Владимирский государственный университет

Д. А. ЯКУБОВИЧ Е. С. ЕРОПОВА

ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ НА БАЗЕ ПЛАТФОРМЫ .NET И ЯЗЫКА C#

Владимир 2025

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»

Д. А. ЯКУБОВИЧ Е. С. ЕРОПОВА

ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ НА БАЗЕ ПЛАТФОРМЫ .NET И ЯЗЫКА C#

Электронное издание

Владимир 2025

ISBN 978-5-9984-2286-7

УДК 004.4
ББК 32.973

Рецензенты:
Кандидат экономических наук, доцент
доцент кафедры менеджмента и бизнес-информатики
Владимирского филиала Финансового университета при Правительстве РФ
С. В. Никифорова

Кандидат физико-математических наук, доцент
зам. директора по учебно-методической работе Педагогического института
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых
В. А. Игонин

Якубович, Д. А. Практикум по программированию на базе платформы .NET и языка C# [Электронный ресурс] / Д. А. Якубович, Е. С. Еропова ; Владим. гос. ун-т им. А. Г. и Н. Г. Столетовых. – Владимир : Изд-во ВлГУ, 2025. – 465 с. – ISBN 978-5-9984-2286-7. – Электрон. дан. (11,1 Мб). – 1 электрон. опт. диск (CD-ROM). – Систем. требования: Intel от 1,3 ГГц ; Windows XP/7/8/10 ; Adobe Reader ; дисковод CD-ROM. – Загл. с титул. экрана.

Содержит темы по обучению студентов программированию на базе платформы .NET и языка программирования C#. Материал систематизирован и проиллюстрирован многочисленными примерами.

Предназначен для проведения практических и лабораторных занятий со студентами педагогических вузов по дисциплинам «Программирование», «Практикум по решению задач на ЭВМ», «Информационные технологии в профессиональной деятельности». Может быть использован для организации самостоятельной работы студентов, чтения курсов повышения квалификации педагогических кадров и самообучения.

Рекомендовано для формирования профессиональных компетенций в соответствии с ФГОС ВО.

Табл. 3. Ил. 294. Библиогр.: 20 назв.

ISBN 978-5-9984-2286-7

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время навыки программирования крайне востребованы в различных сферах профессиональной деятельности. Будущие выпускники педагогических вузов, профиль обучения которых связан с математикой, информатикой, физикой, экономикой и другими науками естественно-научного цикла, должны владеть фундаментальными компетенциями в области разработки современного программного обеспечения (ПО).

Актуальность навыков программирования связана с обширным спектром практических задач, которые могут быть решены посредством реализуемых программ. С другой стороны, понимание принципов работы приложений, алгоритмов обработки данных и их структур помогает специалистам решать профессиональные задачи, например, привлекая возможности систем искусственного интеллекта. Немаловажным является опора на современные языки программирования, платформы и инструментальные средства разработчика.

Современный учитель информатики должен не только владеть ключевыми принципами разработки ПО, но и уметь обучать этим навыкам школьников, демонстрируя разные походы к решению и оптимизации задач, прививать интерес к решению возникающих проблем благодаря задействованию профессионального инструментария.

Первая глава практикума знакомит читателя с платформой .NET и языком программирования C#. Показаны преимущества платформы и языка в обучении. Подробно описывается необходимое для работы ПО, порядок его установки и настройки. Рассматриваются шаблон консольного приложения, его компиляция и запуск.

Вторая глава посвящена фундаментальным понятиям и синтаксису языка C#, включает ключевые команды и управляющие конструкции языка, структуры данных: описание переменных, математические операции, условные и циклические конструкции, методы как подпрограммы. Каждый параграф сопровождается подробно прокомментированными примерами.

Третья глава направлена на формирование у учащихся фундаментальных навыков программирования на языке C# как объектно-ориентированном. Рассматриваются понятие класса и его структурные компоненты, механизмы их функционирования и взаимодействия. Для более детального знакомства весь раздел сопровождается примерами классов, которые расширяются согласно изучаемым возможностям на протяжении всего раздела. Многочисленные примеры изложены в доступной и подробной форме.

Практикум направлен на формирование фундаментальных компетенций студентов по основам объектно-ориентированного программирования и разработке консольных приложений. Каждое занятие сопровождается необходимым теоретическим материалом, многочисленными подробно прокомментированными примерами и иллюстрациями. Практические задания включают многочисленные задачи, необходимые для формирования и закрепления опыта работы с рассматриваемыми технологиями.

Глава 1. ВВЕДЕНИЕ В ПЛАТФОРМУ .NET И ЯЗЫК ПРОГРАММИРОВАНИЯ C#

1.1. Платформа .NET и язык C#

1.1.1. Предисловие к курсу

Значимость навыков программирования

Подготовка будущих учителей включает в себя формирование компетенций в области использования средств ИКТ, которые учитель будет способен применять в своей профессиональной деятельности. Кроме изучения прикладного программного обеспечения (ПО) будущие выпускники должны познакомиться и с разнообразными инструментальным средствами, которые используются в разработке ПО.

Навыки программирования и разработки ПО особенно важны для учителей информатики. В соответствии с Федеральным государственным образовательным стандартом третьего поколения (ФГОС3++) формирование навыков программирования является одной из ключевых компетенций, которая необходима учащимся для понимания принципов работы современных ИКТ. При этом учителю важно не только самому владеть необходимыми навыками алгоритмизации и программирования, но и уметь развивать интерес и мотивацию к изучению основ программирования у учащихся.

Учителю информатики важно выбрать удобный и универсальный язык программирования, на основе которого учащиеся способны освоить фундаментальные навыки алгоритмизации и программирования, а также познакомиться с разработкой ПО на основе выполнения простых учебных проектов прикладного характера. Представленные в текущем пособии технологии могут стать отличной альтернативой для используемых в настоящее время технологий обучения школьников и студентов основам программирования.

Текущий курс по языку C# нацелен на формирование у будущих учителей информатики необходимых навыков проектирования и программирования, которые помогут им не только в освоении технических аспектов дисциплины, но и в способности выбирать рациональные подходы к обучению. Навыки программирования формируют абстрактное, критическое мышление, креативность, дают возможность управлять различными технологиями, что является важным аспектом в условиях быстро меняющегося информационного общества.

Представленный в этом пособии учебно-практический материал описывает мощный и гибкий инструмент разработки ПО, дающий обширные возможности в реализации творческих идей и решения практических задач.

Коротко о планах текущего практического курса

Программа текущего курса включает:

- описание возможностей современной платформы .NET и популярного языка программирования C#;
- установку и конфигурирование необходимого для учебной работы ПО;
- изучение структуры программы на языке C#;
- знакомство с основами синтаксиса C# и основными управляющими конструкциями;
- изучение структуры классов;
- знакомство с основными концепциями объектно-ориентированного программирования.

Для упрощения учебного процесса в этом курсе не затрагиваются вопросы построения сложных алгоритмов, а делается упор на объектно-ориентированную основу. Много внимания уделяется деталям и работе в профессиональном редакторе Visual Studio Code, а примеры и задания ориентированы на практическую составляющую.

Материалы пособия подойдут как для обучения студентов без опыта программирования, так и для систематизации знаний по программированию на примере платформы .NET и языка C#. Читатель убедится, что этот инструмент является современной альтернативой другим технологиям разработки ПО, а также отлично подойдет для обучения школьников основам программирования.

1.1.2. Платформа .NET

Понятие и предпосылки появления

Определение

.NET («дот Нэт») *– это программная платформа для разработки ПО и поддержки его работы на операционных системах. Была разработана компанией Microsoft в 2002 г.*

Технология .NET (изначально .NET Framework) появилась на фоне стремительного развития инструментов разработки ПО и интернет-технологий в конце 90-х годов и прежде всего как ответ уже существовавшему, хотя и достаточно молодому языку Java. В то время существовала острая необходимость в технологиях, способных работать на разных операционных системах.

Язык **Java** (разработанный компанией Sun Microsystems) заложил революционную на тот момент (1995 г.) концепцию работы приложений «Write Once, Run Anywhere» («Напиши один раз и запускай где угодно»). Согласно этой концепции Java-приложения могли работать на любой платформе и операционный системе, где установлена **виртуальная машина** Java (**JVM**).

Однако Java обладал рядом недостатков и критиковался многими разработчиками за производительность и сложность в освоении. Кроме того, права на Java принадлежали компании Sun, что ограничивало возможности развития технологии.

Воодушевившись успехом Java, разработчики Microsoft начали работу над собственным проектом Next Generation Windows Services (NGWS), который впоследствии превратился в **.NET Framework**. Компания пыталась создать более производительную и гибкую платформу для разработки и выполнения приложений, ориентированную на операционные системы линейки Windows.

Современный .NET представляет собой целую экосистему с огромным объемом реализованных решений для разработки различного типа ПО: от простых утилит до проектов со сложным графическим интерфейсом.

Возможности платформы

.NET – это универсальная платформа для разработки, способная решать широкий спектр задач:

- разработка настольных приложений с богатым пользовательским интерфейсом (технология Windows Forms, WPF);
- веб-разработка (динамические веб-сайты и веб-приложения на базе ASP.NET);
- работа с базами данных (технология ADO.NET);
- разработка компонент операционных систем;
- мобильная разработка (например, на базе Xamarin);
- облачные сервисы и вычисления (Azure);
- разработка игр (например – с помощью движка Unity).
- интернет вещей (IoT);
- и др.

NET обладает рядом преимуществ, которые делают её привлекательной для разработчиков:

1. *Универсальность*. .NET поддерживает разные языки программирования на базе единой программной платформы: C#, Visual Basic, C++/CLI, F# или др. Разработчик может выбрать более удобный для него язык программирования среди доступных, а компиляторы преобразуют его в единый специальный промежуточный код, который выполняется одинаково эффективно для любого языка.
2. *Безопасность*. .NET обеспечивает безопасное выполнение программ на операционных системах Windows за счет проверки типов данных, управления доступом к коду и даже криптографии. Работа .NET-приложений контролируется платформой. Это позволяет избежать критических сбоев на уровне операционной системы, если в работе приложения возникли проблемы.
3. *Кроссплатформенность*. .NET отвечает за работу приложений везде, где установлен пакет технологии .NET. Более того, появившиеся позже технологии .NET Core и .NET 5/6/7/8/9 позволяют запускать приложения не только на Windows, но и на Linux и macOS. А в ОС Windows она устанавливается уже изначально, вместе с системой.

4. *Большая библиотека встроенных классов*. Обширный набор классов библиотеки .NET позволяет реализовывать почти любые задачи разработки ПО без необходимости писать код «с нуля», что существенно ускоряет и упрощает разработку больших проектов.
5. *Автоматическое управление памятью*. В .NET реализован **сборщик мусора** – механизм, который автоматически освобождает память, занятую неиспользуемыми объектами программы, что снижает риск утечки памяти и снимает эту задачу с разработчика.
6. *Производительность*. .NET использует **JIT**-компиляцию (Just-In-Time), т.е. компилирует код непосредственно перед выполнением, что обеспечивает достаточно высокую производительность.

Особенности .NET

В основе работы платформы .NET лежат два важных компонента: CLR и FCL.

1. CLR

CLR (**Common Language Runtime**) – это **общеязыковая среда выполнения кода**. Она отвечает за JIT-компиляцию, управление памятью, обработку исключений (возникающих в процессе выполнения ошибок) и обеспечение безопасности. CLR позволяет запускать код, написанный на разных языках программирования, в единой среде.

Разработчик выбирает один из .NET-совместимых языков, синтаксис которого ему близок. Код компилируется в промежуточный универсальный формат и приложение выполняется при поддержке .NET.

2. FCL

FCL (**Framework Class Library**) – это стандартная и огромная **библиотека классов**, которая предоставляет разработчикам готовые инструменты для решения широкого спектра задач. FCL допускает использование везде, где установлен .NET. В ней реализованы классы для работы обработки файлов, баз данных, работы с сетью, графикой, многопоточным программированием и многое другое.

Замечание

.NET – это инструмент разработки ПО и поддержки его работы в операционной системе.

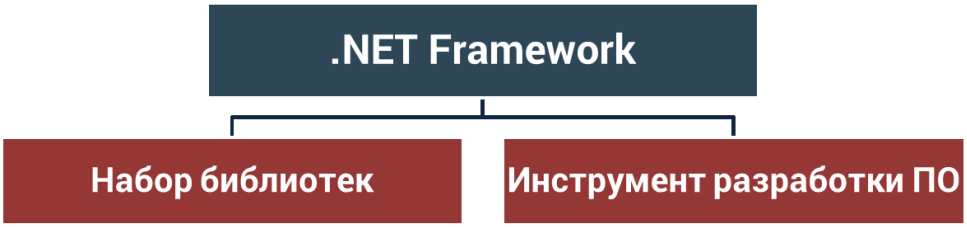


Рис. 1.1. Универсальность .NET

Это важно знать!

Современные версии Windows 10 и 11 уже содержат предустановленный .NET Framework. При необходимости пользователь может бесплатно загрузить последнюю актуальную версию платформы с официального сайта.

В ОС Windows по умолчанию каталог .NET расположен в системном каталоге Windows:

```
C:\WINDOWS\Microsoft.NET\
```

Кратко об истории .NET

Платформа .NET за сравнительно малый срок преодолела большой путь развития.

.NET Framework

Первоначально платформа получила название **.NET Framework** и вышла в релиз в 2002 году. Главной целью создания этой платформы стало упрощение разработки приложений под Windows. Изначально она включала в себя CLR и BCL. Каждая версия платформы добавляла новые возможности в синтаксис поддерживаемых языков программирования и библиотеку классов.

