Programming C#

Second Edition

Jesse Liberty

O'REILLY®

Программирование на C#

Второе издание

Джесс Либерти

Санкт-Петербург — Москва
2003

Джесс Либерти

Программирование на C#, 2-е издание

Перевод С. Иноземцева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научные редакторы	*А. Королев, Н. Секунов*
Редакторы	*А. Лосев, В. Овчинников*
Корректор	*С. Беляева*
Верстка	*Н. Гриценко, А. Дорошенко*

Либерти Д.
Программирование на C#. – Пер. с англ. – СПб: Символ-Плюс, 2003. – 688 с., ил.
ISBN 5-93286-038-3

Созданный в Microsoft специально для новой платформы .NET, язык C# прост, безопасен и нацелен на работу в Интернете. Книга «Программирование на C#» поможет вам быстро приступить к практической разработке веб- и Windows-приложений на платформе .NET, работающих как на локальном компьютере, так и в Интернете.

В первой части книги обсуждаются основы языка и такие понятия ООП, как классы и объекты, наследование и полиморфизм, перегрузка операций, структуры и интерфейсы, массивы, индексаторы и коллекции, строковые объекты и регулярные выражения, исключения и обработка ошибок, делегаты и события. Вторая часть целиком посвящена созданию приложений и содержит обсуждение ADO.NET, ASP.NET и Windows Forms. Технология ASP.NET включает в себя как Web Forms для быстрой разработки веб-приложений, так и Web Services для создания объектов без графического интерфейса, оказывающих услуги в Интернете. В третьей части рассматривается платформа .NET Framework. Особое внимание уделено атрибутам и отражению, удаленным объектам, вычислительным потокам и синхронизации, а также потокам ввода/вывода. Здесь же представлены способы взаимодействия с объектами COM.

ISBN 5-93286-038-3
ISBN 0-596-00309-9 (англ)

Издательство «Символ-Плюс». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 25.11.2002. Формат 70x100 1/16 . Печать офсетная.
Объем 43 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

Приблизительно каждые десять лет новый подход к программированию накатывается, подобно цунами. В начале 80-х новыми технологиями были операционная система Unix, работавшая на рабочих станциях, и новый мощный язык, разработанный в AT&T и названный C. Начало 90-х принесло с собой Windows и C++. Каждая из этих разработок представляла принципиальное изменение подхода к программированию. Пришло время следующей волны – появилась платформа .NET и язык C#. И цель данной книги – помочь читателю оседлать эту волну.

Фирма Microsoft «все поставила на карту» под названием .NET. Когда компания такого размера и с таким влиянием тратит миллиарды долларов на реорганизацию своей структуры для поддержки новой платформы, программисты не могут оставить это без внимания. Оказывается, .NET коренным образом меняет способ мышления программиста. Если коротко, она представляет собой новую платформу разработки программного продукта, предназначенную для упрощения объектно-ориентированного программирования в Интернете. В качестве основного языка этой объектно-ориентированной платформы, нацеленной на Интернет, выбран C#. Его создатели хорошо усвоили уроки, преподанные такими языками, как C (высокая производительность), C++ (объектно-ориентированная структура), Java (сборка мусора, высокая безопасность) и Visual Basic (быстрая разработка). В результате получился язык, идеально подходящий для разработки компонентных n-уровневых распределенных веб-приложений.

Об этой книге

Данная книга является учебником как по языку C#, так и по программированию приложений .NET с его помощью. Читателям, знающим какой-либо язык программирования, достаточно будет бегло ознакомиться с содержимым первых глав, однако главу 1 прочитать все-таки необходимо, поскольку она представляет обзор языка и платформы .NET. Новичку же стоит прочитать эту книгу так, как Король Червей рекомендовал Белому Кролику: «Начни с начала, продолжай, пока не дойдешь до конца. Как дойдешь – кончай!»[1]

[1] Льюис Кэррол «Приключения Алисы в Стране чудес», пер. Н. Демуровой.

Как организована эта книга

Часть I посвящена тонкостям языка C#. В части II говорится, как создавать приложения .NET, а в части III описано, как использовать C# с библиотекой времени выполнения CLR (Common Language Runtime) платформы .NET.

Часть I. Язык программирования C#

Глава 1 «C# и .NET Framework» является введением в язык C# и платформу .NET Framework.

Глава 2 «Начинаем. Программа Hello World» демонстрирует простую программу, создающую контекст для последующего обсуждения, знакомит читателя с инструментальной средой разработки Visual Studio .NET и с некоторыми концепциями языка C#.

Глава 3 «Основы языка программирования C#» представляет основы языка, от базовых типов данных до ключевых слов.

Классы определяют новые типы и позволяют программисту расширить язык так, чтобы лучше смоделировать решаемую задачу. В главе 4 «Классы и объекты» описаны компоненты, являющиеся сердцем и душой языка C#.

Классы могут быть сложными представлениями и абстракциями реальных вещей. В главе 5 «Наследование и полиморфизм» обсуждается, как классы взаимодействуют друг с другом и в каких отношениях они могут находиться.

Глава 6 «Перегрузка операций» учит читателя добавлять операции в типы, определенные пользователем.

Главы 7 и 8 представляют «Структуры» и «Интерфейсы», соответственно. И те и другие являются близкими родственниками классов. Структуры – это облегченные объекты, обладающие меньшими возможностями, чем классы, и менее требовательные к операционной системе и объему памяти. Интерфейсы являются своего рода соглашениями. Они описывают, как работает класс, чтобы другие программисты могли взаимодействовать с объектами четко определенным способом.

Объектно-ориентированные программы нередко создают большое количество объектов. Эти объекты удобно объединять в группы и манипулировать ими как единым целым. Язык C# предоставляет широкую поддержку классов коллекций. В главе 9 «Массивы, индексаторы и классы коллекции» исследуются классы коллекций, предоставляемые библиотекой классов платформы (FCL, Framework Class Library), и демонстрируется, как можно создавать собственные типы коллекций.

В главе 10 «Строки и регулярные выражения» обсуждается, как в языке C# обрабатывается текст. Большинство веб- и Windows-приложений взаимодействует с пользователем, и строки играют важнейшую роль в пользовательском интерфейсе.

В главе 11 «Обработка исключений» поясняется, что делать с исключениями, представляющими собой объектно-ориентированный механизм обработки различных нештатных ситуаций.

Приложения, работающие как в Windows, так и во Всемирной паутине, управляются событиями. В языке C# события являются полноценными элементами языка. В главе 12 «Делегаты и события» описаны принципы работы с событиями и использование для этого *делегатов*, представляющих собой объектно-ориентированный механизм обратного вызова, гарантирующий безопасность преобразования типов.

Часть II. Приемы программирования на C#

Вторая и третья части книги будут интересны всем читателям, независимо от их опыта программирования на других языках. В этих главах исследуется сама платформа .NET.

Часть II посвящена тому, как писать программы для .NET, а именно Windows-приложения с помощью Windows Forms и веб-приложения с помощью Web Forms. Кроме того, в части II показано, как взаимодействовать с базами данных и создавать веб-службы.

На верхнем уровне этой инфраструктуры находится высокоуровневая абстракция операционной системы, предназначенная для облегчения разработки объектно-ориентированного программного продукта. В этот верхний уровень входят ASP.NET и Windows Forms. Технология ASP.NET включает в себя как набор инструментов Web Forms для быстрой разработки веб-приложений, так и веб-службы для создания веб-объектов без пользовательского интерфейса.

C# предоставляет модель быстрой разработки приложений (RAD, Rapid Application Development), аналогичную той, что ранее была доступна только в среде Visual Basic. В главе 13 «Создание Windows-приложений» описано, как пользоваться моделью RAD для создания профессиональных Windows-приложений с помощью среды разработки Windows Forms.

Независимо от того, предназначено приложение для Windows или Всемирной паутины, ему нередко приходится обрабатывать большие объемы информации. В главе 14 «Доступ к данным с помощью ADO.NET» обсуждается уровень ADO.NET платформы .NET Framework и описываются способы взаимодействия с сервером Microsoft SQL Server и другими поставщиками данных.

В главе 15 технология RAD, описанная в главе 13, объединяется с технологиями обработки данных, описанными в главе 14 с целью продемонстрировать «Создание веб-приложений с помощью Web Forms».

Не все приложения имеют пользовательский интерфейс. В главе 16 «Веб-службы» внимание уделено второй части технологии ASP.NET. *Веб-служба* – это распределенное приложение, предоставляющее опре-

деленные функциональные услуги с использованием веб-протоколов, как правило, протоколов XML и HTTP.

Часть III. CLR и .NET Framework

Среда времени выполнения – это окружение, в котором исполняются программы. CLR (Common Language Runtime, общеязыковая среда выполнения) – это сердце .NET. Она включает в себя систему приведения типов данных, которая действует на всей платформе и является общей для всех языков, входящих в .NET. Среда CLR несет ответственность за управление памятью и подсчет ссылок на объекты.

Другой ключевой особенностью среды .NET CLR является *сборка мусора*. В отличие от традиционного программирования на C/C++, разработчик, пишущий на C#, не отвечает за уничтожение объектов. Остались в прошлом долгие часы, потраченные на поиски утечки памяти; среда CLR «прибирает» за программистом, когда тот или иной объект становится ненужным. Сборщик мусора CLR ищет в куче объекты, на которые нет ссылок, и освобождает занимаемую ими память.

