Включает LINQ, .NET 3.5 CLR и библиотеки базовых классов .NET

Третье издание

C# 3.0

СПРАВОЧНИК

Джозеф Албахари
Бен Албахари

O'REILLY® bhv®

C# 3.0

IN A NUTSHELL

C# 3.0

IN A NUTSHELL

Third Edition

Joseph Albahari and Ben Albahari

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Джозеф Албахари
Бен Албахари

C# 3.0
СПРАВОЧНИК

Санкт-Петербург
«БХВ-Петербург»
2009

УДК 681.3.068+800.92C#
ББК 32.973.26-018.1
А45

Албахари, Дж.

А45 C# 3.0. Справочник: Пер. с англ. / Дж. Албахари, Б. Албахари. — 3-е изд. — СПб.: БХВ-Петербург, 2009. — 944 с.: ил.

ISBN 978-5-9775-0245-0

Книга представляет собой подробный справочник по программированию на языке C# 3.0, реализованном в Microsoft Visual Studio 2008. Первые главы целиком посвящены языку C#, начиная с основ синтаксиса, типов и переменных и заканчивая более сложными темами, такими как небезопасный код или препроцессорные директивы. В последующих главах рассмотрено ядро .NET 3.5 Framework, LINQ, XML, коллекции, ввод/вывод и работа в сети, управление памятью, рефлексия, атрибуты, безопасность, домены приложений и взаимодействие с небезопасным кодом. В отличие от предыдущих изданий, материал книги наряду с теоретическими основами сопровождается наглядными практическими примерами, что позволяет сочетать глубину изложения с легкостью понимания.

Для программистов

УДК 681.3.068+800.92C#
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Перевод с английского	*Сергея Иноземцева*
Редактор	*Ирина Иноземцева*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.06.09.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 76,11.
Тираж 2000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.003650.04.08 от 14.04.2008 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

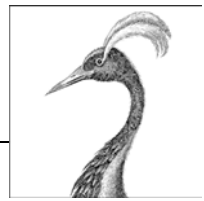
ISBN 978-0-596-52757-0 (англ.)
ISBN 978-5-9775-0245-0 (рус.)

Оглавление

Об авторах

Джозеф Албахари (Joseph Albahari) работает руководителем программных проектов на C# в австралийской компании Egton Software Services, которая является партнером крупнейшего производителя программных продуктов для здравоохранения в Великобритании. Он более 15 лет занимается разработкой крупных коммерческих приложений на .NET и других платформах для таких областей, как медицина, телекоммуникации и образование. Джозеф специализируется на создании заказных компонентов и написал прикладные платформы для трех компаний.

Бен Албахари (Ben Albahari) ранее работал менеджером программных проектов в Microsoft, где он, в частности, занимался созданием .NET Compact Framework и ADO.NET. Затем он стал соучредителем компании Genamics, поставляющей инструментальные средства для программистов, работающих на C# и J++, а также программные продукты для анализа ДНК и белковых последовательностей. Он был соавтором предыдущих изданий этой книги, а также "C# Essentials" ("Основы C#"), первой книги издательства O'Reilly, посвященной этому языку.



Предисловие

Каждый выпуск C# и .NET Framework содержит новые функциональные возможности и имеет больший потенциал для повышения производительности. Версия C# 3.0 содержит самые значительные изменения в языке за все время его развития, в том числе унифицированный синтаксис запросов, называемый LINQ (Language Integrated Query, запрос, интегрированный в язык). Он наводит мост между программами и их источниками данных, традиционно отделяемым друг от друга. Он также приближает C# к функциональным языкам, таким как LISP и Haskell.

Платой за рост функциональности является необходимость изучить больший объем материала. Хотя такие инструменты, как IntelliSense от Microsoft (а также многочисленные справочные материалы), отлично помогают справиться с этой задачей, все они предполагают наличие у читателя *карты концептуального знания*. Наша книга как раз и предлагает подобную карту в сжатом и унифицированном стиле, без излишней информации и длинных предисловий.

Это издание, в отличие от предыдущих, построено исключительно вокруг теоретических концепций и практических примеров, что делает книгу пригодной как для последовательного чтения, так и для просмотра в произвольном порядке. Кроме того, изложение материала ведется более углубленно, чем ранее, поскольку рассчитано на читателя с более низким уровнем начальных знаний. Поэтому книга стала более понятной, чем предыдущие издания.

В книге описываются язык C#, среда CLR и сборки ядра Framework. Мы решили сосредоточиться на этих темах, чтобы уделить больше внимания таким трудным вопросам, как нити выполнения, безопасность и домены приложений. Мы специально отмечаем функциональные возможности, впервые появившиеся в C# 3.0, и соответствующие нововведения во Framework, так что нашу книгу можно использовать и как справочник по C# 2.0.

Целевая аудитория

Книга рассчитана на аудиторию с уровнем подготовки от среднего до высокого. Предварительное знакомство с языком C# не требуется, но определенный опыт программирования необходим. Для новичка книга может послужить дополнением к учебнику программирования (но не заменит его).

Даже если вы знаете C# 2.0, вы найдете для себя полезную информацию на более чем ста страницах, посвященных LINQ и другим нововведениям в C# 3.0. Кроме

того, во многих главах мы постараемся углубить ваши знания языка и ядра платформы Framework.

