УДК 004.438 C#
ББК 32.973.26-018.1
Ф71

Фленов М. Е.

Ф71 Библия C#. — 4-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2019. — 512 с.: ил.

ISBN 978-5-9775-4041-4

Книга посвящена программированию на языке C# для платформы Microsoft .NET, начиная с основ языка и разработки программ для работы в режиме командной строки и заканчивая созданием современных приложений различной сложности (баз данных, графических программ и др.). Материал сопровождается большим количеством практических примеров. Подробно описывается логика выполнения каждого участка программы. Уделено внимание вопросам повторного использования кода. В четвертом издании уделено особое внимание универсальным приложениям Windows и платформе .NET Core, позволяющей писать код, который может выполняться на Windows, macOS и Linux. На сайте издательства находятся примеры программ, дополнительная справочная информация, а также готовые компоненты, тестовые программы и изображения.

Для программистов

УДК 004.438 C#
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Сависте*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн обложки	*Инны Тачиной*
Оформление обложки	*Карины Соловьевой*

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

ISBN 978-5-9775-4041-4

Оглавление

Предисловие

Хотя эта книга и носит название «Библия C#», посвящена она в целом будет .NET — платформе от компании Microsoft, которая состоит из полного набора инструментов для разработчиков (.NET Framework) и для пользователей. В нее входят клиентская и серверная операционные системы (ОС), инструменты разработки и сервисы. В этой книге мы рассмотрим .NET Framework, который нужен программистам для написания программ, а также язык программирования C#, на котором мы и станем писать свой код.

Меня очень часто спрашивают, как я отношусь к .NET. К первой версии .NET Framework я относился не очень хорошо, потому что не понял ее и не увидел преимуществ. Она выглядела скорее как копия Java, причем не совсем привычная. Однако с появлением второй версии .NET кардинально изменилась, а вслед и я полностью изменил свое мнение по поводу C#. Хотя нет — если быть точным, то мое отношение к .NET первой версии так и осталось скорее негативным, чем положительным или нейтральным. А вот ко второй ее версии и ко всем последующим я отношусь не то чтобы положительно, а даже с восхищением.

Большинство языков программирования с богатой историей обладают одним большим недостатком. За время существования в них накапливается много устаревшего и небезопасного, но все это продолжает оставаться в языке для сохранения совместимости с уже написанным кодом. Разработка абсолютно нового языка позволила компании Microsoft избавиться от всего старого и создать что-то новое, чистое и светлое. Наверное, это слишком громкие слова, но сказать их, тем не менее, хочется.

На самом же деле .NET действительно чиста от многих ошибок и проблем прошлого, потому что и новый язык, и платформа были построены с учетом накопившегося опыта.

Чтобы понять, почему мне нравятся .NET и C#, давайте выделим их реальные преимущества. Они уже есть в .NET и никуда не денутся. Итак, к основным преимуществам платформы .NET я бы отнес:

- *универсальный API* — на каком бы языке вы ни программировали, вам предоставляются одни и те же имена классов и методов. Все языки для платформы

.NET отличаются только синтаксисом, а классы используются из .NET Framework. Таким образом, все языки схожи по возможностям, и вы выбираете только тот, который вам ближе именно по синтаксису. Я начинал изучать программирование с Basic, затем пошли Pascal, C, C++, Assembler, Delphi, Java и сейчас — .NET. При переходе с языка на язык приходится очень много времени тратить на изучение нового API. На платформе .NET больше нет такой проблемы.

И тут преимущество не только в том, что все языки одинаковы, а в том, что улучшается возможность взаимодействия программ, написанных на разных языках. Раньше для того, чтобы программа на C++ без проблем взаимодействовала с кодом на Visual Basic или Delphi, приходилось применять различные трюки и уловки. В основном это было связано тем, что каждый язык по-своему обрабатывал и хранил строки и передавал переменные. Сейчас такой проблемы не существует, и все типы данных в C# абсолютно совместимы с Visual Basic .NET или другим языком платформы .NET.

Соответственно, программисты, использующие различные языки, могут работать над одним и тем же проектом и без швов сращивать модули на разных языках.

Все это звучит красиво, но есть и один большой недостаток — поддерживать такой проект будет весьма сложно. Я бы все же не рекомендовал писать один и тот же проект на разных языках. Сейчас C# стал самым популярным для платформы .NET, и на него переходят даже программисты Visual Basic, поэтому лучше выбрать именно этот язык;

- *защищенный код* — платформу Win32 очень часто ругали за ее незащищенность. В ней, действительно, есть очень слабое звено — незащищенность кода и возможность перезаписывать любые участки памяти. Самым страшным в Win32 была работа с массивами, памятью и со строками (последние являются разновидностью массива). С одной стороны, это предоставляет мощные возможности системному программисту, который умеет правильно распоряжаться памятью. С другой стороны, в руках неопытного программиста такая возможность превращается в уязвимость. Сколько раз нам приходилось слышать об ошибках переполнения буфера из-за неправильного выделения памяти? Уже и сосчитать сложно.

  На платформе .NET вероятность такой ошибки стремится к нулю — если вы используете управляемый код, и если Microsoft не допустит ошибок при реализации самой платформы.

  Впрочем, платформа .NET не является абсолютно безопасной, потому что существуют не только ошибки переполнения буфера, — есть еще и ошибки логики работы программы. А такие ошибки являются самыми опасными, и от них нет волшебной таблетки;

- *полная ориентированность на объекты*. Объектно-ориентированное программирование (ООП) — это не просто дань моде, это мощь, удобство и скорость разработки. Платформа Win32 все еще основана на процедурах и наследует все недостатки этого подхода. Любые объектные надстройки — такие как Object

Windows Library (OWL), Microsoft Foundation Classes (MFC) и др. — решают далеко не все задачи;

- *сборка мусора* — начинающие программисты очень часто путаются с тем, когда нужно уничтожать объекты, а когда это делать не обязательно или даже вредно. Приходится долго объяснять, что такое локальные и глобальные переменные, распределение памяти, стек и т. д. Даже опытные программисты нередко допускают ошибки при освобождении памяти. А ведь если память освободить раньше, чем это необходимо сделать, то обращение к несуществующим объектам приведет к краху программы.

  На платформе .NET за уничтожение объектов, хотя вы и можете косвенно повлиять на этот процесс, отвечает сама платформа. В результате у вас не будет утечек памяти, и вместо того, чтобы думать об освобождении ресурсов, вы можете заниматься более интересными вещами. А это приводит и к повышению производительности труда;

- *визуальное программирование* — благодаря объектно-ориентированному подходу стало проще создавать визуальные языки программирования. Если вы программировали на Visual C++, то, наверное, уже знаете, что этот язык далек от идеала, а его визуальные возможности сильно ограниченны по сравнению с такими языками, как Delphi и Visual Basic. Новый язык C# действительно визуален и по своим возможностям практически не уступает самой мощной (по крайней мере, до появления .NET) визуальной среде разработки Delphi. Визуальность упрощает создание графического интерфейса и ускоряет разработку, а значит, ваша программа сможет раньше появиться на рынке и захватить его. Как показывает практика, очень часто первый игрок снимает основные сливки;

- *компонентное представление* — поскольку платформа имеет полностью объектную основу, появилась возможность компонентно-ориентированного программирования (как это сделано в Delphi). В платформе Win32 предпринимались попытки создания компонентной модели с помощью ActiveX, но развертывание подобных приложений было слишком сложной задачей. Большинство разработчиков старались не связываться с этой технологией (в том числе и я), а если жизнь заставляла, то для развертывания приходилось создавать инсталляционные пакеты. Самостоятельно создавать пакет — сложно и очень скучно. На платформе .NET установка новых пакетов сводится к простому копированию файлов без необходимости регистрации в реестре;

- *распределенные вычисления* — платформа .NET ускоряет разработку приложений с распределенными вычислениями, что достаточно важно для корпоративного программного обеспечения. В качестве транспорта при взаимодействии используются технологии HTTP[1], XML[2], SOAP[3] и великолепная и быстрая среда

---

[1] HTTP, HyperText Transfer Protocol — протокол передачи гипертекстовых файлов.

[2] XML, Extensible Markup Language — расширяемый язык разметки.

[3] SOAP, Simple Object Access Protocol — простой протокол доступа к объектам.

WCF (Windows Communication Foundation), которая позволяет связывать сервисы различными протоколами;

- *открытость стандартов* — при рассмотрении предыдущего преимущества мы затронули открытые стандарты HTTP, XML и SOAP. Открытость — это неоспоримое преимущество, потому что предоставляет разработчику большую свободу. Платформа .NET является открытой, и ее может использовать кто угодно. Недавно Microsoft начала перенос .NET на другие платформы.

Список можно продолжать и дальше, но уже видно, что будущее у платформы есть. Каким станет это будущее — пока еще большой и сложный вопрос. Но, глядя на средства, которые были вложены в разработку и рекламную кампанию, можно предположить, что Microsoft не упустит своего и сделает все возможное для обеспечения долгой и счастливой жизни .NET Framework.

Я знаком с .NET с момента появления первой версии и пытался изучать ее, несмотря на то, что синтаксис и возможности платформы, как уже было отмечено ранее, не вызывали у меня восторга. Да, язык C# вобрал в себя все лучшее от C++ и Delphi, но все равно работа с ним особого удовлетворения поначалу не приносила. Тем не менее я продолжал его изучать в свободное время, т. к. люблю находиться на гребне волны, а не стоять в очереди за догоняющими.

Возможно, мое первоначальное отрицательное отношение было связано со скудными возможностями самой среды разработки Visual Studio, потому что после выхода финальной версии Visual Studio .NET и версии .NET Framework 2.0 мое отношение к языку начало меняться к лучшему. Язык и среда разработки стали намного удобнее, а код читабельным, что очень важно при разработке больших проектов. Очень сильно код улучшился благодаря появлению частичных классов (partial classes) и тому, что Microsoft отделила описание визуальной части от основного кода. С этого момента классы стали выглядеть намного чище и красивее.

В настоящее время мое личное отношение к языку и среде разработки Visual Studio .NET более чем положительное. Для меня эта платформа стала номером один в моих личных проектах. И я продолжаю удивляться, как одна компания смогла разработать за такой короткий срок целую платформу, позволяющую быстро решать широкий круг задач.

Благодарности

Я хочу в очередной раз поблагодарить свою семью, которая была вынуждена терпеть мое отсутствие, пока я сидел над книгой.

Я благодарю всех тех, кто помогал мне в работе с ней в издательстве «БХВ-Петербург»: редакторов и корректоров, дизайнеров и верстальщиков — всех, кто старался сделать эту книгу интереснее для читателя.

Спасибо всем моим друзьям — тем, кто меня поддерживал, и всем тем, кто уговорил меня все же написать эту книгу, а это в большинстве своем посетители моего сайта **www.flenov.info**.

Бонус

Я долго не хотел браться за этот проект, имея в виду, что люди в наше время не покупают книги, а качают их из Интернета, да и тратить свободное время, отрывая его от семьи, тяжеловато. Очень жаль, что люди не покупают книг. Издательство «БХВ-Петербург» устанавливает на свои издания не столь уж высокие цены, и можно было бы отблагодарить всех тех людей, которые старались для читателя, небольшой суммой. Если вы скачали книгу, ознакомились с ней и она вам понравилась, то я советую вам купить ее полноценную «бумажную» версию.

Я же со своей стороны постарался сделать книгу максимально интересной, а на FTP-сервере издательства «БХВ-Петербург» мы выложили сопровождающую ее дополнительную информацию в виде статей и исходных кодов для дополнительного улучшения и совершенствования ваших навыков (см. *приложение*).