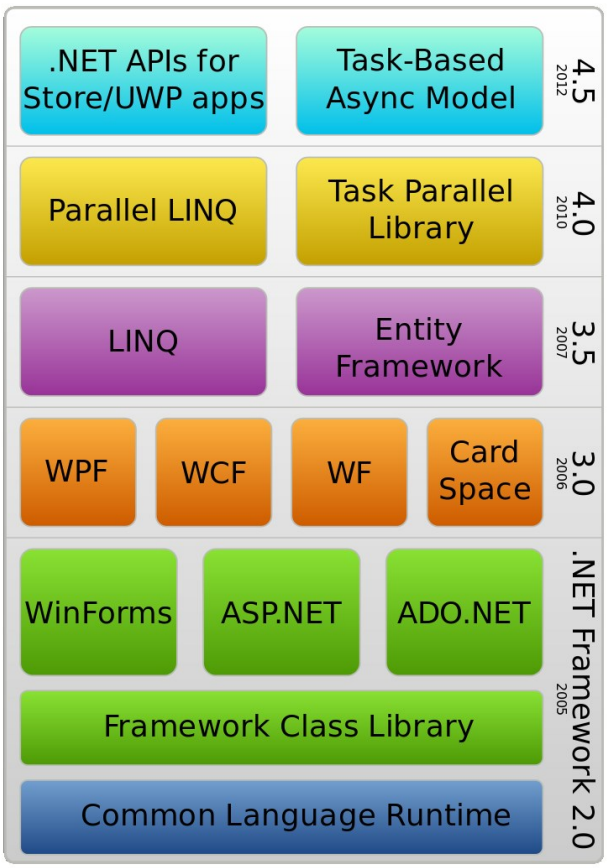


Рис. 1.2. Нововведения платформы .NET Framework в разных версиях

Изначально .NET Framework ориентировалась именно на линейку ОС Windows, но сегодня она находит применение и в других ОС и мобильных платформах.

.NET Core

.NET Framework распространяется бесплатно, но программный код самих библиотек закрыт.

Однако современные тенденции к использованию открытого программного обеспечения подтолкнули Microsoft к более лояльной политике. Так в 2016 году появился проект **.NET Core** как альтернатива .NET Framework, но уже с открытым программным кодом.

Кроме того, платформа .NET Core имеет уже модульную структуру, позволяющую обновлять не всю платформу, а только ее отдельные модули. Много внимания уделено кроссплатформенному выполнению приложений в разных ОС.

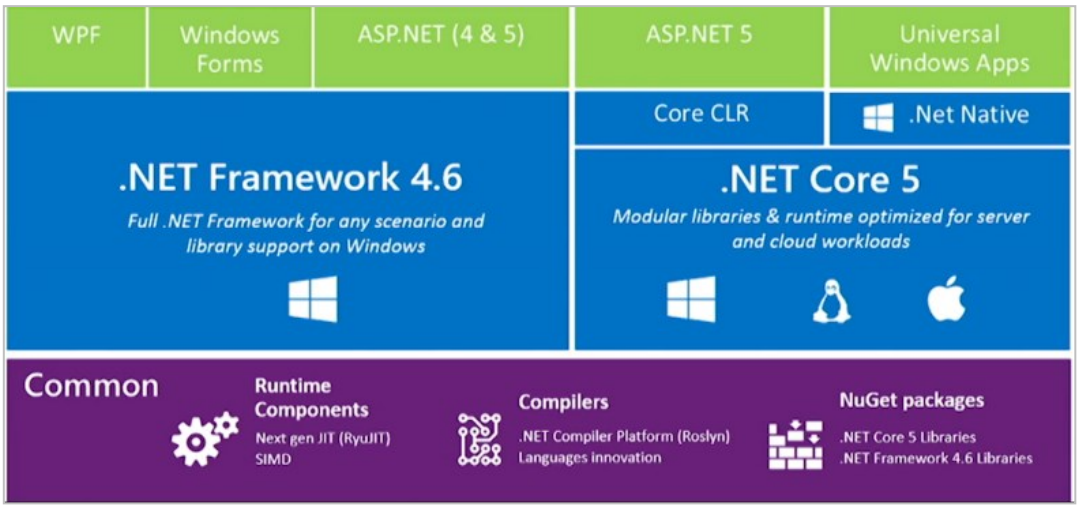


Рис. 1.3. Модульная платформа .NET Core

.NET

10 ноября 2020 года вышел релиз платформы ***.NET 5***, которая объединила множество возможностей .NET Framework и .NET Core в единую универсальную платформу. С этого момента Microsoft развивает платформу именно в таком ключе, предоставляя ежегодные обновления в виде новой версии платформы.

Так, 12 ноября 2024 года состоялся релиз .NET 9, а на ноябрь 2025 года планируется выход .NET 10. Разумеется, каждая последующая версия .NET поддерживает возможности предыдущих (принцип обратной совместимости).

Рис. 1.4. .NET как универсальная модульная платформа

1.1.3. Язык программирования C#

Определение

C# («Си шарп») *– объектно-ориентированный язык программирования, разработанный в 1998-2002 гг. группой инженеров под руководством Андерса Хейлсберга (Microsoft).*

Первоначально проект по разработке нового языка для платформы .NET носил кодовое название «Cool», но вскоре получил свое «музыкальное» имя, символизирующее гармонию и развитие.

Официальный релиз C# состоялся в 2002 году, одновременно с выходом первой версии платформы .NET.

Среди важных достоинств C# выделяют:

- наибольшую популярность среди всех язык на базе платформы .NET;
- схожесть базового синтаксиса с языками C / C++, близость к Java (однако это абсолютно новый язык);
- сочетание разных подходов к программированию (объектно-ориентированный, функциональный, компонентный и др.);
- большое сообщество разработчиков и постоянное совершенствование возможностей синтаксиса.

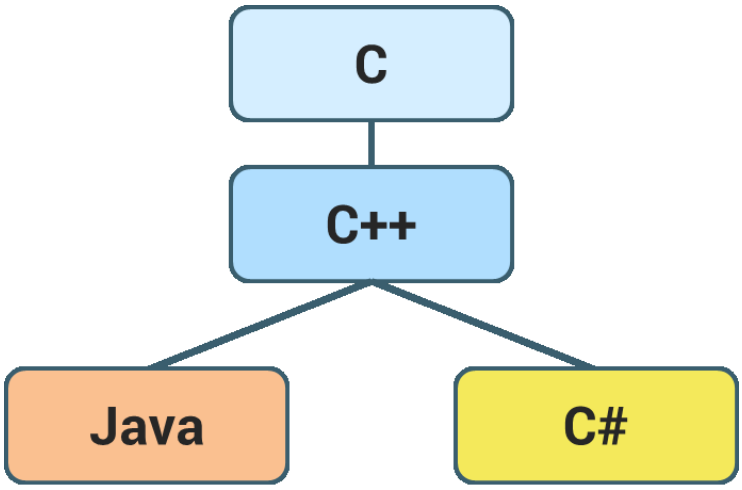


Рис. 1.5. C# похож на языки семейства «Си», унаследовал многие возможности C++ и идеи Java

Таблица 1.1. Версии языка C# в соответствие с .NET

Версия C#	**Версия .NET**	**Дата релиза**
C# 1.0 C# 1.1 C# 1.2	.NET Framework 1.0 .NET Framework 1.1	Январь 2002 Апрель 2003
C# 2.0	.NET Framework 2.0 .NET Framework 3.0	Ноябрь 2005
C# 3.0	.NET Framework 3.5	Ноябрь 2007
C# 4.0	.NET Framework 4	Апрель 2010
C# 5.0	.NET Framework 4.5	Август 2012
C# 6.0	.NET Framework 4.6	Июль 2015
C# 7.0 C# 7.1 C# 7.2 C# 7.3	.NET Framework 4.7 .NET Core 2.0 .NET Core 2.1 .NET Core 2.2 .NET Framework 4.8	Март 2017 Август 2017 Ноябрь 2017 Май 2018
C# 8.0	.NET Core 3.0	Сентябрь 2019
C# 9.0	.NET 5	Ноябрь 2020
C# 10.0	.NET 6	Ноябрь 2021
C# 11.0	.NET 7	Ноябрь 2022
C# 12.0	.NET 8	Ноябрь 2023
C# 13.0	**.NET 9**	**Ноябрь 2024**

Это полезно знать!

На момент написания текущего пособия последняя актуальная версия платформы – .NET 9, а языка – C# 13.

1.1.4. .NET в обучении программированию

Проблема выбора первого языка программирования

В последнее время в школьном курсе информатики для обучения основам программирования обычно выбирают Pascal ABC или Python. Первый отличается грамотной структуризацией синтаксиса и близостью к алгоритмическому языку, а второй лаконичностью.

C# сохраняет в себе баланс между сравнительной простой и прозрачной структурой кода и является хорошим инструментом как для обучения программированию, так и, прежде всего – решения практических задач.

Достоинства C# как учебного языка

1. Простой синтаксис

C# обладает простым и хорошо читаемым синтаксисом, который проще понять новичкам по сравнению с рядом других популярных языков программирования. Учащиеся могут сосредоточиться на логике программы, а не путаться в разнообразии синтаксических конструкций (как, например – в Python).

2. Строгая типизация и качественная модульная структура

С самого начала C# приучает к четкой модульной организации кода. На начальном этапе знакомства с языком учащиеся должны понять важность типизации и структуризации данных в программировании, что и реализует C#.

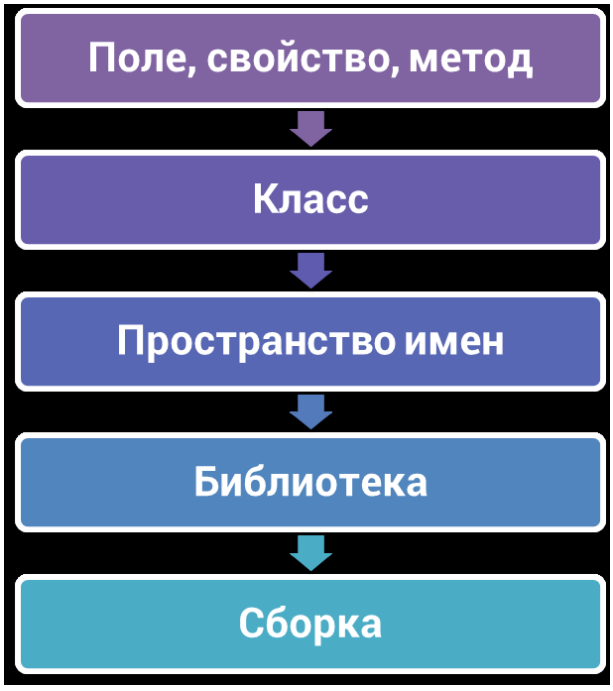


Рис. 1.6. Язык C# обладает четкой модульной структурой кода

3. Обширная библиотека классов

Современный .NET – это мощная платформа для создания коммерческого и некоммерческого ПО. Библиотека готовых классов позволяет решает широкий спектр задач разработки. Это не только вычислительные возможности, но и инструменты для создания интерфейсов приложений.

4. Обширное использование

C# является удобным языком для обучения программированию, в решении личных задач и использовании в разработке ПО для массового пользователя. Язык прекрасно подойдет в качестве альтернативы классическому Pascal, C++, Delphi (Lazarus), минималистичному Python и позволит сформировать более фундаментальное понимание принципов разработки современного ПО.

5. Наличие эргономичного инструментария

C# тесно связан с мощной средой разработки Visual Studio, которая предлагает разнообразные инструменты не только для разработки, но и обучения: от всплывающих подсказок кода до отладки и управления проектами. Возможность работать с профессиональным инструментом существенно повышает интерес учащегося и мотивацию к изучению программирования.

6. Поддержка сообщества

.NET и C# поддерживает и развивает большое сообщество профессиональных разработчиков, поэтому язык активно развивается. На специализированных форумах новички могут получить помощь и консультацию по возникающим проблемам.

Кроме того, по C# доступен большой объем учебной и справочной литературы, прежде всего – в формате веб-ресурсов.

7. Схожесть с другими языками

Поскольку базовый синтаксис C# очень похож и даже частично идентичен популярным языкам C++, Java, JavaScript, то учащиеся без особых трудностей смогут освоить C#, либо наоборот – изучать перечисленные языки в сравнении с C#.

8. Перспективность

C# также востребован и на рынке труда. Специалисты с хорошим знанием и опытом работы востребованы для разработки программного обеспечения, веб-сервисов, мобильных приложений, игр.

Это полезно знать!

Интересный факт: Pascal ABC.NET как учебная среда, реализован на базе платформы .NET Framework. А сам язык поддерживает синтаксис классов и объектов .NET.
В этом курсе вы увидите, что Си-подобный синтаксис C# короче предложенного в Pascal ABC и удобен при создании классов как абстракций ООП.

Вопросы для самопроверки

1. Почему навыки программирования крайне важны в формировании личности будущего специалиста сферы IT?
2. Что представляет собой платформа .NET и какие возможности она реализует?
3. Опишите ключевые компоненты .NET.
4. Что такое виртуальная машина CLR?
5. Какие основные этапы развития прошла технология .NET?
6. В чем преимущества языка C#?
7. Перечислите и кратко опишите достоинства использования C# в качестве инструмента обучения программированию.

Практикум

1. Проверьте, установлен ли .NET Framework на вашем ПК.
2. Уточните версию компонент .NET: для этого откройте *Панель управления* (рис. 1.7) и зайдите в список *Программы и компоненты* (рис. 1.8).
3. В качестве отчета приложите скриншоты.

