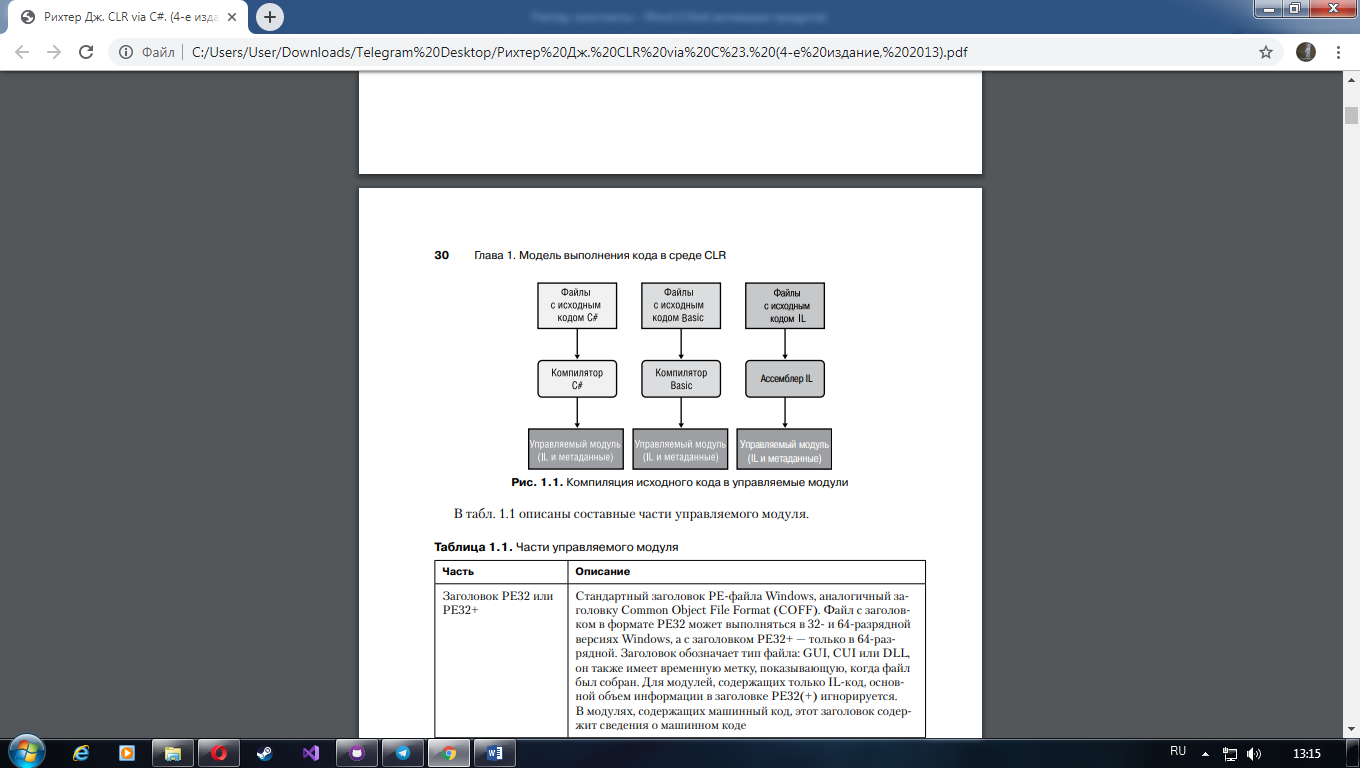
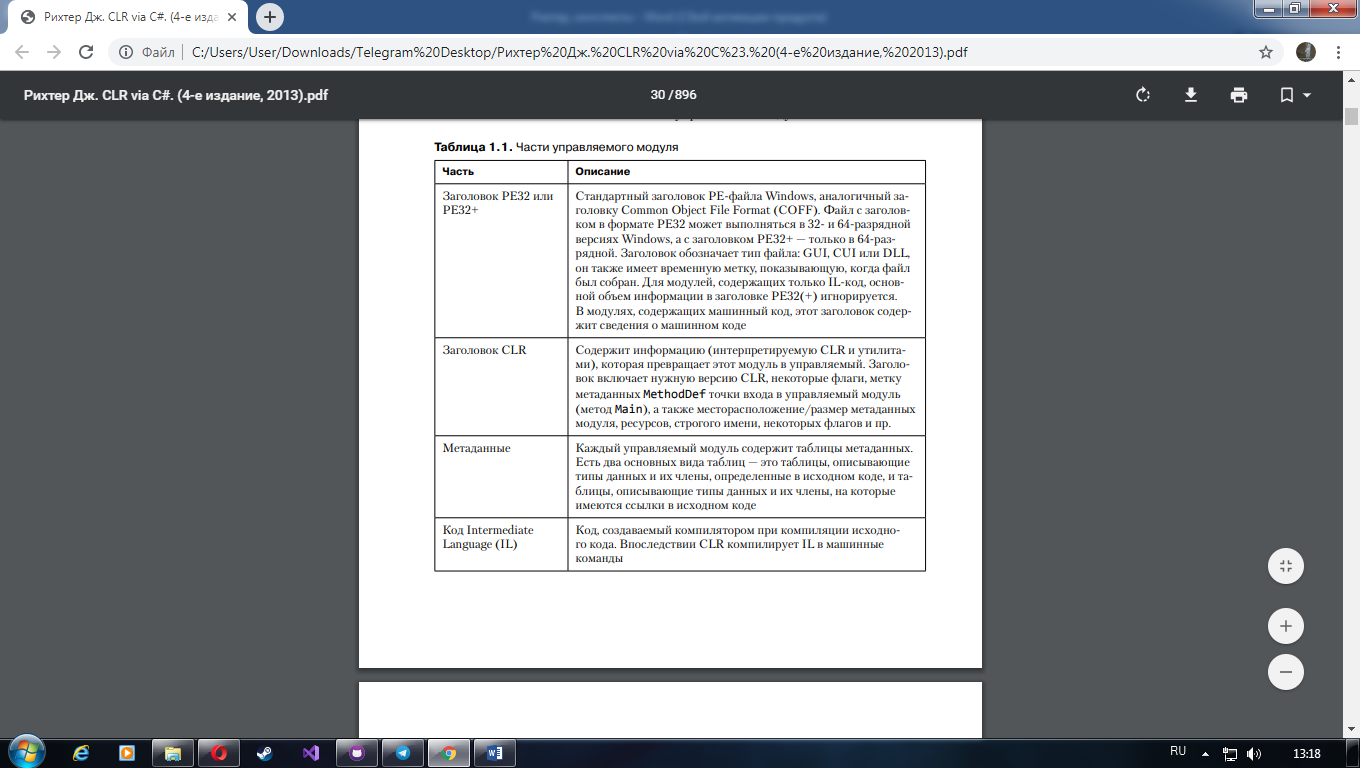
**Глава 1**

