);

          if(link != null) {
            Console.WriteLine("Гиперссылка найдена: " +
                              link);

            Console.Write("Ссылка, Дальше, Выход?");
            answer = Console.ReadLine();

            if(string.Compare(answer, "С", true) == 0) {
              uristr = string.Copy(link);
              break;
            } else if(string.Compare(
                              answer, "В", true) == 0) {
              break;
            } else if(string.Compare(
                              answer, "Д", true) == 0) {
              Console.WriteLine("Поиск следующей ссылки.");
            }
          } else {
            Console.WriteLine("Больше ссылок не найдено.");
            break;
          }

        } while(link.Length > 0);

        // Закрываем поток, содержащий ответ.
        resp.Close();
      } while(uristr != null);

    } catch(WebException exc) {
      Console.WriteLine("Сетевая ошибка: " + exc.Message +
                        "\nКод состояния: " + exc.Status);
    } catch(ProtocolViolationException exc) {
      Console.WriteLine("Ошибка протокола: " + exc.Message);
    } catch(UriFormatException exc) {
      Console.WriteLine("Ошибка в формате URI: " +
                        exc.Message);
    } catch(NotSupportedException exc) {
      Console.WriteLine("Неизвестный протокол: " +
                        exc.Message);
    } catch(IOException exc) {
      Console.WriteLine("I/O Error: " + exc.Message);
    }

    Console.WriteLine("Завершение программы MiniCrawler.");
  }
}
```

Вот как выглядит фрагмент сеанса работы с программой `MiniCrawler`:

```
C:>MiniCrawler http://osborne.com
Переход по адресу http://osborne.com
Гиперссылка найдена: http://www.osborne.com/aboutus/aboutus.shtml
Ссылка, Дальше, Выход? Д
Поиск следующей ссылки.
Гиперссылка найдена: http://www.osborne.com/downloads/downloads.shtml
Ссылка, Дальше, Выход? С
Переход по адресу http://www.osborne.com/downloads/downloads.shtml
.
.
.
```

Теперь рассмотрим, как работает эта программа. URI-адрес, с которого `MiniCrawler` начинает "извоз", задается в командной строке. В методе `Main()` этот URI-адрес запоминается в строке `uristr`. По нему создается запрос, а переменная `uristr` устанавливается равной значению `null`, чтобы нулевая ссылка служила индикатором того, что этот URI-адрес уже использован. Затем запрос отсылается, и на него ожидается ответ. Содержимое полученного ответа считывается посредством вызова метода `GetResponseStream()`, а результат этого вызова (входной поток) представляется в виде `StreamReader`-объекта. После этого вызывается метод `ReadToEnd()`, который возвращает полное содержимое потока в виде строки.

В полученной строке программа выполняет поиск гиперссылки. Это реализуется с помощью статического метода `FindLink()`, который также определен в классе `MiniCrawler`. Методу `FindLink()`, помимо строкового содержимого потока, также передается стартовая позиция, с которой необходимо начать поиск. Параметры, которые принимают эти значения, имеют имена `htmlstr` и `startloc`, соответственно. Обратите внимание на то, что `startloc` является `ref`-параметром. Метод `FindLink()` сначала создает копию "потоковой" строки в "строчном" написании, а затем ищет подстроку `href="http`, которая означает гиперссылку. Если такая подстрока обнаружится, URI-адрес копируется в строку `uri`, а значение параметра `startloc` обновляется и устанавливается равным конечной позиции найденной гиперссылки. Поскольку `startloc` — `ref`-параметр, то соответствующий аргумент будет обновлен и в методе `Main()`, в результате чего следующий поиск начнется с той позиции, на которой остановился предыдущий. В конечном итоге метод `FindLink()` возвращает значение строки `uri`. Если гиперссылка не обнаружится, то, поскольку `uri` была инициализирована `null`-значением, метод и вернет эту нулевую ссылку, которая "засвидельствует" неудачный исход.

Метод `Main()` *при* удачном выполнении метода `FindLink()` отображает значение найденной гиперссылки, и предлагает пользователю выбрать следующее действие из трех возможных. Пользователь может выбрать переход по *с*сылке (нажав клавишу <С>), заказать *д*альнейший поиск другой ссылки (нажав клавишу <Д>) или *в*ыйти из программы (нажав клавишу <В>). Если пользователь нажмет клавишу <С>, будет выполнен новый запрос по известному адресу с получением содержимого соответствующий страницы. И теперь в новом гипертексте будет разыскиваться очередная ссылка. Этот процесс продолжается до тех пор, пока не будут найдены все гиперссылки.

Возможно, вам захочется усилить описанную выше программу `MiniCrawler`. Например, попробуйте автоматизировать Web-такси, заставив его посещать каждую найденную гиперссылку, не спрашивая мнения пользователя. Другими словами, укажите какую-нибудь начальную страницу и отправьте Web-такси в путь, к первой ссылке, которую он найдет. Затем (уже в новой странице) пусть оно снова отправится по первой найденной ссылке и т.д. В том случае, если ваше Web-такси попадет в "тупик", оно должно возвратиться назад на один уровень, найти (в предыдущей странице) следующую гиперссылку и действовать по уже описанной схеме. Чтобы реализовать эту

схему, для хранения URI-адресов и текущей позиции поиска внутри URI-строки следует использовать стек. Поэтому можете воспользоваться коллекцией типа `Stack`. Можно еще усложнить задачу: попробуйте найденные гиперссылки отобразить в виде дерева.

Использование класса `WebClient`

"На десерт" кратко рассмотрим класс `WebClient`. Как упоминалось в начале этой главы, если вам нужно загрузить на удаленный компьютер (или из него) файл, то вместо классов `WebRequest` и `WebResponse` можно для этого использовать класс `WebClient`. Достоинство класса `WebClient` состоит в том, что он выполняет многие действия сам, освобождая таким образом вас.

В классе `WebClient` определен один конструктор:

```
public WebClient()
```

В нем также определены весьма полезные свойства (см. табл. 23.6) и методы (см. табл. 23.7). Все методы генерируют исключение типа `UriFormatException`, если заданный URI-адрес окажется недействительным, и исключение типа `WebException`, если во время передачи данных возникнет ошибка.

Таблица 23.6. Свойства, определенные в классе `WebClient`

Свойство	*Описание*
`public string BaseAddress { get; set; }`	Получает или устанавливает базовый адрес нужного URI. По умолчанию это свойство имеет `null`-значение. Если это свойство установлено, то адреса, заданные методами класса `WebClient`, должны интерпретироваться относительно этого базового адреса
`public ICredentials Credentials { get; set; }`	Получает или устанавливает регистрационную информацию. Это свойство по умолчанию имеет значение `null`
`public WebHeaderCollection Headers{ get; set; }`	Получает или устанавливает коллекцию заголовков запроса
`public NameValueCollection QueryString { get; set; }`	Получает или устанавливает строку запроса, состоящую из пар имя/значение, которые могут быть присоединены к запросу. Строка запроса отделяется от URI символом "?". Если таких пар больше одной, то каждая из них отделяется символом "@"
`public WebHeaderCollection ResponseHeaders{ get; }`	Получает коллекцию заголовков ответа

Таблица 23.7. Методы, определенные в классе `WebClient`

Метод	*Описание*
`public byte[] DownloadData(string uri)`	Загружает информацию с Web-страницы, URI-адрес которой задается параметром *`uri`*. Возвращает результат в виде массива байтов
`public void DownloadFile(string uri, string fname)`	Загружает информацию с Web-страницы, URI-адрес которой задается параметром *`uri`*, и сохраняет результат в файле, имя которого задается параметром *`fname`*
`public Stream OpenRead(string uri)`	Возвращает входной поток, информацию из которого можно прочитать с использованием URI-адреса, заданного параметром *`uri`*. После завершения считывания этот поток необходимо закрыть

Окончание табл. 23.7

Метод	Описание
`public Stream OpenWrite(string uri)`	Возвращает выходной поток, в который можно записать информацию с помощью URI-адреса, заданного параметром `uri`. После завершения записи этот поток необходимо закрыть
`public Stream OpenWrite(string uri, string how)`	Возвращает выходной поток, в который можно записать информацию с использованием URI-адреса, заданного параметром `uri`. После завершения записи этот поток необходимо закрыть. Строка, переданная в параметре `how`, задает способ записи этой информации
`public byte[] UploadData(string uri, byte[] info)`	Записывает информацию, заданную параметром `info`, по URI-адресу, заданному параметром `uri`. Возвращает ответ
`public byte[] UploadData(string uri, string how, byte[] info)`	Записывает информацию, заданную параметром `info`, по URI-адресу, заданному параметром `uri`. Возвращает ответ. Строка, переданная в параметре `how`, задает способ записи этой информации
`public byte[] UploadFile(string uri, string fname)`	Записывает информацию из файла, имя которого задается параметром `fname`, по URI-адресу, заданному параметром `uri`. Возвращает ответ
`public byte[] UploadFile(string uri, string how, string fname)`	Записывает информацию из файла, имя которого задается параметром `fname`, по URI-адресу, заданному параметром `uri`. Возвращает ответ. Строка, переданная в параметре `how`, задает способ записи этой информации
`public byte[] UploadValues( string uri, NameValueCollection vals )`	Записывает значения, хранимые в коллекции, заданной параметром `vals`, по URI-адресу, заданному параметром `uri`. Возвращает ответ
`public byte[] UploadValues( string uri, string how, NameValueCollection vals )`	Записывает значения, хранимые в коллекции, заданной параметром `vals`, по URI-адресу, заданному параметром `uri`. Возвращает ответ Строка, переданная в параметре `how`, задает способ записи этой информации

Использование класса `WebClient` для загрузки данных в файл демонстрируется в следующей программе:

```
// Использование класса WebClient для загрузки
// информации в файл.

using System;
using System.Net;
using System.IO;

class WebClientDemo {
  public static void Main() {
    WebClient user = new WebClient();
    string uri = "http://www.osborne.com";
    string fname = "data.txt";

    try {
      Console.WriteLine(
            "Загрузка данных из Web-страницы " +
            uri + " в файл " + fname);
      user.DownloadFile(uri, fname);
```

```
    } catch (WebException exc) {
      Console.WriteLine(exc);
    } catch (UriFormatException exc) {
      Console.WriteLine(exc);
    }

    Console.WriteLine("Загрузка завершена.");
  }
}
```

Эта программа загружает информацию с Web-сайта `Osborne.com` и помещает ее в файл с именем `data.txt`. Обратите внимание на то, что все это реализуется очень небольшим количеством строк кода, в число которых включены и те, которые обеспечивают обработку возможных исключений. Приведенная программа позволяет загрузить информацию с любого URI-адреса: для этого достаточно изменить соответствующим образом строку, заданную переменной `uri`.

Несмотря на то что классы `WebRequest` и `WebResponse` дают программисту большие возможности для управления и доступ к более обширной информации, средств, инкапсулированных классом `WebClient`, вполне достаточно для потребностей многих приложений. Он особенно полезен в том случае, если приложение должно обеспечить лишь загрузку информации из Web-пространства. Например, с помощью класса `WebClient` вы могли бы организовать получение обновленной документации по интересующим вас продуктам.

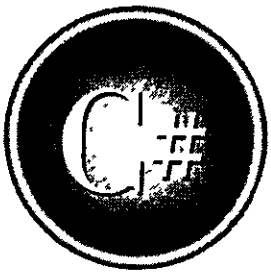
Полный справочник по C#

Часть III

Применение языка C#

В части III представлены три C#-приложения. Первое иллюстрирует создание компонентов и управление ими. Поскольку многие средства C# были разработаны специально для того, чтобы упростить их создание, нетрудно предположить, что компоненты составляют важную часть программирования на C#. Во втором приложении демонстрируется создание Windows-программы с помощью классов, определенных в пространстве имен System.Windows.Forms. Наконец, третье приложение представляет собой пример "чистого кода": речь идет о применении синтаксического анализа методом рекурсивного спуска к вычислению алгебраических выражений.

Полный справочник по



Глава 24

Создание компонентов

Несмотря на то что язык C# можно использовать для написания приложений практически любого типа, одной из самых значительных областей его применения является создание компонентов. Компонентно-ориентированное программирование настолько существенно для C#, что его иногда называют *компонентно-ориентированным языком* (component-oriented language). Поскольку C# и среда .NET Framework разрабатывались с ориентацией на компоненты, компонентная модель программирования здесь в значительной степени упрощена по сравнению с более ранними решениями. Например, если термин *компонент* наводит вас на мысли о COM-компонентах (Component Object Model — модель компонентных объектов) и связанных с этим проблемах, не беспокойтесь. C#-ориентированные компоненты создавать намного проще.

Что представляет собой компонент

Начнем с определения термина *компонент*. Компонент — это независимый модуль, предназначенный для многократного использования и предоставляемый пользователю в двоичном формате. Это определение описывает четыре ключевых характеристики компонента. Рассмотрим их по очереди.

Компонент определен как *независимый* модуль. Это означает, что каждый компонент вполне самодостаточен. Другими словами, компонент обеспечивает полный набор функций. Его внутренняя работа закрыта для "внешнего мира", но при этом реализация может быть изменена без последствий для кода, в котором используется этот компонент.

Компонент *предназначен для многократного применения*. Это означает, что компонент может использовать любая программа, которой требуются его функции. Программа, которая использует компонент, называется *клиентом* (client). Таким образом, компонент может работать с любым количеством клиентов.

Компонент представляет собой отдельный *модуль*. Это очень важно. С точки зрения клиента компонент выполняет конкретную функцию или набор функций. Функциями компонента, может воспользоваться любое приложение, но сам компонент не является автономной программой.

Наконец, компонент должен быть представлен в *двоичном формате*. Это принципиально важно. Хотя использовать компонент могут многие клиенты, они не имеют доступа к его исходному коду. Функциональность компонента открыта для клиентов только посредством его `public`-членов. Другими словами, именно компонент управляет тем, какие функции оставлять открытыми для клиентов, а какие — держать "под замком".

Компонентная модель

Несмотря на то что приведенное выше определение точно описывает программный компонент, для полного его понимания (и использования) этого недостаточно. Принципиально важное значение здесь имеет модель, которая реализует компоненты. Для того чтобы клиент мог использовать компонент, необходимо, чтобы и клиент, и компонент использовали один и тот же набор правил. Набор правил, определяющих форму и поведение компонента, и называется *компонентной моделью* (component model). Именно компонентная модель определяет характер взаимодействия компонента и модели.

Компонентная модель важна хотя бы потому, что предназначенный для многократного использования компонент, предоставляемый пользователю в двоичном фор-

мате может быть реализован различными способами, причем количество этих способов может быть очень большим. Например, существует множество разных способов передачи параметров и приема значений. Можно также по-разному выделять (и освобождать) память (и другие системные ресурсы) для объектов. Поэтому для эффективного использования компонентов клиенты и сами компоненты должны подчиняться правилам, определенным компонентной моделью. По сути компонентная модель представляет своего рода контракт между клиентом и компонентом, который обе стороны согласны выполнять.

До создания C# и среды .NET Framework большинство компонентов были COM-компонентами. Модель COM была разработана для традиционной среды Windows и языка C++. Поэтому она в принципе не в состоянии использовать преимущества новых методов управления памятью, которые реализованы в C# и .NET Framework. Как следствие, COM-контракт был довольно трудным для реализации и ненадежным. К счастью, C# и среда .NET Framework лишены практически всех проблем своих предшественников. Поэтому, если у вас за плечами есть опыт работы с COM-моделью, вы будете приятно удивлены простотой создания компонентов в C#.

Что представляет собой C#-компонент

Благодаря особенностям работы средств языка C#, любой его класс полностью соответствует общему определению компонента. Например, будучи скомпилированным, класс (в его двоичной форме) можно использовать в различных приложениях. Но значит ли это, что любой класс является компонентом? Ответ: нет. Для того чтобы класс стал компонентом, он должен следовать компонентной модели, определенной средой .NET Framework. К счастью, этого совсем не трудно добиться: такой класс должен реализовать интерфейс `System.ComponentModel.IComponent`. При реализации интерфейса `IComponent` компонент удовлетворяет набору правил, позволяющих компоненту быть совместимым со средой .NET Framework.

Несмотря на простоту реализации интерфейса `IComponent`, во многих ситуациях лучше использовать альтернативный вариант — класс `System.ComponentModel.Component`. Класс `Component` обеспечивает стандартную реализацию интерфейса `IComponent`. Он также поддерживает другие полезные средства, свойственные компонентам. Опыт показывает, что большинству создателей компонентов удобнее выводить их из класса `Component`, чем самим реализовать интерфейс `IComponent`, поскольку в первом случае нужно попросту меньше программировать. Следовательно, в C# на создание компонента не требуется "героических усилий" со стороны программиста.

Контейнеры и узлы

С C#-компонентами тесно связаны две другие конструкции: контейнеры и узлы. *Контейнер* — это группа компонентов. Контейнеры упрощают программы, в которых используется множество компонентов. *Узел* позволяет связывать компоненты и контейнеры. Подробнее обе эти конструкции рассматриваются ниже.

Сравнение C#- и COM-компонентов

C#-компоненты гораздо проще реализовать и использовать, чем COM-компоненты. Те, кто знаком с моделью COM, знают, что при использовании COM-компонента необходимо выполнять подсчет ссылок (механизм учета клиентов данного объекта), чтобы гарантировать, что компонент останется в памяти. При такой организации каждый раз, когда добавляется очередная ссылка на компонент, программа

должна вызывать метод AddRef(). При каждом удалении ссылки программа должна вызывать метод Release(). Но самое печальное здесь то, что такой подход чреват ошибки. К счастью, для C#-компонентов подсчета ссылок не требуется. Поскольку C# использует систему сбора мусора, компонент автоматически остается в памяти до тех пор, пока существует хотя бы одна ссылка на него.

Поскольку интерфейс IComponent и класс Component составляют сердцевину программирования компонентов, следующие разделы будут посвящены именно им.

Интерфейс IComponent

Интерфейс IComponent определяет правило, которому должны следовать все компоненты. В интерфейсе IComponent определено только одно свойство и одно событие. Вот как объявляется свойство Site:

```
ISite Site { get; set; }
```

Свойство Site получает или устанавливает узел компонента. Узел идентифицирует компонент. Это свойство имеет null-значение, если компонент не хранится в контейнере.

Событие, определенное в интерфейсе IComponent, носит имя Disposed и объявляется так:

```
event EventHandler Disposed
```

Клиент, которому нужно получить уведомление при разрушении компонента, регистрирует обработчик событий посредством события Disposed.

Интерфейс IComponent также наследует интерфейс System.IDisposable, в котором определен метод Dispose():

```
void Dispose()
```

Этот метод освобождает ресурсы, используемые объектом.

Класс Component

Несмотря на то что для создания компонента достаточно реализовать интерфейс IComponent, намного проще создать класс, производный от класса Component, поскольку он реализует интерфейс IComponent по умолчанию. Именно этот вариант построения компонентов и используется в примерах этой главы. Если класс наследует класс Component, значит, он автоматически выполняет правила, необходимые для получения .NET-совместимого компонента.

В классе Component определен только конструктор по умолчанию. Обычно программисты не создают объект класса Component напрямую, поскольку основное назначение этого класса — быть базовым для создаваемых компонентов.

В классе Component определено два открытых свойства. Объявление первого из них, свойства Container, такое:

```
public IContainer Container { get; }
```

Свойство Container возвращает контейнер, который содержит вызывающий компонент. Если компонента нет в контейнере, возвращается значение null. Следует помнить, что свойство Container устанавливается не компонентом, а контейнером.

Второе свойство, Site, определено в интерфейсе IComponent. В классе Component оно реализовано как виртуальное:

```
public virtual ISite Site {get; set; }
```

Свойство `Site` возвращает или устанавливает объект типа `ISite`, связанный с компонентом. Оно возвращает значение `null`, если компонента в контейнере нет. Свойство `Site` устанавливается не компонентом, а контейнером.

В классе `Component` определено два открытых метода. Первый представляет собой переопределение метода `ToString()`. Второй, метод `Dispose()`, используется в двух формах. Первая из них такова:

```
public void Dispose()
```

Метод `Dispose()`, вызванный в первой форме, освобождает *любые* ресурсы, используемые вызывающим компонентом. Этот метод реализует метод `Dispose()`, определенный в интерфейсе `IDisposable`. Для освобождения компонента и занимаемых им ресурсов клиент вызывает именно эту версию метода `Dispose()`.

Вот как выглядит вторая форма метода `Dispose()`:

```
protected virtual public void Dispose(bool how)
```

Если параметр *how* имеет значение `true`, эта версия метода освобождает как управляемые, так и неуправляемые ресурсы, используемые вызывающим объектом. Если *how* равно значению `false`, освобождаются только неуправляемые ресурсы. Поскольку эта версия метода `Dispose()` защищена (`protected`), ее нельзя вызвать из кода клиента. Поэтому клиент использует первую версию. Другими словами, вызов первой версии метода `Dispose()` генерирует обращение к методу `Dispose(bool)`.

В общем случае компонент, в котором больше нет необходимости, переопределяет версию `Dispose(bool)`, если он содержит ресурсы, которые нужно освободить. Если компонент не занимает никаких ресурсов, то для его освобождения достаточно стандартной реализации метода `Dispose(bool)`, определенной в классе `Component`.

Класс `Component` наследует класс `MarshalByRefObject`, который используется в том случае, когда компонент создается вне локальной среды, например в другом процессе или на другом компьютере, связанном с первым по сети. Для обмена данными (аргументами методов и возвращаемыми значениями) должен существовать механизм, который определит способ пересылки данных. По умолчанию принимается, что информация должна передаваться по значению, но при унаследовании класса `MarshalByRefObject` данные будут передаваться по ссылке. Таким образом, C#-компонент обеспечивает передачу данных по ссылке.

Простой компонент

Итак, настало время от теории перейти к практике и рассмотреть первый пример создания компонента `CipherComp`, который реализует очень простую стратегию шифрования. Ее суть состоит в том, что каждый символ шифруется путем добавления к его коду единицы. Дешифрирование заключается в вычитании единицы. Чтобы зашифровать таким способом строку, достаточно вызвать метод `Encode()`, передав незашифрованный текст в качестве аргумента. Чтобы дешифровать зашифрованную строку, вызовите метод `Decode()`, передав ему как аргумент зашифрованный текст. В обоих случаях возвращаются строки, содержащие результат (шифрования или дешифрирования).

```
// Простой компонент-шифратор.
// Назовите этот файл CipherLib.cs.

using System.ComponentModel;

namespace CipherLib { // Помещаем компонент в его же
                      // пространство имен.
```

```
  // Обратите внимание на то, что класс CipherComp
  // наследует класс Component.
  public class CipherComp : Component {

    // Шифруем строку.
    public string Encode(string msg) {
      string temp = "";

      for(int i=0; i < msg.Length; i++)
        temp += (char) (msg[i] + 1);

      return temp;
    }

    // Дешифруем строку.
    public string Decode(string msg) {
      string temp = "";

      for(int i=0; i < msg.Length; i++)
        temp += (char) (msg[i] - 1);


      return temp;
    }
  }
}
```

Итак, рассмотрим этот код подробнее. Прежде всего, как предлагается в комментарии, назовем этот файл `CipherLib.cs`. Это упростит использование рассматриваемого компонента, если вы работаете в среде разработки Visual Studio IDE. Затем обратите внимание на включение пространства имен `System.ComponentModel`. Как упоминалось выше, оно как раз и предназначено для поддержки программирования компонентов.

Класс `CipherComp` создается в собственном пространстве имен `CipherLib`. Это позволяет защитить глобальное пространство имен от загромождения новыми именами. Несмотря на то что такой подход формально необязателен, предложенный стиль программирования приветствуется.

Класс `CipherComp` наследует класс `Component`. Это означает, что класс `CipherComp` удовлетворяет всем требованиям для того, чтобы быть .NET-совместимым компонентом. Поскольку класс `CipherComp` очень простой, ему не нужно обеспечивать выполнение специфических для компонентов функций. Его действия можно определить как тривиальные.

Обратите также внимание на то, что класс `CipherComp` не распределяет системных ресурсов. Другими словами, он не хранит ссылок на другие объекты. Он просто определяет два метода `Encode()` и `Decode()`. А поскольку в классе `CipherComp` ссылки не хранятся, ему не нужно реализовать метод `Dispose(bool)`. Безусловно, оба метода `Encode()` и `Decode()` возвращают ссылки на строки, но эти ссылки принадлежат вызывающему коду, а не объекту класса `CipherComp`.

Компиляция компонента `CipherLib`

Компонент должен быть скомпилирован с получением `dll`-, а не `exe`-файла. Если вы работаете в среде разработки Visual Studio IDE, то для построения компонента `CipherLib` необходимо создать проект библиотеки классов (Class Library project). Если вы предпочитаете работать с компилятором командной строки, задайте опцию

/t:library. Например, чтобы скомпилировать компонент CipherLib, используйте следующую командную строку:

```
csc /t:library CipherLib.cs
```

В результате выполнения этой команды будет создан файл CipherLib.dll, содержащий компонент CipherComp.

Клиент, использующий компонент CipherComp

После создания компонент "готов к употреблению". Например, следующая программа является клиентом компонента CipherComp, который она использует для шифрования и дешифрирования символьной строки.

```
// Клиент, который использует компонент CipherComp.

using System;
using CipherLib; // Импортируем пространство имен
                 // компонента CipherComp.

class CipherCompClient {
  public static void Main() {
    CipherComp cc = new CipherComp();

    string text = "Это простой тест";

    string ciphertext = cc.Encode(text);
    Console.WriteLine(ciphertext);

    string plaintext = cc.Decode(ciphertext);
    Console.WriteLine(plaintext);

    cc.Dispose(); // Освобождаем ресурсы.
  }
}
```

Заметьте, что клиент включает пространство имен компонента CipherLib. Благодаря этому компонент CipherComp попадает в "поле зрения" клиента. Можно было бы полностью определять каждую ссылку на компонент CipherComp, но включение его пространства имен упрощает работу с компонентом. Во всем остальном компонент CipherComp используется подобно любому другому классу.

Обратите внимание на обращение к методу Dispose() в конце программы. Как разъяснялось выше, посредством вызова метода Dispose() клиент освобождает ресурсы, которые использовал компонент. Компоненты, подобно другим объектам C#, используют один и тот же механизм сбора мусора, который выполняется спорадически. Однако вызов метода Dispose() заставляет компонент немедленно освободить свои ресурсы. Это очень важно в некоторых ситуациях, например, когда компонент удерживает такой ограниченный ресурс, как подключение к сети. Поскольку компонент CipherComp не занимает ресурсы, вызов метода Dispose() в этом примере не актуален. Но так как метод Dispose() является частью контракта компонентной модели, имеет смысл всегда вызывать его при завершении работы с компонентом.

Чтобы скомпилировать программу-клиента, необходимо сообщить компилятору о том, что он должен обратиться к компоненту CipherLib.dll. Для этого используется опция /r. Например, нашу программу client можно скомпилировать с помощью следующей командной строки:

```
csc /r:CipherLib.dll client.cs
```

При использовании среды разработки Visual Studio IDE необходимо для программы-клиента добавить компонент CipherLib.dll как ссылку.

При выполнении программы client получаем следующие результаты:

```
Юуп!рсптупк!ужту
Это простой тест
```

Переопределение метода Dispose()

Представленная в предыдущем разделе версия простейшего компонента CipherComp не занимает системных ресурсов, не создает и не хранит объектов. Поэтому у нас не было серьезных оснований для переопределения метода Dispose(bool). Но если компонент использует системные ресурсы, метод Dispose(bool) нужно переопределять, чтобы эти ресурсы можно было освободить "в принудительном порядке". К счастью, сделать это совсем нетрудно.

Но сначала попробуем разобраться в том, почему иногда необходимо заставить компонент освободить занимаемые им ресурсы, а не полагаться на системный механизм сбора мусора. Как разъяснялось выше в этой книге, процесс сбора мусора — недетерминированное действие. Оно выполняется по системной необходимости, а не потому, что какие-то объекты уже не нужны. Следовательно, если компонент удерживает некоторый ресурс, например, открытый файл, который нужно освободить, чтобы им могла воспользоваться другая программа, должен существовать способ принудительного освобождения ресурса, приводимого в действие, когда клиент завершит использование компонента. Одно лишь удаление ссылок на компонент не решает проблему, поскольку сам компонент будет удерживать ссылку на свой ресурс до тех пор, пока не наступит следующий сеанс сбора мусора. Выход из этой ситуации есть: компонент должен реализовать метод Dispose(bool).

Для переопределения метода Dispose(bool) необходимо следовать следующим правилам.

1. При вызове метода Dispose(bool) с аргументом true переопределяемая версия метода должна освобождать все связанные с компонентом ресурсы — как управляемые, так и неуправляемые. При вызове этого метода с аргументом false переопределяемая версия метода должна освобождать только неуправляемые ресурсы, если таковые имеются.
2. Метод Dispose(bool) должен вызываться многократно, не создавая при этом никаких проблем.
3. Метод Dispose(bool) должен вызывать метод Dispose(bool), реализованный в базовом классе.
4. Деструктор, созданный для компонента, должен вызывать метод Dispose(false).

Чтобы выполнить правило 2, компонент должен отслеживать момент своего освобождения. Обычно это реализуется поддержкой private-поля, которое служит индикатором состояния.

Рассмотрим схематично представленный компонент, который реализует метод Dispose(bool).

```
// Схематичная реализация компонента,
// который использует метод Dispose(bool).
class MyComp : Component {
  bool isDisposed; // true, если компонент освобождается

  public MyComp {
```

```
    isDisposed = false;
    // ...
  }

  ~MyComp() {
    Dispose(false);
  }

  protected override void Dispose(bool dispAll) {
    if(!isDisposed) {
      if(dispAll) {
        // Освобождаем управляемые ресурсы.
        isDisposed = true; // Устанавливаем компонент
                           // в состояние освобождения.
      }
      // Освобождаем неуправляемые ресурсы.
      base.Dispose(dispAll);
    }
  }
}
```

При вызове метода `Dispose()` для конкретного экземпляра компонента автоматически вызывается метод `Dispose(bool)`, чтобы освободить любые ресурсы, которые он занимает.

Демонстрация использования метода `Dispose(bool)`

Чтобы проиллюстрировать использование метода `Dispose(bool)`, усовершенствуем компонент `CipherComp` так, чтобы в нем велся журнал регистрации всех операций шифрования. Поэтому позаботимся о том, чтобы компонент записывал в файл результат каждого вызова метода `Encode()` или `Decode()`. Эти дополнительные действия будут незаметны для пользователя компонента `CipherComp`. На метод `Dispose(bool)` возлагается обязанность закрыть файл, когда этот компонент больше не нужен. Чтобы показать, когда и как вызывается метод `Dispose(bool)`, используются обращения к методу `WriteLine()`.

```
// Улучшенный вариант компонента шифрования, в котором
// поддерживается Файловый журнал регистрации всех операций.

using System;
using System.ComponentModel;
using System.IO;

namespace CipherLib {

  // Компонент шифрования, который поддерживает
  // журнал регистрации.
  public class CipherComp : Component {
    static int useID = 0;
    int id; // Идентификационный номер экземпляра.
    bool isDisposed; // true, если компонент освобождается
    FileStream log;

    // Конструктор
    public CipherComp() {
      isDisposed = false; // Компонент не освобождается.
      try {
```

```
      log = new FileStream("CipherLog" + useID,
                           FileMode.Create);
      id = useID;
      useID++;
    } catch (FileNotFoundException exc) {
      Console.WriteLine(exc);
      log = null;
    }
  }

  // Деструктор
  ~CipherComp() {
     Console.WriteLine("Деструктор для компонента "
                       + id);
     Dispose(false);
  }

  // Шифруем строку. Метод возвращает и
  // сохраняет результат.
  public string Encode(string msg) {

    string temp = "";

    for(int i=0;i < msg.Length; i++)
      temp += (char) (msg[i] + 1);

    // Сохраняем результат шифрования в файле.
    for(int i=0; i < temp.Length; i++)
      log.WriteByte((byte) temp[i]);

    return temp;
  }

  // Дешифруем строку. Метод возвращает и
  // сохраняет результат.
  public string Decode(string msg) {

    string temp = "";

    for(int i=0; i < msg.Length; i++)
      temp += (char) (msg[i] - 1);

    // Сохраняем результат дешифрирования в файле.
    for(int i=0; i < temp.Length; i++)
      log.WriteByte((byte) temp[i]);

    return temp;
  }

  protected override void Dispose(bool dispAll) {
    Console.WriteLine("Dispose(" + dispAll +
                      ") для компонента " + id);

    if(!isDisposed) {
      if(dispAll) {
        Console.WriteLine("Закрытие файла для " +
                          "компонента " + id);
        log.Close(); // Закрываем файл.
```

```
            isDisposed = true;
          }
          // Неуправляемые ресурсы не нужно освобождать.
          base.Dispose(dispAll);
        }
      }
    }
}
```

Рассмотрим подробнее эту версию компонента `CipherComp`. Его код начинается с таких строк:

```
static int useID = 0;
int id; // Идентификационный номер экземпляра.
bool isDisposed; // true, если компонент освобождается
FileStream log;
```

Первое `int`-поле, которое определено как статическое (`static`), предназначено для идентификации каждого экземпляра компонента `CipherComp`. Значение поля `useID` должно включаться в имя файла регистрации, чтобы каждый экземпляр компонента `CipherComp` имел собственный файловый журнал. Поле `id` содержит идентификационный номер компонента, который совпадает со значением поля `useID` в момент создания компонента. Поле `isDisposed` означает, освобожден ли компонент. Четвертое поле (`log`) представляет собой ссылку типа `FileStream`, которая будет указывать на соответствующий файл регистрации.

Теперь рассмотрим конструктор класса `CipherComp`:

```
public CipherComp() {
      isDisposed = false; // Компонент не освобождается.
      try {
        log = new FileStream("CipherLog" + useID,
                             FileMode.Create);
        id = useID;
        useID++;
      } catch (FileNotFoundException exc) {
        Console.WriteLine(exc);
        log = null;
      }
    }
```

В этом конструкторе переменная `isDisposed` инициализируется значением `false`, значит, объект компонента `CipherComp` используется клиентом. Затем открывается файл регистрации. Обратите внимание на то, что имя открываемого файла представляет собой конкатенацию имени "CipherLog" и строкового представления значения `useID`. После этого значение `useID` присваивается переменной `id` и инкрементируется. Таким образом, каждый экземпляр компонента `CipherComp` будет обеспечен отдельным файлом регистрации, а каждый компонент — уникальным идентификационным номером (ID). Здесь важно отметить, что при создании объекта типа `CipherComp` открывается файл, который является системным ресурсом и который необходимо освободить после того, как соответствующий компонент перестанет быть нужным. Однако клиент не в состоянии напрямую освободить этот файл (клиент даже не "знает" о том, что файл был открыт). Следовательно, закрыть файл должен метод `Dispose(bool)`.

Метод `Encode()` шифрует принятый строковый аргумент и возвращает результат, предварительно записав его в файл регистрации. Поскольку `log`-файл остается открытым, при повторных обращениях к методу `Encode()` в него будут добавляться очередные зашифрованные строки. Например, если использовать метод `Encode()` для шифрования трех различных строк, файл регистрации будет содержать все три зашифро-

ванных строки. Метод Decode() работает подобным образом, за исключением того, что дешифрует значение своего аргумента.

Теперь рассмотрим метод Dispose(bool), переопределенный компонентом CipherComp. Для удобства приведем его код здесь еще раз:

```
protected override void Dispose(bool dispAll) {
  Console.WriteLine("Dispose(" + dispAll +
                    ") для компонента " + id);

  if(!isDisposed) {
    if(dispAll) {
      Console.WriteLine("Закрытие файла для " +
                        "компонента " + id);
      log.Close(); // Закрываем файл.
      isDisposed = true;
    }
    // Неуправляемые ресурсы не нужно освобождать.
    base.Dispose(dispAll);
  }
}
```

Обратите внимание на то, что Dispose(bool) определен как protected-метод. Это означает, что его нельзя вызвать из кода клиента. Вместо него в программе клиента вызывается общедоступный метод Dispose(), который реализован в классе Component. В методе Dispose(bool) проверяется значение переменной isDisposed. Если объект уже освобожден, никакие действия не выполняются. Если переменная isDisposed содержит значение false, проверяется значение параметра dispAll. Если оно равно значению true, файл регистрации закрывается, и переменная isDisposed устанавливается равной значению true. Вспомните: если параметр dispAll равен true, все ресурсы должны быть освобождены. Если параметр dispAll равен false, освобождению подлежат только неуправляемые ресурсы (которые в данном случае не задействованы). Наконец, вызывается метод Dispose(bool), реализованный базовым классом (в данном случае им является класс Component). Это гарантирует освобождение любых ресурсов, используемых базовым классом. Обращения к методу WriteLine() здесь используются только в целях иллюстрации, а в реальных приложениях обычно обходятся без них.

Теперь рассмотрим деструктор компонента CipherComp:

```
~CipherComp() {
  Console.WriteLine("Деструктор для компонента "
                    + id);
  Dispose(false);
}
```

Деструктор просто вызывает метод Dispose(bool) с аргументом false. И это вполне понятно: при выполнении деструктора для компонента предполагается, что этот компонент должен быть утилизирован системой сбора мусора. В данном случае автоматически будут освобождены все управляемые ресурсы. Деструктору остается лишь позаботиться об освобождении неуправляемых ресурсов. Обращение к методу WriteLine() здесь используется только в целях иллюстрации, а в реальных приложениях обычно обходятся без него.

Поскольку изменения, внесенные в код компонента CipherComp, не коснулись его интерфейса, мы можем использовать прежний код клиента. Но здесь для примера взята программа-клиент, в которой шифруются и дешифруются две строки:

```
// Еще один вариант программы-клиента, в которой
// используется компонент CipherComp.
```

```
using System;
using CipherLib; // Импортируем пространство имен
                 // компонента CipherComp.

class CipherCompClient {
  public static void Main() {
    CipherComp cc = new CipherComp();

    string text = "Тестирование";

    string ciphertext = cc.Encode(text);
    Console.WriteLine(ciphertext);

    string plaintext = cc.Decode(ciphertext);
    Console.WriteLine(plaintext);

    text = "Компоненты - мощное средство языка C#.";

    ciphertext = cc.Encode(text);
    Console.WriteLine(ciphertext);

    plaintext = cc.Decode(ciphertext);
    Console.WriteLine(plaintext);

    cc.Dispose(); // Освобождаем ресурсы.
  }
}
```

Результаты выполнения этой программы выглядят так:

```
Ужтуйспгбойж
Тестирование
Лпнрпожоуь!.!нпъопж!тсжетугп!?иьлб!D$/
Компоненты - мощное средство языка C#.
Dispose(True) для компонента 0
Закрытие файла для компонента 0
```

После выполнения этой программы файл регистрации `CipherLog0` будет иметь следующее содержимое:

```
УжтуйспгбойжТестированиеЛпнрпожоуь!.!нпъопж!тсжетугп!?иьлб!D$/Компоне
нты - мощное средство языка C#.
```

Эта абракадабра — результат конкатенации двух строк (вернее, их зашифрованных и дешифрованных вариантов).

Обратите внимание на то, что в результатах выполнения программы-клиента метод `Dispose(bool)` вызван с аргументом `true`. Дело в том, что программа вызывает метод `Dispose()` для объекта компонента `CipherComp`. Как разъяснялось выше, метод `Dispose()` затем вызывает метод `Dispose(bool)` с аргументом `true`, что означает гарантированное освобождение всех ресурсов. В качестве эксперимента закомментируйте обращение к методу `Dispose()` в программе-клиенте, а затем скомпилируйте ее и выполните снова. Вы должны получить такие результаты:

```
Ужтуйспгбойж
Тестирование
Лпнрпожоуь!.!нпъопж!тсжетугп!?иьлб!D$/
Компоненты - мощное средство языка C#.
Деструктор для компонента 0
Dispose(False) для компонента 0
Dispose(False) для компонента 0
```

Поскольку обращение к методу Dispose() отсутствовало, компонент CipherComp не был освобожден в явном виде. Но после выполнения программы он, безусловно, был разрушен. Следовательно, при разрушении был вызван его деструктор, что подтверждают результаты выполнения последнего варианта программы, который, в свою очередь, вызвал метод Dispose(bool) с аргументом false. Второй вызов метода Dispose(bool) — это следствие вызова принадлежащей базовому классу версии метода Dispose() из метода Dispose(bool), определенного в компоненте CipherComp. Поэтому метод Dispose(bool) был вызван во второй раз. В данном случае это излишне, но, поскольку метод Dispose() не может "знать", как он был вызван, такое излишество неизбежно, но совершенно безвредно.

Аналогичный подход к реализации метода Dispose(bool) можно использовать при создании любого компонента.

Защита освобожденного компонента от использования

Несмотря на то что компонент CipherComp надлежащим образом "убирает за собой", его все же нельзя назвать безукоризненным. Дело в том, что в нем не предусмотрен способ предотвратить попытку клиента использовать уже разрушенный компонент. Например, ничто не может помешать клиенту освободить компонент CipherComp, а затем попытаться вызвать для него метод Encode(). К счастью, эту проблему легко устранить: достаточно позаботиться о том, чтобы при использовании компонент проверял поле isDisposed. Например, рассмотрим более удачный вариант методов Encode() и Decode():

```
// Метод шифрования строки. Возвращает результат и
// сохраняет его в файле.
public string Encode(string msg) {

  // Предотвращаем использование освобожденного компонента.
  if(isDisposed) {
    Console.WriteLine("Ошибка: компонент уже разрушен.");
    return null;
  }

  string temp = "";

  for(int i=0;i < msg.Length; i++)
    temp += (char) (msg[i] + 1);

  // Сохраняем результат в файле.
  for(int i=0; i < temp.Length; i++)
    log.WriteByte((byte) temp[i]);

  return temp;
}

// Метод дешифрирования строки. Возвращает результат и
// сохраняет его в файле.
public string Decode(string msg) {

  // Предотвращаем использование освобожденного компонента.
  if(isDisposed) {
    Console.WriteLine("Ошибка: компонент уже разрушен.");
    return null;
  }
```

```
    string temp = "";

    for(int i=0; i < msg.Length; i++)
      temp += (char) (msg[i] - 1);

    // Сохраняем результат в файле.
    for(int i=0; i < temp.Length; i++)
      log.WriteByte((byte) temp[i]);

    return temp;
  }
```

В обоих методах проверяется значение поля `isDisposed`, и если оно равно `true`, отображается сообщение об ошибке и никакие действия не выполняются. Но в реальных приложениях при попытке использовать уже разрушенный компонент обычно генерируется исключение.

Использование инструкции `using`

Как разъяснялось в главе 18, инструкцию `using` можно использовать для автоматического освобождения объекта. В этом случае метод `Dispose()` уже не нужно вызывать в явном виде. Вспомним, что инструкция `using` используется в следующих форматах:

```
using (obj) {
  // Использование объекта obj.
}

using (type obj = initializer) {
  // Использование объекта obj.
}
```

Здесь элемент *obj* представляет объект, используемый внутри блока `using`. В первой форме этот объект объявляется вне `using`-инструкции, а во второй — внутри. При завершении блока для объекта *obj* автоматически вызывается метод `Dispose()` (определенный в интерфейсе `System.IDisposable`). Инструкция `using` применяется только к объектам, которые реализуют интерфейс `System.IDisposable` (что, безусловно, относится ко всем компонентам).

Рассмотрим программу-клиент, которая использует компонент `CipherComp` и для его освобождения вместо прямого обращения к методу `Dispose()` применяет инструкцию `using`:

```
// Использование инструкции using.

using System;
using CipherLib; // Импортируем пространство имен
                 // компонента CipherComp.

class CipherCompClient {
  public static void Main() {

    // Объект cc разрушится по завершении этого блока.
    using(CipherComp cc = new CipherComp()) {

      string text = "Инструкция using.";
```

```
        string ciphertext = cc.Encode(text);
        Console.WriteLine(ciphertext);

        string plaintext = cc.Decode(ciphertext);
        Console.WriteLine(plaintext);
    }

  }
}
```

Результаты выполнения этой программы таковы:

```
Йотусфлчй?!vtjoh/
Инструкция using.
Dispose(True) для компонента 0
Закрытие файла для компонента 0
```

Как подтверждают результаты выполнения этой программы-клиента, метод `Dispose()` был вызван автоматически по завершении программного блока. Итак, решение остается за вами: использовать инструкцию `using` или явно вызывать метод `Dispose()`, но вы должны знать, что инструкция `using` упрощает код.

Контейнеры

Используя компоненты, иногда стоит сохранять их в контейнере. Как разъяснялось выше, контейнер определяет группу компонентов. Основное достоинство контейнера состоит в том, что он позволяет управлять целой группой компонентов. Например, вызвав метод `Dispose()` для контейнера, можно освободить сразу все содержащиеся в нем компоненты. В общем случае контейнеры значительно упрощают работу с множеством компонентов.

Чтобы создать контейнер, необходимо создать объект класса `Container`, который определен в пространстве имен `System.ComponentModel`. В классе `Container` определен такой конструктор:

```
public Container()
```

Он создает пустой контейнер.

В уже созданный объект типа `Container` можно добавлять компоненты с помощью метода `Add()`, который используется в двух форматах:

```
public virtual void Add(IComponent comp)
public virtual void Add(IComponent comp,
                        string compName)
```

Первый формат позволяет добавить в контейнер компонент, заданный параметром *comp*. Второй формат не только добавляет в контейнер компонент, заданный параметром *comp*, но и присваивает ему имя, заданное параметром *compName*. Это имя должно быть уникальным с учетом того, что различия между прописными и строчными буквами здесь игнорируются. При попытке использовать имя, которое уже принадлежит компоненту в контейнере, генерируется исключение типа `ArgumentException`. Имя компонента можно получить с помощью свойства `Site`.

Чтобы удалить компонент из контейнера, используйте метод `Remove()`:

```
public virtual void Remove(IComponent comp)
```

Предполагается, что этот метод успешно выполнится в любом случае: либо когда он действительно удалит компонент, заданный параметром *comp*, либо когда подлежащий удалению компонент в контейнере отсутствует. Другими словами, после вызова метода `Remove()` компонента *comp* в контейнере не будет.

Класс Container реализует метод Dispose(). При вызове для контейнера метода Dispose() для всех компонентов, хранимых в этом контейнере, будут вызваны их методы Dispose(). Таким образом, освободить все компоненты, содержащиеся в контейнере, можно одним-единственным вызовом метода Dispose().

В классе Container определено одно свойство Components:

```
public virtual ComponentCollection Components { get; }
```

Это свойство получает коллекцию компонентов, которые хранятся в вызывающем контейнере.

Вспомните, что в классе Component определены свойства Container и Site, которые включаются во все производные компоненты. При сохранении компонента в контейнере свойства Container и Site (относящиеся к объекту компонента) устанавливаются автоматически. Свойство Container называет контейнер, в котором содержится компонент. Свойство Site позволяет получить такие данные о компоненте, как его имя, имя его контейнера и признак режима разработки. Свойство Site возвращает ссылку типа ISite. Интерфейс ISite определяет следующие свойства:

Свойство	*Описание*
`IComponent Component { get; }`	Получает ссылку на компонент
`IContainer Container { get; }`	Получает ссылку на контейнер
`bool DesignMode { get; }`	Возвращает значение `true`, если компонент используется в режиме разработки
`string Name { get; set; }`	Получает или устанавливает имя компонента

Эти свойства можно использовать для получения информации о контейнере или компоненте во время выполнения программы.

Использование контейнера

В следующей программе контейнер используется для хранения двух компонентов типа CipherComp. Первый добавляется в контейнер без задания имени, а второй — с именем "Второй компонент". После выполнения операций над обоими компонентами с помощью свойства Site на экране отображается имя второго компонента. Наконец, для контейнера вызывается метод Dispose(), который освобождает оба компонента. (Конечно же, здесь было бы уместно применить инструкцию using, но в демонстрационных целях показан явный вызов метода Dispose().)

```
// Демонстрация использования контейнера
// для хранения компонентов.

using System;
using System.ComponentModel;
using CipherLib; // Импортируем пространство имен
                 // компонента CipherComp.

class UseContainer {
  public static void Main(string[] args) {
    string str = "Использование контейнеров.";
    Container cont = new Container();

    CipherComp cc = new CipherComp();
    CipherComp cc2 = new CipherComp();
```

```
    cont.Add(cc);
    cont.Add(cc2, "Второй компонент");

    Console.WriteLine("Первое сообщение: " +  str);
    str = cc.Encode(str);
    Console.WriteLine(
            "Первое сообщение в зашифрованном виде: " +
            str);

    str = cc.Decode(str);
    Console.WriteLine(
            "Первое сообщение в дешифрованном виде: " +
            str);

    str = "один, два, три";
    Console.WriteLine("Второе сообщение: " +  str);

    str = cc2.Encode(str);
    Console.WriteLine(
            "Второе сообщение в зашифрованном виде: " +
            str);

    str = cc2.Decode(str);
    Console.WriteLine(
            "Второе сообщение в дешифрованном виде: " +
            str);

    Console.WriteLine("\nИмя объекта cc2: " +
                      cc2.Site.Name);

    Console.WriteLine();

    // Освобождаем оба компонента.
    cont.Dispose();
  }
}
```

Результаты выполнения этой программы клиента таковы:

```
Первое сообщение: Использование контейнеров.
Первое сообщение в зашифрованном виде: Йтрпмэипгбойж!лпоужкожспг/
Первое сообщение в дешифрованном виде: Использование контейнеров.
Второе сообщение: один, два, три
Второе сообщение в зашифрованном виде: пейо-!егб-!усй
Второе сообщение в дешифрованном виде: один, два, три