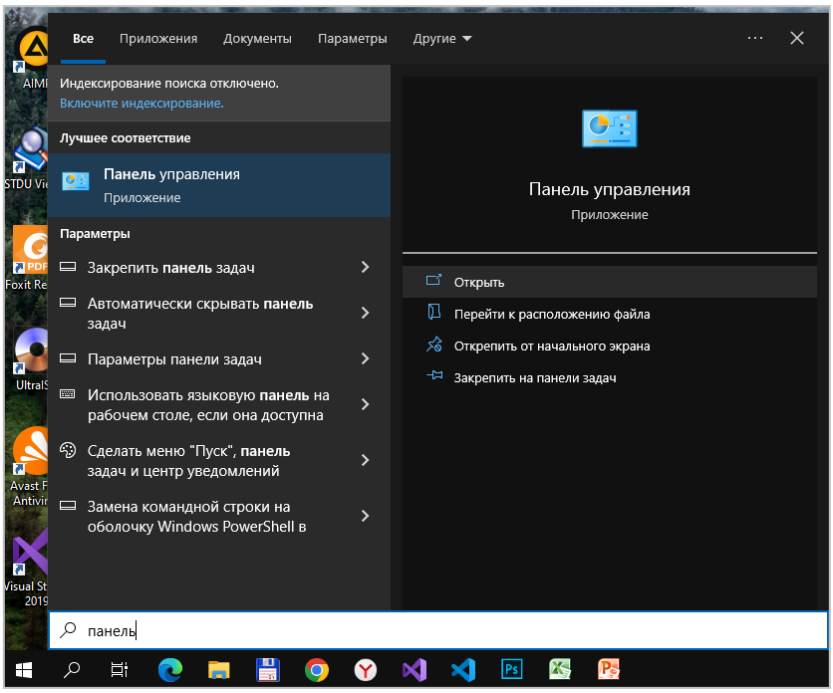


Рис. 1.7. Панель управления операционной системой

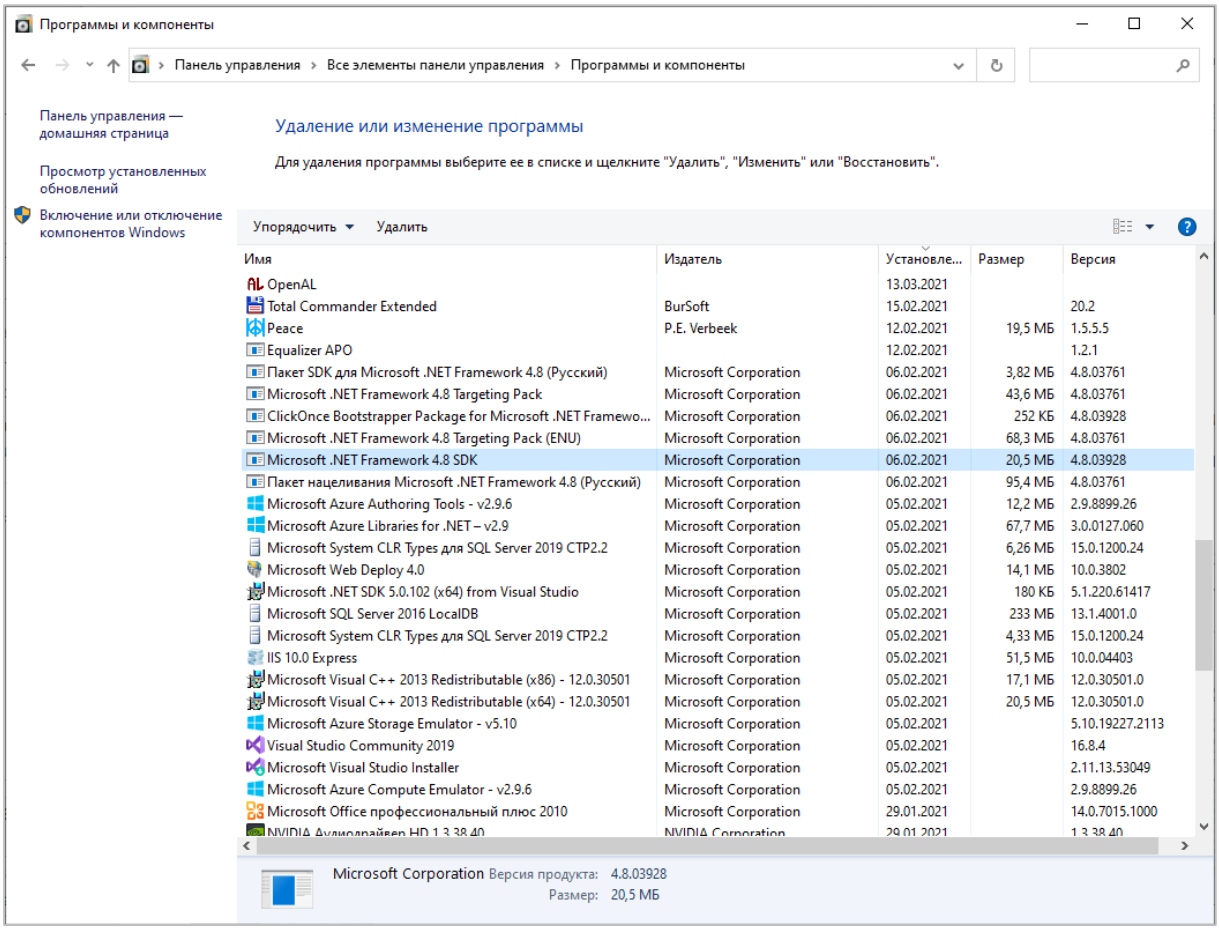


Рис. 1.8. Список установленных приложений

1.2. Инструменты разработчика .NET

1.2.1. Интегрированная среда разработки

Понятие и назначение

Определение

Интегрированная среда разработки (IDE) *– инструментальное программное обеспечение, предназначенное для разработки ПО.*

Интегрированная среда разработки позволяет разработчикам работать на проектами разного типа сложности, предоставляя им комплекс инструментов для написания и отладки кода, проектирования визуального интерфейса приложений, тестирования, сборки и развертывания проектов, анализа производительности. IDE автоматизирует рутинные задачи и предоставляет разработчику удобный интерфейс для работы с кодом и проектом.

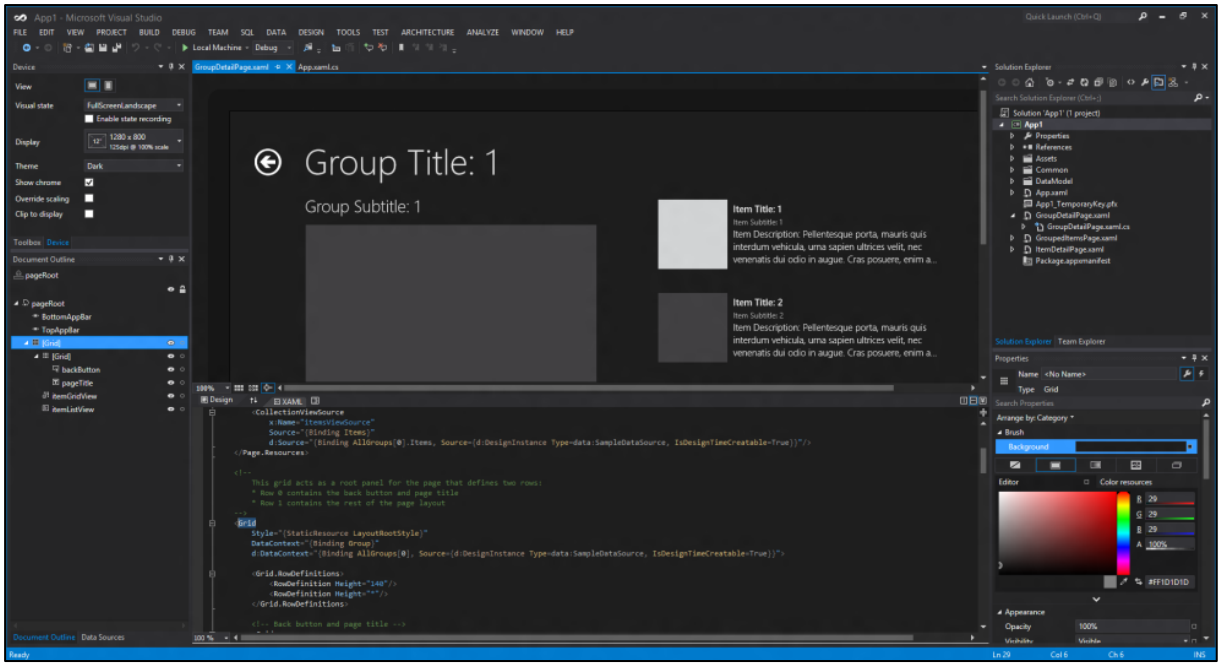


Рис. 1.9. Интерфейс IDE Visual Studio

Состав среды разработки

Продвинутая среда разработки позволяет осуществлять полный цикл разработки программы. Однако возможности сред отличаются.

Выделим наиболее значимый функционал IDE.

1. Текстовый редактор

Предназначен для комфортного написания и редактирования исходного кода программы. Редактор способен подсвечивать синтаксис, выдавать подсказки, осуществлять автоматическое дополнение, содержать множество полезных в работе функций.

2. Транслятор

В зависимости от языка программирования это *компилятор* или *интерпретатор* – программа, которая преобразует высокоуровневый или промежуточный код в низкоуровневые команды, которые исполняет процессор.

3. Отладчик

Программа-отладчик позволяет пошагово выявлять ошибки выполнения программы и следить за содержимым данных и исправлять их по мере необходимости.

4. Средства автоматизации сборки

Упрощают управление сборкой множества файлов проекта в единое приложение.

5. Конструктор визуального интерфейса

Конструктор визуального интерфейса (GUI-дизайнер) – это инструмент, позволяющий проектировать графический интерфейс приложения с помощью перетаскивания компонентов и настройки их свойств.

6. Средства автоматического тестирования

Набор инструментов для организации комплексного тестирования функционала приложения и корректности его работы.

7. Профайлер

Инструмент, анализирующий производительность кода и визуализирующий его в диаграммах и отчетах.

8. Система контроля версий

Позволяет контролировать изменения в программном продукте на всех этапах разработки и работать над проектом в команде. Обычно это интеграция с системами Git, Mercurial или Subversion.

Среды разработки под C#

1. Visual Studio

Visual Studio – интегрированная среда разработки программного обеспечения, разработанная компанией Microsoft.

В настоящий момент Visual Studio является наиболее многофункциональной средой разработки приложений на языке C#:

- позволяет разрабатывать приложения с консольным и графическим интерфейсом;
- включает многофункциональный редактор кода с поддержкой технологии IntelliSense (автоподсказка и автозавершение кода одним нажатием клавиши);
- поддерживает развитый механизм сборки проектов;
- доступны платные (VS Professional) и бесплатные (VS Community) версии.

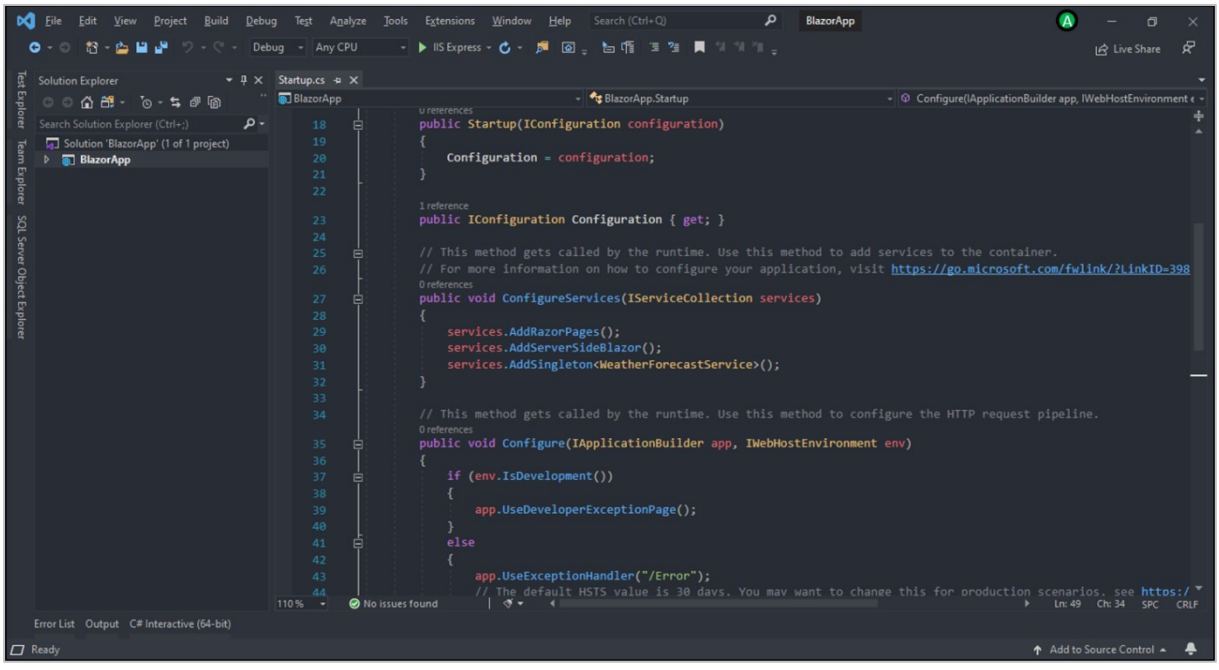


Рис. 1.10. Интерфейс среды разработки Visual Studio

2. *SharpDevelop*

SharpDevelop – легковесная среда разработки под C# и некоторые другие языки программирования. Используется как бесплатная альтернатива Visual Studio.

Однако развитие проекта остановилось в 2016 году, поэтому разработчику придется довольствоваться лишь возможностями старых версий .NET Framework 4.5 и C# 5. Среда:

- включает многофункциональный редактор кода и поддерживает технологию IntelliSense;
- поддерживает автоматическую сборку проектов;
- содержит конструктор приложений с визуальным интерфейсом;
- распространяется свободно;
- потребляет незначительный ресурс ПК.

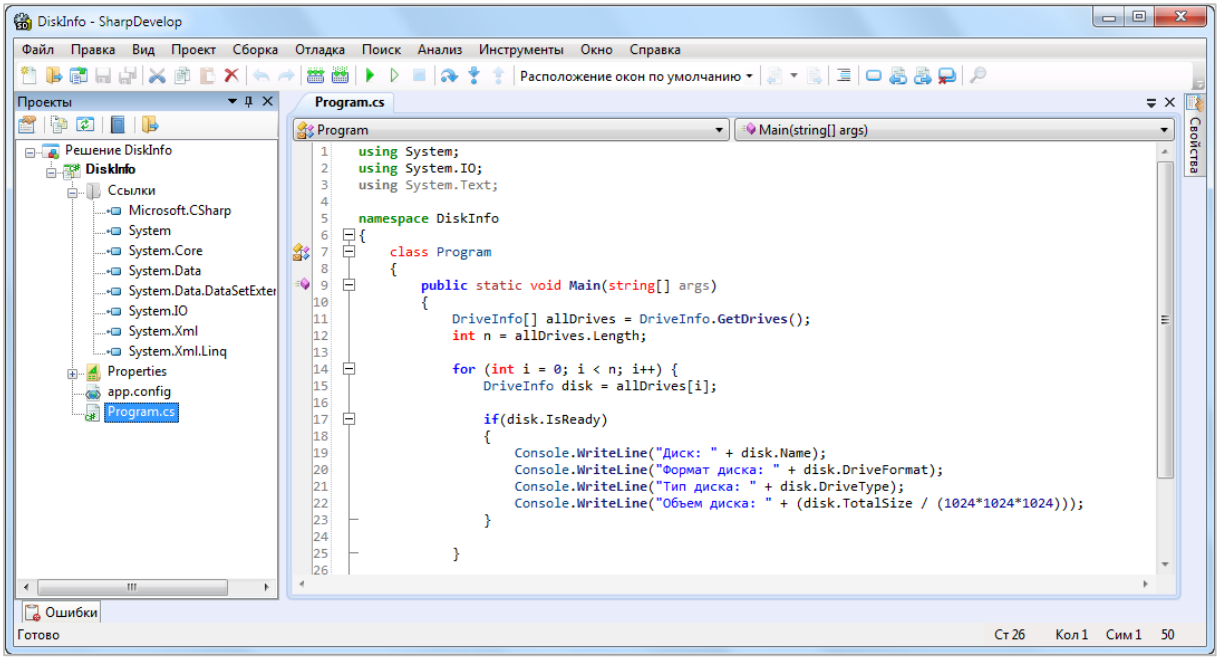


Рис. 1.11. Интерфейс среды разработки SharpDevelop

3. *MonoDevelop*

MonoDevelop – среда разработки приложений на языках C#, Java, C++ и др. Ориентирована, прежде всего, для пользователей Linux.