Электронный архив с этой информацией можно скачать по ссылке **ftp://ftp.bhv.ru/9785977540414.zip** или со страницы книги на сайте **www.bhv.ru**.

Структура книги

В своих предыдущих книгах я старался как можно быстрее приступить к показу визуального программирования в ущерб начальным теоретическим знаниям. В этой же я решил потратить немного времени на погружение в теоретические глубины искусства программирования, чтобы потом рассмотрение визуального программирования пошло как по маслу. А чтобы теория вводной части не была слишком скучной, я старался подносить ее как можно интереснее и, по возможности, придумывал занимательные примеры.

В первых четырех главах мы станем писать программы, не имеющие графического интерфейса, — информация будет выводиться в консоль. В мире, где властвует графический интерфейс: окна, меню, кнопки и панели, — консольная программа может выглядеть несколько дико. Но командная строка еще жива — она, наоборот, набирает популярность, и ярким примером тому является PowerShell (это новая командная строка, которая поставляется в Windows Server 2008 и может быть установлена в Windows 7 или Windows 8/10).

К тому же есть еще и программирование Web, где вообще нет ничего визуального. Я последние 13 лет работаю в Web-индустрии и создаю на языке C# сайты, которые работают под IIS. Мне так же очень часто приходится писать разные консольные утилиты, предназначенные для импорта/экспорта данных из баз, их конвертации и обработки, создания отчетов и т. п.

Визуальное программирование мы начнем изучать только тогда, когда познакомимся с основами .NET и объектно-ориентированным программированием. Мои примеры не всегда будут занимательными, но я старался придумывать что-то познавательное, полезное и максимально приближенное к реальным задачам, которые вам придется решать в будущем, если вы свяжете свою работу с программированием или просто станете создавать что-либо для себя.

Рассмотрев работу с компонентами, мы перейдем к графике и к наиболее важному вопросу для тех, кто профессионально работает в компаниях, — к базам данных. Да, профессиональные программисты чаще всего работают именно с базами данных, потому что в компаниях компьютеры нужны в основном для управления данными и информацией, которая должна быть структурированной и легкодоступной.

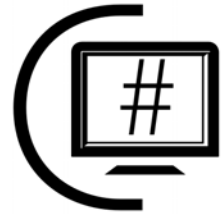
Я не пытался переводить файл справки, потому что Microsoft уже сделала это для нас. Моя задача — научить вас понимать программирование и уметь строить код. А за подробным описанием классов, методов и т. п. всегда можно обратиться к MSDN (Microsoft Development Network) — обширной библиотеке технической информации, расположенной на сайте **www.msdn.com** и доступной теперь и на русском языке, — на сайте имеется перевод всех классов, методов и свойств с подробным описанием и простыми примерами.

В третьем издании книги я исправил некоторые имевшиеся ошибки и добавил описание новых возможностей, которые Microsoft представила миру в .NET 3.5, 4.0 и 4.5. Язык развивается очень быстро и динамично, уже идет работа над .NET 5.0, и я полагаю, что строятся планы и на следующую версию.

В этом, четвертом, издании я обращаю больше внимания на .NET Core, потому что этот стандарт позволяет писать .NET-код, который может выполняться на разных платформах. Это как бы еще одна реинкарнация платформы, в которой Microsoft пытается сделать ее еще лучше. Это не значит, что старый код можно забыть, и .NET Framework мертв. Весь существующий код никуда не делся, его просто стандартизируют. Но более детально с .NET Core мы начнем знакомиться, начиная уже со следующей главы.

Итак, пора переходить к более интересным вещам и начать погружение в мир .NET.

ГЛАВА 1



Введение в .NET

Платформа Microsoft .NET Framework состоит из набора базовых классов и *общеязыковой среды выполнения* (Common Language Runtime, CLR). Базовые классы, которые входят в состав .NET Framework, поражают своей мощью, универсальностью, удобством использования и разнообразием.

Я постараюсь дать только минимум необходимой вводной информации, чтобы как можно быстрее перейти к изучению языка C# и к самому программированию. Я люблю делать упор на практику и считаю ее наиболее важной в нашей жизни. А по ходу практических занятий мы будем ближе знакомиться с теорией.

Для чтения этой и последующих глав рекомендуется иметь установленную среду разработки Visual Studio 2017 или более позднюю. Большинство примеров, приведенных в книге, написаны много лет назад с использованием Visual Studio 2008, но во время работы над этим изданием все примеры открывались и компилировались в новой версии Visual Studio, потому что развитие языка и среды разработки идет с полной обратной совместимостью.

Для большинства примеров достаточно даже бесплатной версии среды разработки Visual C# Community Edition, которую можно свободно скачать с сайта компании Microsoft (**www.visualstudio.com**). И хотя эта версия действительно бесплатная, с ее помощью можно делать даже большие проекты.

1.1. Платформа .NET

В *предисловии* мы уже затронули основные преимущества .NET, а сейчас рассмотрим эту платформу более подробно. До недавнего времени ее поддерживала только Microsoft, но на C# можно было писать программы для Linux (благодаря Mono[1]) и даже игры на Unity[2]. Однако все эти реализации отличались немного друг от дру-

[1] Mono — проект по созданию полноценного воплощения системы .NET Framework на базе свободного программного обеспечения.

[2] Unity — межплатформенная среда разработки компьютерных игр.

га — если написать программу для Windows, то это не значит, что то же самое заработает в Mono на Linux или где-либо еще.

Когда Microsoft начала переносить .NET на Linux и даже на macOS, она явно решила навести порядок в разнообразии реализаций, и появился первый .NET Standard. Сейчас уже существует несколько версий этого стандарта, и последняя из них 2.0. В чем смысл стандарта? Если вы пишете код под стандарт 2.0, вы можете быть уверены, что он будет одинаково работать на любой платформе, которая поддерживает этот стандарт.

Другими словами, стандарт — это набор возможностей, который в программе обязательно должен быть реализован, чтобы она соответствовала этому стандарту.

Я не могу знать реальных причин принятия решений внутри компании Microsoft, но мне кажется, что на появление .NET Core снова повлияло желание перенести .NET на другие платформы. Core по-английски — это *ядро*. Изначально в .NET включили все, что необходимо для написания полноценных приложений, но в каждой ОС своя файловая система, да и многие моменты реализованы по-разному. Нельзя было просто взять и перенести на все платформы все функции .NET, и вместо этого было создано ядро, которое точно будет работать на всех платформах.

При написании консольных приложений на .NET Core или на старом добром .NET Framework вы особой разницы не заметите. А вот для Web разница есть, и если начинать новый Web-проект, то я бы делал его на .NET Core, потому что архитектуру написания Web изменили весьма сильно. Новая архитектура намного лучше, она позволяет компилировать код прямо на лету, а многие удобные возможности поддерживаются прямо «из коробки».

Не знаю, насколько для вас это является преимуществом, но исходные коды .NET Core открыты, а значит, это проект OpenSource, и его исходные коды можно найти на сайте github по адресу: **https://github.com/dotnet**. Мне лично этот факт не важен, потому что в исходные коды я заглядывать не планирую. Но если вам интересно на них посмотреть, вы можете это сделать, а если у вас еще и достаточно опыта, то можно попробовать даже что-то в них улучшить и предложить свое улучшение.

Итак, .NET Core — это реализация .NET-стандарта с открытым исходным кодом, и в этой книге мы будем использовать его для изучения C#. Впрочем, если вы пишете приложение, которое будет выполняться только на Windows, то вполне нормально выбрать в качестве основы и .NET Framework 4.6.

Теперь еще пару слов о том, что мы будем изучать с точки зрения визуального интерфейса. Ранее основным подходом к построению визуального интерфейса являлся Windows Forms, когда вы в визуальном редакторе создавали интерфейс будущей программы, а Visual Studio в фоновом режиме превращал все это в .NET-код. Подход хороший, и таким образом работали многие до недавнего времени. Достаточно отметить, что и в 3-м, предыдущем, издании книги визуальный интерфейс строился именно с помощью Windows Forms, но для этого издания я все переписал с использованием нового подхода — WPF (Windows Presentation Foundation).

Сейчас в «дикой природе» существует огромное количество устройств с разными размерами экранов: от смартфонов с диагональю в 4 дюйма до Surface Hub величи-

ной с телевизор. Чтобы интерфейс приложения адаптировался к разным размерам экрана, в Microsoft используют новый подход — декларативный. Вы все так же можете визуально создавать интерфейс, но визуальный редактор будет хранить его с помощью описания XML. Оно более гибкое и по своему виду чем-то похоже на Web-страницы. HTML, который используется в Web, по своей реализации очень похож на XML, только на сильно упрощенный.

Как Web-страницы адаптируются к разным размерам экрана в браузере, так и интерфейс, созданный на основе WPF, может адаптироваться, что позволяет писать приложения, которые будут работать на любой платформе. Тем самым Microsoft открыла программистам возможность писать программы для любой платформы Windows, — она назвала это Universal Windows Platform (UWP, универсальная платформа Windows). Программа, написанная для этой платформы, сможет запускаться не только на компьютерах под Windows, но и на Windows-планшетах и на приставке Xbox. Раньше еще имело смысл сказать про возможность запуска таких программ даже на смартфоне, но Microsoft, похоже, уходит с этого рынка.

Мне кажется, что UWP пришла всерьез и надолго, поэтому при описании визуальных интерфейсов в этой книге мы будем использовать именно ее.

Ну а теперь все по порядку. Мы узнали, что такое .NET Standard и .NET Core, зачем они нужны и когда их использовать. Теперь пора знакомиться уже с языком C#.

1.1.1. Кубики .NET

Если отбросить всю рекламу, которую нам предлагают, и взглянуть на проблему глазами разработчика, то .NET описывается следующим образом: в среде разработки Visual Studio .NET вы можете с использованием .NET Framework разрабатывать приложения любой сложности, которые очень просто интегрируются с серверами и сервисами от Microsoft.

Основа всего, центральное звено платформы — это .NET Framework. Давайте посмотрим на основные составляющие этой платформы:

- *среда выполнения Common Language Runtime (CLR)*. CLR работает поверх ОС— это и есть виртуальная машина, которая обрабатывает IL-код[1] программы. Код IL — это аналог бинарного кода для платформы Win32 или байт-кода для виртуальной машины Java. Во время запуска приложения IL-код на лету компилируется в машинный код под то «железо», на котором запущена программа. Да, сейчас практически все работает на процессорах Intel, но никто не запрещает реализовать платформу на процессорах других производителей;
- *базовые классы .NET Framework* или *.NET Core* — в зависимости от того, пишете вы приложение, не зависящее от платформы, или для Windows. Как и библиотеки на других платформах, здесь нам предлагается обширный набор классов, которые упрощают создание приложения. С помощью таких компонентов вы можете строить свои приложения как бы из блоков.

[1] IL, Intermediate Language — промежуточный язык.

Когда я услышал такое пояснение в отношении MFC, то не понял его смысла, потому что построение приложений с помощью классов MFC более похоже на рытье нефтяной скважины лопатой, но никак не на строительство из блоков. А вот компоненты .NET реально упрощают программирование, и разработка приложений с помощью расстановки компонентов действительно стала похожа на строительство домика из готовых панелей;

- *расширенные классы .NET Framework*. В предыдущем пункте говорилось о базовых классах, которые реализуют базовые возможности. Также выделяют и более сложные компоненты доступа к базам данных, XML и др.

Надо также понимать, что .NET не является переработкой Java, — у них общее только то, что обе платформы являются виртуальными машинами и выполняют не машинный код, а байт-код. Да, они обе произошли из C++, но «каждый пошел своей дорогой, а поезд пошел своей» (как поет Макаревич).