Имя объекта cc2: Второй компонент

Dispose(True) для компонента 1
Закрытие файла для компонента 1
Dispose(True) для компонента 0
Закрытие файла для компонента 0
```

Как вы убедились, при вызове метода `Dispose()` для контейнера все хранимые в нем компоненты освобождаются. В этом и состоит основное достоинство работы с несколькими компонентами или экземплярами компонентов.

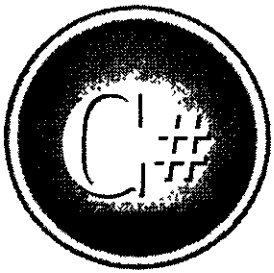
Компоненты — это будущее программирования

Организация приложения в виде набора компонентов — это мощное средство программирования, позволяющее программисту справляться со все более сложными задачами. Программисты в начале своей деятельности замечают, что чем больше программа, тем дольше период ее отладки. С увеличением размера программы обычно растет и ее сложность, но известно, что существует некоторый предел сложности, с которым может справиться человек. С точки зрения чистой комбинаторики, чем больше в программе отдельных строк, тем больше шансов получить побочные эффекты и нежелательные взаимосвязи.

Программные компоненты помогают справиться со сложностью программ по принципу "разделяй и властвуй". Путем разделения программы на независимые компоненты программист может понизить видимый уровень ее сложности. При компонентно-ориентированном подходе программа организуется как набор строго определенных "строительных блоков" (компонентов), которые можно использовать, не вникая в детали их внутренней реализации. Суммарный эффект такого подхода состоит в снижении общей сложности программы. Сам собой напрашивается логический вывод: приложение может состоять только из одних компонентов, связанных между собой таким образом, что один компонент поставляет "питание" для другого. Такую организацию программ можно назвать компонентно-ориентированным программированием.

При такой мощности компонентов и простоте их создания в C# на вопрос: "Компоненты — это будущее программирования?" многие программисты отвечают без колебаний: "Да!"

Полный справочник по



Глава 25

Создание Windows-приложений

Большинство программ, приведенных в этой книге, представляют собой консольные приложения, т.е. приложения, запускаемые из командной строки (и не имеющие графического интерфейса). Консольные приложения прекрасно подходят для демонстрации элементов языка C# и вполне годятся для некоторых типов обслуживающих программ, которые, например, фильтруют файлы. Конечно же, современные приложения в большинстве предназначены для работы в Windows GUI-среде (*g*raphics *u*ser *i*nterface — графический интерфейс пользователя), и эта книга была бы неполной, если не показать в ней, как использовать C# для создания Windows-приложений.

Еще в недалеком прошлом создание Windows-приложений было под силу лишь опытным программистам и внушало ужас новичкам, которым зачастую приходилось в течение нескольких недель осваивать лишь основные элементы построения Windows-приложений. К счастью, C# и среда .NET Framework изменили ситуацию коренным образом. В .NET-библиотеке содержится полная подсистема, которая поддерживает библиотеку Forms, что значительно упрощает создание Windows-программ. С использованием языка C# и библиотеки `System.Windows.Forms` Windows-приложения создавать теперь намного проще.

Windows-программирование — очень большая тема, которой посвящено немало книг. Очевидно, описать все его аспекты в одной главе попросту нереально. Поэтому настоящую главу следует рассматривать как "высокий старт" в направлении Windows-программирования. Здесь вы узнаете, как создать окно, меню и кнопку, а также ответить на сообщение. Прочитав эту главу, вы будете готовы освоить и другие аспекты Windows-программирования, основанного на применении окон.

Краткий экскурс в историю Windows-программирования

Чтобы оценить достоинства языка C# и среды .NET Framework применительно к Windows-программированию, необходимо слегка погрузиться в его историю. Сразу после создания Windows программы взаимодействовали непосредственно с Windows API-интерфейсом (Application Programming Interface — программный интерфейс приложения), представляющим собой большой набор методов, которые могут вызываться программами для доступа к функциям Windows. API-ориентированные программы отличаются большим размером и сложностью. Длина API-ориентированных программ, которые выполняют отдельные функции, измеряется сотнями строк, а реальных приложений — тысячами. Теперь вы можете представить себе, насколько Windows-программы было трудно писать и поддерживать.

Для решения этой проблемы были созданы библиотеки классов, которые инкапсулировали функции API-интерфейса. Возможно, многие читатели уже знакомы с библиотекой MFC (Microsoft Foundation Classes — библиотека базовых классов Microsoft). Она написана на C++; MFC-ориентированные программы также использовали язык C++. Благодаря объектно-ориентированной направленности библиотеки MFC процесс создания Windows-программ заметно упростился. Однако MFC-программы все равно отличались повышенной сложностью, т.е. включали заголовочные файлы, файлы с исходным кодом и файлы ресурсов. Более того, библиотека MFC была по сути лишь "тонкой оболочкой", в которую помещался API-интерфейс, поэтому для реализации Windows-ориентированного поведения приложений по-прежнему требовалось внушительное количество программных инструкций.

Язык C# и .NET Framework-библиотека Forms предлагают полностью объектно-ориентированный подход к Windows-программированию. Вместо обеспечения лишь оболочки для API библиотека Forms определяет простой, интегрированный и логически непротиворечивый способ управления разработкой Windows-приложений. Этот уровень интеграции стал возможным лишь благодаря таким уникальным средствам языка C#, как делегат и события. Более того, благодаря специфическому в C# использованию системы сбора мусора, практически устранена проблема "утечки памяти".

Если вы уже программировали с использованием API-интерфейса или библиотеки MFC, то убедитесь в том, что C#-решения относятся к более высокому уровню. Впервые (с момента создания операционной системы Windows) создание Windows-приложений стало таким же простым, как и создание консольных приложений.

Два способа создания Windows-приложений, основанных на применении окон

Для начала необходимо остановиться вот на чем. Пакет Visual Studio включает обширный набор средств разработки, которые автоматизируют большую часть процесса создания Windows-приложения. С помощью этих средств можно создавать и помещать в нужное место различные элементы управления и меню, используемые приложением. Visual Studio помогает создавать классы и методы, которые необходимы для каждого управляющего элемента. Средства разработки Visual Studio — очень удобный инструмент для создания большинства Windows-приложений, хотя и не единственно возможный. Windows-программы можно также создавать с помощью текстового редактора с последующей компиляцией исходного кода подобно тому, как вы это делали с консольными приложениями.

Поскольку книга посвящена языку C#, а не Visual Studio, да и Windows-программы, представленные в этой главе, довольно невелики по размеру, их создание показано здесь в форме, которая подходит для использования текстового редактора. Но общая структура, разработка и организация программ остаются такими же, как и при использовании автоматизированных средств разработки. Таким образом, материал этой главы применим к любому способу создания программ.

Как Windows взаимодействует с пользователем

Преде чем приступать к Windows-программированию, необходимо понять, как пользователь взаимодействует с Windows, поскольку именно этот фактор определяет архитектуру всех Windows-программ. Это взаимодействие в корне отличается от взаимодействия, реализованного в консольных программах, представленных в других частях этой книги. В случае консольной программы именно ваша программа инициирует взаимодействие с операционной системой. Примером может служить программа, запрашивающая входные данные и выводящая результаты путем вызова методов `Read()` или `WriteLine()`. Таким образом, программы, написанные "традиционным способом", сами обращаются к операционной системе, а не операционная система к ним. Но в отношении "своих" программ Windows предполагает совсем иную модель отношений: именно Windows должна обращаться к вашей программе. Процесс взаимодействия организован следующим образом: программа ожидает до тех пор, пока не получит сообщение от Windows. Получив его, программа должна предпринять соответствующее действие. Отвечая на сообщение, она может вызвать метод, определенный в Windows, но главное здесь то, что инициатором взаимодействия все-таки является

Windows. Таким образом, общий формат всех Windows-программ продиктован *механизмом сообщений*, который и лежит в основе взаимодействия с Windows.

Существует множество различных сообщений, которые Windows может послать программе. Например, при каждом щелчке кнопкой мыши в окне вашей программы будет послано сообщение, связанное со щелчком кнопкой мыши. При щелчке на электронной кнопке будет послано сообщение другого типа, а при выборе элемента меню — третего. Здесь необходимо понимать следующее: с точки зрения программы сообщения поступают случайным образом. Вот почему Windows-программы напоминают программы, управляемые прерываниями. Вы сами не знаете, каким будет следующее сообщение.

Windows-формы

Ядром Windows-программ, написанных на C#, является форма. Форма инкапсулирует основные функции, необходимые для создания окна, его отображения на экране и получения сообщений. Форма может представлять собой окно любого типа, включая основное окно приложения, дочернее или даже диалоговое окно.

Первоначально окно создается пустым. Затем в него добавляются меню и элементы управления, например экранные кнопки, списки и флажки. Таким образом, форму можно представить в виде контейнера для других Windows-объектов.

Когда окну посылается сообщение, оно преобразуется в событие. Следовательно, чтобы обработать Windows-сообщение, достаточно для него зарегистрировать обработчик событий. При получении этого сообщения обработчик событий будет вызываться автоматически.

Класс `Form`

Форма создается посредством реализации объекта класса `Form` или класса, производного от `Form`. Класс `Form` помимо поведения, обусловленого собственными членами, демонстрирует поведение, унаследованное от предков. Среди его базовых классов выделяются своей значимостью классы `System.ComponentModel.Component` (см. главу 24) и `System.Windows.Forms.Control`. Класс `Control` определяет черты, присущие всем Windows-элементам управления. Тот факт, что класс `Form` выведен из класса `Control`, позволяет использовать формы для создания элементов управления. Использование некоторых членов классов `Form` и `Control` демонстрируется в приведенных ниже примерах.

Схематичная Windows-программа, основанная на применении окон

Начнем с создания простейшего Windows-приложения, основанного на применении окон. Это приложение создает и отображает окно, которое не содержит других элементов управления. Тем не менее оно позволяет показать действия, необходимые для построения полнофункционального окна. Это окно представляет собой стартовую площадку, на которой можно построить большинство Windows-приложений. Итак, рассмотрим первую Windows-программу:

```
// Простейшее Windows-приложение, основанное
// на применении окон.
```

```
using System;
using System.Windows.Forms;

// Класс WinSkel - производный от класса Form.
class WinSkel : Form {

  public WinSkel() {
    // Присваиваем окну имя.
    Text = "Рама Windows-окна";
  }

  // Метод Main() используется только для запуска приложения.
  [STAThread]
  public static void Main() {
    WinSkel skel = new WinSkel(); // Создаем форму.

    // Запускаем механизм функционирования окна.
    Application.Run(skel);
  }
}
```

Окно, созданное этой программой, показано на рис. 25.1.

Рис. 25.1. Окно, созданное программой `WinSkel`

Рассмотрим эту программу построчно. Прежде всего обратите внимание на то, что она включает два пространства имен: `System` и `System.Windows.Forms`. Пространство имен `System` необходимо для использования атрибута `STAThread`, который предшествует методу `Main()`, а `System.Windows.Forms` предназначено для поддержки подсистемы Windows Forms.

Затем создается класс `WinSkel`, который наследует класс `Form`. Следовательно, класс `WinSkel` определяет тип формы. В данном случае это самая простая (минимальная) форма.

В теле конструктора класса `WinSkel` содержится только одна строка кода:

```
Text = "Рама Windows-окна";
```

Здесь устанавливается свойство `Text`, которое содержит название окна. Таким образом, после выполнения инструкции присвоения строка заголовка окна будет содержать текст `Рама Windows-окна`. Свойство `Text` определяется так:

```
public virtual string Text { get; set; }
```

Свойство `Text` унаследовано от класса `Control`.

Теперь рассмотрим метод `Main()`, который объявлен подобно другим методам `Main()`, входящим в состав программ этой главы. С этого метода, как вы уже знаете,

начинается выполнение программы. Однако заметьте, что заголовку метода `Main()` предшествует свойство `STAThread`. Microsoft заявляет, что это свойство должен иметь метод `Main()` в каждой Windows-программе. Свойство `STAThread` устанавливает модель организации поточной обработки (threading model), а именно модель с однопоточным управлением, т.е. такую обработку данных, когда все объекты выполняются в едином процессе (single-threaded apartment — STA). Рассмотрение моделей организации поточной обработки выходит за рамки этой главы, но вкратце заметим, что Windows-приложение может использовать одну из двух моделей: с однопоточным или многопоточным управлением.

В методе `Main()` создается объект класса `WinSkel` с именем `skel`. Этот объект затем передается методу `Run()`, определенному в классе `Application`:

```
Application.Run(skel);
```

Эта инструкция запускает механизм функционирования окна. Класс `Application` определяется в пространстве имен `System.Windows.Forms` и инкапсулирует аспекты, присущие всем Windows-приложениям. Вот как определяется используемый здесь метод `Run()`:

```
public static void Run(Form ob)
```

В качестве параметра он принимает ссылку на форму. Поскольку класс `WinSkel` выведен из класса `Form`, объект типа `WinSkel` можно передать методу `Run()`.

Эта программа при выполнении создает окно, показанное на рис. 25.1. Оно имеет стандартный размер (300 пикселей по ширине и 300 пикселей по высоте). Это окно полностью функционально. Можно изменить его размеры, переместить, свернуть, восстановить и закрыть. Таким образом, основные свойства, присущие практически всем окнам, были достигнуты написанием всего нескольких строк программного кода. Для сравнения: такая же программа, написанная на языке C и напрямую вызывающая интерфейс Windows API, потребовала бы приблизительно в пять раз больше программных строк!

Предыдущий пример продемонстрировал основные принципы создания Windows-приложений, основанных на применении окон. Итак, чтобы создать форму, создайте класс, производный от класса `Form`. Затем инициализируйте эту форму в соответствии с требованиями приложения, создайте объект производного класса и вызовите метод `Application.Run()` для этого объекта.

Компиляция первой Windows-программы

Windows-программу можно скомпилировать, используя либо компилятор командной строки, либо среду разработки Visual Studio. Для таких коротких программ, как те, что представлены в этой главе, вполне подойдет компилятор командной строки (как самый простой вариант), но для реальных приложений стоит использовать IDE. (Кроме того, как разъяснялось в начале главы, имеет смысл освоить средства разработки, предоставляемые пакетом Visual Studio.) В любом случае здесь описаны оба варианта компиляции Windows-приложений.

Компиляция из командной строки

Назовите нашу первую Windows-программу `WinSkel.cs` и скомпилируйте ее, вызвав компилятор командной строки с помощью следующей команды:

```
csc /t:winexe WinSkel.cs
```

Ключ `/t:winexe` указывает компилятору на создание Windows-приложения, а не консольной программы. Чтобы выполнить программу, достаточно ввести в командную строку текст `WinSkel`.

Компиляция в интегрированной среде разработки (IDE)

Чтобы скомпилировать программу с помощью Visual Studio IDE, сначала создайте новый проект Windows Application. Для этого выберите команду File⇨New⇨Project (Файл⇨Создать⇨Проект). Затем выберите в диалоговом окне New Project (Создать проект) вариант Windows Application (Windows-приложение). Назовите проект `WinSkel`. С этим проектом будет связан файл `Form1.cs`. Удалите его. Затем, щелкнув правой кнопкой мыши на имени проекта WinSkel, выберите из контекстного меню команду Add⇨Add New Item (Добавить⇨Добавить новый элемент). Вы должны увидеть диалоговое окно, показанное на рис. 25.2.

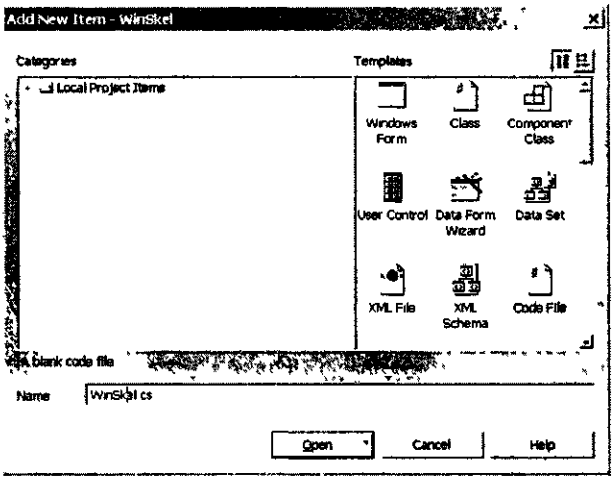


Рис. 25 2. Диалоговое окно Add New Item (Добавить новый элемент)

Выберите на панели Templates (Шаблоны) вариант Code File (Файл с текстом программы) и назовите файл `WinSkel.cs`. Затем введите текст нашей первой Windows-программы и скомпилируйте программу с помощью команды Build⇨Build Solution (Построить⇨Построить решение). Для выполнения программы выберите команду Debug⇨Start Without Debugging (Отладка⇨Начать выполнение без отладки).

Создание кнопки

В общем случае функциональность окна обеспечивается элементами двух типов: элементами управления и меню. Именно с помощью этих элементов и взаимодействует пользователь с программой. Меню описаны ниже в этой главе. Здесь вы узнаете, как поместить в окно элемент управления.

В Windows определены различные типы управляющих элементов, включая экранные кнопки, флажки, переключатели и окна списков. Несмотря на различия между ними, способы их создания примерно одинаковы. В этом разделе мы поместим в окно экранную кнопку, но процедура добавления других типов управляющих элементов будет такой же.

Немного теории

Экранная кнопка инкапсулирована в классе `Button`, который выведен из абстрактного класса `ButtonBase`. Поскольку класс `ButtonBase` предназначен для реализации поведения оконного элемента управления, он наследует класс `Control`. В классе `Button` определен только один конструктор:

```
public Button()
```

Этот конструктор создает кнопку стандартного размера, расположенную внутри окна. Эта кнопка не содержит описания, поэтому, прежде чем использовать ее, необходимо описать ее, присвоив свойству `Text` соответствующую текстовую строку.

Для указания местоположения кнопки в окне необходимо присвоить свойству `Location` координаты ее верхнего левого угла. Свойство `Location` унаследовано от класса `Control` и определяется так:

```
public Point Location { get; set; }
```

Координаты хранятся в структуре `Point`, которая определена в пространстве имен `System.Drawing`. Она включает следующие свойства:

```
public int X { get; set; }
public int Y { get; set; }
```

Таким образом, чтобы создать кнопку с надписью "Щелкните здесь" и привязать ее к точке с координатами 100, 200, используйте следующую последовательность инструкций:

```
Button MyButton = new Button();
MyButton.Text = "Щелкните здесь";
MyButton.Location = new Point(100, 200);
```

Как поместить кнопку на форму

После создания кнопку необходимо поместить на форму. Это реализуется с помощью метода `Add()`, который вызывается для коллекции элементов управления, связанных с формой. Эта коллекция доступна посредством свойства `Controls`, которое унаследовано от класса `Control`. Вот как определяется метод `Add()`:

```
public virtual void Add(Control cntl)
```

Здесь параметр *cntl* означает добавляемый элемент управления. После того как элемент будет добавлен в состав формы, он станет видимым при отображении самой формы.

Простой пример с кнопкой

В следующей программе на созданную ранее форму помещается кнопка. Пока что эта кнопка бездействует, но на ней можно щелкнуть.

```
// Добавление кнопки.

using System;
using System.Windows.Forms;
using System.Drawing;

class ButtonForm : Form {
  Button MyButton = new Button();

  public ButtonForm() {
```

```
    Text = "Использование кнопки";

    MyButton = new Button();
    MyButton.Text = "Щелкните";
    MyButton.Location = new Point(100, 200);

    Controls.Add(MyButton);
  }

  [STAThread]
  public static void Main() {
    ButtonForm skel = new ButtonForm();

    Application.Run(skel);
  }
}
```

В этой программе создается класс `ButtonForm`, который является производным от класса `Form`. Он содержит поле типа `Button` с именем `MyButton`. В конструкторе класса `ButtonForm` кнопка создается, инициализируется и помещается на форму. При выполнении этой программы отображается окно, показанное на рис. 25.3. Вы можете щелкнуть на кнопке, но ничего не произойдет. Чтобы заставить кнопку выполнять какие-либо действия, необходимо добавить в программу обработчик сообщений, как описано в следующем разделе.

Рис. 25.3. Кнопка на форме

Обработка сообщений

Чтобы программа реагировала на щелчок на кнопке (или на какое-либо другое действие пользователя), необходимо обеспечить обработку сообщения, которое генерирует эта кнопка. В общем случае, когда пользователь воздействует на элемент управления, его действие передается программе в виде сообщения. В C#-программе, основанной на применении окон, такие сообщения обрабатываются обработчиками событий. Следовательно, чтобы получить сообщение, программа должна добавить собственный обработчик событий в список обработчиков, вызываемых при генерировании сообщения. Для сообщений, связанных со щелчком на кнопке, это означает добавление обработчика для события `Click`.

Событие `Click` определяется в классе `Button`. (Событие `Click` унаследовано от класса `Control`.) Его общий формат таков:

```
public Event EventHandler Click;
```

Делегат `EventHandler` определяется так:

```
public delegate void EventHandler(object who,
                                  EventArgs args)
```

Объект, который сгенерировал событие, передается в параметре *who*, а информация, связанная с этим событием, — в параметре *args*. Для многих событий в качестве параметра *args* будет служить объект класса, выведенного из класса `EventArgs`. Поскольку щелчок на кнопке не требует дополнительной информации, при обработке события щелчка на кнопке не нужно беспокоиться об аргументах этого события.

Следующая программа основана на предыдущей, но с добавлением кода реакции на щелчок. При каждом щелчке на кнопке будет изменяться ее местоположение.