Платформа .NET и библиотека классов расширяется в платформе среднего уровня, где реализована инфраструктура, включающая в себя классы поддержки, в том числе типы, обеспечивающие связь между процессами, обработку XML, работу с потоками, ввод/вывод, безопасность, диагностику и т. д. В средний уровень входят также компоненты доступа к базам данных, известные под общим именем ADO.NET и обсуждаемые в главе 14.

В третьей части книги рассматриваются взаимоотношения языка C# со средой выполнения CLR и библиотекой классов платформы FCL.

В главе 17 «Сборки и контроль версий» проводится различие между частными (закрытыми) и разделяемыми сборками и демонстрируется, как нужно создавать сборки и управлять ими. В терминологии .NET *сборкой* называется коллекция файлов, представленная пользователю в виде единой библиотеки динамической компоновки (DLL) или выполняемого файла. Сборка является базовой единицей для повторного использования кода, контроля версий, защиты и развертывания.

Сборки платформы .NET содержат обширную информацию о классах, методах, свойствах, событиях и т. п., представленную в виде метаданных. Эти метаданные компилируются в программу и извлекаются из нее использующим ее приложением с помощью отражения. В главе 18 «Атрибуты и отражение» показано, как добавлять метаданные в программу, как создавать пользовательские атрибуты и как читать эти метаданные с помощью отражения. Кроме того, глава содержит обсуждение динамических вызовов, приема программирования, при котором методы вызываются с помощью позднего связывания (то есть на этапе выполнения), а заканчивается глава демонстрацией *динамичес-*

кой генерации кода, нетривиальной техники построения самомодифицирующегося кода.

Платформа .NET Framework была разработана специально для поддержки веб-приложений и распределенных приложений. Компоненты, созданные на языке C#, могут находиться в разных процессах на одном компьютере или на разных компьютерах в локальной сети или Интернете. *Маршалинг* (*marshaling*) представляет собой способ взаимодействия с отсутствующими в данном процессе объектами, а *отдаление* (*remoting*) – технологию связи с такими объектами. Эта тема освещается в главе 19 «Маршалинг и удаленные компоненты».

Библиотеки классов платформы предоставляют широкую поддержку асинхронного ввода/вывода, а также включают в себя другие классы, делающие явную манипуляцию вычислительными потоками необязательной. Тем не менее, в языке C# есть средства управления как вычислительными потоками, так и синхронизацией, обсуждаемые в главе 20 «Потоки и синхронизация».

В главе 21 «Потоки данных» рассматриваются потоки данных, механизм, который позволяет не только взаимодействовать с пользователем, но и получать данные из Интернета. В главе подробно описывается *сериализация* (*serialization*) – технология, позволяющая записать объектный граф на диск и прочитать его с диска.

В главе 22 «Взаимодействие .NET и COM» исследуется взаимодействие с компонентами COM, созданными вне управляемой среды платформы .NET Framework. Существует возможность вызова из кода C# компонентов COM, а также выззова из COM компонентов, созданных на языке C#.

Книга завершается приложением, где перечисляются ключевые слова языка C#.

Для кого эта книга

Эта книга написана для программистов, которые хотят разрабатывать приложения на платформе .NET. Без сомнения, многие из них уже имеют опыт работы с такими языками, как C++, Java или Visual Basic (VB). Возможно, некоторые читатели программируют на других языках, а иные вообще не являются профессиональными программистами, но работали с HTML и другими веб-технологиями. Эта книга ориентирована на все названные категории читателей, хотя людям, никогда не писавшим программы, некоторые разделы покажутся трудными.

Языки C# и Visual Basic .NET

Все языки платформы .NET Framework равны. Однако, перефразируя Джорджа Оруэлла, можно сказать, что некоторые языки равнее дру-

гих. C# является великолепным языком для разработок на платформе .NET. Он исключительно гибок, устойчив и хорошо продуман создателями. В настоящее время он чаще других языков используется в статьях и книгах по .NET-программированию.

Вполне возможно, что некоторые программисты, работающие на VB, предпочтут изучение C# совершенствованию своих навыков в языке VB на уровне VB.NET. Это будет естественным выбором, поскольку переход от VB6 к VB.NET вряд ли окажется легче, чем от VB6 к C# (по крайней мере, с точки зрения автора). Кроме того, исторически сложилось, что программирование на языках семейства C оплачивается выше программирования на языках семейства Basic, хотя, возможно, это и не вполне справедливо. На практике программисты, работающие на VB, никогда не пользовались таким уважением и материальными благами, которых заслуживают, и язык C# предлагает им прекрасную возможность перейти на более высокооплачиваемую работу.

Как бы то ни было, приглашаются все, кто имеет опыт программирования на VB! Автор книги имел в виду и эту категорию читателей и постарался облегчить им переход на другой язык.

Языки C# и Java

Возможно, программисты, пишущие на языке Java, смотрят на C# со смешанным чувством тревоги, радости и обиды. Предполагалось, что C# будет в определенном смысле «оторван» от языка Java. Автор воздержится от комментариев по поводу религиозной войны между Microsoft и «всеми, кто не Microsoft». Однако объективности ради следует признать, что C# многому научился от Java. Но язык Java сам многому научился от C++, а тот позаимствовал синтаксис от C, который, в свою очередь, извлек уроки из опыта других языков. Все мы стоим на плечах гигантов.

Переход на C# для программистов, привыкших к Java, не составит труда – они найдут в нем сходный синтаксис и знакомую и комфортную семантику. Возможно, программисты, работающие на Java, хотят, чтобы особое внимание было сосредоточено на различиях между двумя языками, что позволило бы им использовать C# максимально эффективно. Автор постарался удовлетворить это желание, делая комментарии по ходу изложения (см. замечания, предназначенные для программистов, пишущих на Java).

Языки C# и C++

Хотя программировать в среде .NET на языке C++ можно, это непросто и неестественно. Честно говоря, автор этих строк, программировавший на языке C++ десять лет и написавший дюжину книг по данному предмету, предпочел бы визит к стоматологу программированию на

языке C++ в среде .NET. Возможно, причина тому в более дружественном поведении C#. Как бы то ни было, узнав язык C#, автор не хочет возвращаться к старому.

Впрочем, программисту, привыкшему к C++, следует быть очень внимательным. На его пути встретится немало ловушек, и автор постарался отметить их, расставив в тексте книги предупреждения специально для профессионалов в области C++.

Соглашения, используемые в этой книге

В оформлении книги приняты следующие соглашения:

Курсив используется для обозначения:

- путей в файловой системе, имен файлов и названий программ;
- адресов в Интернете, таких как имена доменов и URL-адреса;
- новых терминов в местах, где даются их определения.

`Моноширинный шрифт` используется для обозначения:

- командных строк и параметров, которые следует ввести в компьютер буквально;
- имен и ключевых слов в примерах программ, включая имена методов, переменных и классов.

`Моноширинный наклонный шрифт` обозначает замещаемые элементы синтаксиса или кода, например изменяемые или необязательные конструкции.

`Моноширинный полужирный шрифт` используется для выделения участков кода.

Особое внимание уделяйте замечаниям, выделенным в тексте следующим образом:

Это совет. Он содержит дополнительную информацию по обсуждаемой теме.

Это предупреждение. Оно помогает справиться с проблемами и даже избежать их.

Техническая поддержка

В числе своих обязанностей автор видит поддержку написанных им книг на веб-сайте:

http://www.LibertyAssociates.com

Там же можно найти исходный код всех примеров, приводимых в этой книге, дискуссионную группу и отдельный раздел с ответами на воп-

росы относительно C#. Однако прежде чем посылать свой вопрос, просмотрите FAQ (перечень наиболее часто задаваемых вопросов) и список замеченных опечаток (errata). Если это не помогло решить проблему, посылайте вопрос в дискуссионный центр.

Самый эффективный способ получить помощь – задать вопрос, сформулированный максимально точно, а лучше написать маленькую программу, которая иллюстрировала бы возникшую проблему. В качестве альтернативы читатель может посетить различные группы новостей или дискуссионные центры в Интернете. Фирма Microsoft ведет целый ряд групп новостей, а Developmentor (*http://www.develop.com*) поддерживает замечательный дискуссионный лист сообщений, пришедших по электронной почте на тему .NET. То же самое можно сказать и про сайт Чарльза Кэррола (Charles Carroll), находящийся по адресу *http://www.asplists.com*.

Обратная связь

Коллектив издательства максимально тщательно проверил информацию, содержащуюся в этой книге, однако не исключено, что некоторые сведения устарели (или даже ошибочны!). Письма с указанием замеченных ошибок и предложениями относительно следующих изданий направляйте, пожалуйста, по адресу:

O'Reilly & Associates, Inc.
005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (телефон в США или Канаде)
(707) 829-0515 (международный/местный телефон)
(707) 829-0104 (факс)

На сайте издательства O'Reilly & Associates есть веб-страница, посвященная этой книге, где находятся примеры, список ошибок и планы на переиздание. Вся эта информация доступна по адресу в Интернете:

http://www.oreilly.com/catalog/progcsharp2

Адрес для замечаний и технических вопросов по книге:

bookquestions@oreilly.com

Более подробную информацию об этой и других книгах, а также технические статьи и дискуссию на тему C# и .NET Framework можно найти на сайте издательства O'Reilly & Associates:

http://www.oreilly.com

и на сайте O'Reilly .NET DevCenter:

http://www.oreillynet.com/dotnet

Благодарности

Чтобы обеспечить точность и полноту изложения и гарантировать, что книга будет соответствовать нуждам и интересам профессиональных программистов, я заручился поддержкой таких замечательных программистов, как Дональд Кси (Donald Xie), Дэн Хэрвиц (Dan Hurwitz), Сет Вайс (Seth Weiss), Сью Линч (Sue Lynch), Клифф Джеральд (Cliff Gerald) и Том Пэтр (Tom Petr). Джим Кулберт (Jim Culbert) не только написал рецензию на книгу и внес ряд ценных предложений, но и постоянно указывал мне на нужды программистов-практиков. Трудно переоценить вклад Джима в эту книгу.