**Модель выполнения кода в среде CLR**

Название среды — общеязыковая среда выполнения (Common Language Runtime, CLR) — говорит само за себя: это среда выполнения, которая подходит для разных языков программирования.

Рисунок 1.1 иллюстрирует процесс компиляции файлов с исходным кодом. Как видно из рисунка, исходный код программы может быть написан на любом языке, поддерживающем среду выполнения CLR. Затем соответствующий компилятор проверяет синтаксис и анализирует исходный код программы. Вне зависимости от типа используемого компилятора результатом компиляции будет являться управляемый модуль (managed module) — стандартный переносимый исполняемый (portable executable, PE) файл 32-разрядной (PE32) или 64-разрядной Windows (PE32+), который требует для своего выполнения CLR.



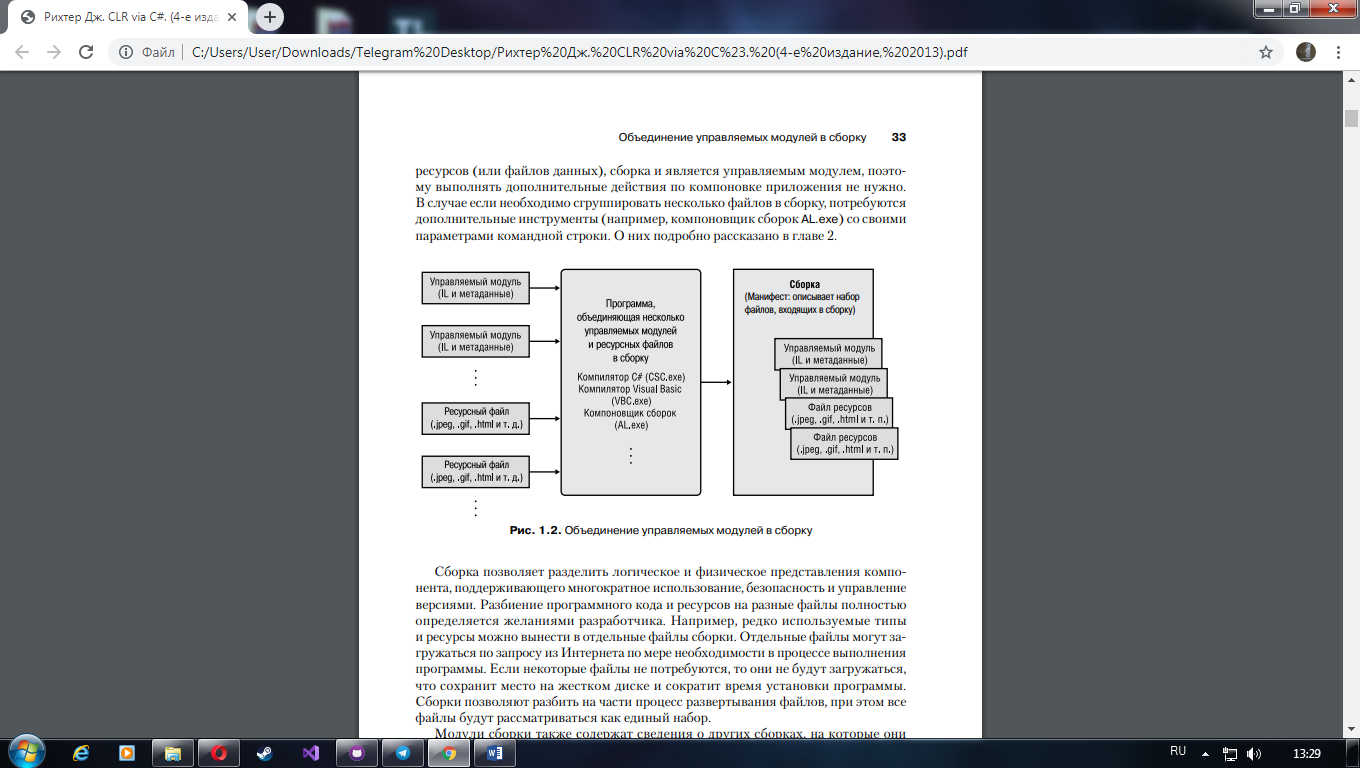


Метаданные находят много применений. Перечислим лишь некоторые из них.

1. Метаданные устраняют необходимость в заголовочных и библиотечных файлах при компиляции, так как все сведения об упоминаемых типах/членах содержатся в файле с реализующим их IL-кодом. Компиляторы могут читать метаданные прямо из управляемых модулей.
2. Среда Microsoft Visual Studio использует метаданные для облегчения написания кода. Ее функция IntelliSense анализирует метаданные и сообщает, какие методы, свойства, события и поля предпочтительны в данном случае и какие именно параметры требуются конкретным методам.