```
// Обработка сообщений от кнопки.

using System;
using System.Windows.Forms;
using System.Drawing;

class ButtonForm : Form {
  Button MyButton = new Button();

  public ButtonForm() {
    Text = "Реакция на щелчок";

    MyButton = new Button();
    MyButton.Text = "Щелкните";
    MyButton.Location = new Point(100, 200);

    // Добавляем в список обработчик событий кнопки.
    MyButton.Click += new EventHandler(MyButtonClick);

    Controls.Add(MyButton);
  }

  [STAThread]
  public static void Main() {
    ButtonForm skel = new ButtonForm();

    Application.Run(skel);
  }

  // Обработчик для кнопки MyButton.
  protected void MyButtonClick(object who, EventArgs e) {

    if(MyButton.Top == 200)
      MyButton.Location = new Point(10, 10);
    else
      MyButton.Location = new Point(100, 200);
  }
}
```

Рассмотрим внимательно код обработчика события щелчка на кнопке:

```
// Обработчик для кнопки MyButton.
protected void MyButtonClick(object who, EventArgs e) {

  if(MyButton.Top == 200)
    MyButton.Location = new Point(10, 10);
  else
    MyButton.Location = new Point(100, 200);
}
```

Обработчик MyButtonClick() использует такую же сигнатуру, как и приведенный выше делегат EventHandler, а это значит, что обработчик можно добавить в цепочку обработчиков для события Click. Обратите внимание на то, что в его определении использован модификатор типа protected. И хотя это не является обязательным требованием, такая модификация имеет смысл, поскольку обработчики событий не предназначены для вызова кодом, а используются лишь для ответа на события.

В коде обработчика координаты верхней границы кнопки определяются с помощью свойства Top. Следующие свойства определяются для всех элементов управления (они задают координаты верхнего левого и нижнего правого углов):

```
public int Top { get; set; }
public int Bottom { get; }
public int Left { get; set; }
public int Right { get; }
```

Обратите внимание на то, что местоположение элемента управления можно изменить с помощью свойств Top и Left, но не свойств Bottom и Right, так как последние предназначены только для чтения. (Для изменения размера элемента управления можно использовать свойства Width и Height.)

При получении события, связанного со щелчком на кнопке, проверяется координата ее верхней границы, и, если она равна начальному значению 200, для кнопки устанавливаются новые координаты: 10, 10. В противном случае кнопка возвращается в исходное положение с координатами 100, 200. Поэтому при каждом щелчке на кнопке ее местоположение меняется.

Прежде чем обработчик MyButtonClick() сможет получать сообщения, его необходимо добавить в цепочку обработчиков событий, связанную с событием Click. Это реализуется в конструкторе класса ButtonForm с помощью такой инструкции:

```
MyButton.Click += new EventHandler(MyButtonClick);
```

После выполнения этой инструкции при каждом щелчке на кнопке и будет вызываться обработчик событий MyButtonClick().

Альтернативная реализация

Интересно отметить, что обработчик событий MyButtonClick() можно написать по-другому. Вспомните, что параметр *who* обработчика событий принимает ссылку на объект, который сгенерировал вызов. Когда происходит событие, связанное со щелчком на кнопке, таким объектом является кнопка, на которой был сделан щелчок. Поэтому обработчик MyButtonClick() может иметь такой код:

```
// Альтернативный вариант обработчика событий,
// связанных со щелчком на кнопке.
protected void MyButtonClick(object who, EventArgs e) {
  Button b = (Button) who;

  if(b.Top == 200)
    b.Location = new Point(10, 10);
  else
    b.Location = new Point(100, 200);
}
```

В этой версии обработчика значение параметра who приводится к типу Button, и именно эта ссылка (а не поле MyButton) используется для доступа к объекту кнопки. И хотя в данном случае этот вариант не демонстрирует никаких преимуществ перед предыдущим, нетрудно представить ситуации, в которых он окажется более подходящим. Например, такой подход позволяет писать обработчик событий кнопки, который не зависит от конкретной кнопки.

Использование окна сообщений

Одним из самых полезных встроенных средств Windows-приложений является окно сообщений. Как понятно по названию, окно сообщений позволяет отображать сообщения. С его помощью можно также получить от пользователя такие простые ответы (на поставленные вопросы), как Да, Нет или OK. В программе, основанной на применении окон, окно сообщений поддерживается классом `MessageBox`. При этом объект класса создавать не нужно. Для отображения окна сообщений достаточно вызвать определенный в этом классе статический метод `Show()`.

Метод `Show()` используется в нескольких форматах. Один из них выглядит так:

```
public static DialogResult Show(
                                string msg,
                                string caption,
                                MessageBoxButtons mbb)
```

Строка, отображаемая внутри окна, передается в параметре *msg*; заголовок окна сообщения — в параметре *caption*. Кнопки, отображаемые в окне, задаются параметром *mbb*. Метод возвращает ответ пользователя.

`MessageBoxButtons` — это перечисление, которое определяет следующие значения:

`AbortRetryIgnore`	`OK`	`OKCancel`
`RetryCancel`	`YesNo`	`YesNoCancel`

Каждое из этих значений описывает кнопки, которые будут включены в окно сообщений. Например, если параметр *mbb* содержит значение `YesNo`, то в окне сообщений будут отображены кнопки Да и Нет.

Значение, возвращаемое методом `Show()`, означает, какая кнопка нажата пользователем. Это может быть одно из следующих значений:

`Abort`	`Cancel`	`Ignore`	`No`
`None`	`OK`	`Retry`	`Yes`

В своей программе вы можете проверить значение, возвращаемое методом `Show()`, и определить линию поведения, избранную пользователем. Например, если в окне сообщения пользователь предупреждается о возможности перезаписи файла, то ваша программа предотвратит перезапись, если пользователь щелкнет на кнопке Отменить, или выполнит ее, если пользователь щелкнет на OK.

Следующая программа основана на предыдущей, но с добавлением кнопки останова и окна сообщений. В обработчике событий, связанных с кнопкой останова, организовано отображение окна сообщений, в котором пользователю предлагается ответить на вопрос, желает ли он остановить программу. Если пользователь ответит щелчком на кнопке Да, программа остановится. Если же он щелкнет на кнопке Нет, выполнение программы будет продолжено.

```
// Добавление кнопки останова.

using System;
using System.Windows.Forms;
using System.Drawing;

class ButtonForm : Form {
  Button MyButton;
  Button StopButton;

  public ButtonForm() {
    Text = "Добавление кнопки Стоп";
```

```
    // Создаем кнопки.
    MyButton = new Button();
    MyButton.Text = "Щелкните";
    MyButton.Location = new Point(100, 200);

    StopButton = new Button();
    StopButton.Text = "Стоп";
    StopButton.Location = new Point(100, 100);

    // Добавляем обработчики событий.
    MyButton.Click += new EventHandler(MyButtonClick);
    Controls.Add(MyButton);
    StopButton.Click += new EventHandler(StopButtonClick);
    Controls.Add(StopButton);
  }

  [STAThread]
  public static void Main() {
    ButtonForm skel = new ButtonForm();

    Application.Run(skel);
  }

  // Обработчик событий для кнопки MyButton.
  protected void MyButtonClick(object who, EventArgs e) {

    if(MyButton.Top == 200)
      MyButton.Location = new Point(10, 10);
    else
      MyButton.Location = new Point(100, 200);
  }

  // Обработчик событий для кнопки StopButton.
  protected void StopButtonClick(object who, EventArgs e) {

    // Если пользователь ответит щелчком на кнопке Yes,
    // программа будет завершена.
    DialogResult result = MessageBox.Show(
                                  "Остановить программу?",
                                  "Завершение",
                                  MessageBoxButtons.YesNo);

    if(result == DialogResult.Yes) Application.Exit();
  }
}
```

Рассмотрим, как используется окно сообщений. Нетрудно заметить, что в конструктор `ButtonForm` добавлена вторая кнопка. Эта кнопка содержит текст "Стоп", и ее обработчик событий связывается с методом `StopButtonClick()`.

В обработчике `StopButtonClick()` с помощью следующей инструкции отображается окно сообщений:

```
// Если пользователь ответит щелчком на кнопке Yes,
// программа будет завершена.
DialogResult result = MessageBox.Show(
                                "Остановить программу?",
                                "Завершение",
                                MessageBoxButtons.YesNo);
```

Здесь окно сообщения имеет заголовок "Завершение". Внутри этого окна отображается текст "Остановить программу?" и кнопки Да и Нет. Результат выполнения метода `Show()`, который содержит ответ пользователя, присваивается переменной `result`. При выполнении следующей строки кода этот ответ проверяется, и от результата проверки зависит дальнейший ход выполнения программы:

```
if(result == DialogResult.Yes) Application.Exit();
```

Если пользователь щелкнет на кнопке Да, программа остановится. Это реализуется вызовом метода `Application.Exit()`, который обеспечивает немедленное завершение программы. В противном случае никакие действия не выполняются, и программа продолжается.

Результат выполнения этой программы показан на рис. 25.4.

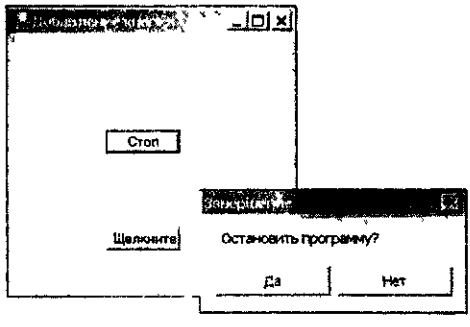


Рис. 25.4. Результат выполнения программы, использующей окно сообщений

Создание меню

Главное окно практически всех Windows-приложений включает меню, расположенное вдоль верхней его границы. Оно называется основным. Основное меню обычно содержит такие категории верхнего уровня, как Файл, Правка и Сервис. Из основного меню можно получить раскрывающиеся меню, содержащие команды, связанные с соответствующей категорией. При выборе элемента меню генерируется сообщение. Следовательно, чтобы обработать команду меню, программа должна присвоить каждому элементу меню обработчик событий.

Основное меню создается путем комбинации двух классов. Первый класс — `MainMenu` — инкапсулирует общую структуру меню, а второй — `MenuItem` — отдельный его элемент. Элемент меню может представлять либо конечное действие (например, Закрыть), либо активизировать другое раскрывающееся меню. Оба класса — `MainMenu` и `MenuItem` — наследуют класс `Menu`. Поскольку меню представляют собой основной ресурс в Windows-программировании, эта тема довольно обширна и включает массу возможностей, связанных с построением меню. К счастью, для создания стандартного меню достаточно освоить основные моменты из этой темы.

При выборе элемента меню генерируется событие `Click`, которое определено в классе `MenuItem`. Следовательно, чтобы обработать выбор элемента меню, в список обработчиков события `Click`, связанный с этим элементом, необходимо добавить соответствующий обработчик.

Каждая форма имеет свойство `Menu`, определенное таким образом:

```
public MainMenu Menu { get; set; }
```

По умолчанию этому свойству не присвоено никакое меню. Чтобы отобразить основное меню, это свойство необходимо "настроить" на создаваемое вами меню.

Для создания основного меню выполните следующие действия.

1. Создайте объект класса `MainMenu`.
2. В объект класса `MainMenu` добавьте объекты класса `MenuItem`, которые описывают категории верхнего уровня. Эти элементы меню добавляются в коллекцию типа `MenuItems`, связанную с основным меню.
3. Для каждого `MenuItem`-объекта верхнего уровня добавьте список `MenuItem`-объектов, который определяет раскрывающееся меню, связанное с элементом меню верхнего уровня. Эти элементы меню добавляются в коллекцию `MenuItems`, связанную с каждым элементом меню верхнего уровня.
4. Добавьте обработчики событий для каждого элемента меню.
5. Присвойте объект класса `MainMenu` свойству `Menu`, связанному с формой.

В следующем фрагменте кода показано, как создать меню Файл, которое содержит три команды: Открыть, Закрыть и Выйти.

```
// Создаем объект основного меню.
MainMenu MyMenu = new MainMenu();

// Добавляем в это меню элемент верхнего уровня.
MenuItem m1 = new MenuItem("Файл");
MyMenu.MenuItems.Add(m1);

// Создаем подменю "Файл".
MenuItem subm1 = new MenuItem("Открыть");
m1.MenuItems.Add(subm1);

MenuItem subm2 = new MenuItem("Закрыть");
m1.MenuItems.Add(subm2);

MenuItem subm3 = new MenuItem("Выйти");
m1.MenuItems.Add(subm3);
```

Эта последовательность инструкций заслуживает внимательного рассмотрения. Она начинается с создания объекта класса `MainMenu` с именем `MyMenu`. Этот объект будет находиться на верхнем уровне структуры меню. Затем создается элемент меню `m1` с заголовком "Файл". Он добавляется непосредственно к объекту `MyMenu`. После этого создается раскрывающееся меню, связанное с командой Файл основного меню. Обратите внимание на то, что элементы раскрывающегося меню добавляются к объекту `m1`, который является элементом Файл основного меню. Если один `MenuItem`-объект добавляется к другому, добавляемый объект становится частью раскрывающегося меню, связанного с элементом, к которому добавляется `MenuItem`-объект. Следовательно, после того как элементы `subm1`—`subm3` будут добавлены к элементу `m1`, при выборе команды File отобразится раскрывающееся меню, содержащее команды Открыть, Закрыть и Выйти.

Создав меню, для каждого его элемента необходимо создать связанные с ним обработчики событий. Как разъяснялось выше, при выборе пользователем команды меню генерируется событие `Click`. Поэтому при выполнении следующей последовательности инструкций элементам `subm1`—`subm3` будут назначены соответствующие обработчики событий.

```
// Добавляем обработчики событий для всех элементов меню.
subm1.Click += new EventHandler(MMOpenClick);
subm2.Click += new EventHandler(MMCloseClick);
subm3.Click += new EventHandler(MMExitClick);
```

Таким образом, если пользователь выберет команду Выйти, выполнится обработчик событий `MMExitClick()`.

Наконец, свойству `Menu` формы нужно присвоить объект класса `MainMenu`:

```
Menu = MyMenu;
```

После выполнения этой инструкции окно будет отображаться вместе с меню, при выборе команд которого будут вызываться соответствующие обработчики событий.

Следующая программа демонстрирует создание основного меню и обработку событий, связанных с выбором команд меню.

```
// Добавляем основное меню.

using System;
using System.Windows.Forms;

class MenuForm : Form {
  MainMenu MyMenu;

  public MenuForm() {
    Text = "Добавление меню";

// Создаем объект основного меню.
MainMenu MyMenu = new MainMenu();

// Добавляем в это меню элемент верхнего уровня.
MenuItem m1 = new MenuItem("Файл");
MyMenu.MenuItems.Add(m1);

MenuItem m2 = new MenuItem("Сервис");
MyMenu.MenuItems.Add(m2);

// Создаем подменю Файл.
MenuItem subm1 = new MenuItem("Открыть");
m1.MenuItems.Add(subm1);

MenuItem subm2 = new MenuItem("Закрыть");
m1.MenuItems.Add(subm2);

MenuItem subm3 = new MenuItem("Выйти");
m1.MenuItems.Add(subm3);

    // Создаем подменю "Сервис".
    MenuItem subm4 = new MenuItem("Координаты");
    m2.MenuItems.Add(subm4);

    MenuItem subm5 = new MenuItem("Изменить размер");
    m2.MenuItems.Add(subm5);

    MenuItem subm6 = new MenuItem("Восстановить");
    m2.MenuItems.Add(subm6);


    // Добавляем обработчики событий для элементов меню.
    subm1.Click += new EventHandler(MMOpenClick);
    subm2.Click += new EventHandler(MMCloseClick);
    subm3.Click += new EventHandler(MMExitClick);
    subm4.Click += new EventHandler(MMCoordClick);
    subm5.Click += new EventHandler(MMChangeClick);
    subm6.Click += new EventHandler(MMRestoreClick);
```

```
    // Назначаем меню форме.
    Menu = MyMenu;
  }

  [STAThread]
  public static void Main() {
    MenuForm skel = new MenuForm();

    Application.Run(skel);
  }

  // Обработчик для команды меню Координаты.
  protected void MMCoordClick(object who, EventArgs e) {
    // Создаем строку, которая содержит три координаты.
    string size =
      String.Format("{0}: {1}, {2}\n{3}: {4}, {5} ",
                    "Вверху, Слева", Top, Left,
                    "Внизу, Справа", Bottom, Right);

    // Отображаем окно сообщений.
    MessageBox.Show(size, "Координаты окна",
                    MessageBoxButtons.OK);
  }

  // Обработчик для команды меню Изменить размер.
  protected void MMChangeClick(object who, EventArgs e) {
    Width = Height = 200;
  }

  // Обработчик для команды меню Восстановить.
  protected void MMRestoreClick(object who, EventArgs e) {
    Width = Height = 300;
  }

  // Обработчик для команды меню Открыть.
  protected void MMOpenClick(object who, EventArgs e) {

    MessageBox.Show("Неактивная команда", "Заглушка",
                    MessageBoxButtons.OK);
  }

  // Обработчик для команды меню Закрыть.
  protected void MMCloseClick(object who, EventArgs e) {

    MessageBox.Show("Неактивная команда", "Заглушка",
                    MessageBoxButtons.OK);
  }

  // Обработчик для команды меню Выйти.
  protected void MMExitClick(object who, EventArgs e) {