Эта книга образует идеальное сочетание с любой из множества книг, посвященных таким прикладным технологиям, как WPF, ASP.NET или WCF. Особенности языка и платформы .NET Framework, которые остались за рамками этих книг, подробно описываются в нашей книге и наоборот.

Если вам нужна книга, где кратко обсуждаются все существующие технологии .NET Framework, то наша книга не для вас. Она также не подходит тому, кто ищет замену для IntelliSense (то есть алфавитные списки типов и их членов, присутствующие в предыдущих изданиях).

Как организована эта книга

Гл. 1 — это введение в C# и .NET Framework. *Гл. 2–4* целиком посвящены языку C#, начиная с основ синтаксиса, типов и переменных и заканчивая более сложными темами, такими как небезопасный код или препроцессорные директивы. Если вы не знакомы с языком, то вам следует прочитать эти главы последовательно, хотя разделы *гл. 4* можно читать в любом порядке.

В остальных главах описывается ядро платформы .NET Framework и затрагиваются такие темы, как LINQ, XML, коллекции, ввод/вывод и работа в сети, управление памятью, отражение, атрибуты, безопасность, нити выполнения, домены приложений и взаимодействие с небезопасным кодом. Порядок чтения большинства глав не принципиален, за исключением *гл. 6, 7* и *13*, в которых закладываются основы для последующих тем. *Гл. 8*, *9* и *10*, посвященные LINQ, тоже лучше читать по порядку.

Что необходимо для чтения этой книги

Для примеров кода, приведенных в книге, требуется компилятор C# 3.0 (или 2.0) и Microsoft .NET Framework 3.5 (или 3.0/2.0). Кроме того, вам понадобится документация Microsoft по платформе .NET. Простейший способ получить все сразу (а также интегрированную среду разработки) сводится к установке Microsoft Visual Studio. Для изучения материала, изложенного в книге, подойдет любое издание Visual Studio, в том числе и Visual Studio Express (которое сейчас можно загрузить бесплатно). Кроме того, Visual Studio содержит экспресс-версию SQL Server, которая потребуется для выполнения примеров LINQ to SQL, и инструментальное средство IntelliSense, выводящее на экран список членов типа по мере того, как вы набираете код.

Если вы ничего не имеете против использования обычного текстового редактора и командной строки, альтернативным решением может быть загрузка .NET Framework SDK. В комплект входят компилятор, документация по .NET и дополнительные утилиты командной строки.

Но проще всего загрузить и установить только Microsoft .NET Framework Runtime. Эта среда содержит компилятор, запускаемый из командной строки, однако в ней нет других утилит и документации.

Соглашения, использованные в книге

На рис. 1 приведена примерная схема изображения типов. Абстрактные классы представлены параллелограммами, а интерфейсы — кружками. Наследование обозначается непрерывной стрелкой, идущей от подтипа к базовому типу. Линия с окрашенным ромбом на конце означает любое другое отношение (ассоциирование, агрегирование или композицию).

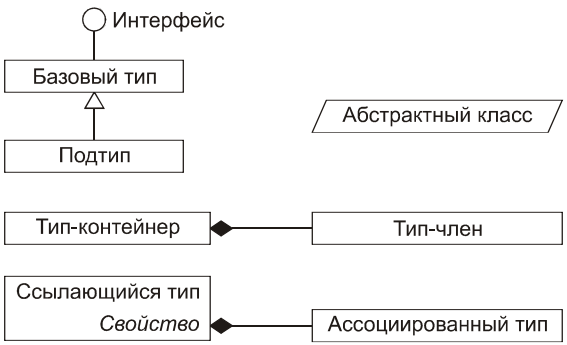


Рис. 1. Примерная схема

В книге использованы следующие типографские соглашения:

- *курсив* выделяет новые термины;
- моноширинный шрифт обозначает код C#, ключевые слова и идентификаторы, а также результат работы программы;
- **моноширинный шрифт, полужирный** выделяет в коде места, на которые следует обратить внимание;
- *моноширинный шрифт, курсив* обозначает текст, который должен быть заменен на пользовательский.

> ***Примечание***
>
> Содержит совет, предложение или замечание общего характера.

> **!** ***Предупреждение***
>
> Содержит предупреждение или предостережение.

Использование фрагментов кода

Эта книга должна помочь вам в работе. Вы можете использовать код, приведенный в этой книге, в своих программах и документах. Вы не обязаны получать наше раз-

решение за исключением тех случаев, когда воспроизводите значительные фрагменты кода. Например, для написания программы, в состав которой входят отрывки кода, приведенные в этой книге, разрешение не требуется. Но продажа или распространение CD-ROM с примерами из книг O'Reilly *требует* разрешения. Когда вы отвечаете на чей-то вопрос, цитируя эту книгу и фрагменты кода из нее, вы не нуждаетесь в разрешении. Если вы включаете значительный объем кода из этой книги в документацию к своему продукту, вы *должны* получить разрешение.

Мы не требуем обязательных ссылок на книгу при цитировании, но будем благодарны, если вы их приведете. Ссылка обычно содержит название книги, автора, издательство и ISBN. Например, "C# 3.0 in a Nutshell, by Joseph Albahari and Ben Albahari. Copyright 2007 Ben Albahari and Joseph Albahari, 978-0-596-52757-0".

Если вы сомневаетесь, не нарушаете ли вы разрешение на использование кода, изложенное выше, не стесняйтесь обратиться по адресу **permissions@oreilly.com**.

