Михаил Фленов

Библия C#

5-е издание

Программирование для .NET на C#
Платформа .NET
Базы данных
Веб-программирование
Сетевое программирование
Повторное использование кода
Изучение языка на полезных примерах

Материалы
на www.bhv.ru

bhv®

УДК 004.438 C#
ББК 32.973.26-018.1
Ф71

Фленов М. Е.

Ф71 Библия C#. — 5-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2022. — 464 с.: ил.

ISBN 978-5-9775-6827-2

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 веб-приложений. Материал сопровождается большим количеством практических примеров. Подробно описывается логика выполнения каждого участка программы. Уделено внимание вопросам повторного использования кода. В пятом издании примеры переписаны с учетом современной платформы .NET 5, а вместо прикладных приложений упор делается на веб-приложения. На сайте издательства находятся коды программ, дополнительная справочная информация и копия базы данных для выполнения примеров из книги.

Для программистов

УДК 004.438 C#
ББК 32.973.26-018.1

Группа подготовки издания:

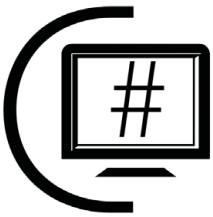
Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Людмила Гауль*
Редактор	*Григорий Добин*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн обложки	*Инны Тачиной*
Оформление обложки	*Карины Соловьевой*

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

ISBN 978-5-9775-6827-2

Оглавление

ГЛАВА 1



Введение в .NET

Платформа Microsoft .NET 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, которую можно скачать с сайта по адресу: **https://visualstudio.microsoft.com**. Большинство примеров, приведенных в книге, написаны много лет назад с использованием Visual Studio 2008, но в процессе 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 всех 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])

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

и даже игры на Unity[1]. Однако все эти реализации немного отличались друг от друга — если написать программу для Windows, то это не значит, что она заработает в Mono на Linux или где-либо еще.

Я не могу знать реальных причин принятия решений внутри компании Microsoft, но мне кажется, что на появление .NET Core 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.

Когда .NET Core вырос и по некоторым возможностям начал обходить существующий .NET Framework, в компании Microsoft решили избавиться от такой путаницы и как бы объединить все усилия в одну ветку. Так что вместо двух веток мы получили одну, и ей дали название просто .NET, хотя на самом деле она является продолжением .NET Core.

При написании консольных приложений на .NET или на старом добром .NET Framework вы особой разницы не заметите. А вот для веб-программирования разница есть, и если начинать новый веб-проект, то я бы делал его на .NET, потому что архитектуру написания веб-приложений изменили весьма сильно. Новая архитектура намного лучше, быстрее и в ней используются современные подходы (паттерны) программирования.

Не знаю, насколько для вас это является преимуществом, но исходные коды .NET Core открыты (то есть это проект OpenSource) и их можно найти на сайте **github** по адресу: **https://github.com/dotnet**. Мне лично этот факт не важен, потому что в исходные коды .NET 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.

В этой книге мы будем изучать язык программирования C#, который является основным языком для платформы .NET.

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 вы можете с использованием платформы .NET разрабатывать приложения любой сложности, которые очень просто интегрируются с серверами и сервисами от Microsoft.

Основа всего, центральное звено платформы — это фреймворк .NET. 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[2] программы. Код

[1] Unity — межплатформенная среда разработки компьютерных игр.

[2] IL, Intermediate Language — промежуточный язык.

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* — 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;
- *расширенные классы .NET*. 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 от 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 здесь использовать мудреные названия типа «компоновочный блок» или «бинарная единица», а просто стану называть вещи простыми 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».

---

[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).

Благодаря тому, что IL-код не является машинным, а интерпретируется JIT-компилятором[1], говорят, что *код управляем* этим JIT-компилятором. Машинный код выполняется напрямую процессором, и ОС не может управлять этим кодом. А вот IL-код выполняется на платформе .NET, 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 и на самом деле не интересует.

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 сама будет 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

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

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, установленный на 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 его 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 2019 Community Edition, но работа с ней не отличается от работы с любой другой версией Visual Studio.