MonoDevelop можно использовать на операционных системах Linux, macOS и Windows. Среда имеет открытый программным код, позволяет разрабатывать проекты, использующие платформы Mono и .NET. Mono (реализует возможности .NET Framework на базе свободного программного обеспечения). Включает в свой состав:

- многофункциональный редактор кода с поддержкой технологии IntelliSense;
- отладчик;
- поддержку плагинов.

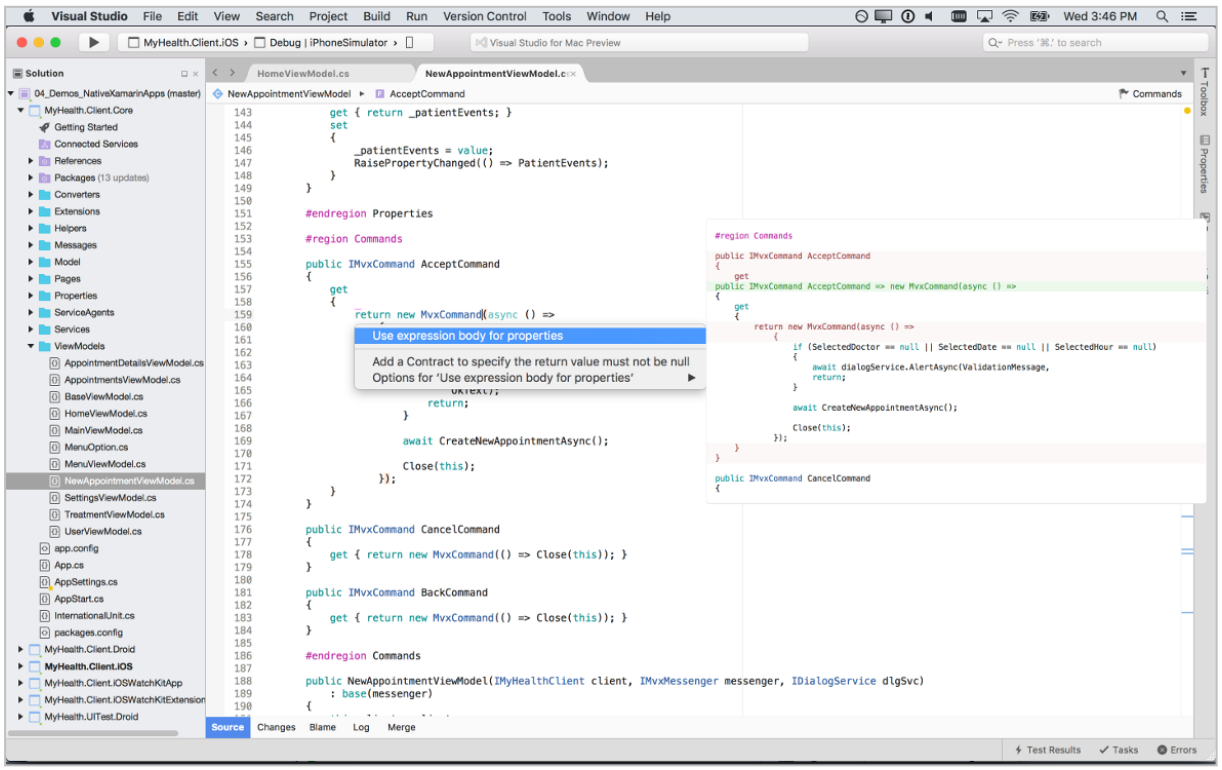


Рис. 1.12. Интерфейс среды разработки MonoDevelop

4. *JetBrains Rider*

JetBrains Rider – это коммерческая кроссплатформенная интегрированная среда разработки программного обеспечения для платформы .NET, разрабатываемая компанией JetBrains.

Позиционируется в качестве основного конкурента Visual Studio:

- позволяет разрабатывать приложения с консольным и графическим интерфейсом;
- включает многофункциональный редактор кода с поддержкой технологии IntelliSense.
- поддерживает развитый механизм сборки проектов;
- содержит инструменты для анализа, отладки, рефакторинга и тестирования программного кода.

В связи с санкциями по состоянию на 2025 г. JetBrains прекратили поддержку своих продуктов для пользователей РФ.

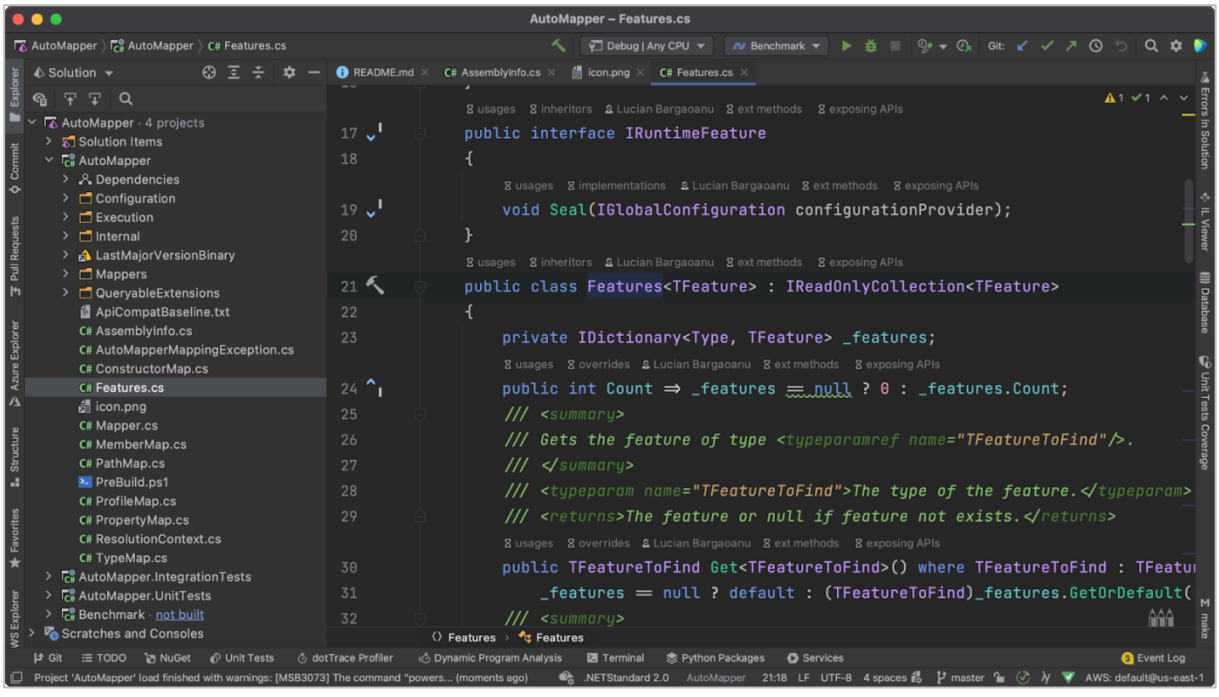


Рис. 1.13. Интерфейс среды разработки JetBrains Rider

1.2.2. Текстовый редактор

Текстовые редакторы как альтернатива IDE

Во многих учебных языках программирования среда разработки непосредственно привязана к компилятору, чтобы упростить работу (например, Pascal ABC).

Однако в общем случае это не требуется. Язык программирования (его компилятор) и фреймворки могут распространяться свободно, а среда разработки являться платной. Разработчик настраивает среду согласно своим предпочтениям.

Вместо среды разработки может использоваться и продвинутый текстовый редактор (например, Vim, Emacs, Sublime Text, Visual Studio Code). Их возможности зачастую не сильно уступают средам разработки, при этом они менее требовательны к ресурсу компьютера. Однако работа с таким приложением может потребовать от программиста более глубокого понимания принципов разработки приложений.

Редактор Visual Studio Code

Определение

Visual Studio Code *– это бесплатно распространяемый профессиональный текстовый редактор, поддерживающий разработку на большинстве популярных языков программирования.*

Visual Studio Code является одним из лучших бесплатных текстовых редакторов, поддерживающий разные языки программирования. При должной настройке он превращается в мощный инструмент разработки, близкий по своим возможностям к Visual Studio. Среди которых:

- многофункциональный редактор кода с поддержкой технологии IntelliSense;
- настройка дополнительных возможностей редактора посредством расширений;
- хорошая оптимизация (по сравнению со средой разработки).

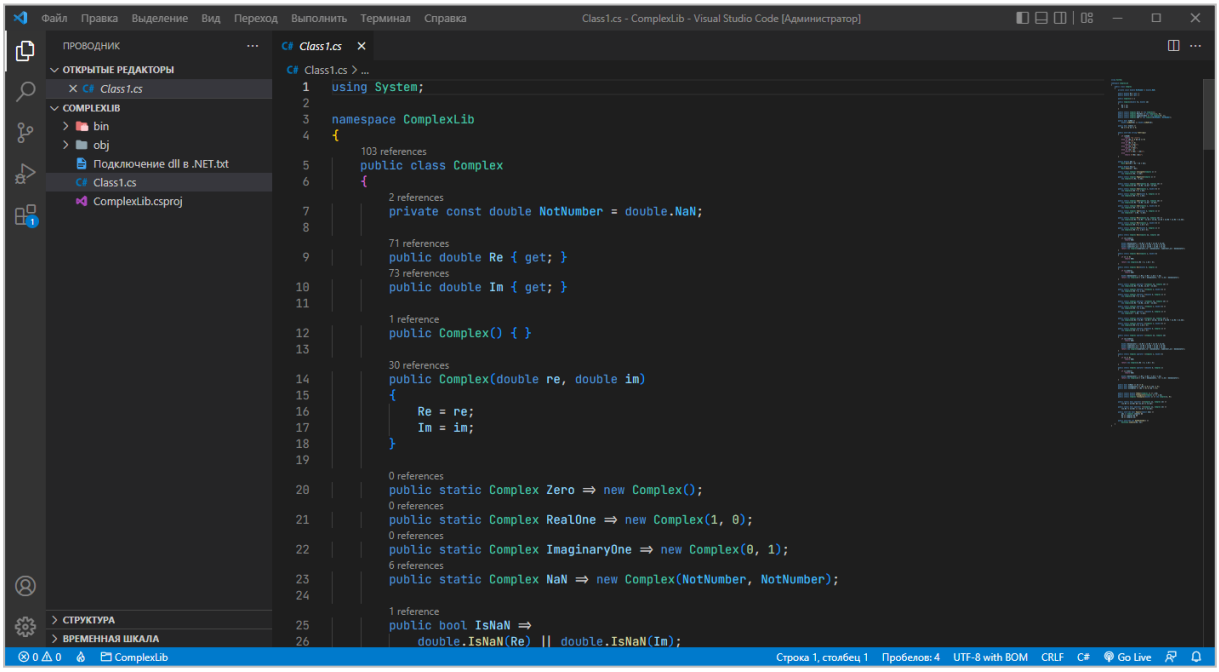


Рис. 1.14. Интерфейс текстового редактора Visual Studio Code

Вопросы для самопроверки

1. Что представляет собой интегрированная среда разработки и какие основные задачи она позволяет решать?
2. Опишите основные компоненты среды разработки.
3. Какими средами можно пользоваться для разработки на языке C#?
4. Чем отличаются возможности текстовых редакторов от возможностей сред разработки? А что общего?
5. Какими текстовыми редакторами вы пользуетесь?

1.3. Установка необходимого дистрибутива

1.3.1. Необходимые для работы программы

Важное замечание!

Дальнейшая работа будет демонстрироваться с использованием редактора Visual Studio Code: в отличие от среды Visual Studio он требует более скромный ресурс ПК. И для изучения основ языка и реализации учебных проектов возможностей этого редактора будет достаточно.

Однако читателю для самостоятельной работы желательно также установить бесплатную среду Visual Studio Community с минимально необходимым набором нагрузок для программирования на C#.

Для работы с языком C# в редакторе Visual Studio Code нам потребуется:

- платформа .NET;
- сам реактор Visual Studio Code;
- некоторые плагины (их устанавливают из редактора).

1.3.2. Установка .NET SDK

1. Установка библиотек .NET Framework (необязательно)

В первую очередь необходимо установить последние версии фреймворка .NET 4.8.2: в него входят компилятор и библиотеки классов.

Важно заметить: скачивание необходимо вести с официального сайта Microsoft: https://support.microsoft.com/.

.NET поставляется в двух вариантах:

- *Runtime* – фреймворк для выполнения приложений под .NET, отвечает только за работу приложений, разработанных под .NET;

- *SDK* – фреймворк как для выполнения программ, так и для разработки под .NET, предоставляет дополнительные возможности разработчику.

.NET Framework 4.8 необходим, прежде всего, для редактора Visual Studio Code. Здесь достаточно установить Runtime-пакет (но рекомендуется SDK).