Благодарности

От Джозефа Албахари

Прежде всего я хочу поблагодарить своего брата и соавтора Бена Албахари за то, что он убедил меня участвовать в этом, как оказалось, весьма успешном проекте. Некоторые главы потребовали плотной совместной работы, которая еще больше сблизила нас. Мне было очень приятно работать с Беном, исследуя особо трудные темы. Он разделяет мое стремление подвергать сомнению прописные истины и упорно разбираться в сложных вопросах, пока не станет до конца понятно "как это работает".

Группа научных редакторов, многие из которых являются теперешними или бывшими сотрудниками Microsoft, была выше всяких похвал. Своим высоким качеством книга обязана глубине и широте познаний Николаса Палдино (Nicholas Paldino). То же самое я должен сказать о Джоэле Побаре (Joel Pobar) и о его глубочайшем знании среды CLR. Моей искренней благодарности заслуживают Кшиштоф Цвалина (Krzysztof Cwalina), Мэтт Уоррен (Matt Warren), Глин Гриффитс (Glyn Griffiths), Ион Василиан (Ion Vasilian), Брэд Абрамс (Brad Abrams), Сэм Джентил (Sam Gentile) и Адам Натан (Adam Nathan).

Я также хочу поблагодарить своего редактора Джона Осборна (John Osborn) за то, что он уделил много времени и сил координации этого проекта, и корректора Одри Дойл (Audrey Doyle) за наведение окончательного блеска. Я благодарю членов моей семьи Соню Албахари (Sonia Albahari) и Мири Албахари (Miri Albahari) за поддержку. Моя сестра Мири, которая сама недавно опубликовала книгу, прекрасно понимала, что такое отсутствие свободного времени!

Приношу благодарность Лутцу Рёдеру (Lutz Roeder), чья утилита .NET Reflector была моим основным инструментом анализа кода. Благодарю читателей моей электронной веб-книги "Threading in C#", чьи комментарии и полезные советы помогли мне в работе над настоящим изданием.

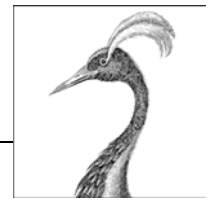
В заключение я хочу поблагодарить свою фирму Egton Software Services за использование технологии LINQ, лишь только появилась такая возможность. Я особенно благодарен Джеймсу Майлсу (James Miles) за его энтузиазм и творческий подход. Наша команда получила бесценный опыт реальной работы, необходимый каждому, кто пишет о технологии до ее официального выпуска.

От Бена Албахари

Поскольку мой брат выразил свою благодарность первым, вы легко догадаетесь, что я хочу сказать. ☺ Мы оба программируем практически с детства (у нас был один на двоих компьютер Apple IIe, брат писал свою операционную систему, а я игру "Виселица"), так что уже давно пора написать книгу вдвоем. Я надеюсь, что богатый опыт, который мы получили, работая над книгой, передастся вам при ее чтении.

Я хотел бы также поблагодарить своих бывших коллег по работе в Microsoft. Там работает много замечательных людей, не только умных, но и обладающих прекрасными человеческими качествами, и я скучаю по ним. В частности, я многому научился у Брайана Бекмана (Brian Beckman), которому весьма обязан.

ГЛАВА 1



Введение в C# и .NET Framework

C# — это универсальный объектно-ориентированный язык программирования, обеспечивающий безопасность преобразования типов. Целью его создания было повышение производительности труда программистов. Поэтому в языке сбалансированы простота, выразительность и эффективность. Главным разработчиком языка, начиная с первой версии, является Андерс Хейлсберг (Anders Hejlsberg), создатель Turbo Pascal и архитектор Delphi. Язык C# нейтрален к платформе, но в силу некоторых своих характеристик особенно хорошо работает на платформе Microsoft .NET Framework.

Объектная ориентированность

Язык C# является полноценной реализацией объектно-ориентированной парадигмы, включающей в себя *инкапсуляцию*, *наследование* и *полиморфизм*. Инкапсуляция означает создание вокруг *объекта* границы, отделяющей его внешнее поведение от внутренней реализации. С точки зрения объектной ориентированности отличительными чертами языка C# являются:

Унифицированная система типов

Основным строительным "кирпичиком" программы на языке C# является инкапсулированная единица данных и функций, называемая *типом*. C# имеет унифицированную систему типов, в которой все типы, в конечном счете, имеют общий базовый тип. Это означает, что все типы, от простых, таких как целые числа, до типов, представляющих сложные пользовательские объекты, имеют один базовый набор функциональных возможностей. Например, любой тип может быть преобразован в строку методом `ToString`.

Классы и интерфейсы

В чистой объектно-ориентированной парадигме единственным видом типа является *класс*. Однако в C# существуют и другие виды типов, к которым относится и *интерфейс* (аналогичный интерфейсам Java). Интерфейс подобен классу с тем отличием, что представляет собой только объявление типа, без реализации. Он особенно полезен в тех случаях, когда требуется множественное наследование (в отличие от таких языков, как C++ и Eiffel, язык C# не поддерживает множественное наследование классов).

Свойства, методы и события

В чистой объектно-ориентированной парадигме все функции являются *методами* (яркий пример — язык Smalltalk). В C# методы представляют собой лишь один из нескольких видов функций, наряду со *свойствами*, *событиями* (и другими видами). Свойства — это функции, инкапсулирующие фрагменты состояния объекта, например, цвет кнопки или текст надписи. События — это функции, которые упрощают действия, связанные с изменением состояния объекта.