Далее в этой главе я попытался подробно предоставить базовую информацию о работе с Visual Studio, но некоторые вещи лучше один раз увидеть, чем подробно прочитать, поэтому на моем сайте я опубликовал видео, в котором показываю базовые основы работы с этой средой разработки, — это видео можно найти здесь: **https://www.flenov.info/books/show/biblia-csharp**. Видео и текстовая информация дополняют друг друга, поэтому я рекомендую вам и прочитать книгу, и просмотреть видео, а не ограничиваться чем-либо одним.

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*).

После запуска Visual Studio может появиться окно приветствия, в котором слева будут расположены проекты, с которыми вы работали ранее, а справа — кнопки

для выполнения базовых команд (рис. 1.1). Самая нижняя кнопка **Create a New Project** позволяет создать новый проект, а имеющаяся чуть ниже ее ссылка **Continue without code** (Продолжить без кода) открывает среду разработки без создания или открытия проекта.

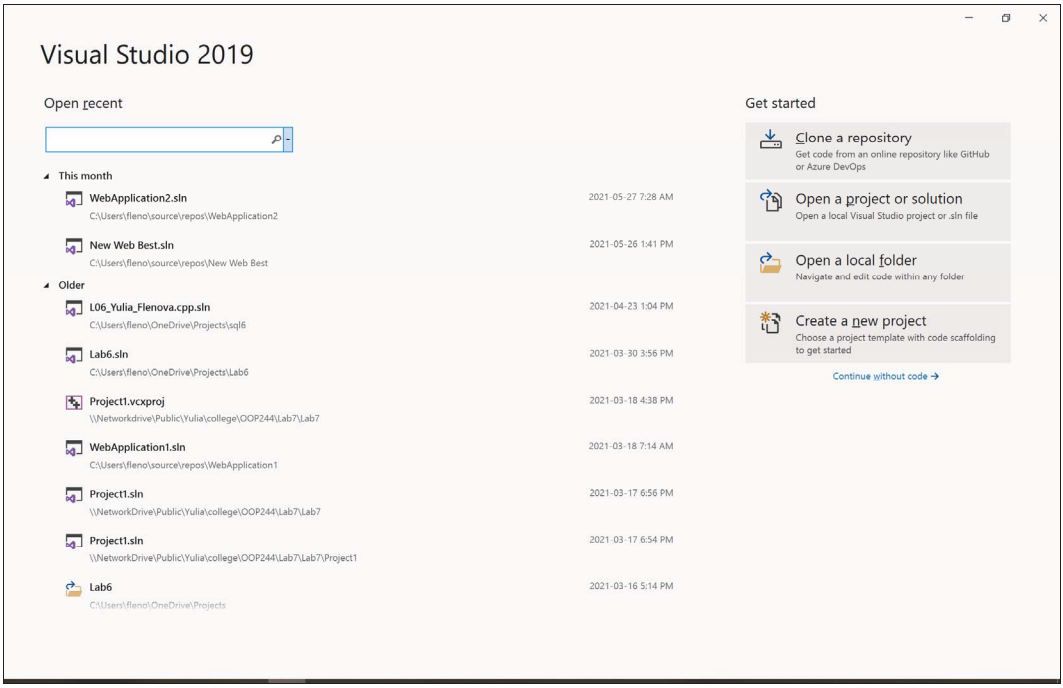


Рис. 1.1. Окно приветствия Visual Studio

Для создания нового проекта выберите в меню Visual Studio команду **File | New | Project** (Файл | Новый | Проект). Перед вами откроется окно **New Project** (Новый проект), в левой части которого расположены шаблоны проектов, которые вы недавно использовали (рис. 1.2). Справа сверху расположены три выпадающих списка:

- среда разработки Visual Studio может работать и компилировать проекты на нескольких языках: Visual C++, Visual C#, Visual Basic и т. д., и в первом выпадающем списке можно выбрать язык программирования;
- из второго списка следует выбрать платформу, под которую вы хотите создать приложение. В Visual Studio можно писать десктопные приложения для Windows, Андроид, iOS, Linux, Azure и т. д.
- третий список предоставляет выбор типа приложения: мобильное, веб-приложение, игры, консольное приложение и т. д.