1.1.2. Сборки

Термин *сборки* в .NET переводят и преподносят по-разному. Я встречал много различных описаний и переводов, например: *компоновочные блоки* или *бинарные единицы*. На самом деле, если не углубляться в терминологию, а посмотреть на сборки глазами простого программиста, то окажется, что это просто файлы, являющиеся результатом компиляции. Именно *конечные файлы*, потому что среда разработки может сохранить на диске после компиляции множество промежуточных файлов.

Наиболее распространены два типа сборок: библиотеки, которые сохраняются в файлах с расширением dll, и исполняемые файлы, которые сохраняются в файлах с расширением exe. Несмотря на то, что расширения файлов такие же, как и у Win32-библиотек и приложений, это все же совершенно разные файлы по своему внутреннему содержимому. Программы .NET содержат не инструкции процессора, как классические Win32-приложения, а IL-код[1]. Этот код создается компилятором и сохраняется в файле. Когда пользователь запускает программу, то она на лету компилируется в машинный код и выполняется на процессоре.

Так что я не буду в книге использовать мудреные названия типа «компоновочный блок» или «бинарная единица», а просто стану называть вещи своими именами: исполняемый файл, библиотека и т. д. — в зависимости от того, что мы компилируем. А если нужно будет определить эти вещи общим названием, не привязываясь к типу, я просто буду говорить «сборка».

Благодаря тому, что IL-код не является машинным, а интерпретируется JIT-компилятором[2], то говорят, что *код управляем* этим JIT-компилятором. Машинный код выполняется напрямую процессором, и ОС не может управлять этим кодом.

---

[1] Как уже отмечалось ранее, IL-код — это промежуточный язык (Intermediate Language). Можно также встретить термины CIL (Common Intermediate Language) или MSIL (Microsoft Intermediate Language).

[2] Just-In-time Compiler — компилятор периода выполнения.

А вот IL-код выполняется на .NET платформе, и уже она решает, как его выполнять, какие процессорные инструкции использовать, а также берет на себя множество рутинных вопросов безопасности и надежности выполнения.

Тут нужно пояснить, почему программы .NET внешне не отличаются от классических приложений, и файлы их имеют те же расширения. Так сделали чтобы скрыть сложности реализации от конечного пользователя. Зачем ему знать, как выполняется программа и на чем она написана? Это для него совершенно не имеет значения. Как я люблю говорить, главное — это качество программы, а как она реализована, на каком языке и на какой платформе, нужно знать только программисту и тем, кто этим специально интересуется. Конечного пользователя это интересовать не должно.

Помимо кода, в сборке хранится информация (метаданные) о задействованных типах данных. Метаданные сборки очень важны с точки зрения описания объектов и их использования. Кроме того, в файле хранятся метаданные не только данных, но и самого исполняемого файла, описывающие версию сборки и содержащие ссылки на подключаемые внешние сборки. В последнем утверждении кроется одна интересная мысль — сборки могут ссылаться на другие сборки, что позволяет нам создавать многомодульные сборки (проще говоря, программы, состоящие из нескольких файлов).

Что-то я вдруг стал использовать много заумных слов, хотя и постоянно стараюсь давать простые пояснения... Вот, например, *метаданные* — это просто описание чего-либо. Таким описанием может быть структурированный текст, в котором указано, что находится в исполняемом файле, какой он версии.

Чаще всего код делят по сборкам по принципу логической завершенности. Допустим, что наша программа реализует работу автомобиля. Все, что касается работы двигателя, можно поместить в отдельный модуль, а все, что касается трансмиссии, — в другой. Помимо этого, каждый модуль будет разбивать составляющие двигателя и трансмиссии на более мелкие составляющие с помощью классов. Код, разбитый на модули, проще сопровождать, обновлять, тестировать и загружать на устройство пользователя. При этом пользователю нет необходимости загружать все приложение — ему можно предоставить возможность обновления через Интернет, и программа будет сама скачивать только те модули, которые были обновлены.

Теперь еще на секунду хочу вернуться к JIT-компиляции и сказать об одном сильном преимуществе этого метода. Когда пользователь запускает программу, то она компилируется так, чтобы максимально эффективно выполняться на «железе» и ОС компьютера, где сборка была запущена на выполнение. Для персональных компьютеров существует множество различных процессоров, и, компилируя программу в машинный код, чтобы он выполнялся на всех компьютерах, программисты очень часто — чтобы не сужать рынок продаж — не учитывают современные инструкции процессоров. А вот JIT-компилятор может учесть эти инструкции и оптимально скомпилировать программу под конкретный процессор, стоящий в конкретном устройстве. Кроме того, разные ОС обладают разными функциями, и JIT-компилятор также может использовать возможности ОС максимально эффективно.

Делает ли такую оптимизацию среда выполнения .NET — я не знаю. Из информации, доступной в Интернете, все ведет к тому, что некоторая оптимизация имеет место. Но достоверно мне это не известно. Впрочем, утверждают, что магазин приложений Windows может перекомпилировать приложения перед тем, как отправлять их пользователю. И когда вы через этот магазин скачиваете что-либо, то получаете специально скомпилированную под ваше устройство версию.

Из-за компиляции кода во время запуска первый запуск на компьютере может оказаться весьма долгим. Но когда платформа сохранит результат компиляции в кэше, последующий запуск будет выполняться намного быстрее.

1.2. Обзор среды разработки Visual Studio .NET

Познакомившись с основами платформы .NET, перейдем непосредственно к практике и бросим беглый взгляд на новую среду разработки — посмотрим, что она предоставляет нам, как программистам.

Мы познакомимся здесь с Visual Studio 2017 Community Edition, но работа с ней не отличается от работы с любой другой версией Visual Studio. В предыдущей версии этой книги рассматривалась версия Visual Studio 2008 Express Edition, которая тогда была бесплатной, однако за 9 лет развития принципы работы с проектами почти не изменились, поэтому при обновлении этой главы понадобилось не так много усилий.

1.2.1. Работа с проектами и решениями

В Visual Studio любая программа заключается в *проект*. Проект — это как каталог для файлов. Он обладает определенными свойствами (например, платформой и языком, для которых создан проект) и может содержать файлы с исходным кодом программы, который необходимо скомпилировать в исполняемый файл. Проекты могут объединяться в *решения* (Solution). Более подробную информацию о решениях и проектах можно найти в файле Documents\Visual Studio 2008.docx из сопровождающего книгу электронного архива (см. *приложение*).

Для создания нового проекта выберите в меню команду **File | New | Project** (Файл | Новый | Проект). Перед вами откроется окно **New Project** (Новый проект), в левой части которого расположено *дерево проектов* (рис. 1.1), — здесь можно выбрать раздел с языком программирования для будущего проекта.

Среда разработки Visual Studio может работать и компилировать проекты на нескольких языках: Visual C++, Visual C#, Visual Basic и т. д. Как можно видеть на рис. 1.1, на первом уровне дерева проектов находятся языки программирования, которые поддерживаются в текущей версии Visual Studio и установлены на компьютере.

В списке шаблонов в центральной области окна создания нового проекта выводятся типы проектов выбранного раздела. У меня сейчас раскрыт раздел языка **Visual C#**,

на втором уровне которого показаны ярлыки шаблонов для разработки проектов на этом языке.

В нижней части окна создания нового проекта имеются три поля ввода:

- **Name** — здесь вы указываете имя будущего проекта;
- **Location** — расположение каталога проекта.
- **Solution name** — имя решения.

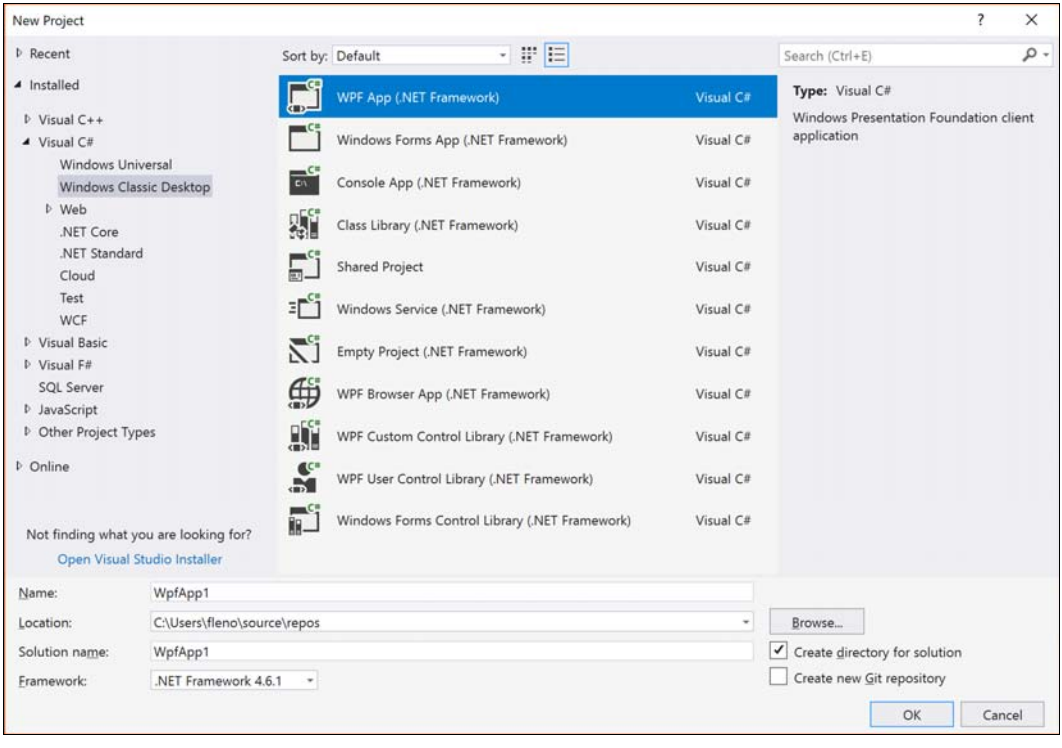


Рис. 1.1. Окно создания нового проекта

Давайте сейчас остановимся на секунду на третьем пункте — с именем решения. Если вы работаете над большим проектом, то он может состоять из нескольких файлов: основного исполняемого файла и динамических библиотек DLL, в которых хранится код, которым приложения могут делиться даже с другими приложениями.

Объединяя библиотеки и исполняемые файлы в одно большое решение, мы упрощаем компиляцию — создание финальных файлов. Мы просто компилируем все решение, а среда разработки проверит, какие изменения произошли, и в зависимости от этого перекомпилирует те проекты (библиотеки или исполняемые файлы), которые изменились.

При задании имени и расположения проекта будьте внимательны. Допустим, что в качестве пути (в поле **Location**) вы выбрали `C:\Projects`, а имя (в поле **Name**) назначили `MyProject`. Созданный проект тогда будет находиться в каталоге

C:\Projects\MyProject. То есть среда разработки создаст каталог с именем проекта, указанным в поле **Name**, в каталоге, указанном в поле **Location**.

Файл проекта, в котором находятся все настройки и описания входящих в проект файлов, имеет расширение csproj. Этот файл создается в формате XML, и его легко просмотреть в любом текстовом редакторе. В текстовом редакторе вы даже можете его редактировать, но, в большинстве случаев, проще использовать для этого специализированные диалоговые окна, которые предоставляет среда разработки.

Настройки, которые находятся в файле проекта csproj, легко изменять с помощью окна **Properties** (Свойства). Это окно можно вызвать, выбрав команду меню **Project | *Xxxx* properties**, где ***Xxxx*** — имя вашего проекта. Если вы назвали проект `HelloWorld`, то имя файла проекта будет HelloWorld.csproj, и команда меню для вызова окна редактирования свойств должна выглядеть так: **Project | HelloWorld properties**.