Безопасность типов

Язык C#, в основном, безопасен в отношении преобразования типов. Это означает, что типы могут взаимодействовать только по протоколам, которые они же и определяют, обеспечивая тем самым внутреннюю непротиворечивость каждого типа. Например, C# не позволяет вам работать со *строковым* типом так, словно он *целочисленный*.

Говоря более конкретно, C# поддерживает *статическую типизацию*, т. е. обеспечивает безопасность типа на этапе компиляции. Это является дополнением к *динамической* безопасности типов, обеспечиваемой средой .NET CLR на этапе выполнения.

Статическая типизация позволяет выявить большое количество ошибок еще до запуска программы. Она избавляет вас от необходимости проводить специальное тестирование, поскольку перекладывает на компилятор проверку согласованности всех типов в программе. В результате большие программы становятся более управляемыми, более предсказуемыми и более устойчивыми. Кроме того, статическая типизация позволяет при написании программы в Visual Studio .NET использовать такие инструменты, как IntelliSense, поскольку всегда известен тип каждой конкретной переменной, и, следовательно, известно, какие именно методы можно вызывать для нее.

Про C# говорят, что это *язык со строгой типизацией*, потому что его правила, действующие в отношении типов (как статически, так и динамически), являются очень строгими. Например, вы не можете передавать число с плавающей точкой в качестве аргумента функции, работающей с целым, если вы предварительно *явно* не преобразуете число с плавающей точкой в целое. Этот принцип позволяет избежать многих ошибок.

Строгая типизация также играет свою роль при работе кода C# в изолированной среде, где каждый аспект безопасности контролируется приложением-хостом. В изолированной среде очень важно то, что вы не можете произвольным образом изменить (и "испортить") состояние объекта, обойдя его правила типизации.

Управление памятью

В программе на языке C# предполагается, что управление памятью производится средой выполнения автоматически. Среда CLR имеет сборщик мусора, работаю-

щий как компонент вашей программы и освобождающий память от объектов, на которые больше нет ссылок. Это избавляет программистов от необходимости явно освобождать память и снимает проблему некорректных указателей, типичную для таких языков, как C++.

Язык C# не исключает использование указателей; он просто делает их ненужными для большинства задач программирования. Вы можете применять указатели в ситуациях, требующих высокой производительности, а также для улучшения взаимодействия с другими программами, но делать это разрешено только в блоках кода, явно помеченных как небезопасные.

Работа на разных платформах

Как правило, язык C# применяется для написания кода, работающего на платформах Windows. Хотя корпорация Microsoft стандартизировала C# и CLR в соответствии с правилами ECMA, существует относительно мало ресурсов (как связанных с Microsoft, так и независимых), предназначенных для поддержки C# на платформах, отличных от Windows. Это означает, что если вашей основной задачей является разработка платформно-независимого приложения, то разумно сделать выбор в пользу такого языка, как Java. Таким образом, язык C# можно применять при написании кроссплатформенного кода в следующих случаях:

- код C# работает на сервере и обслуживает код DHTML, который может выполняться на любой платформе. Это как раз случай ASP.NET;
- код C# выполняется в среде, отличной от Microsoft CLR. Самым известным примером является проект Mono, имеющий собственный компилятор C# и среду выполнения, работающую в Linux, Solaris, Mac OS X и Windows;
- код C# выполняется на хосте, поддерживающем Microsoft Silverlight (в Windows или Mac OS X). Это новая технология, аналогичная Flash Player от Adobe.

Язык C# и среда CLR

Для выполнения кода C# требуется среда, имеющая такие функциональные возможности, как автоматическое управление памятью и обработка исключений. Структура языка C# соответствует структуре среды CLR, обладающей этими возможностями (хотя в техническом отношении C# не зависит от среды CLR). Более того, система типов C# хорошо ложится на систему типов CLR (например, в обеих используются одни и те же объявления базовых типов).

Среда CLR и платформа .NET Framework

Платформа .NET Framework состоит из среды выполнения *CLR* (Common Language Runtime, общеязыковая среда выполнения)) и большого количества библиотек. Библиотеки, в свою очередь, делятся на две группы: библиотеки ядра (которым и посвящена книга) и прикладные, зависящие от библиотек ядра. На рис. 1.1 пред-

ставлен общий вид этих библиотек (который может служить своеобразной картой нашей книги).