Майк Вудринг (Mike Woodring) из Developmentor за неделю сообщил мне больше сведений о CLR, чем удалось бы узнать за полгода самостоятельных изысканий. Победить двух страшных зверей, C# и .NET, мне помогли многие сотрудники Microsoft и O'Reilly, в том числе Эрик Гуннерсон (Eric Gunnerson), Роб Ховард (Rob Howard), Пит Обермейер (Piet Obermeyer), Джонатан Хокинс (Jonathan Hawkins), Питер Дрейтон (Peter Drayton), Брэд Меррил (Brad Merrill), Бен Альбахари (Ben Albahari) и другие. Наверное, самый удивительный программист из всех, кого я знаю, – это Сьюзен Уоррен (Susan Warren). Я глубоко благодарен ей за помощь и советы.

Я нахожусь в неоплатном долгу перед Джоном Осборном (John Osborn), представившим меня издательству O'Reilly. Валери Куэрсиа (Valerie Quercia), Брайан Мак-Дональд (Brian McDonald), Джефф Холком (Jeff Holcomb), Клэр Клутье (Claire Cloutier) и Татьяна Диаз (Tatiana Diaz) сделали книгу гораздо лучше, чем представленная рукопись. Роб Романо (Rob Romano) создал некоторые иллюстрации и улучшил другие.

Многие читатели написали мне, указав на опечатки и небольшие ошибки в первом издании. Их усилия не были потрачены зря. Мы особенно признательны Sol Bick, Brian Cassel, Steve Charbonneau, Randy Eastwood, Andy Gaskall, Bob Kline, Jason Mauss, Mark Phillips, Christian Rodriguez, David Solum, Erwing Steininger, Steve Thomson, Greg Torrance и Ted Volk. Все мы хорошо потрудились над исправлением всех ошибок во втором издании. Еще мы подчистили книгу, чтобы убедиться, что в ней не появилось новых неточностей и что весь код компилируется и запускается правильно с последним релизом Visual Studio .NET. Но если вы все же найдете ошибку, проверьте, пожалуйста, список errata на моем сайте (*http://www.LibertyAssociates.com*) и, если эта ошибка новая, сообщите мне о ней по электронной почте *jliberty@libertyassociates.com*.

Ну и наконец, особая благодарность Брайану Джепсону (Brian Jepson), который отвечал и за повышенное качество второго издания, и за его своевременность. Я очень высоко оценил затраченные им усилия.

12

Делегаты и события

Когда умирает глава какого-нибудь государства, президент США не всегда имеет возможность присутствовать на похоронной церемонии. Вместо себя он посылает делегата. Часто таким делегатом является вице-президент, а если и его присутствие невозможно, президент посылает кого-нибудь другого, например государственного секретаря или даже первую леди. Президент не связывает такую ответственность с каким-то одним человеком; он делегирует свои полномочия тому, кто в состоянии соблюсти международный протокол.

Президент заранее определяет, какие функции должны быть выполнены (присутствие на похоронах), какие параметры следует передать (соболезнования) и что он рассчитывает услышать в ответ (слова признательности). Затем он наделяет конкретного человека делегируемыми полномочиями «на этапе выполнения», и это его, президента, рутинные обязанности.

В программировании нередко приходится сталкиваться с ситуациями, когда требуется выполнить некоторое действие, но заранее не известно, какой метод и даже какой объект будет выполнять его. Например, кнопка на экране знает, что она должна уведомить *некий* объект, когда пользователь щелкнет по ней. Однако она может и не знать, какому объекту или объектам нужно посылать уведомление. Вместо того чтобы жестко связывать кнопку с конкретным объектом, программист связывает ее с *делегатом*, а на этапе выполнения назначает делегатом тот метод, который потребуется.

В стародавние времена, на заре программирования, программа начинала свое выполнение, проделывала предопределенные шаги и завершала работу. Если в этот процесс включался пользователь, взаимодействие с ним было ограниченным и сводилось к заполнению полей.

В наши дни программная модель, основанная на графическом пользовательском интерфейсе, требует другого подхода, известного как *программирование, управляемое событиями*. Современная программа предоставляет пользователю интерфейс и ждет, когда он предпримет какое-либо действие. У пользователя богатый выбор таких действий. Он может выбирать команды меню, нажимать кнопки, вносить изменения в текстовые поля, щелкать по значкам и т. д. Каждое действие приводит к возникновению события. Кроме того, существуют события, непосредственно не связанные с действиями пользователя, например срабатывание таймера, приход сообщения по электронной почте или окончание операции копирования файлов.

Событие инкапсулирует идею «произошло нечто важное», и программа должна на него отреагировать. События и делегаты являются тесно связанными понятиями, поскольку гибкая обработка событий требует точного выбора обработчика. Обработчик события реализуется на языке C#, как правило, в виде делегата.

Делегаты используются и как процедуры обратного вызова, когда один класс как бы говорит другому: «Сделай эту работу, а когда закончишь – дай мне знать». Это второе применение делегатов подробно обсуждается в главе 21. Делегаты еще используются для указания методов, которые становятся известными только на этапе выполнения. Эта тема раскрывается в следующих разделах.

Делегаты

В языке C# делегаты – это полноценные объекты, без оговорок поддерживаемые языком. Технически делегат – это ссылочный тип, инкапсулирующий метод с указанной сигнатурой и возвращаемым типом. В делегате можно инкапсулировать любой подходящий метод. (В C++ и многих других языках программирования аналогичные цели достигаются с помощью указателей на методы класса и на функции. В отличие от них, делегаты объектно-ориентированы и обеспечивают проверку типов.)

Делегат создается ключевым словом `delegate`, за которым указывается возвращаемый тип и сигнатура делегируемых методов, например, так:

```
public delegate int WhichIsFirst(object obj1, object obj2);
```

В этом объявлении определяется делегат с именем `WhichIsFirst`, который инкапсулирует любой метод, принимающий два параметра типа `Object`, и возвращает значение целого типа.

Когда делегат определен, программист может инкапсулировать метод, создав экземпляр делегата путем передачи ему метода, соответствующего по сигнатуре и возвращаемому типу.

Применение делегатов для выбора методов на этапе выполнения

Делегаты указывают методы, которые могут быть использованы при обработке событий, а также для реализации обратных вызовов в программе. Кроме того, они применяются для указания статических методов и методов экземпляра, о которых ничего не известно до этапа выполнения.

Предположим, что программист хочет создать простой контейнерный класс `Pair`, который может принять и упорядочить любые два объекта. Невозможно знать заранее, какие именно объекты будут переданы, но можно внутри объектов определить методы сортировки и делегировать ответственность за упорядочивание объектов внутри класса самим этим объектам.

Разные объекты упорядочиваются по разным критериям. Например, два счетчика будут отсортированы по их числовым значениям, а две кнопки – в алфавитном порядке, по надписям на них. Разработчик класса `Pair` вправе ожидать, что объекты в паре знают, который из них должен быть первым. Поэтому он будет настаивать, чтобы объекты, передаваемые классу `Pair`, имели методы сортировки.

Программист определяет требуемый метод, создавая делегат с сигнатурой и возвращаемым типом метода, который должен быть предоставлен объектом (например кнопкой), чтобы класс `Pair` мог решить, какой объект первый, а какой – второй.

Класс `Pair` определяет делегат `WhichIsFirst`, а метод `Sort()` примет в качестве параметра экземпляр `WhichIsFirst`. Когда классу `Pair` потребуется узнать, в каком порядке хранить объекты, он вызовет делегат, передав ему два объекта в качестве аргументов. Ответственность за определение порядка следования объектов будет возложена на метод, инкапсулируемый делегатом.

Для тестирования делегата создадим два класса, `Dog` (Собака) и `Student` (Студент). У собак и студентов мало общего, но и те и другие знают, как решить, кто из них первый, то есть реализовать методы, которые можно инкапсулировать делегатом `WhichIsFirst`. Следовательно, как объекты класса `Dog`, так и объекты класса `Student` годятся для размещения в классе `Pair`.

В тестовой программе создадим пару объектов класса `Student` и столько же класса `Dog`, а затем сохраним их в объектах класса `Pair`. Создадим объекты делегатов, инкапсулирующие соответствующие методы, удовлетворяющие требованиям на сигнатуру и возвращаемый тип. Наконец, попросим объекты `Pair` отсортировать объекты `Dog` и `Student`. Ниже описывается, как все это реализуется в программе.

Вначале создадим конструктор Pair, который принимает два объекта и помещает их в закрытый массив:

```
public class Pair
{

    // два объекта, сохраняемые в порядке поступления
    public Pair(object firstObject, object secondObject)
    {
        thePair[0] = firstObject;
        thePair [1] = secondObject;
    }
    // массив для двух объектов
    private object[]thePair = new object[2];
```

Затем перегружается метод ToString(), возвращающий строковые значения двух объектов:

```
public override string ToString()
{
    return thePair [0].ToString() + ", " + thePair [1].ToString();
}
```

Итак, в объекте Pair находятся два объекта, и их значения можно вывести. Все готово к сортировке и выводу ее результатов. Поскольку тип объектов заранее неизвестен, ответственность за их упорядочивание в объекте Pair перекладывается на них самих. Поэтому от каждого объекта, хранящегося в Pair, требуется, чтобы он реализовал метод, возвращающий информацию, который из двух объектов первый. Такой метод будет принимать два объекта (произвольного типа) и возвращать перечислимое значение: theFirstComesFirst, если первым является первый параметр, и theSecondComesFirst, если первым должен быть второй.