3. В процессе верификации кода CLR использует метаданные, чтобы убедиться, что код совершает только «безопасные по отношению к типам» операции. (Проверка кода обсуждается далее.)
4. Метаданные позволяют сериализовать поля объекта, а затем передать эти данные по сети на удаленный компьютер и там провести процесс десериализации, восстановив объект и его состояние на удаленном компьютере.
5. Метаданные позволяют сборщику мусора отслеживать жизненный цикл объектов. При помощи метаданных сборщик мусора может определить тип объектов и узнать, какие именно поля в них ссылаются на другие объекты.

На самом деле среда CLR работает не с модулями, а со сборками. Сборка (assembly) — это абстрактное понятие, понять смысл которого на первых порах бывает нелегко. Во-первых, сборка обеспечивает логическую группировку одного или нескольких управляемых модулей или файлов ресурсов. Во-вторых, это наименьшая единица многократного использования, безопасности и управления версиями. Сборка может состоять из одного или нескольких файлов — все зависит от выбранных средств и компиляторов. В контексте среды CLR сборкой называется то, что мы обычно называем компонентом.

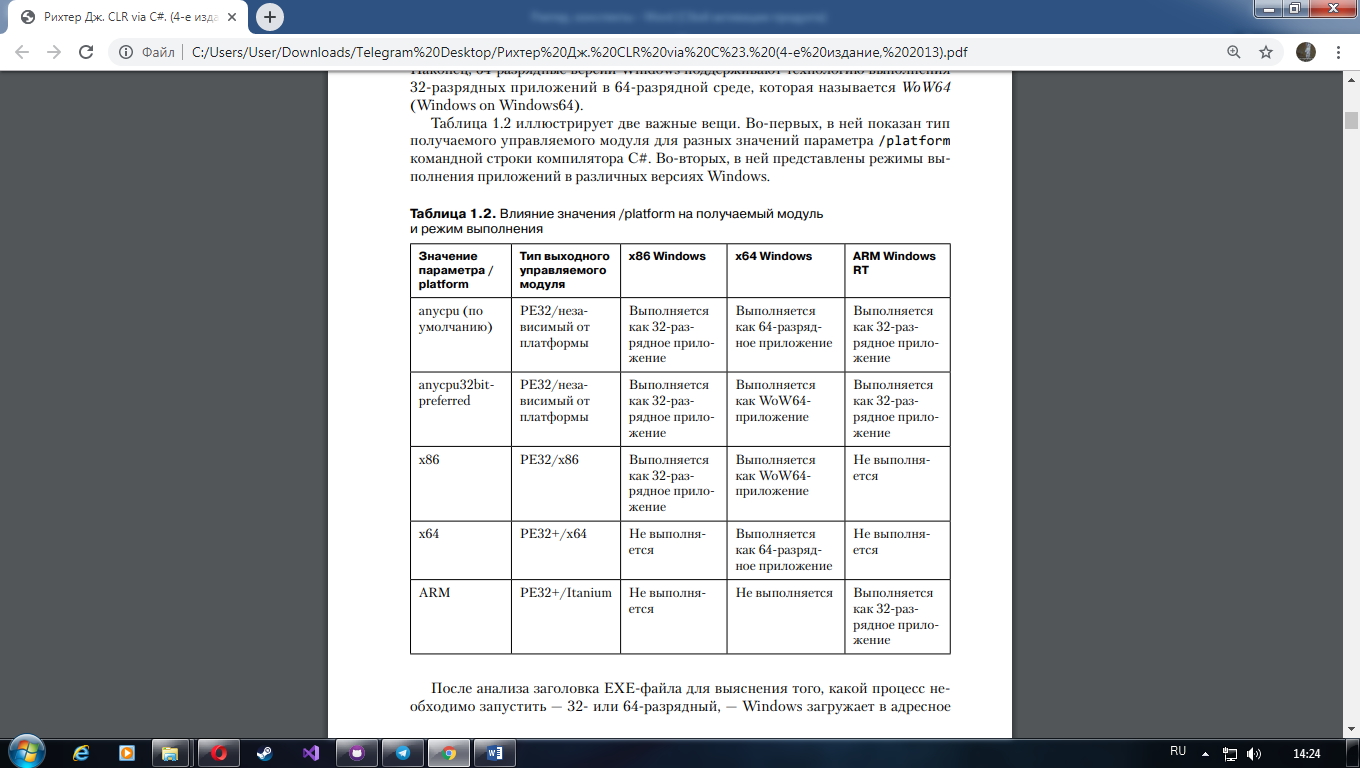
Рисунок 1.2 поможет понять суть сборки.