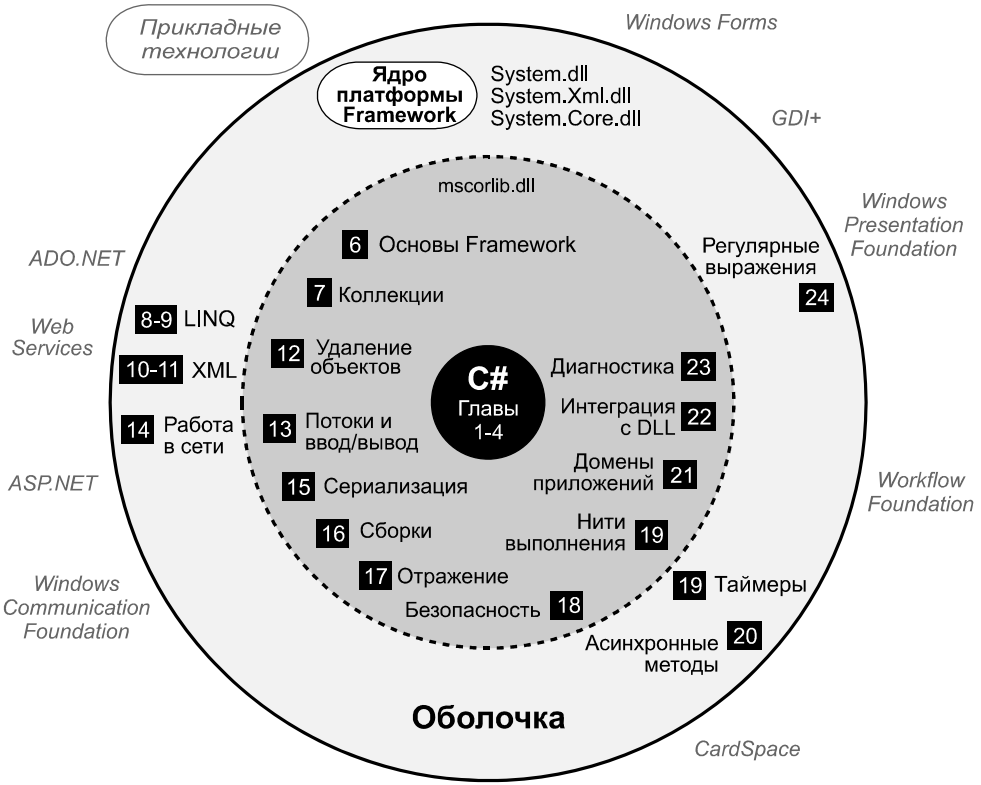


Рис. 1.1. Темы, затрагиваемые в книге, и номера глав, где они обсуждаются. Названия специальных технологий и библиотек, выходящих за рамки этой книги, выделены серым шрифтом и расположены за пределами оболочки

CLR является средой выполнения *управляемого кода*. Язык C# представляет собой один из *управляемых языков*, т. е. он компилируется в управляемый код. Такой код помещается в *сборку*, имеющую форму либо выполняемого файла (с расширением exe), либо библиотеки (с расширением dll) и содержащую также информацию о типах, называемых *метаданными*.

Управляемый код — это код на языке IL (Intermediate Language, промежуточный язык). Когда CLR загружает сборку, IL-код преобразуется в машинный код, например, x86. Это преобразование выполняется компилятором JIT (Just-In-Time, своевременный), являющимся частью CLR. Сборка сохраняет практически все конструкции исходного языка, что облегчает проверку кода и даже его динамическое генерирование.

Примечание

Приложение .NET Reflector, написанное Лутцем Рёдером (Lutz Roeder), является прекрасным инструментом для изучения содержимого сборки. Оно также может быть использовано в качестве декомпилятора.

Среда CLR имеет большое количество служб, действующих на этапе выполнения. В их число входят службы управления памятью, загрузки библиотек и обеспечения безопасности.

Среда CLR нейтральна в языковом отношении и позволяет разработчикам создавать приложения на самых разных языках (C#, Visual Basic .NET, Managed C++, Delphi .NET, Chrome .NET и J#).

Платформа .NET Framework содержит библиотеки, позволяющие написать практически любое Windows-приложение. В *гл. 5* дан обзор библиотек .NET.

Что нового в C# 3.0

Новые функциональные возможности версии C# 3.0 относятся, в основном, к технологии LINQ (Language Integrated Query, запрос, интегрированный в язык). Ее создателей вдохновила работа над проектом Comega (ранее известным под именами X# и Xen). Главный архитектор Эрик Мейер (Erik Meijer) тесно сотрудничал с Андерсом Хейлсбергом при включении этой технологии в состав C#.

LINQ предоставляет возможность писать SQL-подобные запросы прямо в программе на языке C#, причем их корректность проверяется *статически*. Архитектура LINQ позволяет выполнять запросы локально или удаленно. Платформа .NET Framework предоставляет API-интерфейсы, использующие LINQ при обращении к коллекциям, удаленным базам данных и XML-документам. В число функциональных возможностей C# 3.0 входят:

- лямбда-выражения;
- методы расширения;
- неявно типизированные локальные переменные;
- синтаксис, облегчающий восприятие запроса;
- анонимные типы;
- инициализаторы объектов;
- неявно типизированные массивы;
- автоматические свойства;
- частичные методы;
- деревья выражений.

Лямбда-выражения представляют собой миниатюрные функции, создаваемые динамически. Они являются результатом естественного развития анонимных методов, впервые появившихся в C# 2.0. Фактически лямбда-выражения обладают всеми функциональными возможностями анонимных методов. Например, следующее лямбда-выражение возводит в квадрат целое число:

```
Func<int,int> square = x => x * x;
Console.WriteLine (square(3));          // 9
```

Читателям, знакомым с языками *функционального программирования*, такими как Scheme и Haskell, лямбда-выражения не покажутся чем-то необычным. В языке C# лямбда-выражения применяются, в основном, в LINQ-запросах, например:

```
string[] names = { "Tom", "Dick", "Harry" };
IEnumerable<string> filteredNames =              // Выбрать только имена
  Enumerable.Where (names, n => n.Length >= 4); // из 4 и более символов
```