Эти методы будут инкапсулированы делегатом WhichIsFirst, определенным в рамках класса Pair:

```
public delegate comparison
WhichIsFirst(object obj1, object obj2);
```

Возвращаемое значение имеет перечислимый тип comparison:

```
public enum comparison
{
   theFirstComesFirst = 1,
   theSecondComesFirst = 2
}
```

Любой статический метод, который принимает два объекта и возвращает значение типа comparison, может быть инкапсулирован этим делегатом на этапе выполнения.

Теперь для класса Pair можно определить метод Sort():

```
public void Sort(WhichIsFirst theDelegatedFunc)
{
```

```
    if (theDelegatedFunc(thePair[0],thePair[1]) ==
       comparison.theSecondComesFirst)
    {
       object temp = thePair[0];
       thePair[0] = thePair[1];
       thePair[1] = temp;
    }
}
```

Этот метод имеет один параметр, делегат типа `WhichIsFirst` с именем `theDelegatedFunc`. Метод `Sort()` делегирует ответственность за упорядочивание двух объектов в объекте `Pair` методу, инкапсулированному этим делегатом. В теле метода `Sort()` вызывается делегированный метод и анализируется возвращенное им значение, которое является одним из двух перечисляемых значений типа `comparsion`.

Если возвращено значение `theSecondComesFirst`, объекты внутри `Pair` меняются местами; в противном случае ничего не предпринимается.

Обратите внимание, что `theDelegatedFunc` является именем параметра, представляющего метод, инкапсулируемый делегатом. В этом аргументе можно передать любой метод (с соответствующей сигнатурой и возвращаемым типом). Здесь полная аналогия с ситуацией, в которой некий метод принимает в качестве параметра целое число:

```
int SomeMethod (int myParam){//...}
```

В этой строке программы `myParam` – имя параметра, но методу `SomeMethod` можно передать любое значение типа `int`. В рассматриваемой программе имя параметра – `theDelegatedFunc`, но в качестве аргумента ему можно передать любой метод, удовлетворяющий требованиям делегата `WhichIsFirst` к сигнатуре и возвращаемому типу.

Будем сортировать студентов по их именам. Напишем метод, который возвращает значение `theFirstComesFirst`, если имя первого студента идет по алфавиту раньше имени второго, и значение `theSecondComesFirst`, если имя второго идет первым. Получив в качестве параметров «Amy, Beth», метод возвратит `theFirstComesFirst`, а если ему передать «Beth, Amy», он возвратит `theSecondComesFirst`. Когда метод `Sort()` получит значение `theSecondComesFirst`, он поменяет местами элементы массива, ставя Amy на первое место, а Beth на второе.

Теперь напишем еще один метод, `ReverseSort()`, который меняет порядок следования элементов массива на обратный:

```
public void ReverseSort(WhichIsFirst theDelegatedFunc)
{
    if (theDelegatedFunc(thePair[0], thePair[1]) ==
          comparison.theFirstComesFirst)
    {
        object temp = thePair[0];
        thePair[0] = thePair[1];
```

```
            thePair[1] = temp;
        }
    }
```

Логика этого метода идентична логике метода `Sort()`, но метод `ReverseSort()` меняет местами элементы массива, если делегированный метод сообщает, что первый элемент должен быть первым. Итак, если делегированному методу передать «Amy, Beth», то он возвратит `theFirstComesFirst` (то есть имя Amy идет раньше), а метод *обратной* сортировки поменяет имена местами, поставив Beth в начало. Подобный подход позволяет в методе `Sort()` использовать для прямой и обратной сортировки одну и ту же делегируемую функцию, не заставляя объект поддерживать отдельную функцию, которая возвращала бы наименьшее значение из двух.

Теперь нужны какие-нибудь объекты сортировки. Создадим два очень простых класса, `Student` и `Dog`. В момент создания присвоим объекту `Student` какое-либо имя:

```
public class Student
{
    public Student(string name)
    {
        this.name = name;
    }
```

Классу `Student` требуются два метода, один для перегрузки метода `ToString()`, а другой – для инкапсуляции в качестве делегируемого метода.

Класс `Student` должен перегрузить метод `ToString()` так, чтобы одноименный метод в классе `Pair` (вызывающий `ToString()` для объектов, хранящихся в `Pair`) работал корректно. Реализация метода в классе `Student` всего лишь возвращает имя студента, которое уже является строкой:

```
public override string ToString()
{
     return name;
}
```

Класс `Student` должен также реализовать метод, которому метод `Pair.Sort()` будет делегировать ответственность за определение порядка элементов:

```
return (String.Compare(s1.name, s2.name) < 0 ?
   comparison.theFirstComesFirst :
   comparison.theSecondComesFirst);
```

Здесь `String.Compare` является методом, предоставляемым платформой .NET Framework для класса `String`. Он сравнивает две строки и возвра-

щает отрицательное значение, если первая строка меньше, положительное значение, если она больше, и ноль, если строки одинаковы. Более подробно этот метод обсуждался в главе 10. Обратите внимание на логику приведенного выше фрагмента программы. Значение `theFirstComesFirst` возвращается, только если первая строка меньше; если они равны или первая больше, возвращается `theSecondComesFirst`.

Метод `WhichStudentComesFirst()` принимает в качестве параметров два объекта, а возвращает значение типа `comparison`. Он годится на роль делегируемого метода для `Pair.WhichIsFirst`, поскольку удовлетворяет требованиям, предъявляемым к сигнатуре и возвращаемому типу.

Второй класс называется `Dog`. Собак будем сортировать по весу; более легкую поставим перед более тяжелой. Вот полное определение класса `Dog`:

```
public class Dog
 {
    public Dog(int weight)
    {
        this.weight=weight;
    }

      // собаки упорядочиваются по весу
      public static comparison WhichDogComesFirst(
         Object o1, Object o2)
      {
         Dog d1 = (Dog) o1;
         Dog d2 = (Dog) o2;
        return d1.weight > d2.weight ? theSecondComesFirst :
                                       theFirstComesFirst;
      }
    public override string ToString()
    {
        return weight.ToString();
    }
    private int weight;
}
```

Обратите внимание, что в классе `Dog` тоже перегружается метод `ToString()` и реализуется статический метод с соответствующей сигнатурой для делегирования ему полномочий. Кроме того, отметим, что у делегируемых методов классов `Dog` и `Student` разные имена. Они вовсе не обязаны носить одно и то же имя, поскольку назначаются делегатами динамически, на этапе выполнения.

Делегируемым методам можно давать какие угодно имена. Однако если придерживаться определенной системы (например `WhichDogComesFirst` и `WhichStudentComesFirst`), то код будет удобочитаемым, понятным и простым в сопровождении.

Пример 12.1 представляет собой законченную программу, демонстрирующую вызовы делегированных методов.

Пример 12.1. Работа с делегатами

```
namespace Programming_CSharp
{
   using System;

   public enum comparison
   {
      theFirstComesFirst = 1,
      theSecondComesFirst = 2
   }

// простая коллекция из двух элементов
   public class Pair
   {
      // объявление делегата
      public delegate comparison WhichIsFirst(object obj1, object obj2);

      // конструктор принимает два объекта и сохраняет их в порядке
      // поступления
      public Pair(
         object firstObject,
         object secondObject)
      {
         thePair[0] = firstObject;
         thePair[1] = secondObject;
      }

      // открытый метод, упорядочивающий два объекта
      // по определяемому ими критерию
      public void Sort(
         WhichIsFirst theDelegatedFunc)
      {
         if (theDelegatedFunc(thePair[0],thePair[1])
            == comparison.theSecondComesFirst)
         {
            object temp = thePair[0];
            thePair[0] = thePair[1];
            thePair[1] = temp;
         }
      }

      // открытый метод, расставляющий два объекта по порядку, обратному
      // определяемому ими самими критерию
      public void ReverseSort(
         WhichIsFirst theDelegatedFunc)
      {
         if (theDelegatedFunc(thePair[0],thePair[1]) ==
            comparison.theFirstComesFirst)
         {
            object temp = thePair[0];
```

```
            thePair[0] = thePair[1];
            thePair[1] = temp;
         }
      }

      // запросить у двух объектов их строковые значения
      public override string ToString(  )
      {
         return thePair[0].ToString(  ) + ", "
            + thePair[1].ToString(  );
      }

      // закрытый массив для хранения двух объектов
      private object[] thePair = new object[2];
   }

   public class Dog
   {
      public Dog(int weight)
      {
         this.weight=weight;
      }

      // собаки упорядочиваются по весу
      public static comparison WhichDogComesFirst(
         Object o1, Object o2)
      {
         Dog d1 = (Dog) o1;
         Dog d2 = (Dog) o2;
         return d1.weight > d2.weight ?
            comparison.theSecondComesFirst :
            comparison.theFirstComesFirst;
      }
      public override string ToString(  )
      {
         return weight.ToString(  );
      }
      private int weight;
   }

   public class Student
   {
      public Student(string name)
      {
         this.name = name;
      }

      // студенты упорядочиваются по алфавиту
      public static comparison
         WhichStudentComesFirst(Object o1, Object o2)
      {
         Student s1 = (Student) o1;
         Student s2 = (Student) o2;
         return (String.Compare(s1.name, s2.name) < 0 ?
```

```
            comparison.theFirstComesFirst :
            comparison.theSecondComesFirst);
      }
      public override string ToString(  )
      {
         return name;
      }
      private string name;
   }

   public class Test
   {
      public static void Main(  )
      {
         // создать по паре объектов студентов и собак
         // и разместить их в объектах Pair
         Student Jesse = new Student("Jesse");
         Student Stacey = new Student ("Stacey");
         Dog Milo = new Dog(65);
         Dog Fred = new Dog(12);

         Pair studentPair = new Pair(Jesse,Stacey);
         Pair dogPair = new Pair(Milo, Fred);
         Console.WriteLine("studentPair\t\t\t: {0}",
            studentPair.ToString(  ));
         Console.WriteLine("dogPair\t\t\t\t: {0}",
            dogPair.ToString(  ));

         // создать объекты делегатов
         Pair.WhichIsFirst  theStudentDelegate =
            new Pair.WhichIsFirst(
            Student.WhichStudentComesFirst);

         Pair.WhichIsFirst theDogDelegate =
            new Pair.WhichIsFirst(
            Dog.WhichDogComesFirst);

         // сортировка с использованием делегатов
         studentPair.Sort(theStudentDelegate);
         Console.WriteLine("После сортировки studentPair\t\t: {0}",
            studentPair.ToString(  ));
         studentPair.ReverseSort(theStudentDelegate);
         Console.WriteLine("После обратной сортировки studentPair\t: {0}",
            studentPair.ToString(  ));

         dogPair.Sort(theDogDelegate);
         Console.WriteLine("После сортировки dogPair\t\t: {0}",
            dogPair.ToString(  ));
         dogPair.ReverseSort(theDogDelegate);
         Console.WriteLine("После обратной сортировки dogPair\t: {0}",
            dogPair.ToString(  ));
      }
   }
```

```
}
```