При выборе значений из этих выпадающих списков список доступных шаблонов справа внизу будет изменятся. Мы рассматриваем C#, поэтому для всех проектов этой книги в первом списке выбираем **C#**. Во втором можно оставить **All Platforms**,

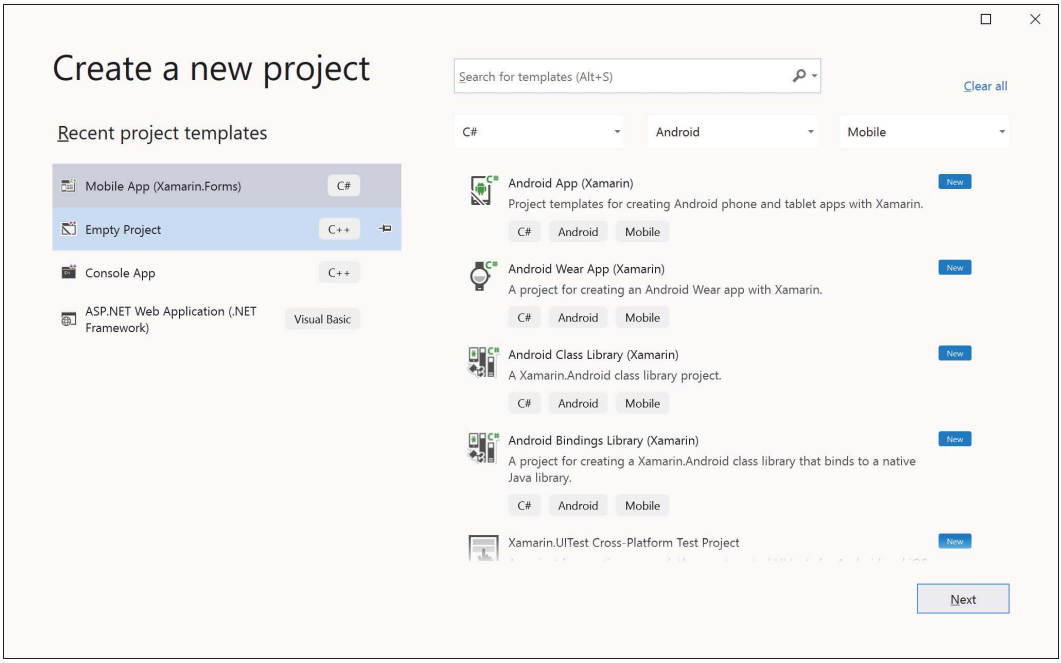


Рис. 1.2. Окно создания нового проекта

а в третьем выберем **Console** для первых приложений, а когда доберемся до веб-приложений, то можно будет выбрать **Web**, чтобы быстрее найти нужный нам шаблон.

Разобравшись со списками, нажимаем кнопку **Next** (Далее), и на втором шаге создания проекта нам предложат ввести три параметра:

- **Project Name** — здесь вы указываете имя будущего проекта;
- **Location** — расположение каталога проекта.
- **Solution name** — имя решения (по умолчанию соответствует имени проекта и, возможно, это поле будет неизменяемым).

Давайте сейчас остановимся на секунду на третьем пункте — имени 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.

Если мы, например, хотим создать приложение Машина, то начнем с создания приложения и решения Машина, а потом можем к этому решению добавить дополнительные проекты: двигатель, коробка передач и т. д. Именно поэтому имя решения по умолчанию может быть неизменяемым и соответствовать имени проекта.

В окне второго шага имеется также флажок **Please solution and project in the same directory** (Поместить решение и проект в один каталог), и если вы планируете помещать все в одну папку, то это скажет мастеру создания проекта, что вы не планируете добавлять новые проекты, и именно поэтому поле ввода имени решения будет недоступно. Но если вы сбросите этот флажок, то для проекта будет создана отдельная папка. Поскольку вы решили завести отдельную папку для проекта, то, возможно, вы уже планируете создавать дополнительные проекты, а значит, имя решения может быть иным, и вам позволят его сменить.

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**.