В нижней части окна создания нового проекта могут располагаться два переключателя или выпадающий список с выбором (если у вас уже открыт какой-то проект в среде разработки):

- **Add to Solution** — добавить в текущее решение;
- **Create new solution** — создать новое решение. Текущее решение будет закрыто и создано новое.

Давайте создадим новый пустой C#-проект — чтобы увидеть, из чего он состоит. Выделите в дереве проектов слева раздел **Visual C#**, затем **Windows Classic Desktop** и в списке шаблонов выберите **WPF App (.NET Framework)**. Обратите внимание, что в нижней части окна появился еще один выпадающий список — **Framework** — здесь мы выбираем версию .NET Framework, под которую хотим компилировать проект. Сейчас на большинстве компьютеров стоит версия 4.6.1, поэтому вполне нормально выбрать самую последнюю версию.

Обратите также внимание, что у имени шаблона в скобках есть указание на то, что перед нами именно приложение .NET Framework. То есть это не .NET Core, а значит, скомпилированный файл будет выполняться под Windows.

В завершение создания нового проекта укажите его имя и расположение и нажмите кнопку **OK**.

Вот теперь панели окна Visual Studio, расположенные справа, заполнились информацией, и имеет смысл рассмотреть их более подробно. Но сначала посмотрим, что появилось в окне слева. Это панель **Toolbox** (Инструментальные средства). У вас она может иметь вид тонкой полоски. Наведите на нее указатель мыши, и панель выдвинется, как показано на рис. 1.2. Чтобы закрепить панель **Toolbox** на экране, щелкните на кнопке с изображением гвоздика-флажка в заголовке панели (слева от кнопки закрытия панели).

В нижней части окна слева могут находиться три панели:

- **Server Explorer** — панель позволяет просматривать соединения с базами данных и редактировать данные в таблицах прямо из среды разработки;

- **Toolbox** — на этой панели располагаются компоненты, которые вы можете устанавливать на форме;
- **Data Sources** — источники данных.

Любая из этих панелей может отсутствовать, а могут присутствовать и другие панели, потому что среда разработки Visual Studio настраиваема, и панели можно закрывать, а можно и отображать на экране, выбирая их имена в меню **View** (Вид). Давайте познакомимся с этими и другими панелями, которые стали нам доступны.

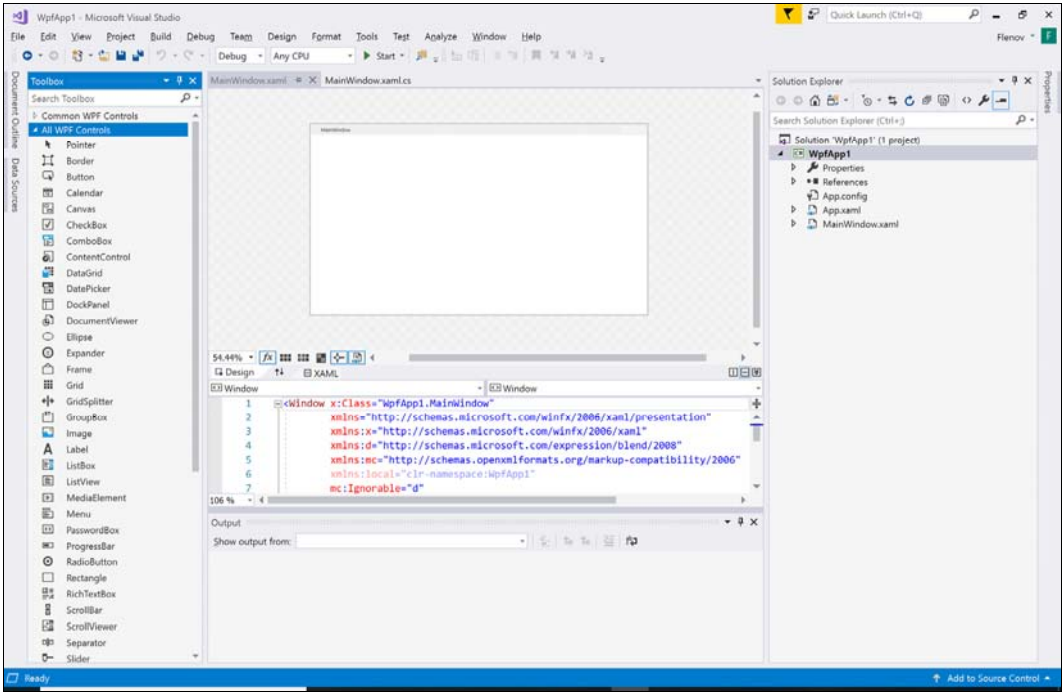


Рис. 1.2. Окно Visual Studio с открытым проектом

1.2.2. Панель *Server Explorer*

Панель **Server Explorer** (рис. 1.3) представляет нам дерево, состоящее из двух основных разделов:

- **Data Connections** — в этом разделе можно создавать и просматривать соединения с базами данных;
- **Servers** — зарегистрированные серверы. По умолчанию зарегистрирован только ваш компьютер. А в панели **Server Explorer** моего компьютера уже имеется ссылка на аккаунт Azure, которым я пользовался, и база данных. Именно для работы с базами данных наиболее удобно это окно.

Щелкните правой кнопкой мыши на разделе **Data Connections**. В открывшемся контекстном меню вы увидите два очень интересных пункта: **Add Connection** (До-

бавить соединение) и **Create New SQL Server Database** (Создать новую базу данных). Мы создавать новую базу не станем, а попробуем создать соединение с уже существующей таблицей или базой данных.

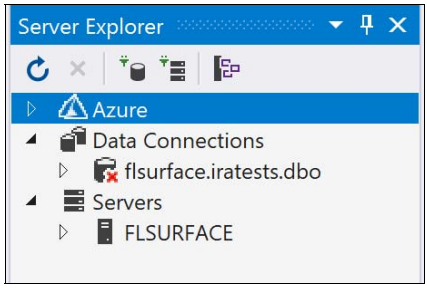


Рис. 1.3. Панель **Server Explorer**

Выберите пункт контекстного меню **Add Connection**, и откроется стандартное окно соединения с базой данных (рис. 1.4). Раскройте ветку созданного соединения, и перед вами появится содержимое этой базы данных.

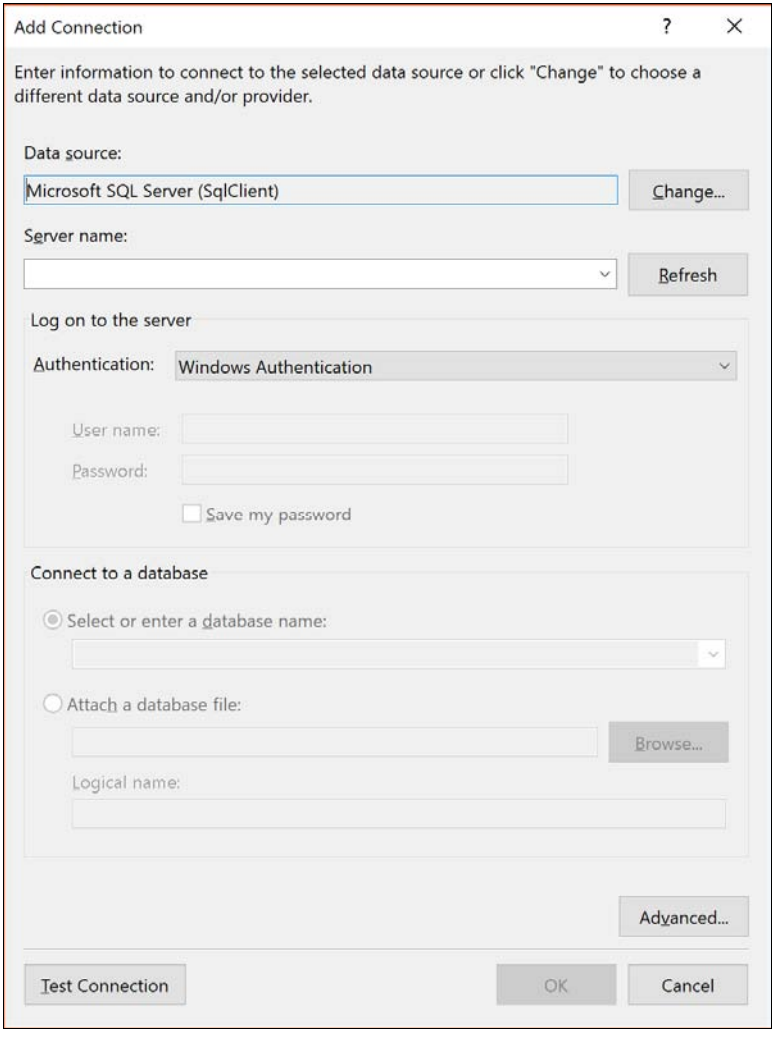


Рис. 1.4. Окно настройки соединения с базой данных

Для MS Access это будут разделы:

- **Tables** — таблицы, которые есть в базе данных;
- **Views** — представления. Я их часто называю *хранимыми запросами выборки*, потому что это запросы `SELECT`, которые хранятся в самой базе данных;
- **Stored Procedures** — хранимые процедуры.

Вы можете просматривать содержимое таблиц и хранимых запросов прямо из среды разработки. Для этого щелкните двойным щелчком на имени таблицы, и ее содержимое появится в отдельном окне рабочей области.

1.2.3. Панель *Toolbox*

Вернемся к панели **Toolbox**. Это наиболее интересная панель. В ней по разделам расположены компоненты, которые можно устанавливать на форму (рис. 1.5). Для каждого типа проекта количество и виды доступных на панели компонентов могут различаться. Различия эти зависят даже не столько от проекта, сколько от контекста. Если открыть в основном окне файл с исходным кодом, где нет визуального интерфейса, то это окно будет пустым. Мы создали простое Windows-приложение, и основные компоненты для этого типа приложения находятся в разделе **All WPF Controls**.

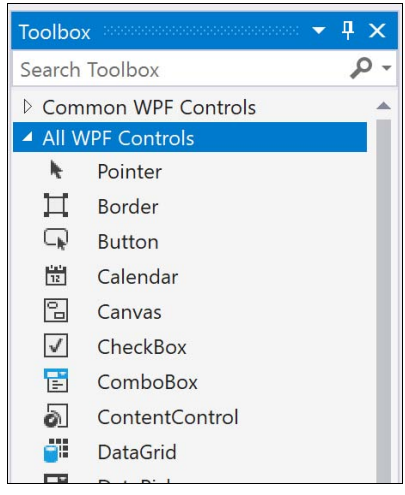


Рис. 1.5. Панель **Toolbox**

Есть несколько способов установить компонент на форму:

1. Перетащить компонент на форму, удерживая левую кнопку мыши. При этом компонент будет создан в той точке, где вы его бросили, а его размеры будут установлены по умолчанию.
2. Щелкнуть на кнопке компонента двойным щелчком. При этом компонент будет создан в левой верхней точке формы, а размеры его будут установлены по умолчанию.

3. Щелкнуть на нужном компоненте, чтобы выделить его. Щелкнуть на форме, чтобы установить компонент. При этом компонент будет создан в той точке, где вы щелкнули, а размеры его будут установлены по умолчанию.
4. Щелкнуть на нужном компоненте, чтобы выделить его. Нажать левую кнопку мыши на форме и протянуть курсор до нужных размеров на форме, чтобы установить компонент. При этом компонент будет создан в той точке, где вы щелкнули, а размеры будут установлены в соответствии с обрисованным на форме прямоугольником.