Методы расширения расширяют существующий тип, добавляя в него новые методы, но не меняя его определение. Они действуют как "синтаксический сахар", создавая впечатление, что статические методы ведут себя как методы экземпляра. Поскольку операторы LINQ-запросов реализованы в виде методов расширения, мы можем упростить предыдущий пример:

```
string[] names = { "Tom", "Dick", "Harry" };
IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
```

Неявно типизированные локальные переменные разрешают вам не указывать тип переменной при ее объявлении, позволяя компилятору распознать его самостоятельно. Поскольку компилятор способен определить тип переменной `filteredNames`, мы можем еще больше упростить наш запрос:

```
var filteredNames = names.Where (n => n.Length == 4);
```

Синтаксис, облегчающий восприятие запросов, позволяет писать запросы в стиле SQL. Он значительно упрощает некоторые виды запросов и тоже выступает в роли "синтаксического сахара" для запросов, имеющих вид лямбда-выражений. Перепишем предыдущий пример в соответствии с синтаксисом, облегчающим восприятие запросов:

```
var filteredNames = from n in names where n.Length >= 4 select n;
```

Анонимные типы — это простые классы, создаваемые динамически и обычно используемые при окончательном выводе запросов:

```
var query = from n in names where n.Length >= 4
            select new {
                         Name = n,
                         Length = n.Length
                       };
```

Вот еще более простой пример:

```
var dude = new { Name = "Bob", Age = 20 };
```

Неявно типизированные массивы исключают необходимость в объявлении типа массива, когда массив конструируется и инициализируется за один шаг:

```
var dudes = new[]
{
   new { Name = "Bob", Age = 20 },
   new { Name = "Rob", Age = 30 }
};
```

Инициализаторы объектов упрощают конструирование, позволяя устанавливать значения полей одновременно с вызовом конструктора. Инициализаторы объектов работают как с анонимными, так и с именованными типами. В следующем примере демонстрируется эта новая возможность языка C# 3.0 и приводится эквивалентный код на C# 2.0:

```
// C# 3.0
Bunny b1 = new Bunny { Name="Bo", LikesCarrots=true, LikesHumans=false };
// C# 2.0
Bunny b2 = new Bunny(  );
b2.Name = "Bo";
b2.LikesHumans = false;
```

Автоматические свойства облегчают работу программиста в тех случаях, когда нужно написать свойство, просто читающее или устанавливающее закрытое поле, содержащее резервное значение для свойства. В следующем примере компилятор автоматически генерирует закрытое резервное поле для свойства `X`:

```
public class Stock
{
  // C# 3.0:
              public decimal X { get; set; }
  // C# 2.0:
              private decimal y;
              public decimal Y
              {
                get { return y;  }
                set { y = value; }
              }
}
```

Частичные методы позволяют автоматически сгенерированному частичному классу предоставлять программисту возможности для ручной настройки. Технология LINQ to SQL использует частичные методы в специализированных классах, отображающих SQL-таблицы. Приведем пример пары частичных методов, один из которых сгенерирован автоматически, а другой написан вручную:

```
// PaymentFormGen.cs – сгенерирован автоматически
partial class PaymentForm
{
  ...
  partial void ValidatePayment (decimal amount);
}
// PaymentForm.cs – написан вручную
partial class PaymentForm
{
  ...
  partial void ValidatePayment (decimal amount)
```

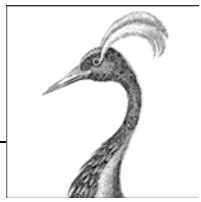
```
  {
    if (amount > 100)
      ...
  }
}
```

Деревья выражений являются миниатюрными объектными моделями, описывающими лямбда-выражения. Компилятор C# 3.0 генерирует деревья выражений, когда лямбда-выражение присваивается специальному типу `Expression<TDelegate>`:

```
Expression<Func<string,bool>> predicate = s => s.Length > 10;
```

Деревья выражений обеспечивают удаленное выполнение LINQ-запросов (например, на сервере базы данных), потому что они могут быть проанализированы и оттранслированы на этапе выполнения (например, в SQL-запрос).

ГЛАВА 2



Основы языка C#

В этой главе мы изложим основы языка C#.

Первая программа на C#

Разберем программу, которая умножает число 12 на 30 и выводит результат (360) на экран. Две косых, идущих подряд, указывают, что часть строчки после них является *комментарием*.

```
using System;                       // импортирование пространства имен
class Test                          // объявление класса
{
  static void Main( )               // объявление метода
  {
    int x = 12 * 30;                // оператор 1
    Console.WriteLine(x);           // оператор 2
  }                                 // конец метода
}                                   // конец класса
```

Основу программы составляют два *оператора*. Операторы в C# выполняются последовательно. Каждый оператор заканчивается точкой с запятой:

```
    int x = 12 * 30;
    Console.WriteLine(x);
```

Первый оператор вычисляет *выражение* `12 * 30` и сохраняет результат в *локальной переменной* `x` целого типа (`int`). Второй оператор вызывает *метод* `WriteLine` класса `Console`, чтобы вывести переменную `x` на экран.