Давайте создадим новый пустой C#-проект, чтобы увидеть, из чего он состоит. В окне создания нового проекта выбираем из первого выпадающего списка **C#**, во втором оставляем все по умолчанию, а из третьего выбираем **Console**. В списке шаблонов у вас должно остаться только **Console Application** (это приложение .NET 5) и **Console App** (приложение .NET Framework). В скобках при **Console App** вы можете видеть **.NET Framework**, что как раз указывает на то, что это старый фреймворк (рис. 1.3). Нас же интересует первый из этих шаблонов. Под именем шаблона показано, что он подходит для консольного приложения, которое потом сможет запускаться на Linux, macOS и Windows.

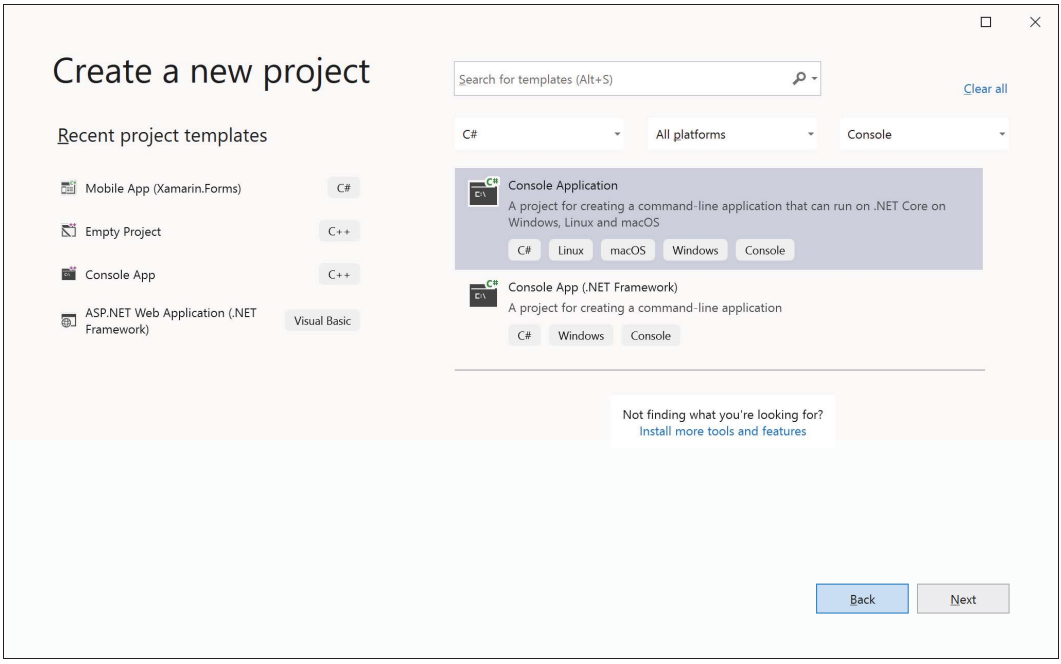


Рис. 1.3. Создаем консольное приложение

Нажимаем в этом окне кнопку **Next** и на следующем шаге вводим имя проекта — `TestApplication`, а место расположения можно оставить по умолчанию. На следующем шаге нам нужно выбрать **Target Framework** (Целевой фреймворк) — давайте выберем наиболее свежий, а на момент подготовки этого издания — это **.NET 5**. С выходом .NET 6 поменяется немногое, потому что тут ничего не менялось уже последние 20 лет, и я не ожидаю каких-либо кардинальных нововведений.

Указав имя проекта и его расположение, для завершения создания нового проекта нажмите кнопку **Create** (Создать). В результате вы увидите окно среды разработки, показанное для нашего проекта на рис. 1.4. Оно включает несколько окон, и одно из главных здесь — это **Solution Explorer** (Проводник решения), которое расположено вдоль правой кромки окна среды разработки. Если по какой-либо причине вы его не видите (оно может быть закрыто), то для его открытия надо выбрать пункт меню **View | Solution Explorer**.