Вы можете настраивать панель **Toolbox** на свой вкус и цвет. Щелкните правой кнопкой мыши в пустом пространстве или на любом компоненте в панели **Toolbox**. Перед вами откроется контекстное меню (рис. 1.6). Рассмотрим его пункты:

- **Cut** — вырезать компонент в буфер обмена;
- **Copy** — скопировать компонент в буфер обмена;
- **Paste** — вставить компонент из буфера обмена;
- **Delete** — удалить компонент;

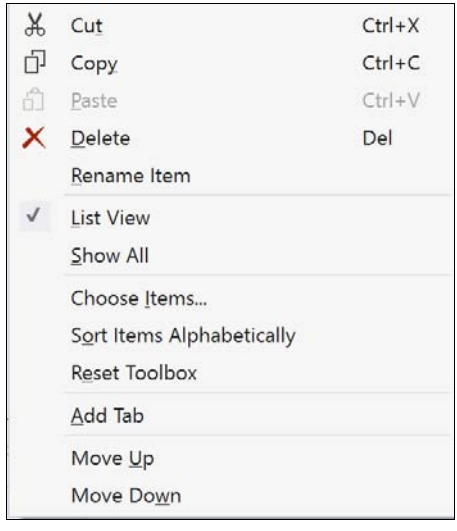


Рис. 1.6. Контекстное меню панели **Toolbox**

- **Rename Item** — переименовать компонент;
- **List View** — просмотреть компоненты в виде списка с именами. Если отключить этот пункт, то на панели **Toolbox** будут отображаться только значки;
- **Show All** — по умолчанию показываются только вкладки, содержащие элементы, которые можно устанавливать на выбранный тип формы, но с помощью этого пункта меню можно отобразить все компоненты;
- **Choose Items** — показать окно добавления/удаления компонентов;
- **Sort Items Alphabetically** — сортировать элементы по алфавиту;
- **Add Tab** — добавить вкладку;

- **Move Up** — выделить предыдущий компонент;
- **Move Down** — выделить следующий компонент.

Наиболее интересным является пункт **Choose Items**, который позволяет с помощью удобного диалогового окна **Choose Toolbox Items** создавать и удалять компоненты.

1.2.4. Панель *Solution Explorer*

Панель **Solution Explorer** (Проводник решения) по умолчанию расположена в правой верхней части главного окна среды разработки (см. рис. 1.2) и позволяет просмотреть, какие проекты и файлы входят в решение. Чтобы открыть файл для редактирования, достаточно щелкнуть на нем мышью двойным щелчком.

На рис. 1.7 показана панель **Solution Explorer** для нашего тестового проекта. Все объекты здесь представлены в виде дерева. Во главе дерева находится название решения. В большинстве случаев оно совпадает с именем проекта. Для переименования решения щелкните на его имени в дереве правой кнопкой мыши и в контекстном меню выберите пункт **Rename** (Переименовать).

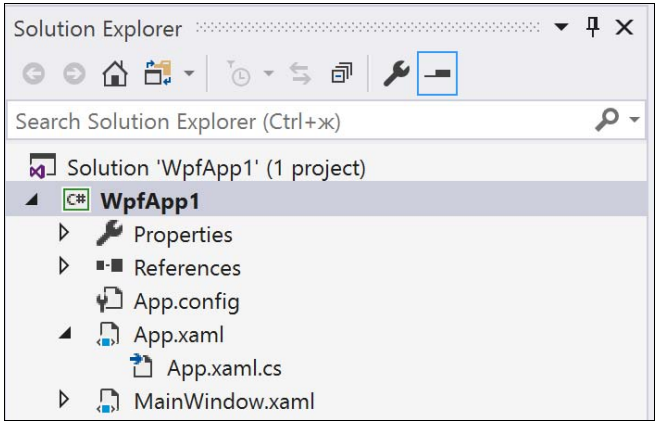


Рис. 1.7. Панель **Solution Explorer**

Для добавления проекта в решение щелкните на имени решения в дереве правой кнопкой мыши и в контекстном меню выберите **Add** (Добавить). В этом меню также можно увидеть следующие пункты:

- **New Project** — создать новый проект. Перед вами откроется окно создания нового проекта, который будет автоматически добавлен в существующее решение;
- **Existing Project** — добавить существующий проект. Этот пункт удобен, если у вас уже есть проект и вы хотите добавить его в это решение;
- **Existing Project From Web** — добавить существующий проект с Web-сайта. Этот пункт удобен для проектов ASP (Active Server Page) и ASP.NET;
- **Add New Item** — добавить новый элемент в решение, а не в отдельный проект. Не так уж и много типов файлов, которые можно добавить прямо в решение.

В диалоговом окне выбора файла вы увидите в основном различные типы текстовых файлов и картинки (значки и растровые изображения);

- **Add Existing Item** — добавить существующий элемент в решение, а не в отдельный проект.

Чтобы создать исполняемые файлы для всех проектов, входящих в решение, щелкните правой кнопкой мыши на имени решения и в контекстном меню выберите **Build Solution** (Собрать решение) — компилироваться будут только измененные файлы, или **Rebuild Solution** (Полностью собрать проект) — компилироваться будут все файлы.

В контекстном меню имени проекта можно увидеть уже знакомые пункты **Build** (Собрать проект), **Rebuild** (Полностью собрать проект), **Add** (Добавить в проект новый или существующий файл), **Rename** (Переименовать проект) и **Remove** (Удалить проект из решения). Все эти команды будут выполняться в отношении выбранного проекта.

Если вы хотите сразу запустить проект на выполнение, то нажмите клавишу <F5> или выберите в главном меню команду **Debug | Start** (Отладка | Запуск). В ответ на это проект будет запущен на выполнение с возможностью отладки, т. е. вы сможете устанавливать точки останова и выполнять код программы построчно. Если отладка не нужна, то нажимаем сочетание клавиш <Ctrl>+<F5> или выбираем команду меню **Debug | Start Without Debugging** (Отладка | Запустить без отладки).

Даже самые простые проекты состоят из множества файлов, поэтому лучше проекты держать каждый в отдельном каталоге. Не пытайтесь объединять несколько проектов в один каталог — из-за этого могут возникнуть проблемы с поддержкой.

Давайте посмотрим, из чего состоит типичный проект. Основной файл, который необходимо открывать в Visual Studio, — файл с расширением csproj. В качестве имени файла будет выступать имя проекта, который вы создали. Если открыть файл проекта csproj с помощью Блокнота, то вы увидите, что в файле реально XML-данные.

В файле проекта с помощью XML-тегов описывается, из чего состоит проект. В тегах описываются и файлы, которые входят в проект, — ведь может быть не один файл, а множество. Когда с помощью Visual Studio командой меню **File | Open | Project** (Файл | Открыть | Проект) вы открываете этот файл, то по служебной информации среда разработки загружает все необходимое и устанавливает соответствующие параметры.

Как уже отмечалось ранее, несколько проектов могут объединяться в одно решение (Solution). Файл решения по умолчанию имеет имя такое же, как у первого проекта, который вы создали, но его можно изменить. Расширение у имени этого файла: sln.

Очень часто программы состоят из нескольких подзадач (проектов), и удобнее управлять ими из одного места. Например, программа может состоять из одного исполняемого файла и двух библиотек. Можно создать решение, в котором все три проекта будут объединены в одно решение. Решение — это что-то вроде виртуаль-

ного каталога для ваших проектов. На рис. 1.8 показан пример решения, состоящего из нескольких проектов. Это мой реальный проект программы CyD Network Utilities, которую можно найти на сайте **www.cydsoft.com**.

Код программы на C# хранится в файлах с расширением cs. Это простой текст без определенного формата, но не стоит редактировать его в Блокноте, потому что среда разработки Visual Studio для редактирования кода намного удобнее.

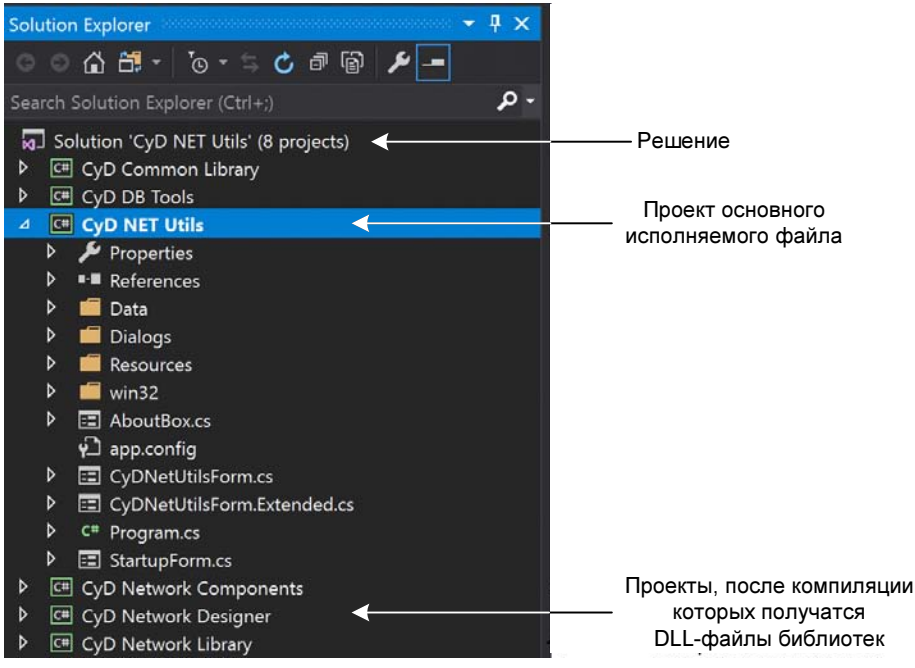


Рис. 1.8. Панель **Solution Explorer** с несколькими открытыми проектами

Среда разработки после компиляции также создает в каталоге проекта два каталога: bin и obj. В каталоге obj сохраняются временные файлы, которые используются для компиляции, а в каталоге bin — результат компиляции. По умолчанию есть два режима компиляции: Debug и Release. При первом методе сборки в исполняемый файл может добавляться дополнительная информация, необходимая для отладки. Такие файлы содержат много лишней информации, особенно если проект создан на C++, и их используют только для тестирования и отладки. Файлы, созданные этим методом сборки, находятся в каталоге bin\Debug. Не поставляйте эти файлы заказчикам!

Release — это режим чистой компиляции, когда в исполняемом файле нет ничего лишнего, и такие файлы поставляют заказчику или включают в установочные пакеты. Файлы, созданные этим методом сборки, находятся в каталоге bin\Release вашего проекта.

На рис. 1.9 показан выпадающий список, с помощью которого можно менять режим компиляции.

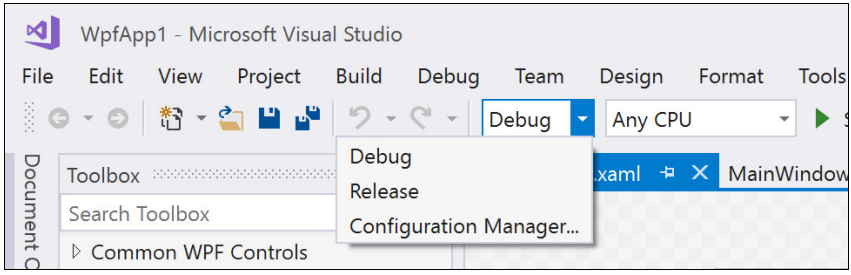


Рис. 1.9. Смена режима компиляции: **Debug** или **Release**

Даже если компилировать в режиме Release, Visual Studio зачем-то помещает в каталог с результатом файлы с расширением pdb. Такие файлы создаются для каждого результирующего файла (библиотеки или исполняемого файла), причем и результирующий, и PDB-файл будут иметь одно имя, но разные расширения.