Ссылка на скачивание:
https://dotnet.microsoft.com/en-us/download/dotnet-framework/net48

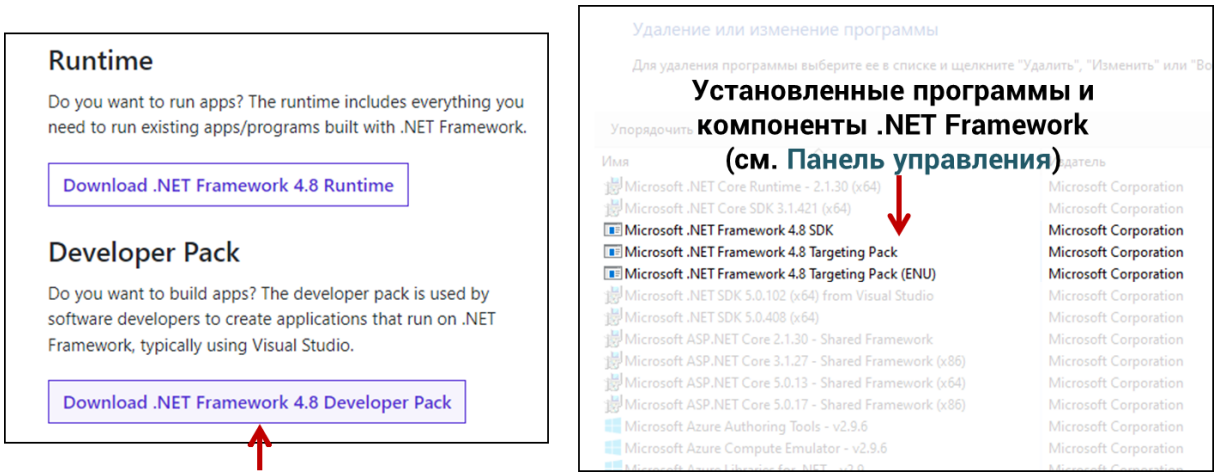


Рис. 1.15. Установка .NET Framework 4.8

Важное замечание!

Для последних версий редактора Visual Studio Code установка .NET Framework 4.8 больше не требуется. Кроме того, операционная система Windows 11 изначально его поддерживает. Описанная процедура актуальна для более ранних операционных систем.

2. Установка .NET

На следующем шаге необходимо установить *.NET 8* или *.NET 9*: именно его библиотеки и компилятор далее будут использоваться для компиляции программ. В отличие от предыдущего шага, здесь обязательно необходимо выбрать SDK-пакет!

Ссылка на скачивание одного из пакетов (на выбор):
https://dotnet.microsoft.com/en-us/download/dotnet/8.0
https://dotnet.microsoft.com/en-us/download/dotnet/9.0

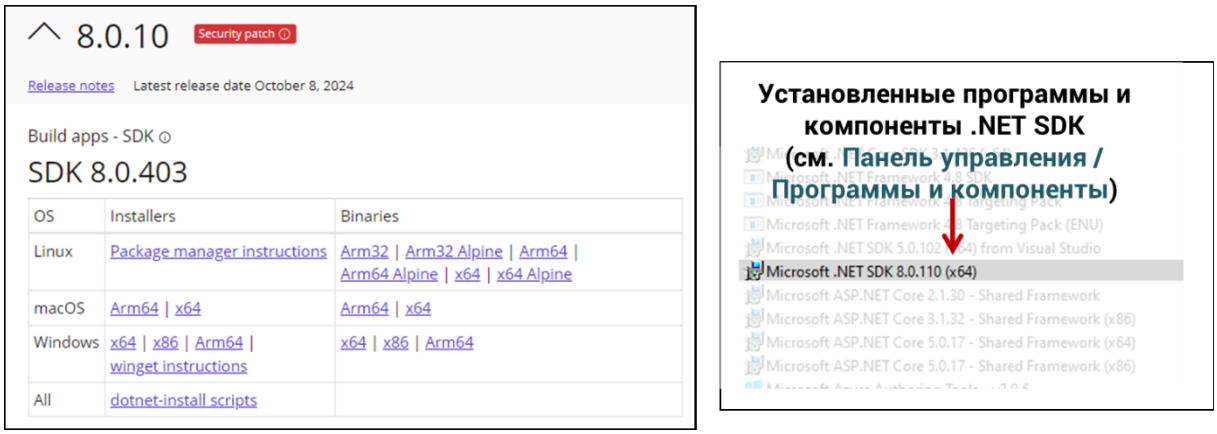


Рис. 1.16. Установка .NET 8 (или 9)

1.3.3. Установка Visual Studio Code

Процедура установки

Перейдите официальный сайт поставщика Visual Studio Code и скачайте последнюю версию редактора:

https://code.visualstudio.com/

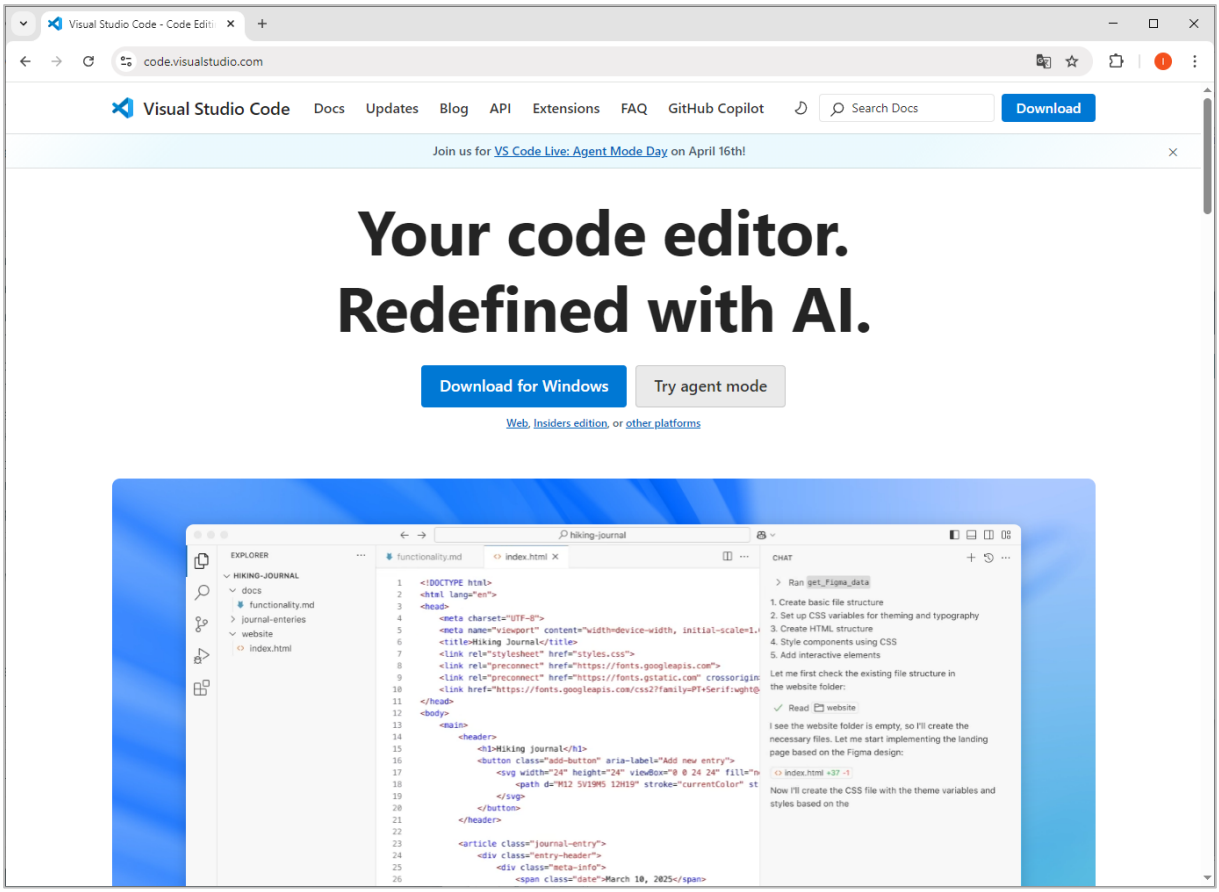


Рис. 1.17. Скачивание редактора Visual Studio Code

Запустите инсталлятор. Осуществите указанные действия до появления окошечка *Выберете дополнительные задачи*:

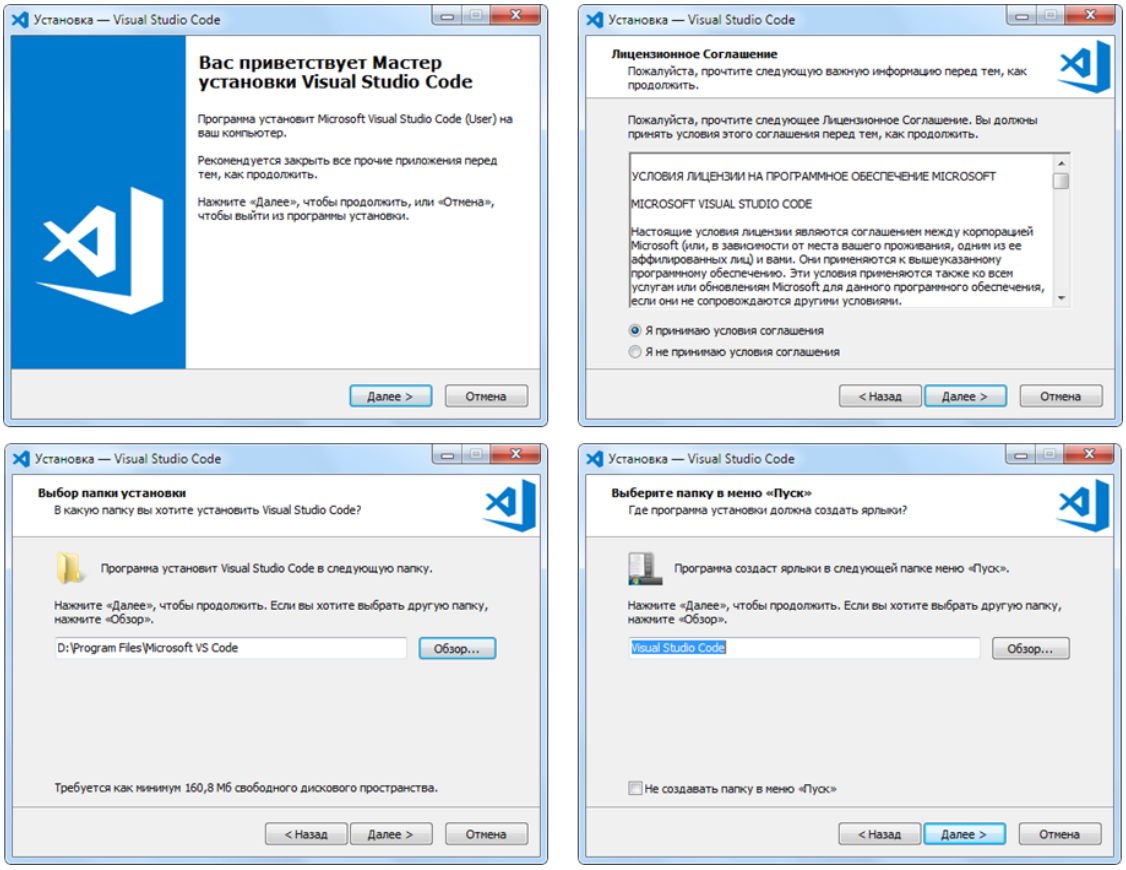


Рис. 1.18. Начало установки редактора Visual Studio Code

Установка первых двух галочек внесет редактор в контекстное меню *правой кнопки мыши* (*ПКМ*), что позволит открывать любые текстовые файлы с помощью Visual Studio Code:

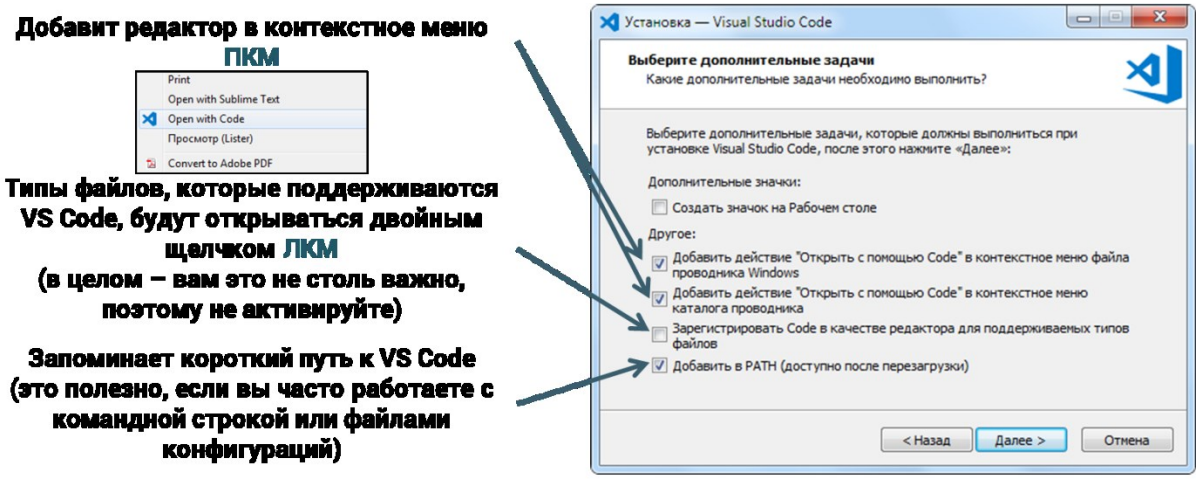


Рис. 1.19. Активация параметров вызова редактора

Далее начнется процесс установки. После его завершения можно открыть редактор:

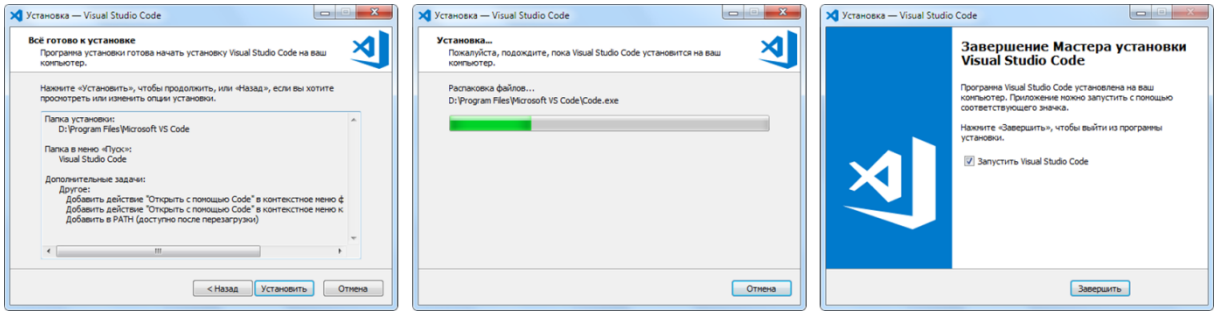


Рис. 1.20. Установка редактора на жесткий диск

Основные компоненты редактора

Проводник

Проводник позволяет оперативно ориентироваться по файлам проекта. Файлы в открытом каталоги можно менять, при этом изменения производятся над соответствующими файлами, хранящимися а жестком диске.