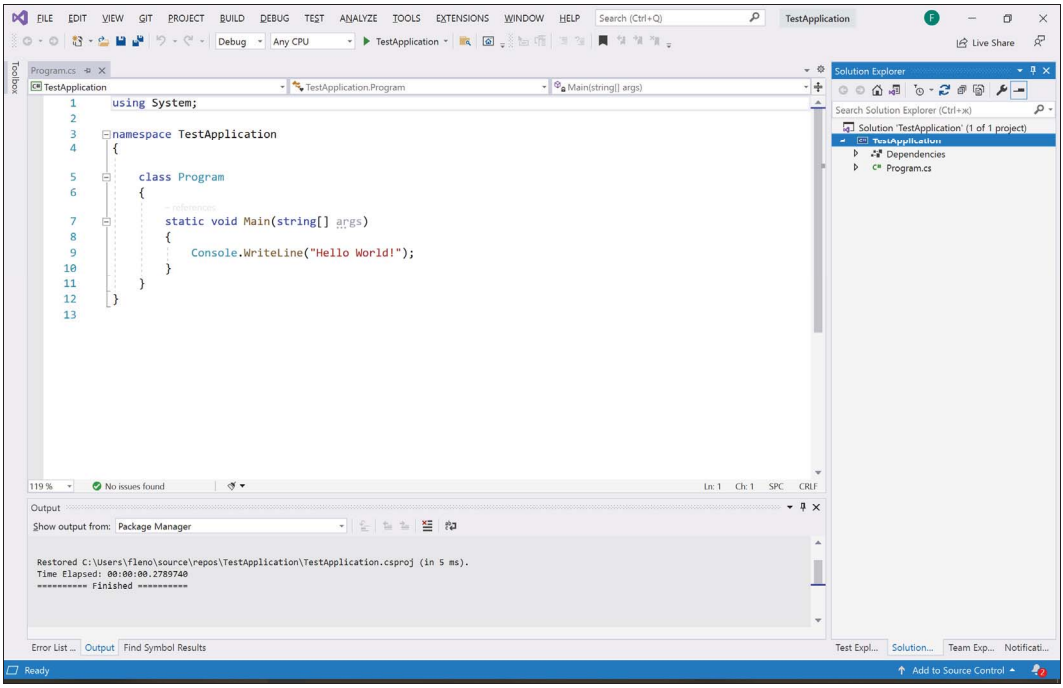


Рис. 1.4. Окно среды разработки с открытым проектом

В окне **Solution Explorer** самая первая запись — имя решения. Чуть ниже в дереве элементов вы увидите имя проекта, которое как бы вложено в решение. В папке с проектом показана папка **Dependencies** (Зависимости) и файлы, которые входят в проект. О зависимостях мы еще поговорим далее — сейчас же мы только знакомимся с проектом. Из файлов пока доступен лишь один — **Program.cs**. Это файл, где мы будем впоследствии писать код. Имена файлов с кодом имеют расширение cs, что означает C Sharp и соответствует языку программирования C#, который рассматривается в этой книге.

Если щелкнуть правой кнопкой мыши на имени проекта и в выпадающем меню выбрать **Open folder in file explorer**, то откроется окно проводника и папка, в которой находятся файлы проекта (рис. 1.5).