    DialogResult result = MessageBox.Show("Остановить программу?",
                                          "Завершение",
                                          MessageBoxButtons.YesNo);

    if(result == DialogResult.Yes) Application.Exit();
  }
}
```

Результат выполнения этой программы показан на рис. 25.5.

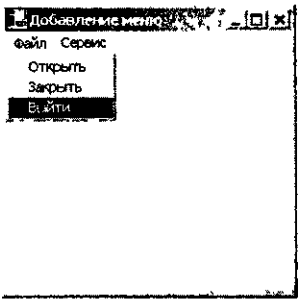


Рис. 25.5. Результат выполнения программы, использующей меню

В этой программе определяется два раскрывающихся меню. Доступ к первому (оно содержит команды Открыть, Закрыть и Выйти) обеспечивается через меню Файл. Обработчики событий для элементов Открыть и Закрыть представляют собой заглушки, которые не выполняют никаких действий, кроме отображения окна сообщения. Обработчик элемента Выйти в собственном окне сообщения предлагает пользователю подтвердить его желание завершить программу. Если пользователь ответит щелчком на кнопке Да, программа будет завершена.

Меню Сервис также содержит три элемента: Координаты, Изменить размер и Восстановить. При выборе команды Координаты в окне сообщения отображаются координаты верхнего левого и нижнего правого углов окна. Попробуйте переместить окно, а затем с помощью этой команды отобразите его координаты. При каждом перемещении окна в новую позицию его координаты будут изменяться соответствующим образом.

При выборе команды Изменить размер окно программы уменьшается в размере так, чтобы его ширина и высота составляли 200 пикселей. Это реализуется с помощью свойств `Width` и `Height`:

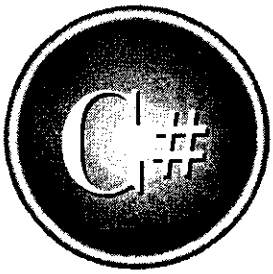
```
public int Width { get; set; }
public int Height { get; set; }
```

По умолчанию окно имеет размеры 300×300. При выборе команды Restore окно возвращается к своему исходному размеру.

Что дальше

Как упоминалось выше, Windows-программирование — очень обширная тема. Windows — это среда с богатыми возможностями, которая предъявляет к программисту большие требования. Если вы — новичок в Windows-программировании, то на его изучение может уйти несколько недель. Начать освоение этого материала можно со знакомства с элементами управления, определенными в пространстве имен `System.Windows.Forms`. Необходимо также научиться обрабатывать Windows-сообщения, в частности те, которые запрашивают повторное отображение (перерисовку) окна. Другая важная подсистема, определенная в пространстве имен `Windows.Drawing`, включает различные классы, которые управляют отображением выходных данных в окне. Пространство имен `Windows.Drawing` инкапсулирует функции, предоставляемые интерфейсом графических устройств Windows GDI (Graphics Device Interface). Постарайтесь написать как можно больше коротких программ, чтобы понять суть каждого нового средства или метода.

Полный справочник по



Глава 26

Синтаксический анализ методом рекурсивного спуска

Как бы вы написали программу, которая в качестве входных данных должна принять строку, содержащую числовое выражение, например (10 – 5) * 3, а затем вычислить правильный ответ? Как правило, когда задаешь этот вопрос, только немногие знают, как это сделать. Для подавляющего большинства программистов кажется очень трудной задачей с помощью языка высокого уровня преобразовать алгебраические выражения в инструкции, которые мог бы выполнить компьютер. Эта процедура называется *анализом выражений* (expression parsing), и именно она лежит в основе всех компиляторов и интерпретаторов, программ табличных вычислений и прочих программ, в которых требуется преобразование числовых выражений в форму, приемлемую для использования компьютером.

Несмотря на то что поставленная выше задача для непосвященных может показаться тайной за семью печатями, анализ выражений представляет собой хорошо структурированную задачу, для которой существует элегантное решение. Решение возможно благодаря тому, что анализ выражений работает в соответствии со строгими алгебраическими правилами. В этой главе предлагается разработка того, что обычно именуется *синтаксическим анализом методом рекурсивного спуска* (recursive-descent parser), а также всех необходимых подпрограмм, которые позволят вычислять числовые выражения. Освоив механизм действия этого анализатора, вы сможете легко усовершенствовать его или модифицировать под свои потребности.

Помимо того, что предлагаемая программа полезна сама по себе, разрабатываемый здесь анализатор также иллюстрирует мощь и диапазон применения языка C#. Наш анализатор — это приложение "чистого кода". Оно не взаимодействует с сетью, не использует GUI-интерфейс или какие-нибудь ограниченные системные ресурсы. В прошлом код такого типа обычно писали на языке C++. Тот факт, что анализатор можно легко создать с помощью языка C#, еще раз доказывает, что C# способен справиться с любой задачей программирования.

Выражения

Поскольку анализатор предназначен для вычисления алгебраических выражений, важно, чтобы вы понимали составляющие их части. Хотя выражения могут состоять из данных всех типов, в этой главе используются лишь числовые. Итак, допустим, в состав числовых выражений входят следующие элементы:

- числа;
- операторы +, –, /, *, ^, % и =;
- круглые скобки;
- переменные.

Здесь знак вставки (^) означает операцию возведения в степень (а не операцию "исключающее ИЛИ", как в C#), а знак "равно" (=) — оператор присвоения. Эти элементы могут объединяться в выражениях в соответствии с правилами алгебры. Вот несколько примеров:

```
10 - 8
(100 - 5) * 14 / 6
a + b - c
10 ^ 5
a = 10 - b
```

Предположим, приоритет упомянутых выше операторов можно представить следующим образом:

наивысший	+ - (унарный)
	^
	* / %
	+ -
самый низкий	=

Операторы с одинаковым приоритетом выполняются слева направо.

На анализатор, представленный в этой главе, налагается ряд ограничений. Во-первых, имена всех переменных должны состоять из одной буквы (другими словами, можно использовать 26 переменных, от `A` до `Z`). При этом анализатор не должен отличать прописное написание буквы от строчного (`a` и `A` воспринимаются как одна и та же переменная). Во-вторых, предполагается, что все числовые значения имеют тип `double`, хотя анализатор нетрудно модифицировать для обработки других типов значений. Наконец, чтобы программа была логически ясной и понятной, в ней предусмотрен отсев только элементарных ошибок.

Анализ выражений: постановка задачи

На первый взгляд может показаться, что анализ выражений — довольно простая задача, но это не совсем так. Чтобы лучше понять проблему, попробуем вычислить это простое выражение:

```
10 - 2 * 3
```

Вы знаете, что это выражение равно 4. И хотя можно легко создать программу, которая вычислит это выражение, проблемой становится создание программы, которая бы давала правильный ответ для любого выражения. Для начала представим нашу будущую программу в виде следующего эскизного варианта:

```
a = получаем первый операнд
while(операнды имеются) {
  op = получаем оператор
  b = получаем второй операнд
  a = a op b
}
```

Согласно этому алгоритму, мы получаем первый операнд, оператор и второй операнд для выполнения первой операции; затем получаем следующий оператор и операнд для выполнения следующей операции и т.д. Но если использовать наш алгоритм, то вычисленное с его помощью выражение 10 — 2 * 3 даст в результате 24 (т.е. 8 * 3), а не 4, поскольку в этой процедуре не учитывается приоритет операторов. Ведь нельзя же просто выполнять операции слева направо, поскольку правила алгебры предписывают, что умножение должно быть выполнено до вычитания. Начинающим программистам может показаться, что эту проблему легко преодолеть, и действительно, в очень редких случаях это возможно. Но проблема еще усложняется, если добавить возможность использования круглых скобок, операции возведения в степень, переменных, унарных операторов и т.п. Хотя известно несколько способов написания программы вычисления выражений, мы рассмотрим здесь самый простой вариант. Этот вариант называется *синтаксическим анализом методом рекурсивного спуска*, и в ходе разработки вы поймете, откуда у этой программы такое название. (В других методах, используемых для написания анализаторов выражений, применяются сложные таблицы, которые должны быть сгенерированы другой программой. Такие программы получили название анализаторов *с табличным управлением*.)

Анализ выражения

Анализировать и вычислять выражения можно различными способами. Что касается синтаксического анализатора, использующего метод рекурсивного спуска, то представим себе выражения в виде *рекурсивных структур данных* (recursive data structures), т.е. выражений, которые определяются на основе самих себя. Если предположить, что такие выражения могут использовать только операторы +, -, *, / и круглые скобки, то все выражения могут быть определены с использованием следующих правил:

```
выражение -> член [+ член] [- член]
член -> множитель [* множитель] [/ множитель]
множитель -> переменная, число или (выражение)
```

Квадратные скобки означают необязательный элемент, а символ -> заменяет словосочетание "представляет собой". В действительности эти правила называются *порождающими правилами* выражения. Следовательно, относительно определения понятия *член* можно сказать: член представляет собой множитель, умноженный на множитель, или множитель, разделенный на множитель. Обратите внимание на то, что приоритет выполнения операторов неявно выражен самим способом определения выражения.

Выражение

```
10 + 5 * B
```

имеет два члена: `10` и `5 * B`. Второй член состоит из двух множителей: `5` и `B`. Эти множители состоят из числа и из переменной, соответственно.

А выражение

```
14 * (7 - C)
```

имеет два множителя: `14` и `(7 - C)`. Множители состоят из одного числа и одного выражения, заключенного в круглые скобки. Выражение в круглых скобках состоит из двух членов: числа и переменной.

Этот процесс составляет основу для разработки рекурсивного анализатора, который представляет собой набор взаимно рекурсивных методов, работающих в цепеобразном стиле и реализующих порождающие правила. На каждом этапе анализатор выполняет определенные операции в алгебраически корректной последовательности. Чтобы понять, как эти порождающие правила используются для анализа выражения, рассмотрим следующий пример:

```
9/3 - (100 + 56)
```

Для анализа этого выражения мы должны выполнить следующие действия.

1. Получить первый член, `9/3`.
2. Получить каждый множитель и выполнить операцию деления целых чисел. Результат равен числу `3`.
3. Получить второй член, `(100 + 56)`. На этом этапе необходимо начать с рекурсивного анализа второго выражения.
4. Получить каждый член и сложить их. Результат равен числу `156`.
5. Вернуться из рекурсивного вызова и вычесть число `156` из числа `3`. Ответ равен `-153`.

Если вы почувствовали, что сбиты с толку, не расстраивайтесь. Это довольно сложная концепция, которая требует определенных навыков. При рассмотрении выражений с "рекурсивной" точки зрения необходимо помнить два основных принципа. Во-первых, приоритет операторов неявно выражен в самом характере определения

порождающих правил. Во-вторых, этот метод анализа и вычисления выражений очень близок тому, как математические выражения вычисляются человеком.

В этой главе мы разработаем два анализатора. Первый предназначен для анализа и вычисления выражений с плавающей точкой (типа `double`), которые состоят только из литеральных значений. Этот анализатор иллюстрирует основы анализа выражений методом рекурсивного спуска. Второй строится на базе первого, но позволяет также использовать переменные.

Разбор выражения

Чтобы вычислить выражение, необходимо предоставить анализатору отдельные компоненты этого выражения. Например, выражение

```
A * B - (W + 10)
```

содержит такие отдельные компоненты: `A`, `*`, `B`, `-`, `(`, `W`, `+`, `10` и `)`. На языке грамматического разбора каждый компонент выражения называется *лексемой*, а лексема представляет собой неделимый элемент выражения. Поскольку разбиение выражения на лексемы имеет принципиальное значение для процесса анализа, ему следует уделить особое внимание, а затем уже заняться самим анализатором.

Чтобы представить выражение в виде лексем, необходимо иметь метод, который бы последовательно возвращал каждую лексему выражения в направлении от начала к концу. Этот метод должен "уметь перескакивать" через пробелы и обнаруживать конец выражения. В нашем анализаторе метод, который выполняет эту задачу, именуется `GetToken()`.

Оба анализатора, представленные в этой главе, инкапсулированы в классе `Parser`. Хотя этот класс подробно описан ниже, нам уже теперь необходимо познакомиться с его первой частью, чтобы понять работу метода `GetToken()`. Итак, класс `Parser` начинается с определения следующих перечислений и полей:

```
class Parser {
  // Перечисляем типы лексем.
  enum Types { NONE, DELIMITER, VARIABLE, NUMBER };
  // Перечисляем типы ошибок.
  enum Errors { SYNTAX, UNBALPARENS, NOEXP, DIVBYZERO };

  string exp;     // Ссылка на строку выражения.
  int expIdx;     // Текущий индекс в выражении.
  string token;   // Текущая лексема.
  Types tokType; // Тип лексемы.
```

Первое перечисление названо именем `Types`. При анализе выражения имеет значение тип лексемы. В разработанных здесь анализаторах мы имеем дело только с тремя типами лексем: переменная, число и разделитель. Они представлены значениями `VARIABLE`, `NUMBER` и `DELIMITER`, которые определены в перечислении `Types`. Категория `DELIMITER` используется как для операторов, так и для круглых скобок. Тип `NONE` служит в качестве специального значения для неопределенной лексемы. Перечисление `Errors` представляет различные ошибки, которые могут возникнуть в процессе анализа.

Ссылка на строку, которая содержит анализируемое выражение, хранится в переменной `exp`. Таким образом, `exp` будет ссылаться на такую строку, как `"10+4"`. Индекс следующей лексемы этой строки содержится в поле `expIdx` и первоначально равен нулю. Выделенная из строки лексема будет храниться в переменной `token`, а ее тип — в переменной `tokType`.

Ниже приведен метод GetToken(). При каждом вызове он получает следующую лексему из выражения, содержащегося в строке, адресуемой ссылкой exp, начиная с индекса expIdx, т.е. следующая лексема начинается с элемента exp[expIdx]. Эта лексема помещается в поле token, а ее тип — в поле tokType. Метод GetToken() использует метод IsDelim(), который определяет, является ли заданный символ разделителем.

```
// Получаем следующую лексему.
  void GetToken()
  {
    tokType = Types.NONE;
    token = "";

    if(expIdx == exp.Length) return; // конец выражения

    // Пропускаем пробелы.
    while(expIdx < exp.Length &&
          Char.IsWhiteSpace(exp[expIdx])) ++expIdx;

    // Хвостовой пробел завершает выражение.
    if(expIdx == exp.Length) return;

    if(IsDelim(exp[expIdx])) { // Это оператор?
      token += exp[expIdx];
      expIdx++;
      tokType = Types.DELIMITER;
    }
    else if(Char.IsLetter(exp[expIdx])) { // Это
                                          // переменная?
      while(!IsDelim(exp[expIdx])) {
        token += exp[expIdx];
        expIdx++;
        if(expIdx >= exp.Length) break;
      }
      tokType = Types.VARIABLE;
    }
    else if(Char.IsDigit(exp[expIdx])) { // Это число?
      while(!IsDelim(exp[expIdx])) {
        token += exp[expIdx];
        expIdx++;
        if(expIdx >= exp.Length) break;
      }
      tokType = Types.NUMBER;
    }
  }

  // Метод возвращает значение true, если
  // символ c является разделителем.
  bool IsDelim(char c)
  {
    if((" +-/*%^=()".IndexOf(c) != -1))
      return true;
    return false;
  }
```

Рассмотрим подробно код метода GetToken(). После выполнения первых инструкций инициализации метод проверяет, не достигнут ли конец выражения, сравнивая значение переменной expIdx со значением свойства exp.Length. Поскольку

expIdx — индекс строки, содержащей анализируемое выражение, то если он равен длине строки, значит, выражение полностью проанализировано (разбито на лексемы).

Если из выражения еще можно извлечь очередную лексему, метод GetToken() опускает все начальные пробелы. После этого элемент exp[expIdx] будет содержать либо цифру, либо переменную, либо оператор, либо пробел (если анализируемое выражение завершают хвостовые пробелы). Если следующим символом является оператор, то он возвращается в виде строки, сохраненной в переменной token, а в поле tokType сохраняется значение DELIMITER. Если следующим символом является буква, то предполагается, что это одна из переменных выражения. Она возвращается в виде строки, сохраненной в переменной token, а в поле tokType сохраняется значение VARIABLE. Если следующим символом является цифра, то считывается число, которое возвращается в виде строки, сохраненной в переменной token, а в поле tokType сохраняется значение NUMBER. Наконец, если следующий символ не попадает ни под одну из предыдущих категорий, переменная token будет содержать нулевую строку.

Чтобы не загромождать деталями код метода GetToken(), здесь упрощена проверка наличия ошибок и принят ряд допущений. Например, выражение может завершаться нераспознаваемым символом, если ему предшествует пробел. В этой версии имена переменных могут иметь любую длину, но значимой является только первая буква имени. При необходимости вы можете расширить диапазон распознаваемых ошибок самостоятельно.

Чтобы лучше понять, как работает метод GetToken(), рассмотрим это выражение:

```
A + 100 - (B * C) / 2
```

При разбиении выражения на лексемы метод GetToken() получает следующие результаты (лексема и ее тип):

Лексема	*Тип лексемы*
A	VARIABLE
+	DELIMITER
100	NUMBER
-	DELIMITER
(	DELIMITER
B	VARIABLE
*	DELIMITER
C	VARIABLE
)	DELIMITER
/	DELIMITER
2	NUMBER

Вспомните, что лексема всегда содержит строку, даже если она состоит из одного символа

Простой анализатор выражений

Ниже приведена первая версия анализатора. Она может вычислять выражения, которые состоят исключительно из литералов, операторов и круглых скобок. И хотя метод GetToken() может обрабатывать переменные, этот анализатор их не воспринима-

ет. Если вы поймете работу такого упрощенного анализатора, мы расширим его возможности, "научив" обрабатывать переменные.

```
/*
   Этот модуль содержит рекурсивный нисходящий
   синтаксический анализатор, который не использует
   переменных.
*/

using System;

// Класс исключений для обнаружения ошибок анализатора.
class ParserException : ApplicationException {
  public ParserException(string str) : base(str) { }

  public override string ToString() {
    return Message;
  }
}

class Parser {
// Перечисляем типы лексем.
  enum Types { NONE, DELIMITER, VARIABLE, NUMBER };
  // Перечисляем типы ошибок.
  enum Errors { SYNTAX, UNBALPARENS, NOEXP, DIVBYZERO };

  string exp;     // Ссылка на строку выражения.
  int expIdx;     // Текущий индекс в выражении.
  string token;   // Текущая лексема.
  Types tokType; // Тип лексемы.