Метод выполняет некое действие с помощью последовательности операторов, называемой *операторным блоком* и представляющей собой пару фигурных скобок, содержащую несколько операторов (возможно, ни одного). Мы определили метод по имени `Main`:

```
  static void Main( )
  {
    ...
  }
```

Применение функций высокого уровня, обращающихся к функциям низкого уровня, упрощает код. Мы можем переработать нашу программу так, что она будет содержать метод многократного использования, умножающий целое число на 12:

```
using System;
class Test
{
  static void Main( )
  {
    Console.WriteLine(FeetToInches(30));      // 360
    Console.WriteLine(FeetToInches(100));     // 1200
  }
  static int FeetToInches(int feet)
  {
    int inches = feet * 12;
    return inches;
  }
}
```

Метод может получать *входные* данные от вызывающей функции, для чего в нем определяются *параметры*, и возвращать ей *выходные* данные, и с этой целью в нем определяется *тип возвращаемого значения*. Мы создали метод по имени `FeetToInches` с одним параметром для ввода значения в футах и с типом возвращаемого значения для вывода значения в дюймах:

```
static int FeetToInches(int feet) {...}
```

Литералы 30 и 100 являются *аргументами*, передаваемыми методу `FeetToInches`. В нашем примере метод `Main` ничего не содержит в круглых скобках, потому что у него нет параметров, а поскольку он не возвращает никакого значения, то перед его именем стоит ключевое слово `void`:

```
static void Main( )
```

C# распознает метод по имени `Main` как точку входа в программу, определяемую по умолчанию. Метод `Main`, в принципе, может возвращать целое значение (а не `void`), передавая его вычислительной среде. Кроме того, он может принимать массив строковых аргументов (в который будут занесены аргументы, передаваемые программе). Например:

```
static int Main(string[] args) {...}
```

> ***Примечание***
>
> Массив (например, `string[]`) представляет собой набор из фиксированного количества элементов некоторого типа. Массивы определяются с помощью пары квадратных скобок, помещенной после типа элемента (см. *разд. "Массивы"* далее в этой главе).

Методы — это один из нескольких видов функций в C#. Другим видом являются *знаки операций*, например, используемый здесь знак "*", обозначающий умножение. Существуют также *конструкторы*, *свойства*, *события*, *индексаторы* и *деструкторы*.

В приведенном примере два метода сгруппированы в класс. *Класс* объединяет в себе функции и данные, образующие объектно-ориентированный "кирпичик" программы. Члены класса `Console` обрабатывают ввод/вывод, выполняемый через командную строку; примером метода этого класса является `WriteLine`. Класс `Test` имеет два метода, `Main` и `FeetToInches`. Класс аналогичен *типу*, в чем мы убедимся в *разд. "Основные сведения о типах"* далее в этой главе.

На самом внешнем уровне программы типы организованы в *пространства имен*. Директива `using` использована для того, чтобы пространство имен `System` стало доступным нашему приложению, и чтобы можно было работать с классом `Console`. Мы могли бы определить все свои классы в пространстве имен `TestPrograms` таким способом:

```
using System;
namespace TestPrograms
{
  class Test  {...}
  class Test2 {...}
}
```

Платформа .NET Framework организована в виде вложенных пространств имен. Вот пример пространства имен, содержащего типы для обработки текста:

```
using System.Text;
```

Директива `using` поставлена здесь для удобства. Кроме того, вы можете ссылаться на тип по его полному имени, т. е. по имени с префиксом, обозначающим пространство имен, например, `System.Text.StringBuilder`.

Компиляция

Компилятор C# преобразует исходный код, представленный в виде набора файлов с расширением cs, в сборку. *Сборка* — это единица компоновки и развертывания в .NET. Сборка может представлять собой либо *приложение*, либо *библиотеку*. Обычное консольное приложение или Windows-приложение имеет метод `Main` и является EXE-файлом. Библиотека — это файл с расширением dll; она эквивалентна EXE-файлу без точки входа. Ее предназначение заключается в том, что она вызывается из приложения или других библиотек. (Иногда говорят, что приложение или другая библиотека ссылается на эту библиотеку.) Платформа .NET Framework является набором библиотек.

Компилятор C# представляет собой файл csc.exe. Вы можете воспользоваться интегрированной средой разработки, например Visual Studio .NET, для автоматического запуска компилятора или скомпилировать программу вручную из командной строки. Во втором случае вы должны вначале сохранить программу в файле, например, MyFirstProgram.cs, а затем вызвать `csc` (расположенный в *<windows>* /Microsoft.NET/Framework) из командной строки следующим образом:

```
csc MyFirstProgram.cs
```

В результате будет создано приложение MyFirstProgram.exe.

Примечание

Сборки обсуждаются в *гл. 16*.

Синтаксис

Синтаксис C# основан на синтаксисе языков C и C++. В этом разделе мы обсудим элементы синтаксиса C# на примере следующей программы:

```
using System;
class Test
{
  static void Main( )
  {
    int x = 12 * 30;
    Console.WriteLine(x);
  }
}
```

Идентификаторы и ключевые слова

Идентификаторы — это имена, которыми программисты называют классы, методы, переменные и т. д. Перечислим идентификаторы в нашей программе в порядке их появления:

```
System   Test   Main   x   Console   WriteLine
```

Идентификатор должен быть словом, состоящим из символов Unicode и начинающимся с буквы или символа подчеркивания. Идентификаторы C# чувствительны к регистру. В соответствии с соглашением аргументы, локальные переменные и закрытые поля именуются в стиле camel (например, `myVariable`), а все остальные идентификаторы — в стиле Pascal (например, `MyMethod`).