На этом рисунке изображены некоторые управляемые модули и файлы ресурсов (или данных), с которыми работает некоторая программа. Эта программа создает единственный файл PE32(+), который обеспечивает логическую группировку файлов. При этом в файл PE32(+) включаетсяч блок данных, называемый манифестом (manifest). Манифест представляет собой обычный набор таблиц метаданных. Эти таблицы описывают файлы, которые входят в сборку, общедоступные экспортируемые типы, реализованные в файлах сборки, а также относящиеся к сборке файлы ресурсов или данных.

**Загрузка CLR**

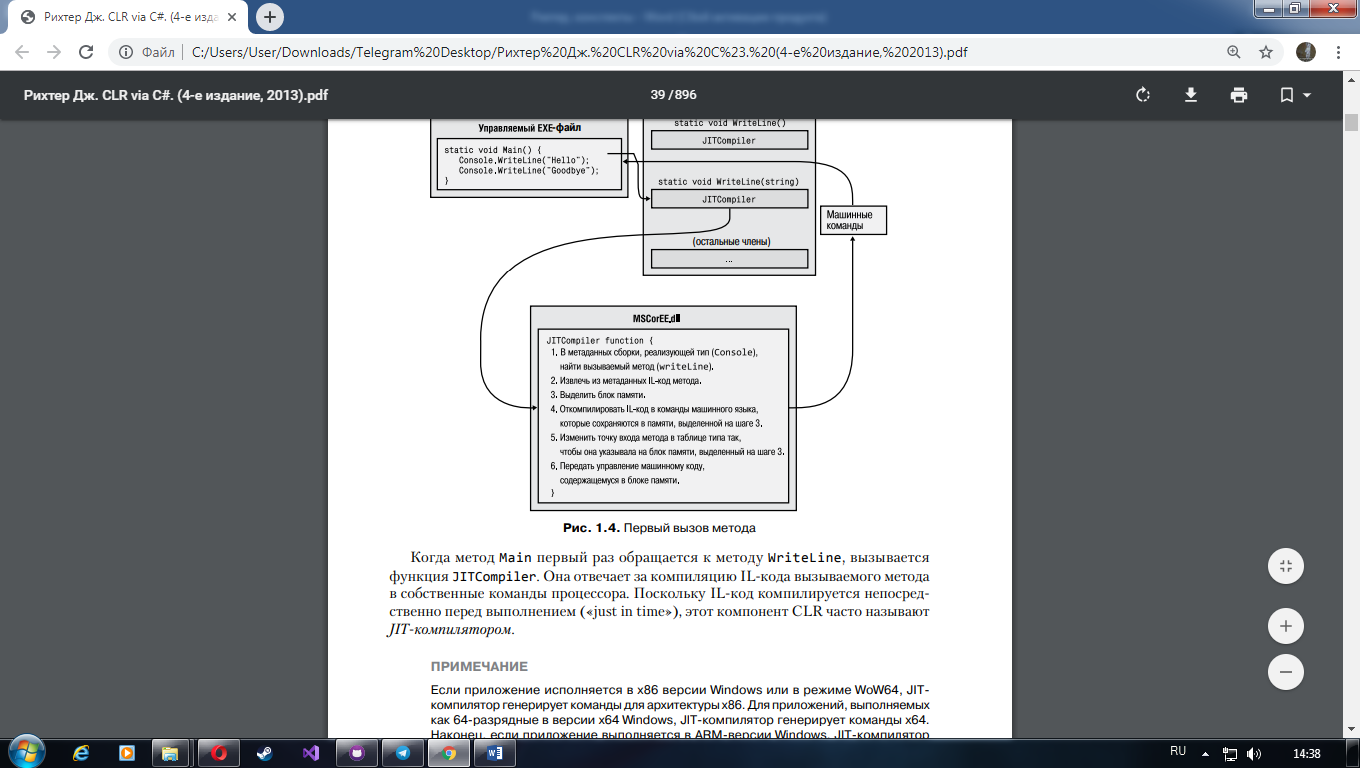
Каждая создаваемая сборка представляет собой либо исполняемое приложение, либо библиотеку DLL, содержащую набор типов для использования в исполняемом приложении. Разумеется, среда CLR отвечает за управление исполнением кода. Это значит, что на компьютере, выполняющем данное приложение, должна быть установлена платформа .NET Framework. В компании Microsoft был создан дистрибутивный пакет .NET Framework для свободного распространения, который вы можете бесплатно поставлять своим клиентам.