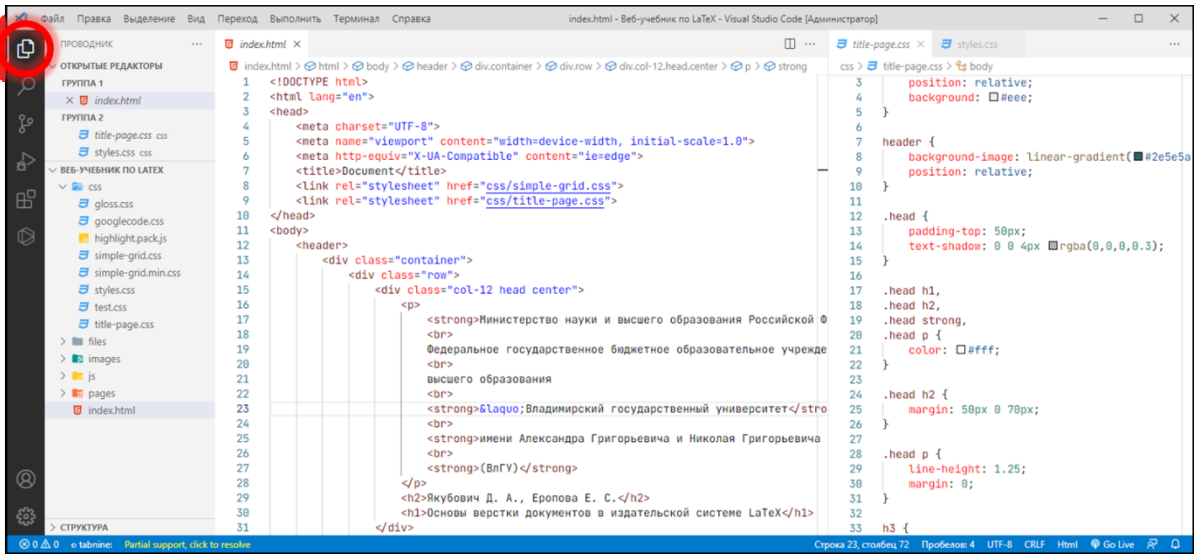


Рис. 1.21. Проводник по структуре проекта

Поиск

Поиск поможет найти текст и заменить его во всех файлах.

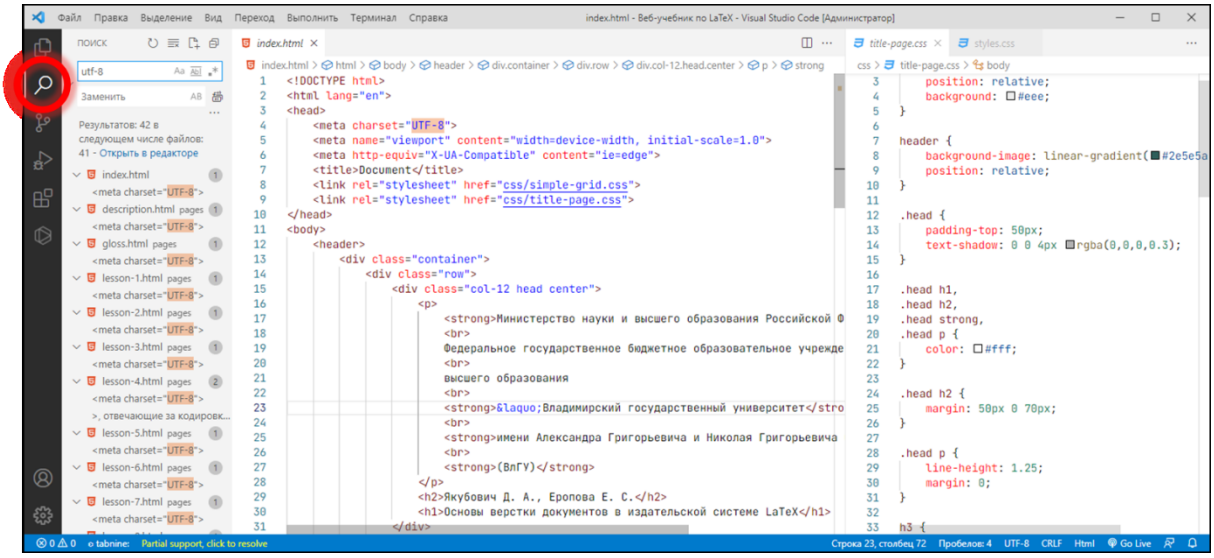


Рис. 1.22. Поиск и замена найденных соответствий в тексте документа

Расширения

Расширения – дополнительные плагины, расширяющие функционал редактора. Их скачивание производится из сети интернет.

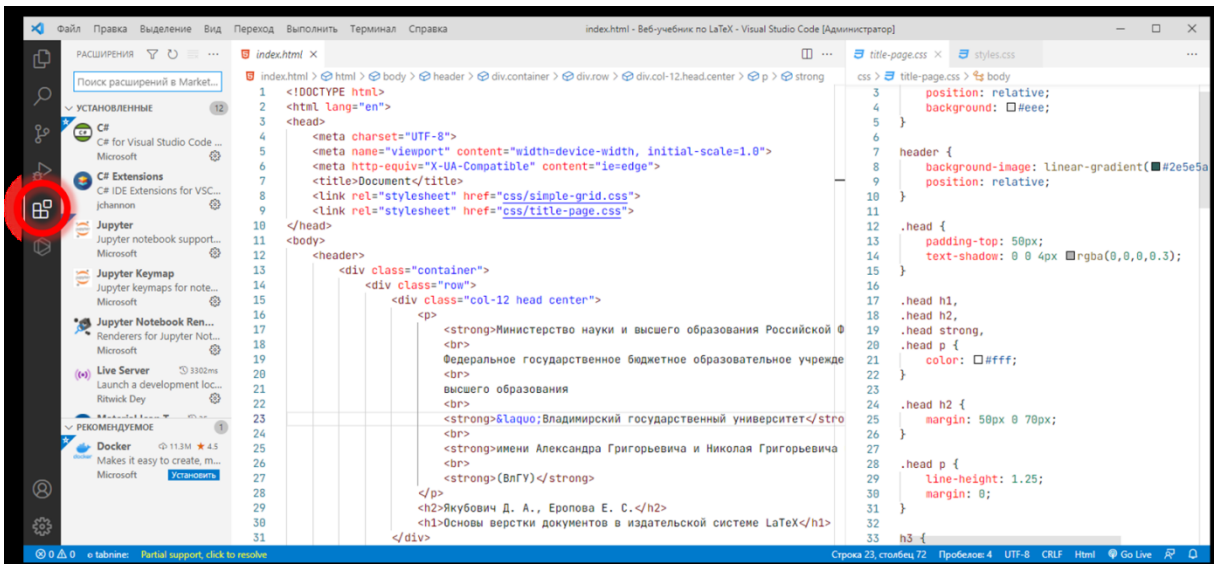


Рис. 1.23. Расширения редактора

Установка расширений для работы с C#

В отличие от среды Visual Studio, редактор Visual Studio Code для работы с C# требует установки ряда расширений.

Первоначально можно установить руссификацию интерфейса редактора. Для этого нажмите на кнопку *Расширения*, в поисковой строке наберите «russian» и выберете плагин *Russian Language Pack for Visual Studio Code*. Нажмите на *Install* и дождитесь установки пакета (расширения скачиваются из сети Интернет). Обязательно перезагрузите редактор.

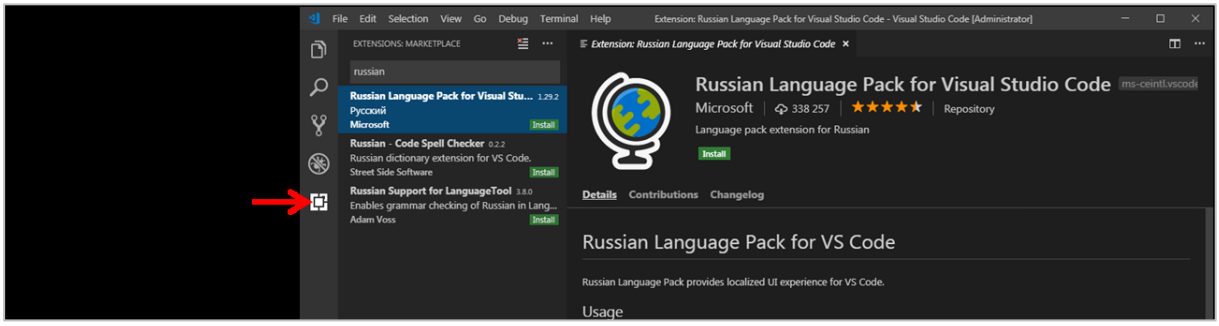


Рис. 1.24. Руссификация интерфейса редактора

Следующим шагом установите необходимый набор плагинов для работы с C#. Для этого установите расширение *C# Dev Kit*, которое также самостоятельно загрузит расширения *C#* и *.NET Install Tools* (работают совместно). Это основные расширения для программирования под .NET.

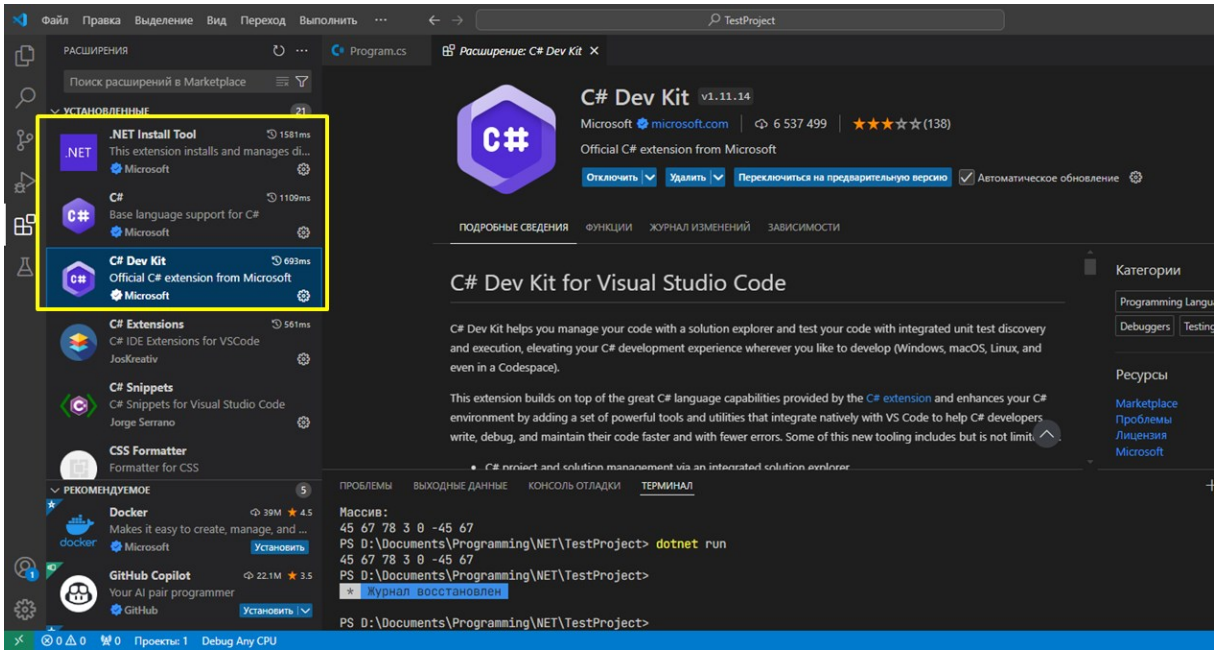


Рис. 1.25. Установка расширений для работы с .NET и C#

1.3.4. Компиляция консольного приложения

Подробное описание последовательности действий

В этом курсе мы будем работать с консольными приложениями.

1. Создание проекта

Создайте отдельный пустой каталог (папку) для хранения своих проектов. Каждый новый проект (задача практикума) далее должен быть расположен в отдельной папке.

Внутри создайте пустую папку с названием *FirstProject* и перетащите ее в область *Проводника*. Редактор создаст виртуальную рабочую область проекта.

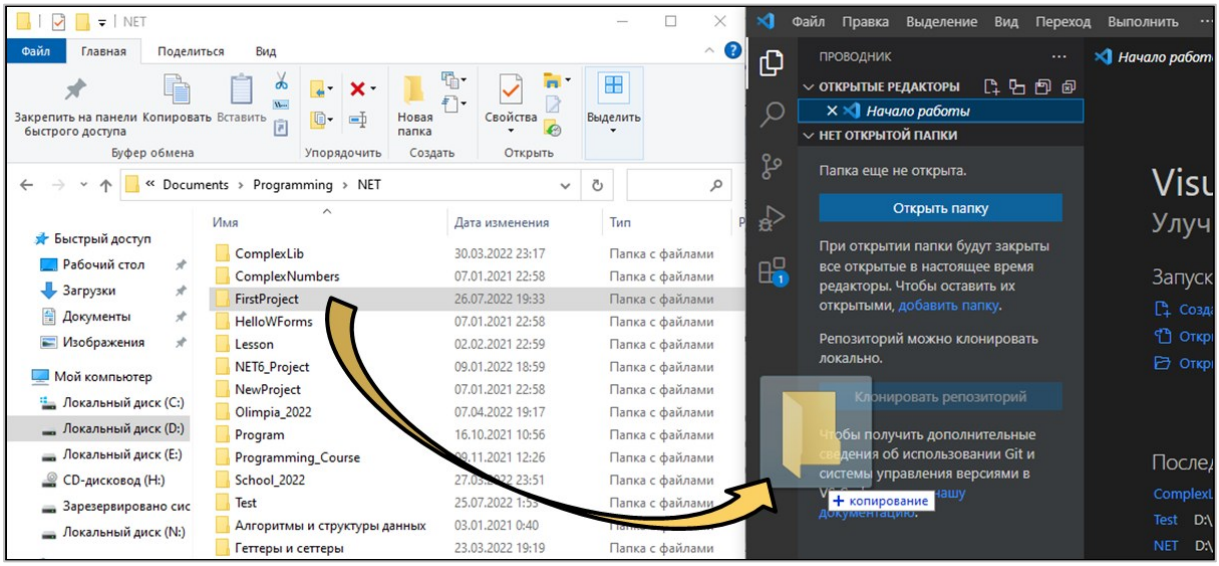


Рис. 1.26. Загрузка корневого каталога проекта

В панели меню нажмите на *Вид / Терминал*. В нижней части редактора появится окошко *терминала*, т.е. командной строки.