В PDB-файлах содержится служебная информация, упрощающая отладку, — она откроет пользователю слишком много лишней информации об устройстве кода. Если произойдет ошибка работы программы, и рядом с исполняемым файлом будет находиться его PDB-файл, то пользователю покажут даже в какой строчке кода какого файла произошел сбой, чего пользователю совершенно не нужно знать. Если же PDB-файла пользователю не давать, то в ошибке будет показана лишь поверхностная информация о сбое. Так что не распространяйте эти файлы вместе со своим приложением без особой надобности.

Впрочем, если вы разрабатываете Web-сайт, то подобный файл можно поместить на Web-сервер, если вы правильно обрабатываете ошибки. В этом случае в журнал ошибок будет записана подробная информация о каждом сбое, которая упростит поиск и исправление ошибки.

1.2.5. Панель *Class View*

Панель **Class View** (Просмотр класса) позволяет просмотреть в виде дерева содержимое файла с исходным кодом. Как уже отмечалось ранее, если эта панель отсутствует на экране, вы можете открыть ее, выбрав в главном меню команду **View | Class View**. Эта панель наиболее эффективна, если в одном файле объявлено несколько классов, — тогда вы можете увидеть их иерархию. Щелкнув на элементе двойным щелчком, можно перейти на его описание в файле с исходным кодом.

Если вы не имели опыта программирования, то слова «класс», «метод», «свойство» и пр. будут для вас ясны как темный лес. Поэтому лучше оставим эту тему до лучших времен, которые наступят очень скоро. Панель **Class View** достаточно простая, и когда вы познакомитесь со всеми понятиями ООП, то все встанет на свои места, и вы сами разберетесь, что здесь для чего.

1.2.6. Работа с файлами

Чтобы начать редактирование файла, необходимо щелкнуть на нем двойным щелчком в панели **Solution Explorer**. В ответ на это действие в рабочей области окна появится вкладка с редактором соответствующего файла. Содержимое и вид окна сильно зависят от типа файла, который вы открыли.

Мы, в основном, будем работать с языком C#. В качестве расширения имени файлов для хранения исходного кода этот язык использует комбинацию символов cs (C Sharp).

Файлы с визуальным интерфейсом (с расширением xaml) по умолчанию открываются в режиме визуального редактора. Для того чтобы увидеть исходный код формы, нажмите клавишу <F7> — в рабочей области откроется новая вкладка. Таким образом, у вас будут две вкладки: визуальное представление формы и исходный код формы. Чтобы быстро перейти из вкладки с исходным кодом в визуальный редактор, нажмите комбинацию клавиш <Shift>+<F7>.

Если необходимо сразу же открыть файл в режиме редактирования кода, то в панели **Solution Explorer** щелкните на файле правой кнопкой мыши и выберите в контекстном меню команду **View Code** (Посмотреть код).

1.3. Простейший пример .NET-приложения

Большинство руководств и книг по программированию начинается с описания простого примера, который чаще всего называют «Hello World» («Здравствуй, мир»). Если ребенок, родившись на свет, издает крик, то первая программа будущего программиста в такой ситуации говорит «Hello World». Что, будем как все, — или назовем свою первую программу: «А вот и я» или «Не ждали»? Простите мне мой канадский юмор, который будет регулярно появляться на страницах книги.

Честно сказать, название не имеет особого значения, главное — показать простоту создания проекта и при этом рассмотреть основы. Начинать с чего-то более сложного и интересного нет смысла, потому что программирование — занятие не из простых, а необходимо рассказать очень многое.

В этой главе мы создадим простое приложение на C# для платформы .NET Core. С помощью этого примера мы разберемся в основах новой платформы, а потом уже начнем усложнять задачу.

1.3.1. Проект на языке C#

Давайте создадим простой проект, который проиллюстрирует работу C#-приложения. Выберите в главном меню команду **File | New | Project** (Файл | Создать | Проект). Перед вами отобразится окно создания проекта (см. рис. 1.1). Выберите в дереве левой области окна раздел **Visual C#** и затем раздел **.NET Core**, а в правом списке — **Console Application (.NET Core)** (Консольное приложение). Введите имя проекта: `EasyCSharp`. Нажмите кнопку **OK** — среда разработки создаст вам новый проект.

Так как мы создали .NET Core версию консольного приложения, то наше приложение должно будет работать на Linux и macOS.

В разделе **Visual C#** окна создания проекта имеется также шаблон **Windows Classic Desktop**. Если выбрать этот шаблон, то все описываемое далее тоже будет прекрасно работать, но только результирующий файл сможет запускаться лишь под Windows. Если вы знаете, что ваш код будет запускаться только под Windows, или вам нужно в проекте использовать какие-либо специфичные для этой платформы возможности, то можете выбрать этот пункт.

Теперь в панели **Solution Explorer** удалите все подчиненные элементы для проекта **EasyCSharp** — у вас должен остаться только файл **Class1.cs**. Щелкните на нем двойным щелчком, и в рабочей области окна откроется редактор кода файла. Удалите из него все, что там есть, а наберите следующее:

```
using System;

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

Примечание

Исходный код примера к этому разделу можно найти в папке *Source\Chapter1\EasyCSharp* сопровождающего книгу электронного архива (см. *приложение*).

1.3.2. Компиляция и запуск проекта на языке C#

Сначала скомпилируем проект и посмотрим на результат работы. Для компиляции необходимо выбрать в меню команду **Build | Build Solution** (Построить | Построить решение) или просто нажать комбинацию клавиш <Ctrl>+<Shift>+<B>. В панели **Output** (Вывод) отобразится информация о компиляции. Если этой панели у вас пока нет, то она появится после выбора указанной команды меню, или ее можно открыть, выбрав команду меню **View | Output**. Посмотрим на результат в панели вывода:

```
------------------- Завершено -------------------
Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped
Построено: 1 удачно, 0 с ошибками, 0 не требующих обновления, 0 пропущено
```

Я привел перевод только последней строки, потому что в ней пока кроется самое интересное. Здесь написано, что один проект скомпилирован удачно, и ошибок нет. Еще бы — ведь в нашем проекте и кода почти нет.

Результирующий файл можно найти в каталоге вашего проекта — во вложенном в него каталоге bin*ТекущаяКонфигурация*\netcoreapp*X.X* (*ТекущаяКонфигурация* здесь — это каталог Release или Debug), а *X.X* — это версия .NET Core. Если создавать приложение .NET Framework, то последней части пути не будет.

Выделите в панели **Solution Explorer** корневой элемент (решение), который имеет имя, как и у созданного проекта. В панели **Properties** должны появиться свойства активного решения. В свойстве **Active Config** (Текущая конфигурация) есть выпадающий список, который состоит из двух элементов:

- **Debug** — если выбран этот режим компиляции, то в созданный исполняемый файл помимо самого кода программы будет помещена еще и отладочная информация. Хотя на самом деле основная отладочная информация будет находиться в отдельном файле, и я точно не знаю, произойдут ли какие-нибудь изменения в основном файле. Эта информация необходима для отладки приложения. Такой тип компиляции лучше всего использовать на этапе разработки;
- **Release** — в результирующий файл попадет только байт-код без отладочной информации. Именно эти сборки нужно поставлять вашим клиентам, чтобы они работали с программой.

Для смены режима компиляции можно использовать и выпадающий список **Solution Configurations** (Конфигурации решения) на панели инструментов.

Получается, что исполняемый файл может находиться как в каталоге bin\Debug, так и в каталоге bin\Release, — это зависит от типа компиляции.

Обратите внимание, что в свойствах решения есть еще и свойство **Startup project** (Исполняемый проект). Если у вас в решение входят несколько проектов, то в этом списке можно выбрать тот проект, который будет запускаться из среды разработки. Для запуска проекта выберите команду меню **Debug | Start** (Отладка | Выполнить).

Запускать наше приложение из среды разработки нет смысла, потому что окно консоли появится только на мгновение, и вы не успеете увидеть результат, но попробуйте все же нажать клавишу <F5>.

Попробуем модифицировать код следующим образом:

```
using System;

class EasyCSharp
{
  public static void Main()
  {
    Console.WriteLine("Hello World!!!");
    Console.ReadLine();
  }
}
```

В этом примере добавлена строка `Console.ReadLine()`, которая заставляет программу дождаться, когда пользователь нажмет клавишу <Enter>. Если теперь снова скомпилировать проект и запустить его на выполнение, то на фоне черного экрана

командной строки вы сможете увидеть заветную надпись, которую видят большинство начинающих программистов: **Hello World!!!** (кавычки, окружающие эту надпись в коде, на экран выведены не будут). Хоть я и обещал другую, более оригинальную надпись, не стоит все же выделяться из общей массы, так что давайте посмотрим именно на эти великие слова.

Итак, у нас получилось первое приложение, и вы можете считать, что первый шаг на долгом пути к программированию мы уже сделали. Шаги будут постепенными, и сейчас абсолютно не нужно понимать, что происходит в этом примере, — нам необходимо знать только две строки:

```
Console.WriteLine("Hello World!!!");
Console.ReadLine();
```

Первая строка выводит в консоль текст, который указан в скобках. Кстати, текст должен быть размещен в двойных кавычках, если это именно *текст*, а не *переменная* (о переменных читайте в *разд. 2.2* и *2.4*). О строках мы тоже позднее поговорим отдельно, но я решил все же сделать это небольшое уточнение уже сейчас.

Вторая строка запрашивает у пользователя строку символов. Концом строки считается символ возврата каретки, который невидим, а чтобы ввести его с клавиатуры, мы нажимаем клавишу <Enter>. В любом текстовом редакторе, чтобы перейти на новую строку (завершить текущую), мы нажимаем эту клавишу, и точно так же здесь.

Когда вы нажмете <Enter>, программа продолжит выполнение. Но у нас же больше ничего в коде нет, только символы фигурных скобок. А раз ничего нет, значит, программа должна завершить работу. Вот и вся логика этого примера, и ее нам будет достаточно на следующую главу, в процессе чтения которой мы станем рассматривать скучные, нудные, но очень необходимые примеры.

1.4. Компиляция приложений

В этом разделе мы чуть подробнее поговорим о компиляции C#-приложений. Для того чтобы создать сборку (мы уже создавали исполняемый файл для великого приложения «Hello World»), необходимо нажать комбинацию клавиш <Ctrl>+ +<Shift>++<B>, или клавишу <F6>, или выбрать в меню команду **Build | Build Solution**. Компиляция C#-кода в IL происходит достаточно быстро, если сравнивать этот процесс с компиляцией классических C++-приложений.

Но Visual Studio — это всего лишь удобная оболочка, в которой писать код — одно наслаждение. На самом же деле, код можно писать в любой другой среде разработки или даже в Блокноте, а для компиляции использовать утилиту командной строки.

1.4.1. Компиляция в .NET Framework

Для компиляции под платформу .NET Framework используется программа компилятора csc.exe (от англ. C-Sharp Compiler), которую можно найти в каталоге C:\Windows\Microsoft.NET\Framework\v*X.X*, где *X.X* — версия .NET Framework. На моем

компьютере этот компилятор находится в папке C:\Windows\Microsoft.NET\Framework\v3.5\csc.exe (у вас вложенная папка v3.5 может иметь другое имя, которое зависит от версии .NET).

> ***Примечание***
>
> Раньше компилятор C# находился в папке *C:\Program Files\Microsoft.NET\SDK\vX.X\Bin*. Сейчас там можно найти множество утилит .NET как командной строки, так и с визуальным интерфейсом.

Чтобы проще было работать с утилитой командной строки, путь к ней лучше добавить в переменные окружения. Для этого щелкните правой кнопкой мыши на ярлыку **Мой компьютер** (My Computer) на рабочем столе и выберите **Свойства** (Properties). В Windows XP появится окно свойств, в котором нужно перейти на вкладку **Дополнительно** (Advanced), а в Windows 7 или Windows 8/10 откроется окно, похожее на окно панели управления, в котором нужно выбрать пункт **Дополнительные параметры системы** (Advanced system settings). Далее нажмите кнопку **Переменные среды** (Environment Variables) и в системный параметр **Path** добавьте через точку с запятой путь к каталогу, где находится компилятор csc, который вы будете использовать.

Теперь попробуем скомпилировать пример «Hello World», который мы написали ранее, а для этого в командной строке нужно выполнить следующую команду:

```
csc.exe /target:exe test.cs
```

В нашем случае `test.cs` — это файл с исходным кодом. Я просто его переименовал, чтобы не было никаких `Class1.cs`. Обратите внимание, что имя файла компилятора указано полностью (с расширением), и именно так и должно быть, иначе он не запустится, если только путь к этому файлу в системном окружении вы не указали самым первым. Дело в том, что самым первым в системном окружении стоит путь к каталогу C:\Windows, а в этом каталоге уже есть каталог CSC, и именно его будет пытаться открыть система, если не указано расширение файла.

После имени файла идет ключ `/target` — он указывает на тип файла, который вы хотите получить в результате сборки. Нас интересует исполняемый файл, поэтому после ключа и двоеточия надо указать `exe`. Далее идет имя файла. Я не указываю здесь полный путь, потому что запускаю команду в том же каталоге, где находится файл test.cs, иначе пришлось бы указывать полный путь к файлу.

В результате компиляции мы получаем исполняемый файл test.exe. Запустите его и убедитесь, что он работает корректно. Обратите внимание на имя исполняемого файла. Если при компиляции из среды разработки оно соответствовало имени проекта, то здесь — имени файла. Это потому, что в командной строке мы компилируем не проект, а файл test.cs.

Конечно же, для больших проектов использовать утилиту командной строки проблематично и неудобно, поэтому и мы не станем этого делать. Но для компиляции небольшой программки из 10 строк, которая только показывает или изменяет что-то в системе, эта утилита вполне пригодна, и нет необходимости ставить тяжеловесный пакет Visual Studio. Где еще ее можно использовать? Вы можете создать

собственную среду разработки и использовать csc.exe для компиляции проектов, но на самом деле я не вижу смысла это делать.

В составе .NET Framework есть еще одна утилита — msbuild, которая умеет компилировать целые проекты из командной строки. В качестве параметра утилита принимает имя файла проекта или решения, и она уже делает все то, что умеет делать Visual Studio во время компиляции в IDE.

С недавнего времени эту утилиту включили в состав Visual Studio, что не совсем понятно. Ведь вся сила msbuild заключается в том, что с ее помощью можно компилировать даже без установки VS, а теперь получается, что мне придется все равно использовать установочный пакет среды разработки.

Я на работе разрабатываю достаточно большой сайт, а компилировать мы его должны на сервере, где нельзя установить Visual Studio. Однако, установив только .NET Framework, мы можем из командной строки собрать проект и выполнить скрипт для запуска обновленного кода на «боевых» серверах. Честно говоря, сервер, на котором мы собираем код, находится в другой стране, и к нему мы подключаемся удаленно. Так вот, используя msbuild, очень просто подключиться к серверу удаленно и все выполнить из командной строки.

1.4.2. Компиляция в .NET Core

При создании .NET Core приложения создается DLL — библиотечный файл, а не исполняемый. Для его запуска используется утилита `dotnet run`. Перейдите в каталог проекта, где расположен файл EasyCSharp.csproj и просто выполните команду:

```
dotnet run
```

Платформа `dotnet` запустит DLL-файл.

Вы можете сказать, что это неудобно, и пользователи не будут счастливы, что им нужно помнить эту команду, но точно такая же проблема есть и у Java. Такие платформы могут создать промежуточный IL-код, но кто-то должен инициировать выполнение кода.

Нельзя создать универсальный исполняемый файл, который будет запускаться в любой ОС, потому что у Linux и macOS другой формат исполняемого файла. Но платформа может создать исполняемый файл-заглушку, которая сделает все для нас, и для создания этого файла нужно опубликовать наше приложение:

```
dotnet publish --runtime win7-x64
```

Эта команда публикует приложение, которое создает заглушку для исполнения нашего приложения в Windows, начиная с версии 7, и на компьютере с 64-битным процессором.

Можно указать также следующие платформы для исполнения:

- `osx.10.11-x64` — для macOS;
- `ubuntu.16.04-x64` — для Linux.

1.5. Поставка сборок

Теперь немного поговорим о поставке скомпилированных приложений конечному пользователю. Платформу проектировали так, чтобы избежать двух проблем: сложной регистрации, присущей COM-объектам, и DLL hell[1], характерной для платформы Win32, когда более старая версия библиотеки могла заменить более новую версию и привести к краху системы. Обе эти проблемы были решены весьма успешно, и теперь мы можем забыть о них, как о пережитке прошлого.

Установка сборок на компьютер пользователя сводится к банальному копированию библиотек (DLL-файлов) и исполняемых файлов. По спецификации компании Microsoft для работы программы должно хватать простого копирования, и в большинстве случаев дело обстоит именно так. Но разработчики — вольные птицы, и они могут придумать какие-то привязки, которые все же потребуют инсталляции с первоначальной настройкой. Я не рекомендую делать что-то подобное без особой надобности, старайтесь организовать все так, чтобы процесс установки оставался максимально простым, хотя копирование может выполнять и программа установки.

Когда библиотека DLL хранится в том же каталоге, что и программа, то именно программа несет ответственность за целостность библиотеки и ее версию. Но помимо этого есть еще разделяемые библиотеки, которые устанавливаются в систему, чтобы любая программа могла использовать их ресурсы. Именно в таких случаях чаще всего возникает DLL hell. Допустим, что пользователь установил программу A, в которой используется библиотека XXX.dll версии 5.0. В данном случае XXX — это просто какое-то имя, которое не имеет ничего общего с «клубничкой», с тем же успехом мы могли написать и YYY.dll. Вернемся к библиотеке. Допустим, что теперь пользователь ставит программу B, где тоже есть библиотека XXX.dll, но версии 1.0. Очень старая версия библиотеки не совместима с версией 5.0, и теперь при запуске программы A компьютер превращается в чертенка и «идет в ад».

Как решить эту проблему? Чтобы сделать библиотеку разделяемой, вы можете поместить ее в каталог C:\Windows\System32 (по умолчанию этот каталог в системе скрыт), но от «ада» вам не уйти, и за версиями придется следить самому. Проектировщики .NET поступили гениально — вместо этого нам предложили другой, более совершенный метод разделять библиотеки .NET между приложениями — GAC[2], который располагается в каталоге C:\Windows\assembly. Чтобы зарегистрировать библиотеку в кэше, нужно, чтобы у нее была указана версия, и она была бы подписана ключом. Как это сделать — отдельная история, сейчас мы пока рассматриваем только механизм защиты.

Глобальный кэш сборок относится к .NET Framework. Пока что .NET Core вроде бы не планирует реализовывать это и будет работать только с приватными сборками.

Посмотрите на кэш сборок — каталог C:\Windows\assembly. Здесь может быть несколько файлов с одним и тем же именем, но разных версий, — например,

---

[1] DLL hell (DLL-кошмар, буквально: DLL-ад) — тупиковая ситуация, связанная с управлением динамическими библиотеками DLL в операционной системе.

[2] Global Assembly Cache — глобальный кэш сборок.

Microsoft.Build.Engine. Если запустить поиск по имени `Microsoft.Build.Engine.ni.dll`, вы найдете несколько файлов, но эти сборки будут иметь разные версии: 2.0 и 3.5.

За скопированную в кэш библиотеку с версией 2.0 система отвечает головой, и если вы попытаетесь скопировать в кэш еще одну такую же библиотеку, но с версией 3.5, то старая версия не затрется, — обе версии будут сосуществовать без проблем. Таким образом, программы под 2.0 будут прекрасно видеть свою версию библиотеки, а программы под 3.5 — свою, и вы не получите краха системы из-за некорректной версии DLL.

Тут вы можете спросить: а что если я создам свою библиотеку и дам ей имя Microsoft.Build.Engine — смогу ли я затереть библиотеку Microsoft и устроить крах для DLL-файлов? По идее, это невозможно, если только разработчики не допустили ошибки в реализации. Каждая сборка подписывается, и открытый ключ вы можете видеть в столбце **Маркер открытого ключа** на рис. 1.10. Если в кэше окажутся две библиотеки с одинаковыми именами, но с разными подписями, они будут мирно сосуществовать в кэше, и программы станут видеть ту библиотеку (сборку), которую и задумывал разработчик. Когда программист указывает, что в его проекте нужна определенная сборка из GAC (именно из GAC, а не локально), то в метаданных сохраняется имя сборки, версия и ключ, и при запуске Framework ищет именно

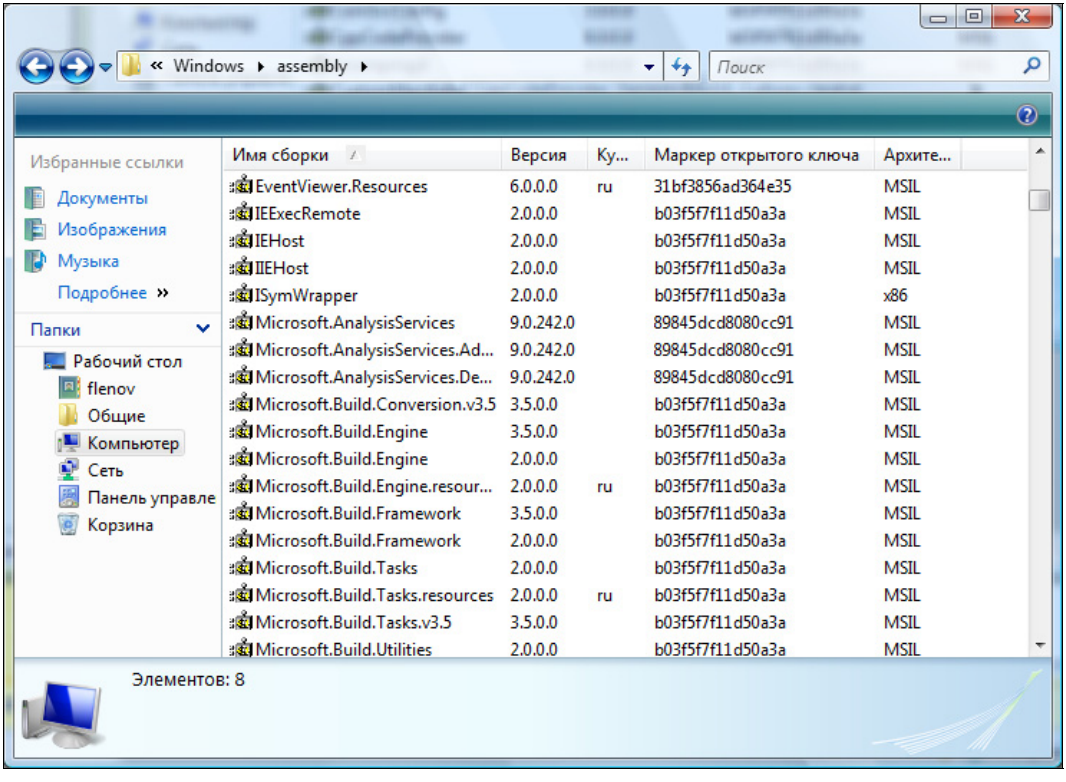


Рис. 1.10. GAC и маркер открытого ключа

эту библиотеку. Так что проблем не должно быть, и пока ни у кого не было, значит, разработчики в Microsoft все сделали хорошо.

С помощью подписей легко бороться и с возможной подменой DLL-файла. Когда вы ссылаетесь на библиотеку, то можете указать, что вам нужна для работы именно эта версия, и VS запомнит версию и открытый ключ. Если что-то не будет совпадать, то приложение работать не станет, а значит, закрываются такие потенциальные проблемы, как вирусы, которые занимаются подменой файлов.

Сборки, которые находятся в том же каталоге, что и программа, называются *приватными* (private), потому что должны использоваться только этим приложением. Сборки, зарегистрированные в GAC, называются *совместными* или *разделяемыми* (shared).

Теперь посмотрим, из чего состоит версия сборки, — ведь по ней система определяет, соответствует ли она требованиям программы или нет, и можно ли ее использовать. Итак, версия состоит из четырех чисел:

- Major — основная версия;
- Minor — подверсия сборки;
- Build — номер полной компиляции (построения) в данной версии;
- Revision — номер ревизии для текущей компиляции.

Первые два числа характеризуют основную часть версии, а остальные два являются дополнительными. При запуске исполняемого файла, если ему нужен файл из GAC, система ищет такой файл, в котором основная часть версии строго совпадает. Если найдено несколько сборок с нужной основной версией, то система выбирает из них ту, что содержит максимальный номер подверсии. Поэтому, если вы только исправляете ошибку в сборке или делаете небольшую корректировку, изменяйте значение Build и Revision. Но если вы изменяете логику или наращиваете функционал (а это может привести к несовместимости), то следует изменять основную версию или подверсию, — это зависит от количества изменений и вашего настроения. В таком случае система сможет контролировать версии и обезопасит вас от проблем с файлами DLL.

Щелкните правой кнопкой мыши на имени проекта в панели **Solution Explorer** и выберите в контекстном меню команду **Properties**. На первой вкладке **Application** (Приложение) нажмите кнопку **Assembly Information** (Информация о сборке). Перед вами откроется окно (рис. 1.11), в котором можно указать информацию о сборке, в том числе и ее версию в полях **Assembly Version**.

Уже ясно, что исполняемые файлы, написанные под платформу .NET, не могут выполняться на процессоре, потому что не содержат машинного кода. Хотя нет, содержат, но только заглушку, которая сообщит пользователю о том, что нет виртуальной машины. Чтобы запустить программу для платформы .NET, у пользователя должна быть установлена ее нужная версия. В Windows 8/10 или Windows 7 все необходимое уже есть, а для Windows XP можно скачать и установить специальный пакет отдельно. Он распространяется компанией Microsoft бесплатно и по статистике уже установлен на большинстве пользовательских компьютеров.

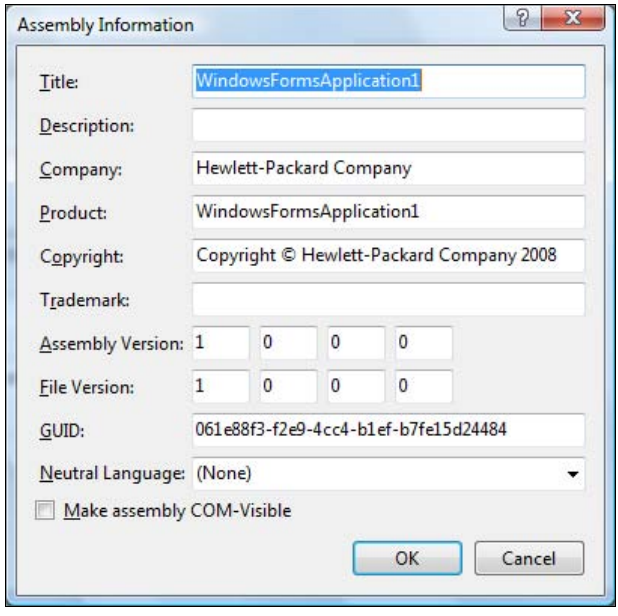


Рис. 1.11. Информация о сборке

Пакет .NET, позволяющий выполнять .NET-приложения, вы можете скачать с сайта Microsoft (**www.microsoft.com/net/Download.aspx**) — его имя dotnetfx.exe. Для .NET 2.0 этот файл занимает 23 Мбайт, а для .NET 3.0 — уже более 50 Мбайт. Ощущаете, как быстро и как сильно расширяется платформа? Но не это главное, главное — это качество, и пока оно соответствует ожиданиям большого количества разработчиков.

1.6. Формат исполняемого файла .NET

Для того чтобы вы лучше могли понимать работу .NET, давайте рассмотрим формат исполняемого файла этой платформы. Классические исполняемые файлы Win32 включали в себя:

- заголовок — описывает принадлежность исполняемого файла, основные характеристики и, самое главное, — точку, с которой начинается выполнение программы;
- код — непосредственно байт-код программы, который будет выполняться;
- данные (ресурсы) — в исполняемом файле могут храниться какие-то данные, например, строки или ресурсы.

В .NET-приложении в исполняемый файл добавили еще и метаданные.

C# является высокоуровневым языком, и он прячет от нас всю сложность машинного кода. Вы когда-нибудь просматривали Win32-программу с помощью дизассемблера или программировали на языке ассемблера? Если да, то должны представлять себе весь тот ужас, из которого состоит программа.

Приложения .NET не проще, если на них посмотреть через призму дизассемблера, только тут машинный код имеет другой вид. Чтобы увидеть, из чего состоит ваша программа, запустите утилиту ildasm.exe, которая входит в поставку .NET SDK. Если вы работаете со средой разработки Visual Studio .NET, то эту утилиту можно найти в каталоге C:\Program Files (x86)\Microsoft SDKs\Windows\ v7.0A\Bin (если вы используете 7-ю версию SDK).

Запустите эту утилиту, и перед вами откроется окно программы дизассемблера. Давайте загрузим наш проект `EasyCSharp` и посмотрим, из чего он состоит. Для этого выбираем в главном меню утилиты команду **File | Open** и в открывшемся окне — исполняемый файл. В результате на экране будет отображена структура исполняемого файла.

В нашем исходном коде был только один метод — `Main()`, но, несмотря на это, в структуре можно увидеть еще метод `.ctor`. Мы его не описывали, но он создается автоматически.

Щелкните двойным щелчком на методе `Main()` и посмотрите на его код:

```
.method public hidebysig static void  Main() cil managed
{
 .entrypoint
 // Code size       11 (0xb)
 .maxstack  1
 IL_0000:  ldstr "Hello World!!!"
 IL_0005:  call  void [mscorlib]System.Console::WriteLine(string)
 IL_000a:  ret
} // end of method EasyCSharp::Main
```

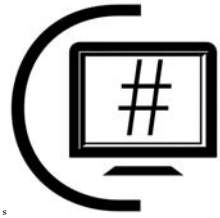
Это и есть низкоуровневый код .NET-приложения, который является аналогом ассемблера для Win32-приложения. Помимо этого, вы можете увидеть в дереве структуры исполняемого файла пункт **MANIFEST**. Это манифест (текстовая информация или метаданные) исполняемого файла, о котором мы говорили ранее.

Как уже отмечалось, компиляторы .NET создают не байт-код, который мог бы выполняться в системе, а специальный IL-код. Исполняемый файл, скомпилированный с помощью любого компилятора .NET, будет состоять из стандартного PE-заголовка, который способен выполняться на Win32-системах, IL-кода и процедуры заглушки `CorExeMain()` из библиотеки mscoree.dll. Когда вы запускаете программу, сначала идет стандартный заголовок, после чего управление передается заглушке. Только после этого начинается выполнение IL-кода из исполняемого файла.

Программы .NET Core не содержат Win32-кода. Вместо этого при публикации создается отдельный исполняемый файл, который и будет отвечать за инициацию выполнения сборки .NET Core.

Поскольку IL-код программы не является машинным и не может исполняться, то он компилируется в машинный байт-код на лету с помощью специализированного JIT-компилятора. Конечно же, за счет компиляции на лету выполнение программы замедляется, но это только при первом запуске. Полученный машинный код сохраняется в специальном буфере на вашем компьютере и используется при последующих запусках программы.

ГЛАВА 2



Основы C#

Теоретических данных у нас теперь достаточно, и мы можем перейти к практической части и продолжить знакомство с языком C# на конкретных примерах. Именно практика позволяет лучше понять, что происходит и почему. Книжные картинки и текст — это хорошо, но когда вы сами сможете запустить программу и увидеть результат, то лучшего объяснения придумать просто невозможно.

Конечно, сразу же писать серьезные примеры мы не сможем, потому что программирование — весьма сложный процесс, для погружения в который нужно обладать достаточно глубокими базовыми знаниями, так что на начальных этапах нам помогут простые решения. Но о них — чуть позже. А начнем мы эту главу с рассказа о том, как создаются пояснения к коду (комментарии), и поговорим немного о типах данных и пространстве имен, — т. е. заложим основы, которые понадобятся нам при рассмотрении последующих глав, и без которых мы не сможем написать визуальное приложение. Если вы знакомы с такими основами, то эту главу можете пропустить, но я рекомендовал бы вам читать все подряд.

2.1. Комментарии

Комментарии — это текст, который не влияет на код программы и не компилируется в выходной файл. А зачем тогда они нужны? С помощью комментариев я буду здесь вставлять в код пояснения, чтобы вам проще было с ним разбираться, поэтому мы и рассматриваем их первыми.

Некоторые используют комментарии для того, чтобы сделать код более читабельным. Хотя вопрос стиля выходит за рамки этой книги, я все же сделаю короткое замечание, что код должен быть все же читабельным и без комментариев.

Существуют два типа комментариев: однострочный и многострочный. *Однострочный* комментарий начинается с двух символов `//`. Все, что находится в этой же строке далее, — комментарий.

Следующий пример наглядно иллюстрирует сказанное:

```
// Объявление класса EasyCSharp
class EasyCSharp
{ // Начало

  // Функция Main
  public static void Main()
  {
    OutString()
  }
}  // Конец
```

Символы комментария // не обязательно должны быть в начале строки. Все, что находится слева от них, воспринимается как код, а все, что находится справа до конца строки, — это комментарий, который игнорируется во время компиляции. Я предпочитаю ставить комментарии перед строкой кода, которую комментирую, и, иногда, в конце строки кода.

Многострочные комментарии в C# заключаются в символы /* и */. Например:

```
/*
  Это многострочный
  комментарий.
*/
```

Если вы будете создавать многострочные комментарии в среде разработки Visual Studio, то она автоматически к каждой новой строке добавит в начало символ звездочки. Например:

```
/*
 * Это многострочный
 * комментарий.
*/
```

Это не является ошибкой и иногда даже помогает отличить строку комментария от строки кода. Во всяком случае, такой комментарий выглядит элегантно и удобен для восприятия.

2.2. Переменная

Понятие *переменная* является одним из ключевых в программировании. Что это такое и для чего нужно? Любая программа работает с данными (числами, строками), которые могут вводиться пользователем или жестко прописываться в коде программы. Эти данные надо где-то хранить. Где? Постоянные данные хранятся в том или ином виде на жестком диске, а временные данные — в оперативной памяти компьютера.