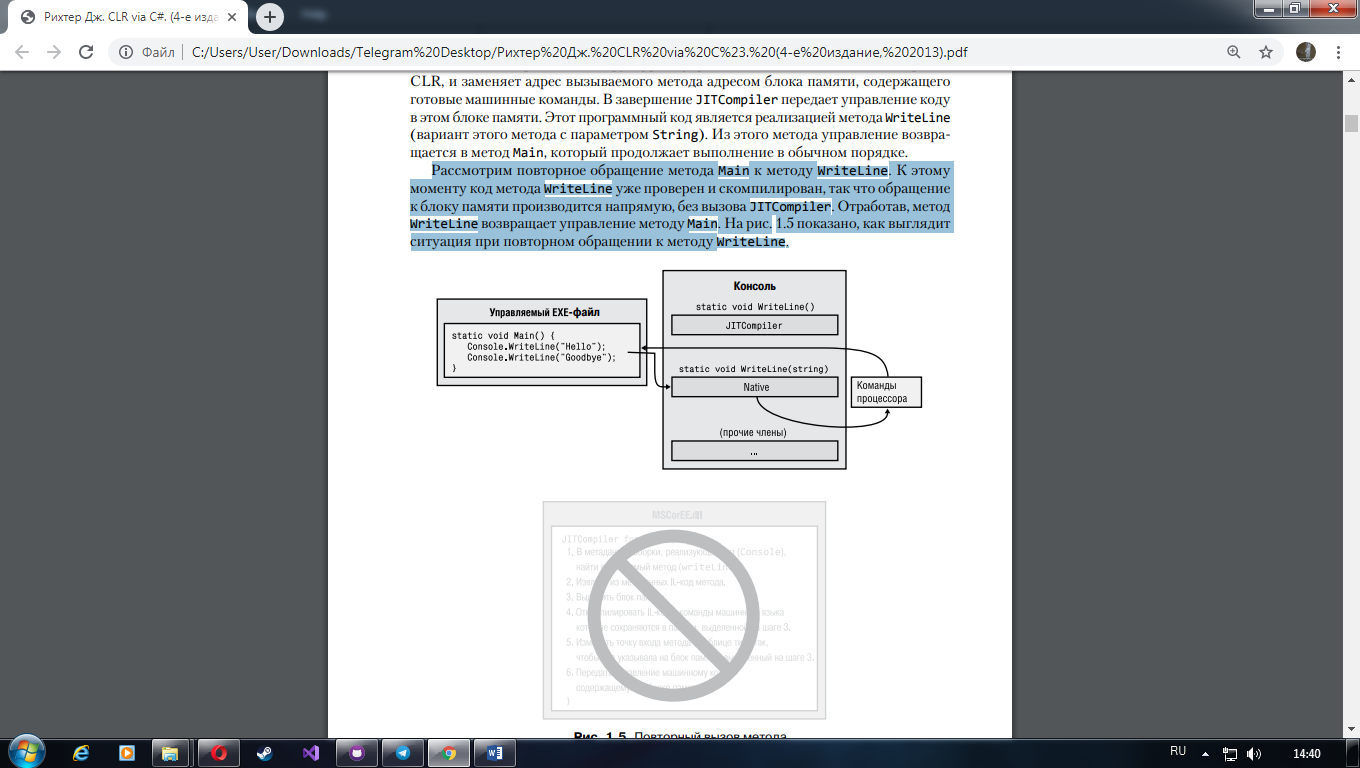
**Исполнение кода сборки.**

Как говорилось ранее, управляемые модули содержат метаданные и программный код IL. Это не зависящий от процессора машинный язык, разработанный компанией Microsoft после консультаций с несколькими коммерческими и академическими организациями, специализирующимися на разработке языков и компиляторов. IL — язык более высокого уровня по сравнению с большинством других машинных языков. Он позволяет работать с объектами и имеет команды для создания и инициализации объектов, вызова виртуальных методов и непосредственного манипулирования элементами массивов. В нем даже есть команды инициирования и перехвата исключений для обработки ошибок. IL можно рассматривать как объектно-ориентированный машинный язык. Обычно разработчики программируют на высокоуровневых языках, таких как C#, Visual Basic или F#. Компиляторы этих языков генерируют IL-код. Однако такой код может быть написан и на языке ассемблера, так, Microsoft предоставляет ассемблер IL (ILAsm.exe), а также дизассемблер IL (ILDasm.exe).