Вывод:

```
studentPair                              : Jesse, Stacey
dogPair                                  : 65, 12
После сортировки studentPair             : Jesse, Stacey
После обратной сортировки studentPair    : Stacey, Jesse
После сортировки dogPair                 : 12, 65
После обратной сортировки dogPair        : 65, 12
```

Программа `Test` создает два объекта класса `Student` и два объекта класса `Dog`, которые помещаются в объекты `Pair`. Конструктор `Student()` принимает строку с именем студента, а конструктор `Dog()` – вес собаки, выраженный целым числом.

```
Student Jesse = new Student("Jesse");
Student Stacey = new Student ("Stacey");
Dog Milo = new Dog(65);
Dog Fred = new Dog(12);

Pair studentPair = new Pair(Jesse,Stacey);
Pair dogPair = new Pair(Milo, Fred);
Console.WriteLine("studentPair\t\t\t: {0}", studentPair.ToString());
Console.WriteLine("dogPair\t\t\t\t: {0}", dogPair.ToString());
```

Затем выводится содержимое двух контейнеров `Pair`, демонстрирующее, в каком порядке расположены поступившие объекты:

```
studentPair           : Jesse, Stacey
dogPair               : 65, 12
```

Как и следовало ожидать, они хранятся в порядке поступления. Далее создаются объекты двух делегатов:

```
Pair.WhichIsFirst  theStudentDelegate =
   new Pair.WhichIsFirst(Student.WhichStudentComesFirst);

Pair.WhichIsFirst theDogDelegate =
   new Pair.WhichIsFirst(Dog.WhichDogComesFirst);
```

Первый делегат, `theStudentDelegate`, создается передачей соответствующего статического метода класса `Student`. Второй, `theDogDelegate`, передачей статического метода класса `Dog`.

Теперь делегаты являются объектами, которые могут быть переданы методам. Вначале они передаются методу `Sort()` объекта `Pair`, а затем – методу `ReverseSort()`. Результаты выводятся на экран:

```
После сортировки studentPair          : Jesse, Stacey
После обратной сортировки studentPair   : Stacey, Jesse
После сортировки dogPair              : 12, 65
После обратной сортировки dogPair       : 65, 12
```

Статические делегаты

У программы из примера 12.1 есть один недостаток. Вызывающему классу, в данном случае классу Test, приходится создавать экземпляры делегатов, нужных ему для упорядочивания объектов в объекте Pair. Было бы гораздо удобнее получать делегат прямо от класса Student или Dog. Этого нетрудно добиться, если включить в каждый класс собственный статический делегат. Внесем изменения в класс Student, добавив в него делегат:

```
public static readonly Pair.WhichIsFirst OrderStudents =
   new Pair.WhichIsFirst(Student.WhichStudentComesFirst);
```

Будет создан статический делегат с именем OrderStudents, доступный только для чтения.

Модификатор readonly в определении OrderStudents гарантирует, что после создания этого статического поля оно не будет изменено.

Аналогичным образом создается делегат в классе Dog:

```
public static readonly Pair.WhichIsFirst OrderDogs =
    new Pair.WhichIsFirst(Dog. WhichDogComesFirst);
```

Теперь делегаты – это статические поля соответствующих классов. Каждое из них жестко связано с соответствующим методом класса. Эти делегаты можно вызывать без создания локальных экземпляров. Достаточно будет передать методам класса Pair статический делегат соответствующего класса:

```
studentPair.Sort(Student.OrderStudents);
Console.WriteLine("После сортировки studentPair\t\t: {0}",
studentPair.ToString());
 studentPair.ReverseSort(Student.OrderStudents);
 Console.WriteLine("После обратной сортировки studentPair\t: {0}",
    studentPair.ToString());

 dogPair.Sort(Dog.OrderDogs);
 Console.WriteLine("После сортировки dogPair\t\t: {0}", dogPair.ToString());
 dogPair.ReverseSort(Dog.OrderDogs);
 Console.WriteLine("После обратной сортировки dogPair\t: {0}",
dogPair.ToString());
```

После внесения этих изменений в программу результат ее работы не изменится.

Делегаты как свойства

Статические делегаты имеют один недостаток: их экземпляры должны быть созданы независимо от того, будут ли они использованы, как

это было с классами Student и Dog в предыдущем примере. Положение можно исправить, если заменить статические поля делегатов на свойства.

Уберем из класса Student определение:

```
public static readonly Pair.WhichIsFirst OrderStudents =
   new Pair.WhichIsFirst(Student.WhichStudentComesFirst);
```

и поставим вместо него:

```
public static Pair.WhichIsFirst OrderStudents
{
    get
    {
        return new Pair.WhichIsFirst(WhichStudentComesFirst);
    }
}
```

Аналогично поступим со статическим полем класса Dog:

```
public static Pair.WhichIsFirst OrderDogs
{
    get
    {
        return new Pair.WhichIsFirst(WhichDogComesFirst);
    }
}
```

Передача делегатов в качестве параметров остается прежней:

```
studentPair.Sort(Student.OrderStudents);
dogPair.Sort(Dog.OrderDogs);
```

При обращении к свойству OrderStudent создается делегат:

```
return new Pair.WhichIsFirst(WhichStudentComesFirst);
```

Основным достоинством такого подхода является тот факт, что делегат не создается, пока не будет запрошен. Это позволяет классу Test самому решать, когда ему нужен делегат, причем ответственность за создание делегата по-прежнему лежит на классе Student (или Dog).

Определение порядка вызова методов с помощью массива делегатов

Делегаты помогают программисту создавать системы, в которых пользователь динамически определяет порядок выполнения действий. Предположим, имеется система обработки изображений, в которой изображение может быть подвергнуто ряду операций, таких как изме-

нение резкости, поворот, фильтрация и т. д. Предположим также, что порядок применения этих операций к изображению достаточно важен. Пользователь хочет выбрать эффекты из меню и сообщить программе, в каком порядке нужно их применить к изображению.

Чтобы удовлетворить сформулированным требованиям, можно для каждой операции создать делегат и разместить их в коллекции, например в массиве, в порядке, указанном пользователем. Когда все делегаты созданы и сохранены в коллекции, достаточно перебрать его элементы в цикле, поочередно вызывая каждый делегированный метод.

Начнем с создания класса `Image`, представляющего изображение, обрабатываемое классом `ImageProcessor`:

```
public class Image
{
    public Image()
    {
        Console.WriteLine("Изображение создано");
    }
}
```

Будем считать, что объект `Image` представляет файл в формате *.gif* или *.jpeg* (или в каком-нибудь другом).

В классе `ImageProcessor` объявим делегат. Программист может выбирать любые параметры и любой тип возвращаемого значения. Пусть в данном примере делегат инкапсулирует метод, который не принимает аргументов и возвращает `void`:

```
public delegate void DoEffect();
```

Теперь нужно объявить ряд методов, каждый из которых обрабатывает объект `Image` и соответствует сигнатуре и возвращаемому типу делегата:

```
public static void Blur()
 {
     Console.WriteLine("Растушевка изображения");
 }

 public static void Filter()
 {
     Console.WriteLine("Наложение фильтра");
 }
 public static void Sharpen()
 {
     Console.WriteLine("Настройка резкости");
 }

 public static void Rotate()
 {
     Console.WriteLine("Вращение изображения");
 }
```

В реальной ситуации эти методы должны выполнять фактические действия по растушевке, наложению фильтра, настройке резкости и повороту изображения соответственно. Конечно, они будут очень сложны.