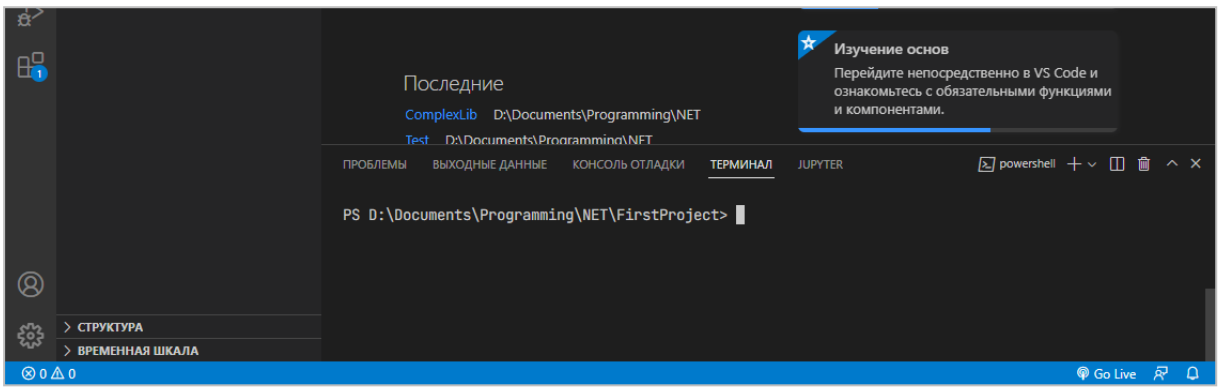


Рис. 1.27. Окно терминала

Многие операции по сборке проекта в текстовых редакторах требуют работы с командной строкой.

Чтобы создать новый проект на языке C#, напишите команду

```
dotnet new console
```

и нажмите *Enter*. Эта команда обращается к компилятору .NET и требует подготовить файлы для консольного типа проекта. Ее необходимо использовать только один раз – при создании проекта.

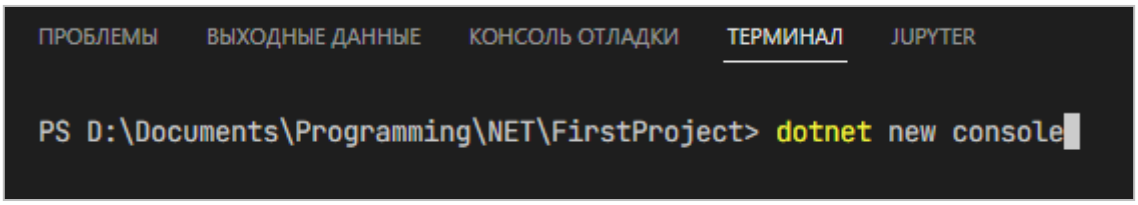


Рис. 1.28. Подготовка файлов проекта

Через некоторое время в проводнике добавится ряд новых файлов с метаданными о проекте:

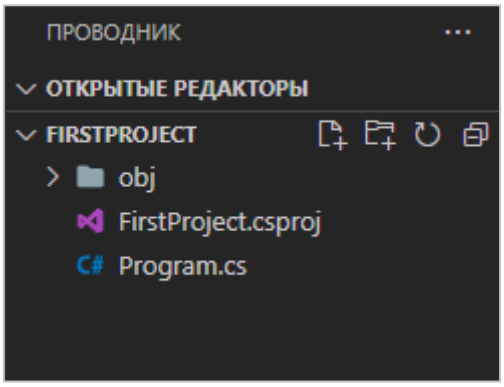


Рис. 1.29. Первоначально сгенерированные файлы проекта

Нажмите на файл *Program.cs*: в нем VSC автоматически сгенерирует стартовый шаблон консольного приложения (код шаблона будет другим, если вы работаете в современных версиях .NET). Это основной файл с программным кодом.

Подождите некоторое время. Редактор должен активировать корректную работу плагинов для языка C#.

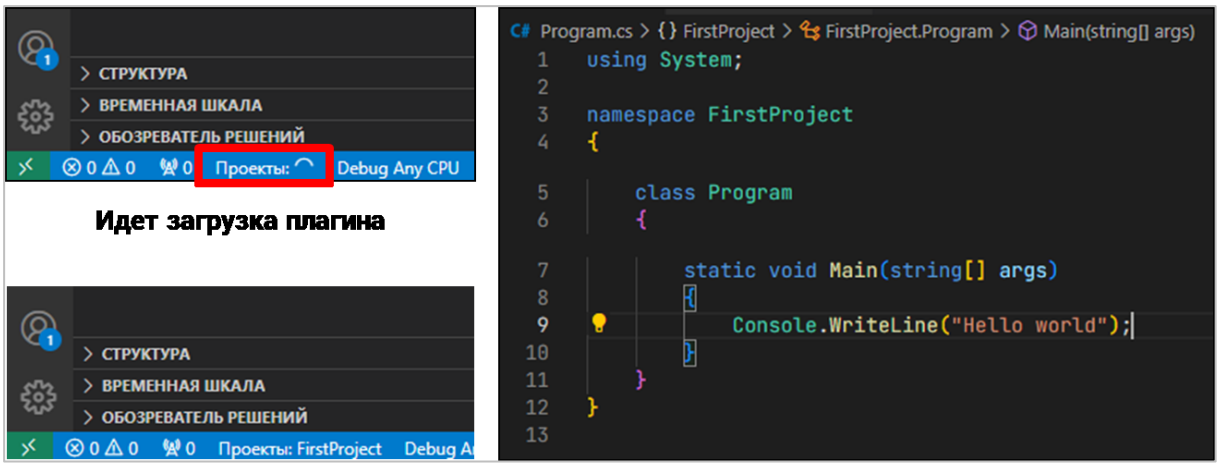


Рис. 1.30. Загрузка плагина для проекта и стартовый шаблон (необходимо дождаться загрузки модулей плагина, чтобы все функции по автоматической подсказке и дополнению кода были активированы)

Важное замечание!

Начиная с .NET 6 шаблон консольного приложения отличается от приведенного выше. В C# 10 (и более поздних версиях) используются операторы ***верхнего уровня****, которые позволяют не писать явно класс* **`Program`** *и метод* **`Main`***, а сразу реализовывать его тело.*
Однако мы будем использовать «классический» подход, соблюдая стандартную структуру кода. Поэтому скопируйте (наберите) код как изображено на рис. 1.30.

Параллельно с этим процессом в проводник добавится еще ряд файлов:

- в каталог **`obj`** будут записываются промежуточные файлы, генерируемые при компиляции проекта;
- в каталог **`bin`** записываются окончательные файлы сборки проекта;
- в файле с расширением **`.csproj`** хранятся метаданные о версии .NET и типе исполняемого файла.

При первом использовании расширения может потребоваться некоторое время на загрузку необходимых компонент для редактора (см. информацию на вкладке *Входные данные*).

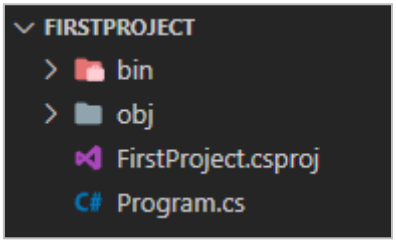


Рис. 1.31. Автоматическое добавление файлов в проект

Дополнительно редактор предложит подготовить проект под возможность отладки кода (это может быть полезно в дальнейшем): будет создан еще один каталог **`.vscode`**.

2. Компиляция и сборка проекта

Чтобы скомпилировать проект, в терминал введите команду

```
dotnet run
```

Эта команда не только компилирует, но и запускает итоговый исполняемый файл. Поскольку проект консольный, то информация выводится в терминале.

```
Восстановление выполнено.

PS D:\Documents\Programming\NET\FirstProject> dotnet run
Hello World!
PS D:\Documents\Programming\NET\FirstProject>
```

Рис. 1.32. Компиляция программы с помощью командной строки (терминала)

Если требуется только компиляция (сборка без запуска), то вводится команда

```
dotnet build
```

В случае наличия ошибок в коде процесс компиляции завершится неудачей, а информация об ошибке(ах) будет выведена в терминале.

Внесем некоторые изменения в код проекта:

Глава 1\FirstProject\Program.cs

```
using System;

namespace FirstProject
{
    class Program
    {
        static void Main(string[] args)
        {
            string myName = "Денис";

            Console.WriteLine("Привет, мир!");
            Console.WriteLine(
                $"{myName} начинает изучение языка C#"
            );
        }
    }
}
```

Сохраните файл и повторно выполните команду:

```
dotnet run
```

```
PS D:\Documents\Programming\NET\FirstProject> dotnet run
Привет, мир!
Денис начинает изучение языка C#
PS D:\Documents\Programming\NET\FirstProject>
```

Рис. 1.33. Повторная компиляция и запуск обновленной программы

Важное замечание!

Перед компиляцией не забывайте сохранять файлы с кодом, в которые были внесены изменения! Можно использовать комбинацию ***Ctrl + S****.*

Число на значке проводника покажет количество несохраненных вкладок:

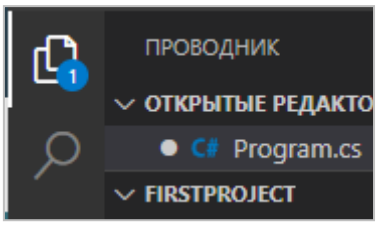


Рис. 1.34. В проекте один из файлов не сохранен после изменений

Вы также можете включить автосохранение в меню *Файл / Автосохранение*:

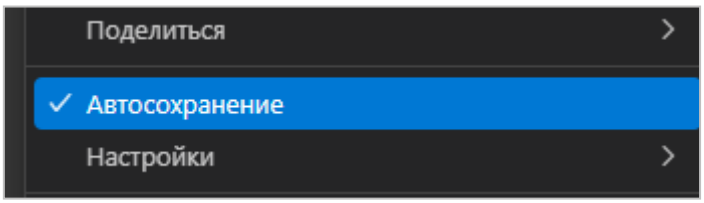


Рис. 1.35. Автоматическое сохранение файлов после изменения

3. Выгрузка проекта

Когда работа над проектом завершена, его следует выгрузить из рабочей области редактора. Для этого используйте меню *Файл / Закрыть папку*.

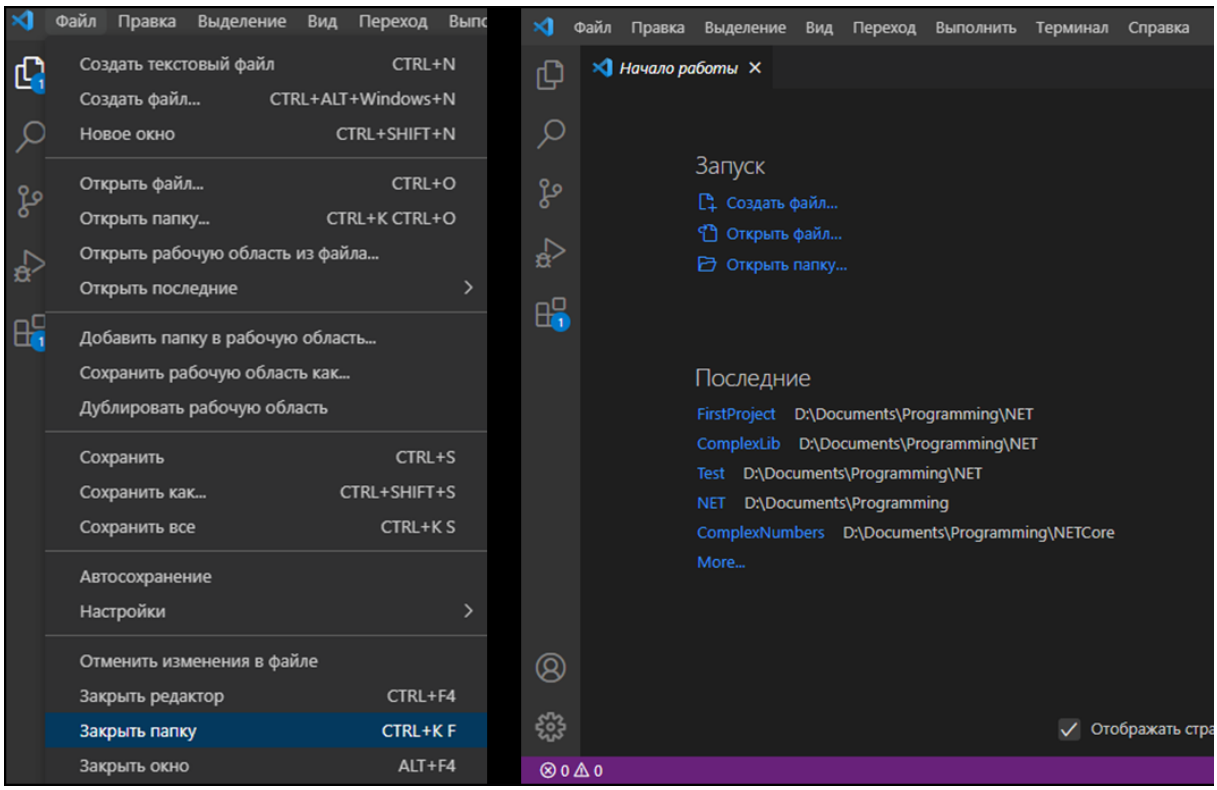


Рис. 1.36. Выгрузка каталога с проектом

Это полезно знать!

*Терминал редактора запоминает последние введенные команды. Чтобы вернуться к ним, используйте стрелки **вверх / вниз** на клавиатуре.*
Т.о. вводить команду для компиляции постоянно не требуется используйте буфер команд терминала.

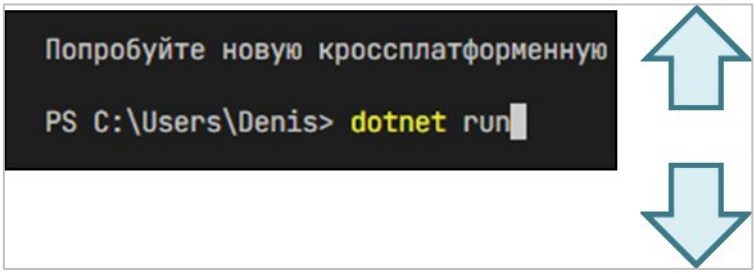


Рис. 1.37. Повтор ранее введенных в терминале команд

Обобщение алгоритма сборки консольного приложения

1. Создаем новый каталог и перетаскиваем его в *Проводник* редактора.
2. Вызываем терминал и создаем новый проект командой `dotnet new console`.
3. Редактируем код, сохраняем его.
4. Компилируем код и запускаем исполняемый файл приложения командой `dotnet run`. После любых изменений в коде повторяем шаг.
5. После завершения работы над проектом папку можно выгрузить из редактора.