Для выполнения какого-либо метода его IL-код должен быть преобразован в машинные команды. Этим занимается JIT-компилятор (Just-In-Time) среды CLR. На рис. 1.4 показано, что происходит при первом вызове метода. Непосредственно перед исполнением метода Main среда CLR находит все типы данных, на которые ссылается программный код метода Main. При этом CLR выделяет внутренние структуры данных, используемые для управления доступом к типам, на которые есть ссылки. На рис. 1.4 метод Main ссылается на единственный тип — Console, и среда CLR выделяет единственную внутреннюю структуру. Эта внутренняя структура данных содержит по одной записи для каждого метода, определенного в типе Console. Каждая запись содержит адрес, по которому можно найти реализацию метода. При инициализации этой структуры CLR заносит в каждую запись адрес внутренней недокументированной функции, содержащейся в самой среде CLR. Я обозначаю эту функцию JITCompiler:



Рассмотрим повторное обращение метода Main к методу WriteLine. К этому моменту код метода WriteLine уже проверен и скомпилирован, так что обращение к блоку памяти производится напрямую, без вызова JITCompiler. Отработав, метод WriteLine возвращает управление методу Main. На рис. 1.5 показано, как выглядит ситуация при повторном обращении к методу WriteLine.



**Небезопасный код.**

По умолчанию компилятор C# компании Microsoft генерирует безопасный код. Под этим термином понимается код, безопасность которого подтверждается в процессе верификации. Тем не менее компилятор Microsoft C# также позволяет разработчикам писать небезопасный код, способный напрямую работать с адресами памяти и манипулировать с байтами по этим адресам. Как правило, эти чрезвычайно мощные средства применяются для взаимодействия с неуправляемым кодом или для оптимизации алгоритмов, критичных по времени. Однако использование небезопасного кода создает значительный риск: небезопасный код может повредить структуры данных и использовать (или даже создавать) уязвимости в системе безопасности. По этой причине компилятор C# требует, чтобы все методы, содержащие небезопасный код, помечались ключевым словом unsafe, а при компиляции исходного кода использовался параметр компилятора /unsafe. Когда JIT-компилятор пытается откомпилировать небезопасный метод, он сначала убеждается в том, что сборке, содержащей метод, были предоставлены разрешения System.Security.Permissions.SecurityPermission с установленным флагом SkipVerification из перечисления System.Security.Permissions. SecurityPermissionFlag. Если флаг установлен, JIT-компилятор компилирует небезопасный код и разрешает его выполнение. CLR доверяет этому коду и надеется, что прямой доступ к памяти и манипуляции с байтами не причинят вреда. Если флаг не установлен, JIT-компилятор выдает исключение System. InvalidProgramException или System.Security.VerificationException, предотвращая выполнение метода. Скорее всего, в этот момент приложение аварийно завершится, но по крайней мере без причинения вреда. Компания Microsoft предоставляет утилиту PEVerify.exe, которая проверяет все методы сборки и сообщает обо всех методах, содержащих небезопасный код. Возможно, вам стоит запустить PEVerify.exe для всех сборок, на которые вы ссылаетесь; это позволит узнать о возможных проблемах с запуском ваших приложений по интрасети или Интернету.