Классу `ImageProcessor` нужен массив для хранения делегатов, выбранных пользователем, а также переменная, отслеживающая размер массива. Естественно, нужна и переменная для хранения изображения:

```
DoEffect[] arrayOfEffects;
Image image;
int numEffectsRegistered = 0;
```

Далее, классу `ImageProcessor` необходим метод для записи делегатов в массив:

```
public void AddToEffects(DoEffect theEffect)
{
    if (numEffectsRegistered >= 10)
    {
        throw new Exception("Слишком много элементов в массиве");
    }
    arrayOfEffects[numEffectsRegistered++] = theEffect;

}
```

Еще ему нужен метод, вызывающий делегированные методы по очереди:

```
public void ProcessImages()
{
    for (int i = 0;i < numEffectsRegistered;i++)
    {
        arrayOfEffects[i]();
    }
}
```

Наконец, остается объявить статические делегаты, которые будет вызывать пользователь, и привязать к ним методы обработки изображения:

```
public DoEffect BlurEffect = new DoEffect(Blur);
public DoEffect SharpenEffect = new DoEffect(Sharpen);
public DoEffect FilterEffect = new DoEffect(Filter);
public DoEffect RotateEffect = new DoEffect(Rotate);
```

В реальном приложении, где таких эффектов будут десятки, разумнее реализовать их в виде свойств, а не статических методов. Тогда можно будет сэкономить на том, что свойства создаются лишь по мере необходимости. Правда, эта экономия достигается за счет некоторого усложнения программы.

Программа пользователя, как правило, включает в себя компонент, реализующий пользовательский интерфейс. В примере 12.2 имитируется выбор эффектов, после чего программа добавляет их в массив и вызывает метод `ProcessImage()`.

Пример 12.2. Массив делегатов

```
namespace Programming_CSharp
{
   using System;

   // изображение, подвергающееся обработке
   public class Image
   {
      public Image()
      {
         Console.WriteLine("Изображение создано");
      }
   }

   public class ImageProcessor
   {
      // объявить делегат
      public delegate void DoEffect();

      // создать различные статические делегаты,
      // связанные с методами класса
      public DoEffect BlurEffect =
         new DoEffect(Blur);
      public DoEffect SharpenEffect =
         new DoEffect(Sharpen);
      public DoEffect FilterEffect =
         new DoEffect(Filter);
      public DoEffect RotateEffect =
         new DoEffect(Rotate);

      // конструктор инициализирует изображение и массив
      public ImageProcessor(Image image)
      {
         this.image = image;
         arrayOfEffects = new DoEffect[10];
      }

      // в реальном приложении был бы использован
      // более гибкий класс коллекции
      public void AddToEffects(DoEffect theEffect)
      {
         if (numEffectsRegistered >= 10)
         {
            throw new Exception(
               "Слишком много элементов в массиве");
         }
         arrayOfEffects[numEffectsRegistered++]
            = theEffect;
```

```
    }

    // методы обработки изображения...
    public static void Blur()
    {
        Console.WriteLine("Растушевка изображения");
    }

    public static void Filter()
    {
        Console.WriteLine("Наложение фильтра");
    }

    public static void Sharpen()
    {
        Console.WriteLine("Регулировка контрастности");
    }

    public static void Rotate()
    {
        Console.WriteLine("Вращение изображения");
    }

    public void ProcessImages()
    {
        for (int i = 0;i < numEffectsRegistered;i++)
        {
            arrayOfEffects[i]();
        }
    }

    // закрытые переменные класса...
    private DoEffect[] arrayOfEffects;

    private Image image;
    private int numEffectsRegistered = 0;

}

// тестовый класс
public class Test
{
    public static void Main()
    {
        Image theImage = new Image();

        // ради простоты пользовательский интерфейс отсутствует,
        // программа просто добавляет методы в массив и
        // вызывает класс-обработчик, вызывающий
        // методы в порядке добавления
        ImageProcessor theProc =
            new ImageProcessor(theImage);
        theProc.AddToEffects(theProc.BlurEffect);
        theProc.AddToEffects(theProc.FilterEffect);
        theProc.AddToEffects(theProc.RotateEffect);
```

```
            theProc.AddToEffects(theProc.SharpenEffect);
            theProc.ProcessImages();
        }
    }
}
```

Вывод:

```
Изображение создано
Растушевка изображения
Наложение фильтра
Вращение изображения
Регулировка контрастности
```

В классе `Test` из примера 12.2 создается экземпляр класса `ImageProcessor`, и в массив добавляются различные эффекты. Если пользователь предпочитает сначала произвести растушевку изображения, а потом применить фильтр, то именно в таком порядке делегаты добавляются в массив. Аналогичным образом, если пользователь захочет повторить какую-нибудь операцию, в коллекцию можно еще раз добавить соответствующий делегат.

В реальном приложении операции могут быть представлены в виде списка, позволяющего пользователю менять порядок следования элементов. Если он переместит какие-либо эффекты в списке, программе будет достаточно переместить элементы коллекции. Более того, список операций можно хранить в базе данных, загружая их динамически и создавая экземпляры делегатов в соответствии с записями в базе данных.

Делегаты обеспечивают гибкость реализации, позволяя динамически определять, какие методы следует вызвать, в каком порядке и как часто.

Множественное делегирование

Иногда бывает желательно выполнять *множественное делегирование* (*multicast*), то есть вызов двух методов из одного делегата. Это становится особенно актуально при обработке событий (обсуждаемых далее в этой главе).

Поставленная цель – иметь делегат, вызывающий несколько методов, – отличается от ситуации, в которой присутствует коллекция делегатов, каждый из которых вызывает один метод. В предыдущем примере коллекция использовалась для упорядочивания различных делегатов. Допускалось многократное добавление одного и того же делегата в коллекцию, а также переупорядочивание делегатов в коллекции с целью изменения порядка их вызовов.

При множественном делегировании создается один делегат, вызывающий несколько инкапсулированных методов. Например, когда пользователь щелкает по кнопке, может потребоваться, чтобы приложение

предприняло несколько действий. Конечно, можно реализовать это требование с помощью коллекции делегатов, однако проще и элегантнее организовать множественное делегирование.

Множественное делегирование можно использовать с делегатами, возвращающими значение (имеется в виду значение, отличное от `void`). В этом случае вы получите только одно значение: значение последнего вызванного делегата.

Два делегата можно объединить с помощью операции сложения (+). Результатом будет новый множественный делегат, вызывающий методы обоих объединенных делегатов. Например, пусть `Writer` и `Logger` являются делегатами. В следующей строчке кода создается новый множественный делегат по имени `myMulticastDelegate`:

```
myMulticastDelegate = Writer + Logger;
```

К множественному делегату можно добавить делегат с помощью оператора +=. Делегат, стоящий справа от знака операции, добавляется к множественному делегату, стоящему слева. Например, если `Transmitter` и `myMulticastDelegate` являются делегатами, то следующая строчка кода соединит их в один:

```
myMulticastDelegate += Transmitter;
```

Чтобы понять, как создаются и используются множественные делегаты, разберем пример. В примере 12.3 создается класс `MyClassWithDelegate`, который объявляет делегат, принимающий строку и возвращающий `void`:

```
public delegate void StringDelegate(string s);
```

Затем определяется класс `MyImplementingClass` с тремя методами, каждый из которых принимает строку и возвращает `void`: `WriteString`, `LogString` и `TransmitString`. Первый выводит строку в стандартный поток вывода, второй имитирует запись в регистрационный журнал, а третий имитирует передачу строки в Интернете. Для вызова соответствующих методов создаются объекты делегатов:

```
Writer("Строка передана методу Writer\n");
Logger("Строка передана методу Logger\n");
Transmitter("Строка передана методу Transmitter\n");
```

Чтобы проследить за объединением делегатов, создадим еще один объект делегата:

```
MyClassWithDelegate.StringDelegate myMulticastDelegate;
```

и присвоим ему результат «сложения» двух имеющихся:

```
myMulticastDelegate = Writer + Logger;
```

С помощью операции += добавим к нему еще один делегат:

```
myMulticastDelegate += Transmitter;
```

Наконец, из делегата удаляется один, для чего используется операция -=:

```
myMulticastDelegate -= Logger;
```

Пример 12.3. Объединение делегатов

```
namespace Programming_CSharp
{
   using System;

   public class MyClassWithDelegate
   {
      // объявление делегата
      public delegate void StringDelegate(string s);

   }

   public class MyImplementingClass
   {
      public static void WriteString(string s)
      {
         Console.WriteLine("Вывод строки {0}", s);
      }
      public static void LogString(string s)
      {
         Console.WriteLine("Запись строки в журнал {0}", s);
      }

      public static void TransmitString(string s)
      {
         Console.WriteLine("Передача строки {0}", s);
      }

   }


   public class Test
   {
      public static void Main()
      {
         // определить три делегата StringDelegate
         MyClassWithDelegate.StringDelegate
            Writer, Logger, Transmitter;

         // определить еще один объект StringDelegate,
         // который будет множественным делегатом
         MyClassWithDelegate.StringDelegate
            myMulticastDelegate;

         // создать экземпляры первых трех делегатов,
         // передавая методы для инкапсуляции
```

```
        Writer = new MyClassWithDelegate.StringDelegate(
            MyImplementingClass.WriteString);
        Logger = new MyClassWithDelegate.StringDelegate(
            MyImplementingClass.LogString);
        Transmitter =
            new MyClassWithDelegate.StringDelegate(
            MyImplementingClass.TransmitString);

        // вызвать делегированный метод Writer
        Writer("Строка передана методу Writer\n");

        // вызвать делегированный метод Logger
        Logger("Строка передана методу Logger\n");

        // вызвать делегированный метод Transmitter
        Transmitter("Строка передана методу Transmitter\n");

        // сообщить о готовности объединить
        // два делегата в один множественный
        Console.WriteLine(
            "myMulticastDelegate = Writer + Logger");

        // объединить два делегата и
        // присвоить результат делегату myMulticastDelegate
        myMulticastDelegate = Writer + Logger;

        // вызвать делегированные методы -
        // будут вызваны два метода
        myMulticastDelegate(
            "Первая строка передана методу Collector");

        // сообщить о готовности добавить
        // третий делегат к множественному делегату
        Console.WriteLine(
            "\nmyMulticastDelegate += Transmitter");

        // добавить третий делегат
        myMulticastDelegate += Transmitter;

        // вызвать три делегированных метода
        myMulticastDelegate(
            "Вторая строка передана методу Collector");

        // сообщить о готовности удалить
        // один делегат
        Console.WriteLine(
            "\nmyMulticastDelegate -= Logger");

        // удалить один делегат
        myMulticastDelegate -= Logger;

        // вызвать два оставшихся
        // делегированных метода
        myMulticastDelegate(
            "Третья строка передана методу Collector");
    }
```

```
    }
}
```