Ключевые слова — это имена, зарезервированные компилятором, которые нельзя использовать в качестве идентификаторов. В нашей программе присутствуют следующие ключевые слова:

```
using   class   static   void   int
```

Приведем полный список ключевых слов C#:

```
abstract        As              base            bool            break
byte            Case            catch           char            checked
class           Const           continue        decimal         default
delegate        Do              double          else            enum
event           Explicit        extern          false           finally
fixed           Float           for             foreach         goto
if              Implicit        in              int             interface
internal        Is              lock            long            namespace
```

new	Null	object	operator	out
override	Params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
while				

Как избежать конфликтов

Если вам действительно необходимо использовать идентификатор, совпадающий с ключевым словом, вы можете квалифицировать его с помощью префикса `@`. Например:

```
class class  {...}      // так нельзя
class @class {...}      // так можно
```

Символ `@` не считается частью идентификатора. Следовательно, `@myVariable` и `myVariable` — одно и то же.

Контекстные ключевые слова

В языке C# также присутствуют *контекстные ключевые слова*. Хотя они и распознаются компилятором, их можно использовать в качестве идентификаторов, не квалифицируя. Их список:

add	ascending	by	descending	equals
from	get	global	group	in
into	join	let	on	orderby
partial	remove	select	set	value
var	where	yield		

В контексте применения этих ключевых слов двусмысленность не возникает.

Литералы, пунктуаторы и знаки операций

Литералы — это простейшие элементы данных, статически встроенные в программу. В нашем примере были использованы следующие литералы:

```
12   30
```

Пунктуаторы помогают разметить структуру программы. В нашем примере встречаются такие пунктуаторы:

```
;   {   }
```

Точка с запятой служит для завершения оператора. Это означает, что сам оператор может занимать несколько строчек:

```
Console.WriteLine
  (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

Фигурные скобки объединяют несколько операторов в один операторный блок.

Знаки операций участвуют в выражениях. Большинство знаков операций в C# состоит из одного символа, например, для умножения используется символ `*`. Знаки операций подробно обсуждаются далее в этой главе. В нашей программе были использованы следующие знаки операций:

```
.  (  )  *  =
```

Точка означает, что конструктивный элемент является членом более общего элемента. Круглые скобки используются при объявлении или вызове метода. Если метод не принимает никаких аргументов, ставятся пустые скобки. Знак равенства служит для *присваивания* (двойной знак равенства, `==`, используется при сравнении на равенство, о чем мы поговорим чуть позже).

Комментарии

В C# существуют два разных стиля документирования исходного кода: *однострочные комментарии* и *многострочные комметарии*. Однострочный комментарий начинается с двух косых (`//`) и продолжается до конца строки. Например:

```
int x = 3;   // переменной x присваивается значение 3
```

Многострочный комментарий начинается с символов `/*` и заканчивается символами `*/`. Например:

```
int x = 3;   /* этот комментарий
                занимает две строки */
```

Комментарии могут содержать документирующие XML-теги *(см. разд. "XML-документирование" гл. 4)*.

Основные сведения о типах

Тип определяет, каким должно быть значение. *Значение* — это место в памяти, отведенное под *переменную* или *константу*. Переменная представляет значение, которое может измениться, а константа — постоянную величину (о константах мы поговорим далее в этой главе). В нашей первой программе мы создали локальную переменную по имени `x`:

```
static void Main( )
{
  int x = 12 * 30;
  Console.WriteLine(x);
}
```

Все значения в C# являются *экземплярами* определенного типа. Характер и множество допустимых значений переменной зависят от ее типа. Переменная `x` имеет тип `int`.

Примеры стандартных типов

Стандартные типы — это типы, поддерживаемые компилятором. Тип `int` является базовым стандартным типом, представляющим целые числа, попадающие в диапазон от -2^{31} до $2^{31}-1$ и занимающие 32 бита в памяти. С экземплярами типа `int` мы можем выполнять, например, арифметические действия:

```
int x = 12 * 30;
```

Другим стандартным типом C# является тип `string`. Он представляет последовательность символов, например, ".NET" или "http://oreilly.com". Мы можем манипулировать строками, вызывая специальные функции:

```
string message = "Hello world";
string upperMessage = message.ToUpper( );
Console.WriteLine(upperMessage);       // выводится: HELLO WORLD
int x = 2007;
message = message + x.ToString( );
Console.WriteLine(message);            // выводится: Hello world2007
```

Базовый тип `bool` имеет только два значения, `true` и `false`. Этот тип обычно применяется для реализации ветвления в вычислениях с помощью условного оператора `if`. Например:

```
bool simpleVar = false;
if (simpleVar)
  Console.WriteLine("This will not print");
int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
  Console.WriteLine ("This will print");
```

> ***ПРИМЕЧАНИЕ***
>
> В языке C# стандартные типы (которые также называются встроенными) обозначаются при помощи ключевых слов. Пространство имен `System` в .NET Framework содержит много важных типов, не являющихся стандартными (например, `DateTime`).

Примеры пользовательских типов

Подобно тому, как мы строим сложные функции из простых, мы можем строить сложные типы из базовых. В следующем примере определяется пользовательский тип `UnitConverter`. Это класс, который используется в качестве основы для преобразования единиц измерения:

```
using System;

public class UnitConverter
{
     int ratio;                                          // поле
     public UnitConverter(int unitRatio) {ratio = unitRatio; } // конструктор
     public int Convert(int unit)        {return unit * ratio; } // метод
}
```