**NGen.exe**

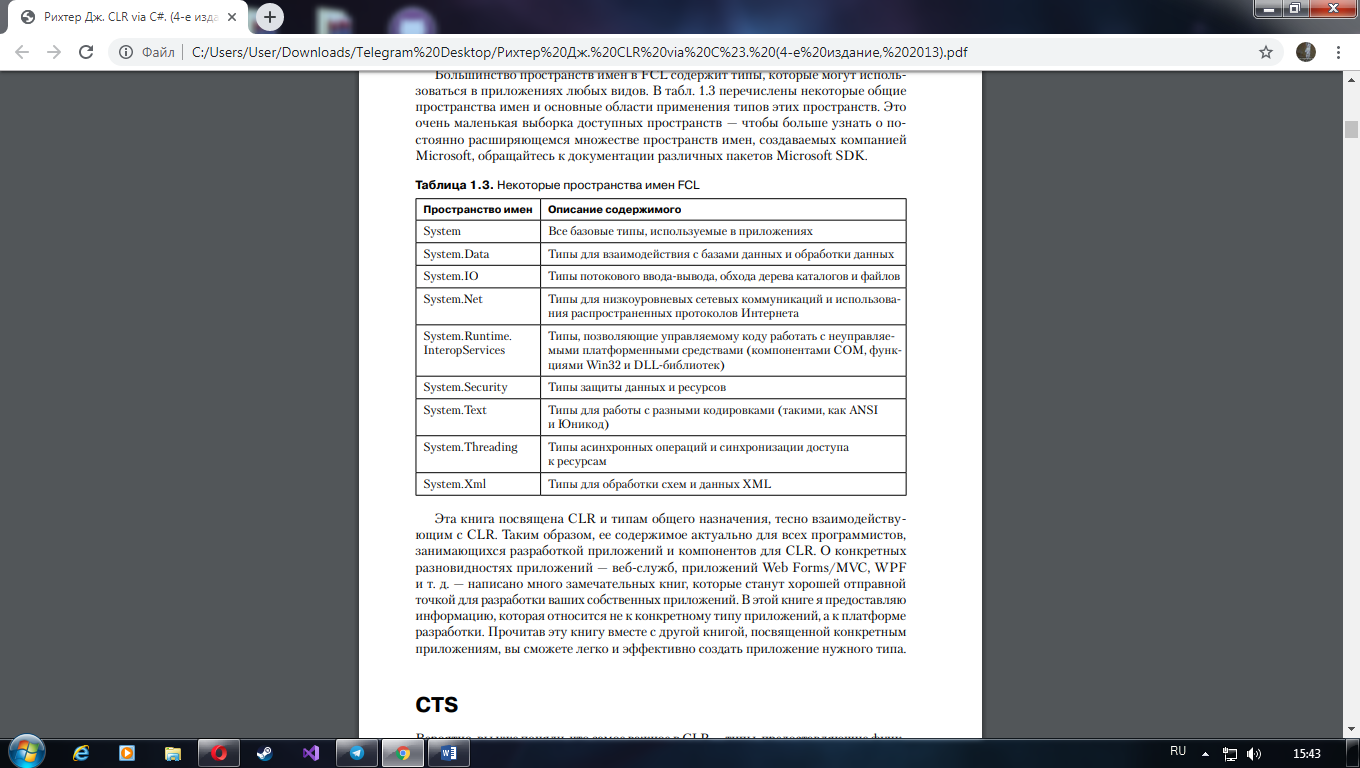
Программа NGen.exe, входящая в поставку .NET Framework, может использоваться для компиляции IL-кода в машинный код при установке приложения на машине пользователя. Так как код компилируется на стадии установки, JIT-компилятору CLR не приходится компилировать его во время выполнения, что может улучшить быстродействие приложения. Программа NGen.exe полезна в двух ситуациях. Ускорение запуска приложения. Запуск NGen.exe ускоряет запуск, потому что код уже откомпилирован в машинную форму, и компиляцию не нужно выполнять на стадии выполнения. Сокращение рабочего набора приложения. Если вы ожидаете, что сборка будет загружаться в нескольких процессах одновременно, обработка ее программой NGen.exe может сократить рабочий набор приложения. Дело в том, что NGen.exe преобразует IL в машинный код и сохраняет результат в отдельном файле. Этот файл может отображаться на память в нескольких адресных пространствах одновременно, а код будет использоваться совместно, без использования каждым процессом собственного экземпляра кода.

**Библиотека FCL**

Одним из компонентов .NET Framework является FCL (Framework Class Library) — набор сборок в формате DLL, содержащих несколько тысяч определений типов, каждый из которых предоставляет некоторую функциональность.

Так как FCL содержит буквально тысячи типов, взаимосвязанные типы объединяются в одно пространство имен. Например, пространство имен System (которое вам стоит изучить как можно лучше) содержит базовый тип Object — «предок» всех остальных типов в системе. Кроме того, пространство имен System содержит типы для целых чисел, символов, строк, обработки исключений и консольного ввода-вывода, а также набор вспомогательных типов, осуществляющих безопасные преобразования между типами данных, форматирование, генерирование случайных чисел и выполняющих математические функции. Все приложения используют типы из пространства имен System

Большинство пространств имен в FCL содержит типы, которые могут использоваться в приложениях любых видов. В табл. 1.3 перечислены некоторые общие пространства имен и основные области применения типов этих пространств. Это очень маленькая выборка доступных пространств — чтобы больше узнать о постоянно расширяющемся множестве пространств имен, создаваемых компанией Microsoft, обращайтесь к документации различных пакетов Microsoft SDK.



**CTS**

Поскольку типы занимают центральное место в CLR, компания Microsoft разработала формальную спецификацию CTS (Common Type System), которая описывает способ определения и поведение типов.

Согласно спецификации CTS, тип может содержать нуль и более членов.