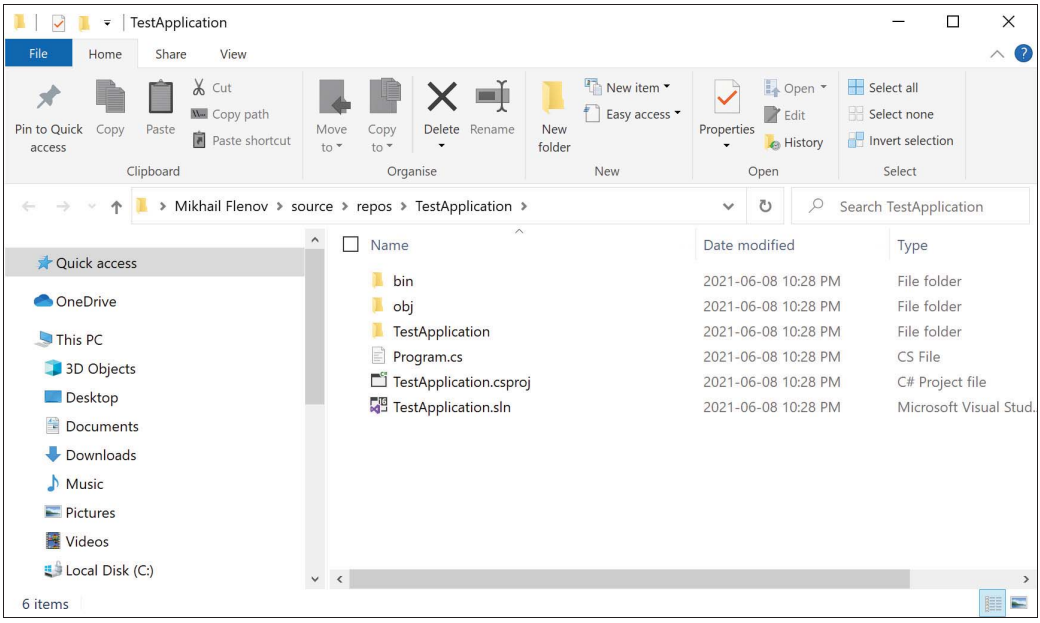


Рис. 1.5. Файлы проекта

Здесь мы видим файл нашего решения, который имеет расширение sln (от solution, решение), файл проекта с расширением csproj (C Sharp Project, проект C#) и файл с исходным кодом Program.cs, который мы видели и в окне **Solution Explorer**.

Файл проекта, в котором находятся все настройки и описания входящих в проект файлов, как уже отмечено, 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.

Чтобы посмотреть или изменить свойства проекта, можно щелкнуть на имени проекта в окне **Solution Explorer** и выбрать из появившегося меню пункт **Properties** (Свойства).

Чтобы открыть файл для редактирования, достаточно щелкнуть на нем мышью двойным щелчком в окне **Solution Explorer**.

Для переименования решения щелкните на его имени в дереве правой кнопкой мыши и в контекстном меню выберите пункт **Rename** (Переименовать). Таким же образом можно переименовывать и файлы.

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 Web Site** — добавить существующий проект с веб-сайта;
- **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 или **Build Solution** (Собрать решение) — компилироваться будут только измененные файлы, или **Rebuild Solution** (Полностью собрать проект) — компилироваться будут все файлы. Такие же пункты меню есть и в основном меню **Build**. Для сборки проекта проще использовать комбинацию клавиш <Ctrl>+<Shift>+<B>.

Если же щелкнуть правой кнопкой мыши на имени проекта, то в контекстном меню 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 окна команду **Debug | Start Debugging** (Отладка | Запуск). В ответ на это проект будет запущен на выполнение с возможностью отладки — то есть вы сможете устанавливать точки останова и выполнять код программы построчно. Если отладка не нужна, то нажимаем комбинацию 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.

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 применяются два режима компиляции: Debug и Release:

- в режиме Debug в исполняемый файл может добавляться дополнительная информация, необходимая для тестирования и отладки. Такие файлы содержат много лишней информации, особенно если проект создан на C++, то их только для тестирования и отладки и используют. Файлы, созданные в этом режиме

компиляции, 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 в этом режиме компиляции, находятся в каталоге bin\Release вашего проекта.

На рис. 1.6 показан выпадающий список, с помощью которого можно менять режим компиляции.

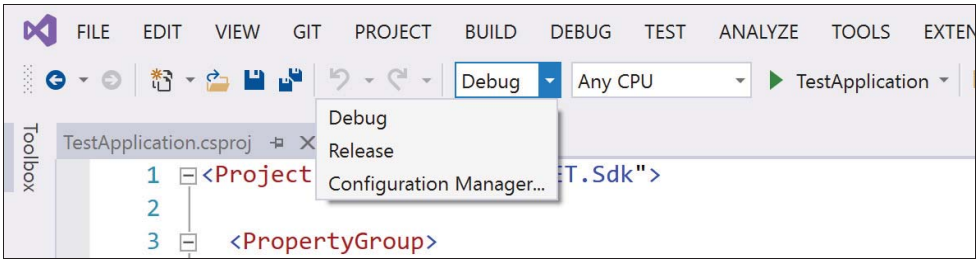


Рис. 1.6. Смена режима компиляции: **Debug** или **Release**

Впрочем, даже если компилировать проект 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. И если произойдет ошибка работы программы, а рядом с исполняемым файлом будет находиться его PDB-файл, то пользователю даже покажут, 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 веб-сайт, то подобный файл можно поместить на веб-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.

1.2.2. 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.