  // Входная точка анализатора.
  public double Evaluate(string expstr)
  {
    double result;

    exp = expstr;
    expIdx = 0;

    try {
      GetToken();
      if(token == "") {
        SyntaxErr(Errors.NOEXP); // Выражение
                                 // отсутствует.
        return 0.0;
      }

      EvalExp2(out result);

      if(token != "") // Последняя лексема должна
                      // быть null-значением.
        SyntaxErr(Errors.SYNTAX);

      return result;
    } catch (ParserException exc) {
      // При необходимости добавьте сюда обработку
      // других ошибок.
      Console.WriteLine(exc);
```

```
    return 0.0;
  }
}

// Сложение или вычитание двух членов выражения.
void EvalExp2(out double result)
{
  string op;
  double partialResult;

  EvalExp3(out result);
  while((op = token) == "+" || op == "-") {
    GetToken();
    EvalExp3(out partialResult);
    switch(op) {
      case "-":
        result = result - partialResult;
        break;
      case "+":
        result = result + partialResult;
        break;
    }
  }
}

// Умножение или деление двух множителей.
void EvalExp3(out double result)
{
  string op;
  double partialResult = 0.0;

  EvalExp4(out result);
  while((op = token) == "*" ||
        op == "/" || op == "%") {
    GetToken();
    EvalExp4(out partialResult);
    switch(op) {
      case "*":
        result = result * partialResult;
        break;
      case "/":
        if(partialResult == 0.0)
          SyntaxErr(Errors.DIVBYZERO);
        result = result / partialResult;
        break;
      case "%":
        if(partialResult == 0.0)
          SyntaxErr(Errors.DIVBYZERO);
        result = (int) result % (int) partialResult;
        break;
    }
  }
}

// Возведение в степень.
void EvalExp4(out double result)
{
  double partialResult, ex;
```

```
  int t;

  EvalExp5(out result);
  if(token == "^") {
    GetToken();
    EvalExp4(out partialResult);
    ex = result;
    if(partialResult == 0.0) {
      result = 1.0;
      return;
    }
    for(t=(int)partialResult-1; t > 0; t--)
      result = result * (double)ex;
  }
}

// Выполнение операции унарного + или -.
void EvalExp5(out double result)
{
  string  op;

  op = "";
  if((tokType == Types.DELIMITER) &&
      token == "+" || token == "-") {
    op = token;
    GetToken();
  }
  EvalExp6(out result);
  if(op == "-") result = -result;
}

// Обработка выражения в круглых скобках.
void EvalExp6(out double result)
{
  if((token == "(")) {
    GetToken();
    EvalExp2(out result);
    if(token != ")")
      SyntaxErr(Errors.UNBALPARENS);
    GetToken();
  }
  else Atom(out result);
}

// Получаем значение числа.
void Atom(out double result)
{
  switch(tokType) {
    case Types.NUMBER:
      try {
        result = Double.Parse(token);
      } catch (FormatException) {
        result = 0.0;
        SyntaxErr(Errors.SYNTAX);
      }
      GetToken();
      return;
    default:
```

```
        result = 0.0;
        SyntaxErr(Errors.SYNTAX);
        break;
    }
  }

  // Обрабатываем синтаксическую ошибку.
  void SyntaxErr(Errors error)
  {
    string[] err = {
      "Синтаксическая ошибка",
      "Дисбаланс скобок",
      "Выражение отсутствует",
      "Деление на нуль"
    };

    throw new ParserException(err[(int)error]);
  }

// Получаем следующую лексему.
  void GetToken()
  {
    tokType = Types.NONE;
    token = "";

    if(expIdx == exp.Length) return; // конец выражения

    // Пропускаем пробелы.
    while(expIdx < exp.Length &&
          Char.IsWhiteSpace(exp[expIdx])) ++expIdx;

    // Хвостовой пробел завершает выражение.
    if(expIdx == exp.Length) return;

    if(IsDelim(exp[expIdx])) { // Это оператор?
      token += exp[expIdx];
      expIdx++;
      tokType = Types.DELIMITER;
    }
    else if(Char.IsLetter(exp[expIdx])) { // Это
                                          // переменная?
      while(!IsDelim(exp[expIdx])) {
        token += exp[expIdx];
        expIdx++;
        if(expIdx >= exp.Length) break;
      }
      tokType = Types.VARIABLE;
    }
    else if(Char.IsDigit(exp[expIdx])) { // Это число?
      while(!IsDelim(exp[expIdx])) {
        token += exp[expIdx];
        expIdx++;
        if(expIdx >= exp.Length) break;
      }
      tokType = Types.NUMBER;
    }
  }
```

```
  // Метод возвращает значение true, если
  // символ c является разделителем.
  bool IsDelim(char c)
  {
    if((" +-/*%^=()".IndexOf(c) != -1))
      return true;
    return false;
  }

}
```

Приведенный здесь анализатор может обрабатывать следующие операторы: +, -, *, / и %. Кроме того, он может выполнить операцию возведения в степень (^) при использовании целочисленного показателя степени, а также корректно обработать оператор "унарный минус" и выражение в круглых скобках.

Чтобы использовать анализатор, сначала создайте объект типа `Parser`. Затем вызовите метод `Evaluate()`, передав ему в качестве аргумента подлежащее вычислению выражение, представленное в строковом виде. Вам останется лишь получить результат, возвращаемый методом `Evaluate()`. Использование анализатора демонстрируется в следующей программе:

```
// Демонстрация использования анализатора выражений.

using System;

class ParserDemo {
  public static void Main()
  {
    string expr;
    Parser p = new Parser();

    Console.WriteLine(
      "Для выхода из программы введите пустое выражение.");

    for(;;) {
      Console.Write("Введите выражение: ");
      expr = Console.ReadLine();
      if(expr == "") break;
      Console.WriteLine("Результат: " + p.Evaluate(expr));
    }
  }
}
```

Вот пример использования анализатора:

```
Для выхода из программы введите пустое выражение.
Введите выражение: 10-2*3
Результат: 4
Введите выражение: (10-2)*3
Результат: 24
Введите выражение: 10/3
Результат: 3.33333333333333
Введите выражение: 10/3.5
Результат: 2.85714285714286
Введите выражение:
```

Осмысление механизма анализа

Рассмотрим детально класс `Parser`. Как упоминалось при рассмотрении метода `GetToken()`, класс `Parser` содержит четыре закрытых поля. Вычисляемое выражение хранится в строке `exp`. Это поле устанавливается при каждом вызове метода `Evaluate()`. Важно помнить, что анализатор вычисляет выражения, которые запоминаются в стандартной C#-строке. Например, следующие строки содержат выражения, которые способен вычислить наш анализатор:

```
"10 - 5"
"2 * 3.3 / (3.1416 * 3.3)"
```

Текущий индекс символа в строке `exp` хранится в поле `expIdx`. В начале анализа переменная `expIdx` устанавливается равной нулю. Значение `expIdx` инкрементируется по мере "перемещения" по анализируемому выражению. Поле `token` содержит текущую лексему, а поле `tokType` — ее тип.

Входной точкой анализатора является метод `Evaluate()`, которому при вызове необходимо передать строку, содержащую выражение, предназначенное для анализа. Методы `EvalExp2()`—`EvalExp6()` вместе с методом `Atom()` и составляют рекурсивный нисходящий синтаксический анализатор. Они реализуют расширенный набор описанных выше порождающих правил. Комментарии, приведенные в начале каждого метода, описывают выполняемые ими функции. В следующую версию анализатора будет добавлен метод `EvalExp1()`.

Метод `SyntaxErr()` обрабатывает синтаксические ошибки в задаваемых пользователем выражениях. Методы `GetToken()` и `IsDelim()`, как описано выше, разбивают выражение на составные части. В нашем анализаторе метод `GetToken()` используется для выделения из выражения лексем. На основе типа лексемы принимается решение о том, какие действия предпримет анализатор.

Чтобы понять, как анализатор вычисляет выражение, возьмем для примера следующее выражение:

```
10 - 3 * 2
```

При первом вызове метод `Evaluate()` возвращает первую лексему. Если она представляет собой пустую строку (""), на экране отобразится сообщение `"Выражение отсутствует"`, и метод `Evaluate()` возвратит значение `0.0`. Но в данном случае лексема содержит число `10`. Затем вызывается метод `EvalExp2()`. Метод `EvalExp2()` вызывает метод `EvalExp3()`, который вызывает метод `EvalExp4()`, а он — метод `EvalExp5()`. После этого метод `EvalExp5()` определяет, является ли лексема унарным плюсом или унарным минусом. В нашем случае это не так, поэтому вызывается метод `EvalExp6()`. В этой точке программы метод `EvalExp6()` либо рекурсивно вызывает метод `EvalExp2()` (в том случае, если выражение заключено в круглые скобки), либо метод `Atom()` для получения значения числа. Поскольку лексема не является левой скобкой, выполняется метод `Atom()` и переменной `result` присваивается значение `10`. Затем считывается следующая лексема, и выполнение этого метода завершается. Управление передается вызвавшему методу, который, завершаясь (в свою очередь), передает управление "вверх" по цепочке вызовов, т.е. происходит последовательный возврат из методов, входящих в состав цепочки. Так как новая лексема представляет собой оператор "-", цепочка возвращает нас к методу `EvalExp2()`.

Следующий шаг очень важен. Поскольку текущая лексема оказалась оператором "-", она сохраняется в переменной `op`. Затем анализатор получает очередную лексему, которая является числом `3`, и снова начинает "спуск" по цепочке методов. После вызова метода `Atom()` возвращаемое им число `3` записывается в поле `result` и считывается лексема "*". Тип лексемы заставляет вернуться по цепочке к методу

EvalExp3(), в котором считывается последняя лексема, 2. Здесь выполняется первая арифметическая операция, а именно умножение 2 на 3. Результат возвращается методу EvalExp2(), который выполняет операцию вычитания. В результате вычитания получаем 4. Хотя описанный процесс на первый взгляд может показаться сложным, "пройдите" его вручную на различных примерах; это поможет понять его работу, а главное, вы убедитесь в том, что он функционирует корректно для разных исходных выражений.

При обнаружении ошибки в процессе анализа вызывается метод SyntaxErr(), который генерирует исключение типа ParserException, позволяющее установить причину ошибки. Исключение ParserException — не встроенное; оно определено в начале файла, содержащего определение класса Parser. Этот анализатор выражений, как проиллюстрировано предыдущей программой, можно использовать для простого настольного калькулятора. Но прежде чем пытаться встроить его в базу данных или в более усовершенствованный калькулятор, необходимо наделить его способностью обрабатывать переменные. Это и является предметом обсуждения следующего раздела.

Добавление в анализатор переменных

Во всех языках программирования, многих калькуляторах и программах табличных вычислений переменные служат для хранения значений, которые будут задействованы позже. Чтобы наш анализатор можно было использовать для таких приложений, необходимо расширить его возможности в направлении обработки переменных. Для этого прежде всего нужно добавить сами переменные. Как упоминалось выше, в качестве имен переменных мы будем использовать буквы от A до Z. Выделим для хранения переменных массив в классе Parser. Каждая переменная может претендовать только на одну позицию в 26-элементном массиве типа doubles. Итак, добавим в класс Parser следующее поле:

```
double[] vars = new double[26];
```

В класс Parser необходимо также добавить следующий конструктор, который инициализирует переменные:

```
public Parser() {
  // Инициализируем переменные нулевыми значениями.
  for(int i=0; i < vars.Length; i++)
    vars[i] = 0.0;
}
```

"Из любезности" к пользователю переменные теперь инициализированы значениями 0.0.

Нам также потребуется метод поиска значения заданной переменной. Поскольку переменные носят имена от A до Z, их легко использовать для индексации массива vars путем вычитания ASCII-значения буквы A из имени переменной. Этим занимается метод FindVar(). Вот его код:

```
// Метод возвращает значение переменной.
  double FindVar(string vname)
  {
    if(!Char.IsLetter(vname[0])){
      SyntaxErr(Errors.SYNTAX);
      return 0.0;
    }
    return vars[Char.ToUpper(vname[0])-'A'];
  }
```

Как видно из кода, метод действительно принимает длинные (т.е. не однобуквенные) имена переменных, например A12 или test, но значимой является только первая буква имени. При необходимости эту ситуацию можно изменить в соответствии с конкретными требованиями.

Мы должны также модифицировать метод Atom(), чтобы он мог обрабатывать как числа, так и переменные. Вот как выглядит его новая версия:

```
// Получаем значение числа или переменной.
  void Atom(out double result)
  {
    switch(tokType) {
      case Types.NUMBER:
        try {
          result = Double.Parse(token);
        } catch (FormatException) {
          result = 0.0;
          SyntaxErr(Errors.SYNTAX);
        }
        GetToken();
        return;
      case Types.VARIABLE:
        result = FindVar(token);
        GetToken();
        return;
      default:
        result = 0.0;
        SyntaxErr(Errors.SYNTAX);
        break;
    }
  }
```

По сути, этих дополнений вполне достаточно для того, чтобы анализатор корректно использовал переменные, но пока не реализован способ присвоения этим переменным значений. Чтобы переменной можно было придать некоторое значение, анализатор должен "уметь" обрабатывать оператор присвоения (=). Для реализации присвоения добавим в класс Parser еще один метод — EvalExp1(). Теперь именно этот метод будет начинать рекурсивную нисходящую цепочку. Это означает, что для того, чтобы начать анализ выражения, из метода Evaluate() теперь должен вызываться метод EvalExp1(), а не метод EvalExp2(). Вот как выглядит код метода EvalExp1():

```
// Обработка операции присвоения.
  void EvalExp1(out double result)
  {
    int varIdx;
    Types ttokType;
    string temptoken;

    if(tokType == Types.VARIABLE) {
      // Сохраняем старую лексему.
      temptoken = String.Copy(token);
      ttokType = tokType;

      // Вычисляем индекс переменной.
      varIdx = Char.ToUpper(token[0]) - 'A';

      GetToken();
      if(token != "=") {
```

```
        PutBack(); // Возвращаем текущую лексему в поток
                   // и восстанавливаем старую,
                   // поскольку у нас не присвоение.
        token = String.Copy(temptoken);
        tokType = ttokType;
      }
      else {
        GetToken(); // Получаем следующую часть exp.
        EvalExp2(out result);
        vars[varIdx] = result;
        return;
      }
    }

    EvalExp2(out result);
  }
```

В методе `EvalExp1()` необходимо узнать, имеет ли место присвоение. Дело в том, что оператору присвоения всегда предшествует имя переменной, но наличие "одинокого" имени переменной еще не гарантирует, что за ним обязательно следует выражение присвоения. Другими словами, анализатор воспримет выражение `A = 100` как присвоение, но он достаточно интеллектуален, чтобы "сообразить", что выражение `A/10` не является присвоением. Для этого метод `EvalExp1()` считывает следующую лексему из входного потока. Если она не содержит знак равенства, эта лексема возвращается во входной поток для последующего вызова метода `PutBack()`, код которого приводится ниже:

```
// Возвращаем лексему во входной поток.
void PutBack()
{
  for(int i=0; i < token.Length; i++) expIdx--;
}
```

После внесения необходимых изменений анализатор примет следующий вид:

```
/*
   Этот модуль содержит рекурсивный нисходящий
   анализатор, который распознает переменные.
*/

using System;

// Класс исключений для ошибок анализатора.
class ParserException : ApplicationException {
  public ParserException(string str) : base(str) { }

  public override string ToString() {
    return Message;
  }
}

class Parser {
  // Перечисляем типы лексем.
  enum Types { NONE, DELIMITER, VARIABLE, NUMBER };
  // Перечисляем типы ошибок.
  enum Errors { SYNTAX, UNBALPARENS, NOEXP, DIVBYZERO };

  string exp;     // Ссылка на строку выражения.
  int expIdx;     // Текущий индекс в выражении.
  string token;   // Текущая лексема.
```

```
  Types tokType; // Тип лексемы.

  // Массив для переменных.
  double[] vars = new double[26];

  public Parser() {
    // Инициализируем переменные нулевыми значениями.
    for(int i=0; i < vars.Length; i++)
      vars[i] = 0.0;
  }

  // Входная точка анализатора.
  public double Evaluate(string expstr)
  {
    double result;

    exp = expstr;
    expIdx = 0;

    try {
      GetToken();
      if(token == "") {
        SyntaxErr(Errors.NOEXP); // Выражение отсутствует.
        return 0.0;
      }

      EvalExp1(out result); // В этом варианте анализатора
                            // сначала вызывается
                            // метод EvalExp1().

      if(token != "") // Последняя лексема должна
                      // быть нулевой.
        SyntaxErr(Errors.SYNTAX);

      return result;
    } catch (ParserException exc) {
      // При желании добавляем здесь обработку ошибок.
      Console.WriteLine(exc);
      return 0.0;
    }
  }

  // Обрабатываем присвоение.
  void EvalExp1(out double result)
  {
    int varIdx;
    Types ttokType;
    string temptoken;

    if(tokType == Types.VARIABLE) {
      // Сохраняем старую лексему.
      temptoken = String.Copy(token);
      ttokType = tokType;

      // Вычисляем индекс переменной.
      varIdx = Char.ToUpper(token[0]) - 'A';

      GetToken();
```

```
    if(token != "=") {
      PutBack(); // Возвращаем текущую лексему в поток
                 // и восстанавливаем старую,
                 // поскольку отсутствует присвоение.
      token = String.Copy(temptoken);
      tokType = ttokType;
    }
    else {
      GetToken(); // Получаем следующую часть
                  // выражения exp.
      EvalExp2(out result);
      vars[varIdx] = result;
      return;
    }
  }

  EvalExp2(out result);
}

// Складываем или вычитаем два члена выражения.
void EvalExp2(out double result)
{
  string op;
  double partialResult;

  EvalExp3(out result);
  while((op = token) == "+" || op == "-") {
    GetToken();
    EvalExp3(out partialResult);
    switch(op) {
      case "-":
        result = result - partialResult;
        break;
      case "+":
        result = result + partialResult;
        break;
    }
  }
}

// Выполняем умножение или деление двух множителей.
void EvalExp3(out double result)
{
  string op;
  double partialResult = 0.0;

  EvalExp4(out result);
  while((op = token) == "*" ||
        op == "/" || op == "%") {
    GetToken();
    EvalExp4(out partialResult);
    switch(op) {
      case "*":
        result = result * partialResult;
        break;
      case "/":
        if(partialResult == 0.0)
          SyntaxErr(Errors.DIVBYZERO);
```

```
        result = result / partialResult;
        break;
      case "%":
        if(partialResult == 0.0)
          SyntaxErr(Errors.DIVBYZERO);
        result = (int) result % (int) partialResult;
        break;
    }
  }
}

// Выполняем возведение в степень.
void EvalExp4(out double result)
{
  double partialResult, ex;
  int t;

  EvalExp5(out result);
  if(token == "^") {
    GetToken();
    EvalExp4(out partialResult);
    ex = result;
    if(partialResult == 0.0) {
      result = 1.0;
      return;
    }
    for(t=(int)partialResult-1; t > 0; t--)
      result = result * (double)ex;
  }
}

// Выполняем операцию унарного + или -.
void EvalExp5(out double result)
{
  string  op;

  op = "";
  if((tokType == Types.DELIMITER) &&
      token == "+" || token == "-") {
    op = token;
    GetToken();
  }
  EvalExp6(out result);
  if(op == "-") result = -result;
}

// Обрабатываем выражение в круглых скобках.
void EvalExp6(out double result)
{
  if((token == "(")) {
    GetToken();
    EvalExp2(out result);
    if(token != ")")
      SyntaxErr(Errors.UNBALPARENS);
    GetToken();
  }
  else Atom(out result);
}
```

```
// Получаем значение числа или переменной.
void Atom(out double result)
{
  switch(tokType) {
    case Types.NUMBER:
      try {
        result = Double.Parse(token);
      } catch (FormatException) {
        result = 0.0;
        SyntaxErr(Errors.SYNTAX);
      }
      GetToken();
      return;
    case Types.VARIABLE:
      result = FindVar(token);
      GetToken();
      return;
    default:
      result = 0.0;
      SyntaxErr(Errors.SYNTAX);
      break;
  }
}

 // Возвращаем значение переменной.
double FindVar(string vname)
{
  if(!Char.IsLetter(vname[0])){
    SyntaxErr(Errors.SYNTAX);
    return 0.0;
  }
  return vars[Char.ToUpper(vname[0])-'A'];
}

// Возвращаем лексему во входной поток.
void PutBack()
{
  for(int i=0; i < token.Length; i++) expIdx--;
}

// Обрабатываем синтаксическую ошибку.
void SyntaxErr(Errors error)
{
  string[] err = {
    "Синтаксическая ошибка",
    "Дисбаланс скобок",
    "Выражение отсутствует",
    "Деление на нуль"
  };

  throw new ParserException(err[(int)error]);
}

// Получаем следующую лексему.
void GetToken()
{
  tokType = Types.NONE;
  token = "";
```

```
    if(expIdx == exp.Length) return; // Конец выражения.