1. Поле — переменная, являющаяся частью состояния объекта. Поля идентифицируются именем и типом.
2. Метод — функция, выполняющая операцию с объектом, часто с изменением его состояния. Метод обладает именем, сигнатурой и модификаторами. Сигнатура определяет количество параметров (и порядок их следования), типы параметров, наличие возвращаемого значения, и если оно имеется — тип значения, возвращаемого методом.
3. Свойство — с точки зрения вызывающей стороны выглядит как поле, но в реализации типа представляет собой метод (или два). Свойства позволяют организовать проверку параметров или состояния объекта перед обращением к значению и/или вычислять его значение только при необходимости. Кроме того, они упрощают синтаксис работы с данными и позволяют создавать «поля», доступные только для чтения или записи.
4. Событие — используется для создания механизма оповещения между объектом и другими заинтересованными объектами. Например, кнопка может поддерживать событие, оповещающее другие объекты о щелчке на ней.

CTS также задает правила видимости типов и доступа к членам типа. Например, помечая тип как открытый (ключевое слово public), вы тем самым экспортируете этот тип, делая его видимым и доступным для любой сборки. С другой стороны, пометка типа на уровне сборки (ключевое слово internal в C#) делает его видимым и доступным для кода той же сборки. Таким образом, CTS устанавливает правила, по которым сборки формируют границу видимости типа, а CLR обеспечивает выполнение правил видимости. CTS 51 Тип, видимый для вызывающей стороны, может установить дополнительные ограничения на возможность обращения к своим членам. Ниже перечислены варианты ограничения доступа к членам типа:

1. Закрытый (приватный) доступ — член типа доступен только для других членов того же типа.  Доступ в семействе — член типа доступен для производных типов независимо от того, принадлежат ли они той же сборке или нет. Обратите внимание: во многих языках (таких, как C# и C++) доступ в семействе обозначается ключевым словом protected.
2. Доступ в семействе и сборке — член типа доступен для производных типов, но только в том случае, если они определяются в той же сборке. Многие языки (например, C# и Visual Basic) не поддерживают этот уровень доступа. Разумеется, в IL-коде он поддерживается.
3. Доступ в сборке — член типа доступен для любого кода, входящего в ту же сборку. Во многих языках доступ в сборке обозначается ключевым словом internal.
4. Доступ в семействе или сборке — член типа доступен для производных типов из любой сборки, а также для любых типов в той же сборке. В C# этот вариант доступа обозначается ключевыми словами protected internal.
5. Открытый доступ — член типа доступен для любого кода в любой сборке

А вот еще одно правило CTS: все типы должны быть производными (прямо или опосредованно) от предопределенного типа System.Object (то есть от типа Object из пространства имен System). Тип Object является корнем иерархии типов, а следовательно, гарантирует, что каждый экземпляр типа обладает минимальным набором аспектов поведения. А если говорить конкретнее, тип System.Object позволяет сделать следующее:  сравнить два экземпляра на равенство;  получить хеш-код экземпляра;  запросить фактический тип экземпляра;  выполнить поверхностное (поразрядное) копирование экземпляра;  получить строковое представление текущего состояния экземпляра.

**Взаимодействие с неуправляемым кодом**

.NET Framework обладает множеством преимуществ перед другими платформами разработки. Впрочем, лишь немногие компании могут позволить себе заново спроектировать и реализовать весь существующий код. Компания Microsoft понимает это, поэтому среда CLR была спроектирована так, чтобы приложения могли состоять как из управляемых, так и из неуправляемых компонентов. А если говорить конкретнее, CLR поддерживает три сценария взаимодействий: Управляемый код может вызывать неуправляемые функции из DLL с использованием механизма P/Invoke (сокращение от «Platform Invoke»). В конце концов, многие типы, определяемые в FCL, во внутренней реализации вызывают функции, экспортируемые из Kernel32.dll, User32.dll и т. д. Многие языки программирования предоставляют средства, упрощающие вызов неуправляемых функций из DLL в управляемом коде. Например, приложение C# может вызвать функцию CreateSemaphore, экспортируемую библиотекой Kernel32.dll. Управляемый код может использовать готовые компоненты COM. Многие компании уже реализовали большое количество неуправляемых компонентов COM. На основе библиотек типов из этих компонентов можно создать управляемую сборку с описанием компонента COM. Управляемый код обращается к типу из управляемой сборки точно так же, как к любому другому управляемому типу. За дополнительной информацией обращайтесь к описанию программы TlbImp.exe, входящей в поставку .NET Framework SDK. Неуправляемый код может использовать управляемый тип. Большая часть существующего неуправляемого кода требует наличия компонента COM. Такие компоненты гораздо проще реализуются с управляемым кодом, что позволяет избежать служебного кода, связанного с подсчетом ссылок и интерфейсами. Например, на C# можно написать элемент управления ActiveX или расширение командного процессора. За дополнительной информацией обращайтесь к описанию программ TlbExp.exe и RegAsm.exe, входящих в поставку .NET Framework SDK.