Вывод:

```
Вывод строки Строка передана методу Writer

Запись строки в журнал Строка передана методу Logger

Передача строки Строка передана методу Transmitter

myMulticastDelegate = Writer + Logger
вывод строки Первая строка передана методу Collector
Запись строки в журнал Первая строка передана методу Collector

myMulticastDelegate += Transmitter
Вывод строки Вторая строка передана методу Collector
Запись строки в журнал Вторая строка передана методу Collector
Передача строки Вторая строка передана методу Collector

myMulticastDelegate -= Logger
Вывод строки Третья строка передана методу Collector
Передача строки Третья строка передана методу Collector
```

В тестирующей части программы из примера 12.3 определяются объекты делегатов и вызываются первые три из них (`Writer`, `Logger` и `Transmitter`). Четвертому делегату, `myMulticastDelegate`, присваивается объединение первых двух, после чего он тоже вызывается. Это приводит к вызову обоих делегированных методов. Третий делегат добавляется к делегату `myMulticastDelegate`, и, когда тот вызывается, срабатывают все три метода. Наконец, делегат `Logger` удаляется из множественного делегата. Теперь вызов делегата `myMulticastDelegate` заканчивается вызовом двух оставшихся методов.

Преимущества множественного делегирования легче оценить, когда речь заходит о событиях, рассматриваемых в следующем разделе. Когда возникает некоторое событие, например нажатие кнопки, связанный с ним делегат может вызвать целый ряд обработчиков для реакции на данное событие.

События

Графические пользовательские интерфейсы (GUI), система Windows и веб-броузеры (например, выпущенные компанией Microsoft) требуют, чтобы программы реагировали на *события* (*event*). Событием может быть нажатие кнопки, выбор команды меню, завершение операции передачи файла и т. д. Короче говоря, если происходит что-то важное, программа должна отреагировать. Порядок возникновения событий предсказать невозможно. Система ждет возникновения события, а затем предпринимает действия по его обработке.

В среде GUI любые элементы управления могут *вызвать* событие. Например, если щелкнуть по кнопке, вызывается событие `Click`, а ес-

ли добавить элемент в раскрывающийся список, может быть вызвано событие `ListChanged`.

Реакция на событие является предметом заботы других классов, и она совсем не интересует класс, вызвавший событие. Кнопка как бы говорит: «По мне щелкнули», а реагировать должны другие классы.

Публикация и подписка

В языке C# любой объект может *опубликовать* набор событий, а другие классы могут на них *подписаться*. Когда публикующий класс вызывает событие, все подписавшиеся классы уведомляются об этом.

Подобное проектное решение является реализацией паттерна «издатель/подписчик», описанного в основополагающем труде «Паттерны проектирования» Гаммы и др. («Design Patterns», by Gamma, et al.), вышедшем в издательстве Addison Wesley в 1995 году.[1] Гамма формулирует предназначение данного образца следующим образом: «Определить между объектами зависимость «один-ко-многим», чтобы при смене состояния одного объекта все зависимые объекты уведомлялись об этом и автоматически меняли свои состояния».

Когда имеется такой механизм, объект говорит: «Вот события, о которых я могу уведомить». Другие классы подписываются, говоря: «Да, уведомите меня, когда оно произойдет». Например, кнопка уведомляет заинтересованных наблюдателей, когда пользователь щелкнет по ней. Кнопка называется *издателем*, потому что публикует событие `Click`, а другие классы называются *подписчиками*, потому что подписываются на это событие.

События и делегаты

События в C# обрабатываются с помощью делегатов. Класс-издатель определяет делегат, который должен быть реализован классами-подписчиками. Когда возникает событие, методы подписчиков вызываются через этот делегат.

Метод, реагирующий на события, называется *обработчиком события* (*event handler*). Программист объявляет обработчик события как любой другой делегат.

Существует соглашение, по которому обработчики событий в .NET Framework возвращают `void` и принимают два параметра. Первый –

[1] Гамма, Хелм, Джонсон, Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования». – Пер. с англ. – СПб:Питер, 2000 г.

это *источник* события, то есть объект-издатель. Вторым параметром передается объект, производный от класса EventArgs. Рекомендуется, чтобы обработчики событий, создаваемые всеми программистами, следовали этому соглашению.

EventArgs является классом, базовым для всех классов событий. В отличие от своего конструктора, этот класс наследует все методы от класса Object, хотя он и добавляет открытое статическое поле empty, представляющее событие без состояния (чтобы обеспечить эффективную обработку таких событий). Классы, производные от EventArgs, содержат информацию о возникшем событии.

Событие является свойством класса, опубликовавшего его. Ключевое слово event управляет доступом классов-подписчиков к событию. Оно предназначено для обеспечения идиомы «публикация/подписка».

Предположим, создается класс Clock, который с помощью события уведомляет классы-подписчики о том, что местное время увеличилось на одну секунду. Назовем это событие OnSecondChange. Событие и тип делегата для обработки события определяются следующим образом:

```
[атрибуты] [модификаторы] event тип
       имя
```

Например:

```
public event SecondChangeHandler OnSecondChange;
```

Здесь не указаны атрибуты (которые обсуждаются в главе 18). В качестве модификатора указывается ключевое слово abstract, new, override, static, virtual или один из четырех модификаторов права доступа, в данном случае public.

За модификатором стоит ключевое слово event.

Тип – это делегат, который должен быть связан с событием, в данном случае SecondChangeHandler.

Имя – имя события, в рассматриваемом примере это OnSecondChange. Принято начинать имена событий с префикса On.

Итак, приведенное объявление утверждает, что OnSecondChange является событием, реализованным с помощью делегата SecondChangeHandler.

Объявление этого делегата выглядит так:

```
public delegate void SecondChangeHandler(
    object clock,
    TimeInfoEventArgs timeInformation
    );
```

Здесь объявляется делегат. Как говорилось выше, в соответствии с соглашением обработчик события должен возвращать void и принимать два параметра: источник события (объект Clock) и объект класса, про-

изводного от EventArgs (в данном примере TimeInfoEventArgs). Класс TimeInfoEventArgs определяется следующим образом:

```
public class TimeInfoEventArgs : EventArgs
{
     public TimeInfoEventArgs(int hour, int minute, int second)
     {
         this.hour = hour;
         this.minute = minute;
         this.second = second;
     }

     public readonly int hour;
     public readonly int minute;
     public readonly int second;
}
```

Объект TimeInfoEventArgs будет содержать информацию о текущем времени (часы, минуты и секунды). Он определяет конструктор и три открытые переменные, доступные только для чтения.

Кроме делегата и события класс Clock содержит три переменные — hour, minute и second — и один метод Run():

```
public void Run()
{
    for(;;)
    {
        // ждать 10 миллисекунд
        Thread.Sleep(10);

        // получить текущее время
        System.DateTime dt = System.DateTime.Now;

        // если количество секунд изменилось,
        // уведомить подписчиков
        if (dt.Second != second)
        {
            // создать объект TimeInfoEventArgs
            // для передачи подписчику
            TimeInfoEventArgs timeInformation =
                new TimeInfoEventArgs(dt.Hour,dt.Minute,dt.Second);

            // если есть подписчики, уведомить их
            if (OnSecondChange != null)
            {
                OnSecondChange(this,timeInformation);
            }
        }

        // обновить состояние
        this.second = dt.Second;
        this.minute = dt.Minute;
```

```
            this.hour = dt.Hour;
        }
    }
```

В методе `Run()` работает бесконечный цикл `for`, в котором периодически проверяется системное время. Если оно отличается от значения, хранящегося в объекте `Clock`, он уведомляет подписчиков и обновляет свое состояние.

Первое, что делает метод, – переходит в режим ожидания на 10 миллисекунд:

```
Thread.Sleep(10);
```

Здесь вызывается статический метод класса `Thread`, принадлежащего пространству имен `System.Threading`. Более подробно этот класс обсуждается в главе 20. Вызов метода `Sleep()` не дает циклу `for` занимать практически все время процессора.