    // Опускаем пробел.
    while(expIdx < exp.Length &&
          Char.IsWhiteSpace(exp[expIdx])) ++expIdx;

    // Хвостовой пробел завершает выражение.
    if(expIdx == exp.Length) return;

    if(IsDelim(exp[expIdx])) { // Это оператор?
      token += exp[expIdx];
      expIdx++;
      tokType = Types.DELIMITER;
    }
    else if(Char.IsLetter(exp[expIdx])) { // Это
                                          // переменная?
      while(!IsDelim(exp[expIdx])) {
        token += exp[expIdx];
        expIdx++;
        if(expIdx >= exp.Length) break;
      }
      tokType = Types.VARIABLE;
    }
    else if(Char.IsDigit(exp[expIdx])) { // Это число?
      while(!IsDelim(exp[expIdx])) {
        token += exp[expIdx];
        expIdx++;
        if(expIdx >= exp.Length) break;
      }
      tokType = Types.NUMBER;
    }
  }

  // Метод возвращает значение true,
  // если с --  разделитель.
  bool IsDelim(char c)
  {
    if((" +-/*%^=()".IndexOf(c) != -1))
      return true;
    return false;
  }
}
```

Чтобы опробовать этот усовершенствованный анализатор, используйте ту же программу, которую мы применяли для запуска упрощенного варианта. Но теперь можно вводить выражения, подобные следующим:

```
A = 10/4
A - B
C = A * (F - 21)
```

Синтаксический контроль в рекурсивном нисходящем анализаторе

В процессе анализа выражений под синтаксической ошибкой понимается ситуация, когда входное выражение не соответствует правилам функционирования анализатора. В большинстве случаев имеет место ошибка оператора — обычно это просто опечатка (ошибка, допускаемая при вводе с клавиатуры). Например, следующие выражения недействительны для анализаторов, представленных в этой главе:

```
10 ** 8
((10 - 5) * 9
/8
```

Первое выражение содержит два оператора рядом, во втором — дисбаланс скобок, а в последнем знак деления стоит в начале выражения. Все эти ситуации недопустимы для анализатора. Поскольку синтаксические ошибки могут "спровоцировать" анализатор на выдачу неверных результатов, необходимо позаботиться о защите от них.

При изучении кода анализатора вы, вероятно, обратили внимание на метод `SyntaxErr()`, который вызывается при определенных обстоятельствах. В отличие от других типов анализаторов, в рекурсивно-нисходящем используется простой контроль синтаксиса, который реализован в методах `Atom()`, `FindVar()` или `EvalExp6()`, где проверяется баланс скобок.

При вызове метода `SyntaxErr()` генерируется исключение типа `ParserException`, которое содержит описание ошибки. Это исключение перехватывается в методе `Evaluate()`. Следовательно, при обнаружении ошибки работа анализатора немедленно останавливается. При необходимости такое поведение можно изменить.

Что еще можно сделать

Как упоминалось выше в этой главе, в анализаторе предусмотрена проверка ошибок лишь в минимальном диапазоне. Возможно, вы хотели бы детализировать описание ошибок, например, выделять в выражении позицию, в которой обнаружена ошибка. Это позволило бы быстро находить и исправлять ошибки.

Приведенный здесь анализатор предназначен для вычисления только числовых выражений. Но, внеся определенные дополнения, можно "научить" его вычислять другие типы выражений, обрабатывать строки, пространственные координаты или комплексные числа. Например, чтобы анализатор мог обрабатывать строковые объекты, необходимо внести следующие изменения:

- определить новый тип лексемы `STRING`;
- усовершенствовать метод `GetToken()`, чтобы он мог распознавать строки;
- добавить новую `case`-ветвь в методе `Atom()` для обработки лексем типа `STRING`.

После реализации этих действий анализатор сможет обрабатывать строковые выражения, подобные следующим:

```
a = "стерео"
b = "звучание"
c = a + b
```

В результате строка `c` должна содержать конкатенацию строк `a` и `b`, т.е. `"стереозвучание"`.

Предлагаем вам неплохое применение для анализатора: создайте простой всплывающий мини-калькулятор, который принимает вводимое пользователем выражение и отображает результат. Такой калькулятор будет прекрасным дополнением почти к любому коммерческому приложению.

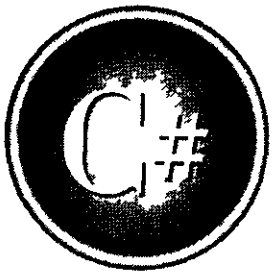
Наконец, попробуйте преобразовать класс `Parser` в компонент. Это совсем нетрудно сделать. Сначала сделайте класс `Parser` производным от класса `Component`. Затем реализуйте метод `Dispose(bool)`. И все! После этого калькулятор будет доступным для любого приложения.

Полный справочник по C#

Приложения

Создание языка C#

Полный справочник по



Приложение А

Краткий обзор языка комментариев XML

Язык C# поддерживает три типа комментариев. Первые два характеризуются символами // и /* */. Третий тип основан на XML-тегах и называется XML-комментарием. (XML-комментарии также называется комментариями документации.) Каждая строка XML-комментария начинается с символов ///. XML-комментариями обычно предваряется определение классов, пространств имен, методов, свойств и событий. Используя XML-комментарии, можно встраивать информацию о программе в саму программу. При компиляции такой программы вы получите XML-комментарии собранными в XML-файле. XML-комментарии можно утилизировать с помощью средства IntelliSense системы Visual Studio.

Теги языка комментариев XML

C# поддерживает теги XML-документации, представленные в табл. А.1. Назначение большинства из них можно понять по описанию, да и работают они подобно другим XML-тегам, с которыми многие программисты уже знакомы. Однако тег <list> (<список>) несколько сложнее других. Список содержит два компонента: заголовок списка и элементы списка. Базовый формат заголовка списка такой:

```
<listheader>
  <term> имя </term>
  <description> текст </description>
</listheader>
```

Здесь *текст* описывает *имя*. Для таблицы элемент *текст* не используется. Базовый формат элемента списка таков:

```
<item>
  <term> имя_элемента </term>
  <description> текст </description>
</item>
```

Здесь *текст* описывает *имя_элемента*. Для маркированных или нумерованных списков или таблиц элемент *имя_элемента* не используется. Можно использовать несколько тегов <item>.

Таблица А.1. Теги языка комментариев XML

Тег	Описание
<c> *код* </c>	Определяет текст, заданный элементом *код*, как программный код
<code> *код* </code>	Определяет несколько строк текста, заданных элементом *код*, как программный код
<example> *трактовка* </example>	Текст, соответствующий элементу *трактовка*, описывает пример кода
<exception cref = "*имя*"> *explanation* </exception>	Описывает исключительную ситуацию, заданную элементом *имя*
<include file = '*имя_файла*' path = '*путь* [@*имя_тега* = "*ID_тега*"]' />	Определяет файл, содержащий XML-комментарии для текущего файла. Файл задается элементом *имя_файла*. Каталоговый путь к тегу, имя тега и ID тега задаются элементами *путь*, *имя_тега*, и *ID_тега*, соответственно
<list type = "*тип*"> *заголовок_списка* *элементы_списка* </list>	Определяет список. Тип списка задается элементом *тип*, который может принимать одно из следующих значений: `bullet` (маркированный), `number` (нумерованный) или `table` (таблица)

Окончание табл. А.1

Тег	Описание
`<para> текст </para>`	Определяет абзац текста в другом теге
`<param name = 'имя_параметра'> трактовка </param>`	Описывает параметр, заданный элементом *имя_параметра*. Описание содержится в тексте, соответствующем элементу *трактовка*
`<paramref name = "имя_параметра" />`	Означает, что элемент *имя_параметра* представляет собой имя параметра
`<permission cref = "идентификатор"> трактовка </permission>`	Описывает параметр разрешения, связанный с членами класса, заданными элементом *идентификатор*. Параметры разрешения описаны в тексте, соответствующем элементу *трактовка*
`<remarks> трактовка </remarks>`	Текст, заданный элементом *трактовка*, представляет собой общие комментарии, которые часто используются для описания класса или структуры
`<returns> трактовка </returns>`	Текст, заданный элементом *трактовка*, описывает значение, возвращаемое методом
`<see cref = "идентификатор" />`	Объявляет ссылку на другой элемент, заданный элементом *идентификатор*
`<seealso cref = "идентификатор" />`	Объявляет перекрестную ссылку типа "см. также" на *идентификатор*
`<summary> трактовка </summary>`	Текст, заданный элементом *трактовка*, представляет собой общие комментарии, которые часто используются для описания метода или другого члена класса
`<value> трактовка </value>`	Текст, заданный элементом *трактовка*, описывает свойство

Компиляция XML-документа

Чтобы создать XML-файл, который содержит комментарии к документу, задайте опцию `/doc`. Например, чтобы скомпилировать файл `DocTest.cs`, содержащий XML-комментарии, используйте такую командную строку:

```
csc DocTest.cs /doc:DocTest.xml
```

Чтобы создать выходной XML-файл при использовании интегрированной среды Visual Studio IDE, необходимо использовать диалоговое окно Property Pages (Страницы свойств), которое активизируется при выборе команды View⇨Property Pages (Вид⇨Страницы свойств). Затем выберите команду Configuration Properties⇨Build (Свойства конфигурации⇨Построить). После этого укажите имя XML-файла в свойстве XML Documentation File (XML-файл документации).

Пример XML-документа

Рассмотрим пример использования XML-комментариев:

```
// Пример XML-документа.

using System;
```

```
/// <remark>
/// Это пример XML-документа.
/// В классе Test показано использование ряда тегов.
/// </remark>

class Test {
  /// <summary>
  /// Выполнение начинается с метода Main().
  /// </summary>
  public static void Main() {
    int sum;

    sum = Summation(5);
        Console.WriteLine("Сумма последовательных " +
                          5 + " чисел равна " + sum);
  }

  /// <summary>
  /// Метод Summation() возвращает сумму ряда чисел.
  /// <param name = "val">
  /// Последнее слагаемое передается в параметре val.
  /// </param>
  /// <see cref="int"> </see>
  /// <returns>
  /// Результат возвращается как int-значение.
  /// </returns>
  /// </summary>
  static int Summation(int val) {
    int result = 0;

    for(int i=1; i <= val; i++)
      result += i;

    return result;
  }
}
```

Предположим, что приведенная выше программа называется `XmlTest.cs`. С помощью следующей строки

```
csc XmlTest.cs /doc:XmlTest.xml
```

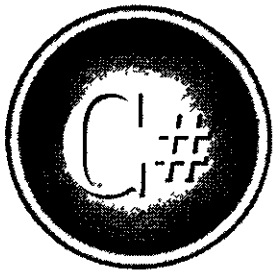
эту программу можно скомпилировать, а в результате компиляции будет создан файл `XmlTest.xml`, который должен содержать следующие комментарии:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>t</name>
    </assembly>
    <members>
        <member name="T:Test">
            <remark>
            Это пример XML-документа.
            В классе Test показано использование ряда тегов.
            </remark>
        </member>
        <member name="M:Test.Main">
            <summary>
            Выполнение начинается с метода Main().
```

```
            </summary>
        </member>
        <member name="M:Test.Summation(System.Int32)">
            <summary>
            Метод Summation() возвращает сумму ряда чисел.
            <param name="val">
            Последнее слагаемое передается в параметре val.
            </param>
            <see cref="T:System.Int32"> </see>
            <returns>
            Результат возвращается как int-значение.
            </returns>
            </summary>
        </member>
    </members>
</doc>
```

Обратите внимание на то, что каждому документированному элементу присваивается уникальный идентификатор. Эти идентификаторы могут использовать другие программы, которые включают XML-комментарии.

Полный справочник по



Приложение Б

C# и робототехника

Своим хобби я считаю робототехнику Много лет тому назад это было также моей работой, поскольку я разрабатывал и внедрял язык управления промышленными роботами Мне было очень интересно заниматься роботами, потому что они оживляли логику программ, которые мы писали Кроме того, они взаимодействовали с реальным миром Несмотря на то что в течение долгого времени я уже не сталкивался с робототехникой на профессиональном уровне, эта область всегда оставалась для меня очень важной и волнующей И здесь я вспомнил о своем давнем увлечении по той простой причине, что язык C# предлагает специалистам в области подготовки программ для роботов воспользоваться некоторыми преимуществами

Обычно, когда говорят о программах управления роботами, подразумевают высокоэффективные процедуры, написанные на языке C++ Но C# может изменить это устоявшееся мнение Дело в том, что программы управления роботами часто имеют огромный размер и содержат множество таких подсистем, как дистанционное управление, искусственное (техническое) зрение, распознавание образов, управление двигателем и т д Эти подсистемы можно организовать (и реализовать) в виде коллекции C#-компонентов Применение компонентов к программированию роботов поможет преодолеть сложность, которая прежде была присуща программам, создаваемым в этой области Компоненты позволят также легко изменять подсистемы и модернизировать их

Если вам близка тема робототехники (особенно, если вы занимаетесь созданием собственного робота), то робот, показанный на рис Б 1, может вас заинтересовать Это мой тестовый робот И интересен он вот чем Во-первых, он содержит встроенный микропроцессор, который обеспечивает базовое управление двигателем и сенсорную обратную связь Во-вторых, он содержит приемопередатчик RS-232, который используется для получения инструкций от главного компьютера и возвращения результатов Такая организация позволяет удаленному компьютеру выполнять интенсивную обработку данных (и это помимо того, чем нагружен сам робот) В-третьих, он содержит видеокамеру, которая связана с беспроводным видеопередатчиком

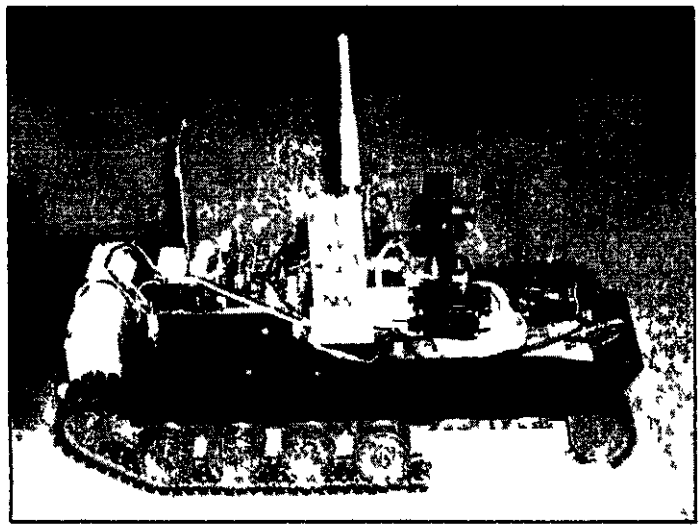


Рис Б 1 Простои, но деиствующии тестовыи робот

Этот робот создан на базе ходовой части танка Hobbico M1 Abrams R/C. (Я обнаружил, что шасси модели R/C танков и автомобилей можно успешно использовать для робота.) Из танка была убрана большая часть внутрикорпусных устройств, включая приемник (ресивер) и устройство переключения передач, но оставлены двигатели. Танк Hobbico прекрасно подходит для платформы робота благодаря его устойчивости и отличным двигателям; он может перевозить большой груз, а гусеницы не дают ему опрокинуться. Кроме того, благодаря гусеницам, робот обладает нулевым радиусом поворота и может перемещаться по неровной поверхности. Длина ходовой части около 18, а ширина — около 8 дюймов.

К пустой ходовой части я добавил детали. Для обеспечения автономного управления я использовал микропроцессор BASIC Stamp 2, который отличается простотой и мощностью (фирма-изготовитель — Parallax, Inc. (`www.parallaxinc.com`). Приемопередатчик RS-232, видеокамера и радиопередатчик также от компании Parallax. Как радиопередатчик RS-232, так и видеопередатчик имеют диапазон покрытия около 300 футов. Я также добавил для танкового двигателя электронные регуляторы скорости, которые управляются микропроцессором BASIC Stamp.

Рассмотрим теперь, как функционирует робот. Удаленный компьютер запускает основную программу управления роботом. Эта программа отрабатывает такие "тяжелые" подсистемы, как техническое зрение, дистанционное управление и пространственная ориентация. Она также может запомнить последовательность действий, а затем повторить их. Удаленный компьютер передает роботу команды управления перемещением (через радиосвязь RS-232). Микропроцессор BASIC Stamp принимает эти команды и выполняет соответствующие действия. Например, при получении команды "вперед" микропроцессор BASIC Stamp посылает соответствующие сигналы электронным регуляторам скорости, связанным с двигателями. После выполнения команды робот возвращает код подтверждения приема. Таким образом, между удаленным компьютером и роботом имеется двунаправленную связь, благодаря которой робот может подтверждать успешное выполнение каждой команды.

Поскольку основная обработка данных по роботу выполняется в удаленном компьютере, на объем обрабатываемых данных не налагается строгих ограничений. Например, на момент написания книги робот мог следить за объектом с помощью системы технического зрения. Реализация этой возможности требует обработки довольно большого объема данных, которую было бы трудно перенести на микропроцессор робота.

Насколько мне известно, большинство программ управления роботами по-прежнему пишется на языке C++. Но как только у меня появится свободное время, я обязательно займусь их переводом на C#. И начать эту работу я планирую с подсистемы, которая повторяет заданную последовательность команд. Эту "списочную" задачу нетрудно решить, опираясь на соответствующую C#-коллекцию.

Предметный указатель

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ш

Я