Прождав 10 миллисекунд, метод проверяет текущее время:

```
System.DateTime dt = System.DateTime.Now;
```

Приблизительно один раз за сто проверок увеличивается значение переменной `second`. Когда метод обнаруживает это, он оповещает своих подписчиков. С этой целью создается новый объект `TimeInfoEventArgs`:

```
if (dt.Second != second)
{
   // создать объект TimeInfoEventArgs
   // для передачи подписчику
   TimeInfoEventArgs timeInformation =
      new TimeInfoEventArgs(dt.Hour,dt.Minute,dt.Second);
```

Затем метод `Run()` уведомляет подписчиков, вызывая событие `OnSecondChange`:

```
   // если есть подписчики, уведомить их
   if (OnSecondChange != null)
   {
      OnSecondChange(this,timeInformation);
   }
}
```

Если у события нет зарегистрированных подписчиков, оно имеет значение `null`. Условие в операторе `if` сравнивает событие со значением `null`. Следует убедиться в наличии подписчиков прежде, чем вызывать событие `OnSecondChange`.

Читатель помнит, что обработчик события `OnSecondChange` принимает два аргумента – источник события и объект класса, производного от `EventArgs`. Во фрагменте программы, приведенном выше, видно, что

первым аргументом является ссылка this, поскольку сам объект Clock и является источником события. Второй аргумент – это объект TimeInfoEventArgs, созданный строчкой раньше.

Вызов события приведет к вызову методов, зарегистрированных классом Clock с помощью делегата. Этот вопрос будет обсужден чуть позже.

После вызова события класс Clock обновляет свое состояние:

```
this.second = dt.Second;
this.minute = dt.Minute;
this.hour = dt.Hour;
```

Обратите внимание, что этот код не обеспечивает безопасность при работе с несколькими потоками. Безопасность и синхронизация обсуждаются в главе 20.

Итак, остается лишь создать классы, подписывающиеся на это событие. Создадим два. Первый назовем DisplayClock. Его задача сводится к показу текущего времени на экране компьютера.

В рассматриваемом примере этот класс будет максимально упрощен и ограничится двумя методами. Первый, вспомогательный, называется Subscribe. Он будет подписывать класс на событие OnSecondChange класса Clock. Вторым методом будет обработчик события по имени TimeHasChanged:

```
public class DisplayClock
{
    public void Subscribe(Clock theClock)
    {
        theClock.OnSecondChange +=
            new Clock.SecondChangeHandler(TimeHasChanged);
    }

    public void TimeHasChanged(
        object theClock, TimeInfoEventArgs ti)
    {
            Console.WriteLine("Текущее время: {0}:{1}:{2}",
                ti.hour.ToString(),
                ti.minute.ToString(),
                ti.second.ToString());
   }
}
```

Когда вызывается метод Subscribe(), он создает новый делегат SecondChangeHandler, передавая ему обработчик события TimeHasChanged(). Затем он подписывает этот делегат на событие OnSecondChange класса Clock.

Второй класс, реагирующий на событие, будет называться LogCurrentTime. В реальном приложении он регистрировал бы событие в фай-

ле журнала, но здесь, в демонстрационном примере, он будет выводить сообщение на экран:

```
public class LogCurrentTime
{
   public void Subscribe(Clock theClock)
    {
        theClock.OnSecondChange += new
Clock.SecondChangeHandler(WriteLogEntry);
    }

    // этот метод должен выводить информацию в файл, но здесь
    // он выводит ее на экран в демонстрационных целях;
    // этот объект не поддерживает никакое состояние
    public void WriteLogEntry(
        object theClock, TimeInfoEventArgs ti)
    {
        Console.WriteLine("Записано в журнал: {0}:{1}:{2}",
            ti.hour.ToString(),
            ti.minute.ToString(),
            ti.second.ToString());
    }
}
```

Хотя в данном примере классы-подписчики имеют много общего, в реальном приложении на событие может подписаться любое количество совершенно непохожих классов.

Обратим внимание, что события добавляются к объекту операцией `+=`. Такой подход позволяет впоследствии добавить новые события к событию `OnSecondChange` объекта `Clock`, не разрушая при этом уже существующие. Когда класс `LogCurrentTime` подписывается на событие `OnSecondChange`, нельзя допустить, чтобы оно «забыло», что на него уже подписан класс `DisplayClock`.

Итак, нужно создать класс `Clock`, а также класс `DisplayClock` и заставить их подписаться на событие. Затем создадим класс `LogCurrentTime` и тоже подпишем его на событие. Наконец, выполним класс `Clock`. Все это демонстрируется в примере 12.4.

Пример 12.4. Работа с событиями

```
namespace Programming_CSharp
{
   using System;
   using System.Threading;

   // класс для хранения информации о событии;
   // в данном случае он содержит только сведения,
   // полученные от класса Clock, но мог бы
   // хранить и состояние
   public class TimeInfoEventArgs : EventArgs
   {
```

```
   public TimeInfoEventArgs(int hour, int minute, int second)
   {
      this.hour = hour;
      this.minute = minute;
      this.second = second;
   }
   public readonly int hour;
   public readonly int minute;
   public readonly int second;
}

// главное действующее лицо - класс, за которым наблюдают
// другие классы; он публикует одно событие, OnSecondChange;
// наблюдатели подписываются на это событие
public class Clock
{
   // делегат, который должен быть реализован подписчиками
   public delegate void SecondChangeHandler
      (
      object clock,
      TimeInfoEventArgs timeInformation
      );

   // публикуемое событие
   public event SecondChangeHandler OnSecondChange;

   // запустить класс Clock;
   // он будет вызывать событие каждую секунду
   public void Run()
   {
      for(;;)
      {
         // ждать 10 миллисекунд
         Thread.Sleep(10);

         // получить текущее время
         System.DateTime dt = System.DateTime.Now;

         // если количество секунд изменилось,
         // уведомить подписчиков
         if (dt.Second != second)
         {
            // создать объект TimeInfoEventArgs
            // для передачи подписчику
            TimeInfoEventArgs timeInformation =
               new TimeInfoEventArgs(
                  dt.Hour,dt.Minute,dt.Second);

            // если есть подписчики, уведомить их
            if (OnSecondChange != null)
            {
               OnSecondChange(
                  this,timeInformation);
            }
```

```
            }
            // обновить состояние
            this.second = dt.Second;
            this.minute = dt.Minute;
            this.hour = dt.Hour;

         }
      }
      private int hour;
      private int minute;
      private int second;
   }

   // наблюдатель; класс DisplayClock подписывается на
   // событие класса Clock; его задача -
   // выводить на консоль текущее время
   public class DisplayClock
   {
      // подписаться на событие SecondChangeHandler
      // класса Clock
      public void Subscribe(Clock theClock)
      {
         theClock.OnSecondChange +=
            new Clock.SecondChangeHandler(TimeHasChanged);
      }

      // метод, реализующий делегируемую функциональность
      public void TimeHasChanged(
         object theClock, TimeInfoEventArgs ti)
      {
         Console.WriteLine("Текущее время: {0}:{1}:{2}",
            ti.hour.ToString(),
            ti.minute.ToString(),
            ti.second.ToString());
      }
   }

   // второй подписчик, который должен выводить информацию в файл
   public class LogCurrentTime
   {
      public void Subscribe(Clock theClock)
      {
         theClock.OnSecondChange +=
            new Clock.SecondChangeHandler(WriteLogEntry);
      }

      // этот метод должен выводить информацию в файл, но здесь
      // он выводит ее на экран в демонстрационных целях;
      // этот объект не поддерживает никакое состояние
      public void WriteLogEntry(
         object theClock, TimeInfoEventArgs ti)
      {
```

```
            Console.WriteLine("Записано в журнал: {0}:{1}:{2}",
               ti.hour.ToString(),
               ti.minute.ToString(),
               ti.second.ToString());
         }
      }

      public class Test
      {
         public static void Main()
         {
            // создать новый объект Clock
            Clock theClock = new Clock();

            // создать объект DisplayClock и заставить его
            // подписаться на событие только что созданного класса
            DisplayClock dc = new DisplayClock();
            dc.Subscribe(theClock);

            // создать объект LogCurrentTime и заставить его
            // подписаться на событие класса Clock
            LogCurrentTime lct = new LogCurrentTime();
            lct.Subscribe(theClock);

            // запустить класс Clock
            theClock.Run();
         }
      }
   }
```

Вывод:

```
Текущее время: 14:53:56
Записано в журнал: 14:53:56
Текущее время: 14:53:57
Записано в журнал: 14:53:57
Текущее время: 14:53:58
Записано в журнал: 14:53:58
Текущее время: 14:53:59
Записано в журнал: 14:53:59
Текущее время: 14:54:0
Записано в журнал: 14:54:0
```

«Чистым» результатом этой программы является создание двух классов, `DisplayClock` и `LogCurrentTime`, каждый из которых подписан на событие третьего класса, а именно `Clock.OnSecondChange`.

Отделение издателей от подписчиков

Класс `Clock` и сам мог бы выводить время, не вызывая никакое событие, так к чему все эти хлопоты с делегатами? Достоинство идиомы «публикация/подписка» состоит в возможности уведомить любое количество классов о возникшем событии. Классам-подписчикам нет дела

до того, как работает класс `Clock`, а ему – до того, как они отреагируют на событие. Аналогичным образом кнопка может опубликовать событие `Onclick`, и любое количество самых разных объектов может подписаться на него и получать уведомление каждый раз, когда пользователь щелкнет по кнопке.

Издатель и подписчик отделены друг от друга делегатом. Это очень удобно, поскольку такое решение позволяет писать гибкий и устойчивый код. Класс `Clock` может изменить способ определения времени, не нарушая при этом работу классов-подписчиков. Со своей стороны, подписчики могут менять реакцию на событие, не нарушая работу класса `Clock`. Независимое существование классов облегчает сопровождение кода.