Вопросы для самопроверки

1. Какое ПО необходимо для работы с C#?
2. Опишите процедуру установки .NET и Visual Studio. На какие моменты важно обратить внимание?
3. Чем отличается .NET Runtime от SDK-пакета?
4. Перечислите основные шаги при создании консольного приложения в Visual Studio Code.

Практикум

1. Установка и конфигурирование ПО

Задание 1

1. Проверьте, установлен ли на вашем ПК .NET Framework 4.8 и .NET 8-9 SDK (или предыдущие версии .NET 5-7)? Используйте список установленных программ на *Панели управления*.
2. Если не установлен, то поочередно установите фреймворки, используя указанные ссылки.
3. Протестируйте работу компилятора C#. Для этого в меню Пуск вызовите *Командную строку* и введите команду `dotnet --version`. Если компилятор успешно установлен, то отобразится его версия (рис. 1.38).
4. В качестве отчета приложите скриншоты.

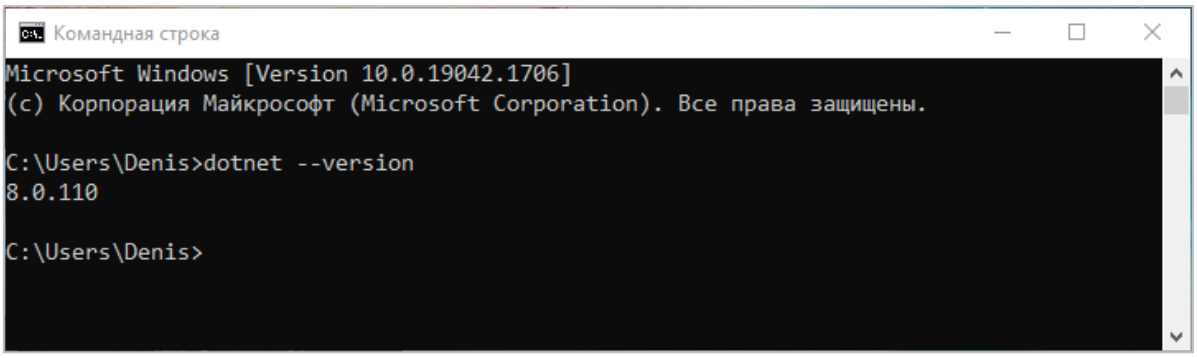


Рис. 1.38. Версия установленной платформы .NET SDK

Задание 2

1. Установите редактор Visual Studio Code.
2. Проверьте наличие обозначенных в п. 1.3.3 плагинов. В случае отсутствия их также требуется загрузить.

2. Запуск проектов

Задание 1

1. Следуя описанным инструкциям, создайте консольный проект с названием *FirstProject* и осуществите его сбоку (компиляцию и запуск). Для тестирования приведенный ранее пример.

2. После вывода текста добавьте команду `Console.ReadKey();`, которая ожидает нажатия какой-либо клавиши (задержит закрытие консоли). Не забывайте сохранять файл перед компиляцией проекта.
3. Вне редактора перейдите в каталог `bin` и найдите исполняемый exe-файл – это автономное .NET-приложение. Запустите его (рис. 1.39).
4. В некоторых случаях запуск файла может быть неуспешен или выдавать ошибку. Это может быть связано с работой вашего антивируса. Попробуйте отключить «экраны» антивируса на 10 мин и повторить процесс. Если приложение запустится, значит следует прописать путь к проектам в исключения карантина антивируса.

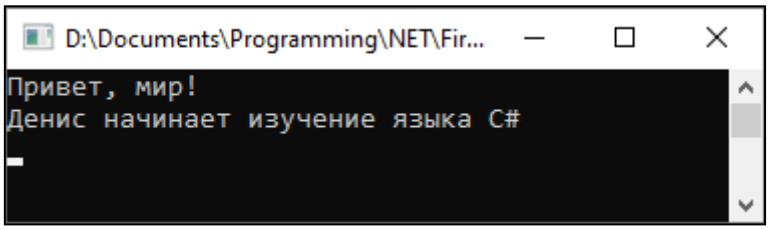


Рис. 1.39. Запуск EXE-файлы скомпилированного .NET-приложения с консольным интерфейсом

Задание 2

1. Попрактикуйтесь в использовании функции автодополнения команд (IntelliSense).
2. Начните ввод названия класса `Console`, достаточно первые несколько букв. В появившемся окне подсказки представятся команды с таким же началом. Чтобы завершить команду, нажмите *Tab* (можно *Enter* или щелчок *ПКМ*):

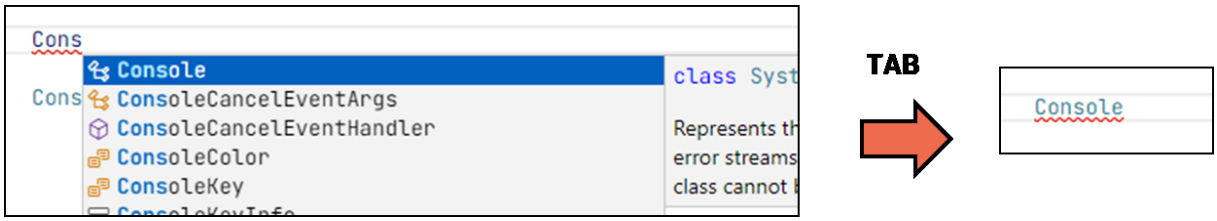


Рис. 1.40. Автозавершение команд

3. Поставьте точку. Откроется список доступных методов и свойств этого класса. По аналогии введите первые буквы

метода `WriteLine` и клавишей *Tab* завершите его. Допишите текст сообщения «Привет мир»:

4. С помощью стрелок клавиатуры можно прокручивать команды из списка автоподсказки.
5. Закрепите работу с функцией автозавершения кода, набрав следующий:

```
using System;

namespace FirstProject
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            string myName = "Денис";

            Console.WriteLine($"Привет, {myName}!");
            Console.Write("Для продолжения нажмите любую клавишу...");
            Console.ReadKey();

            Console.Clear();

            DayOfWeek today = DateTime.Now.DayOfWeek;
            int year = DateTime.Now.Year;

            Console.WriteLine($"Сегодня {today}, {year} год");
        }
    }
}
```

Рис. 1.41. Образец выполнения задания 2

Это полезно знать!

Используйте автозавершение при наборе кода. Держите мизинец или безымянный палец рядом с ***Tab****. Это не только ускорит процесс, но и существенно уменьшит частоту опечаток.*

Если подсказка вдруг пропала, нажмите ***Ctrl + Пробел****.*

Это полезно знать!

Настройка свойств редактора VSC осуществляется в меню ***Файл / Настройки / Параметры*** *или по нажатию иконку шестеренки в левом нижнем углу.*

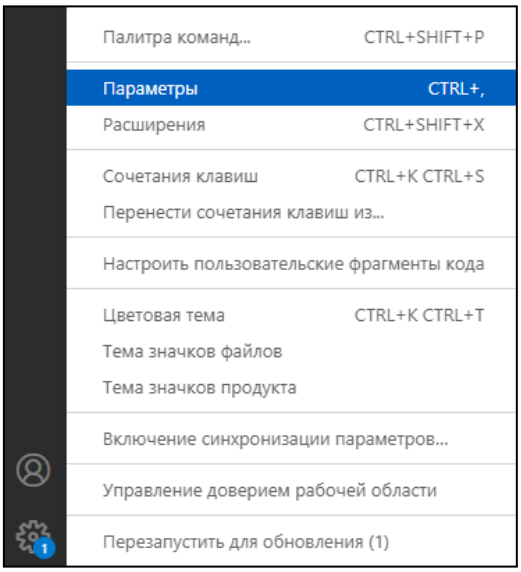


Рис. 1.42. Настройка редактора

Это полезно знать!

В том же разделе можно изменить тему оформления редактора: ***Управление / Цветовая тема****.*

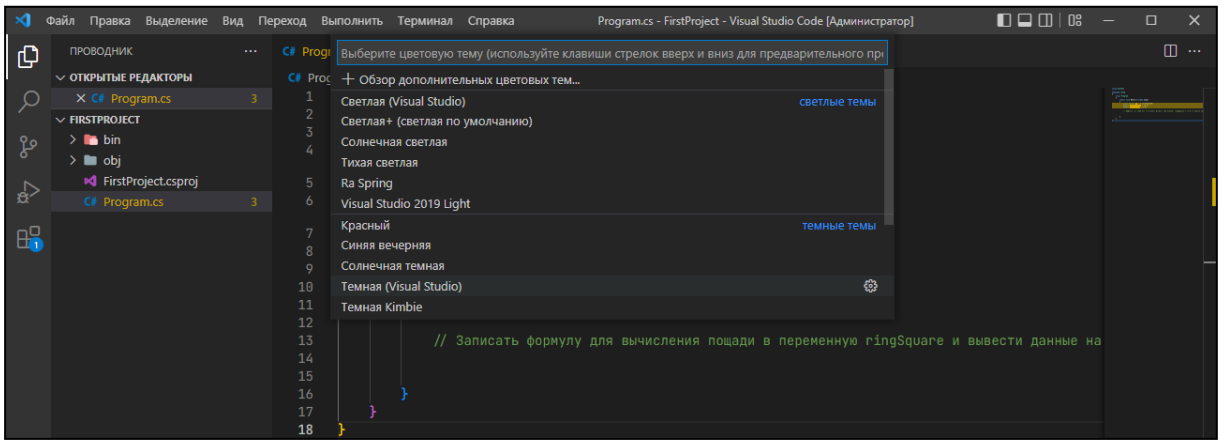
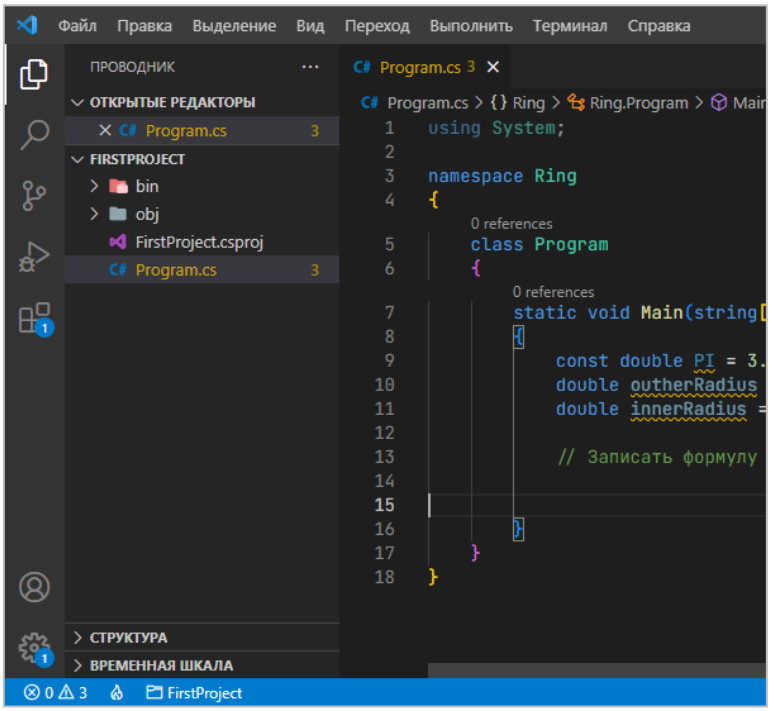


Рис. 1.43. Выбор цветовой темы редактора

Это полезно знать!

Масштабировать редактор позволяют комбинации ***Ctrl*** *+* ***+*** *и* ***Ctrl*** *+* ***-****.*



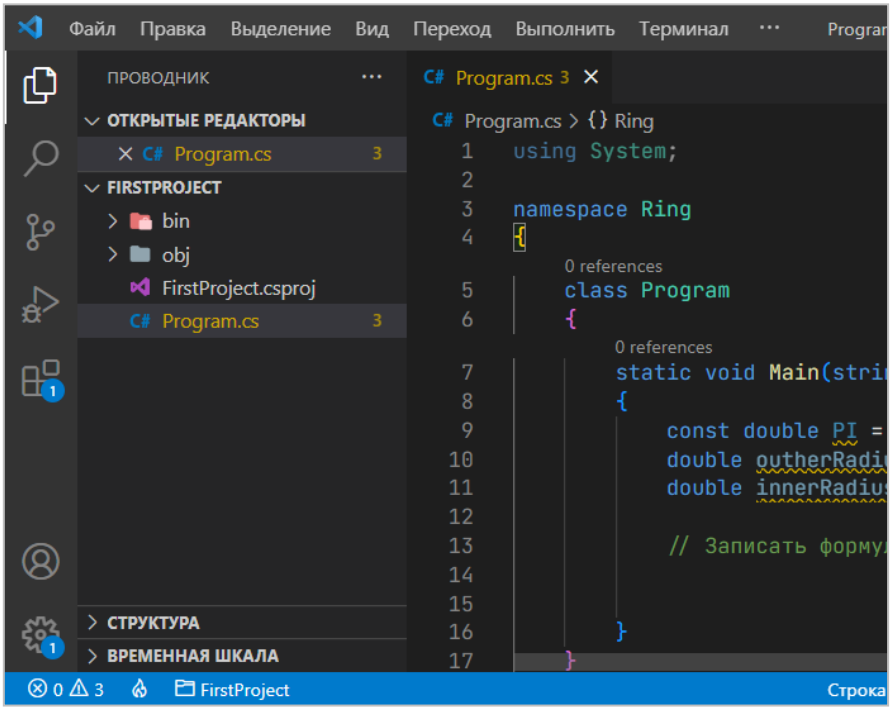


Рис. 1.44. Масштабирование окна редактора

1.4. Структура программы

1.4.1. Базовый шаблон консольного приложения

Прежде чем приступить к изучению структуры консольного приложения, рекомендуется установит еще одно расширение – **C# Snippets**. Это расширение добавляет в меню подсказки и автодополнения кода различные фрагменты часто используемого кода.

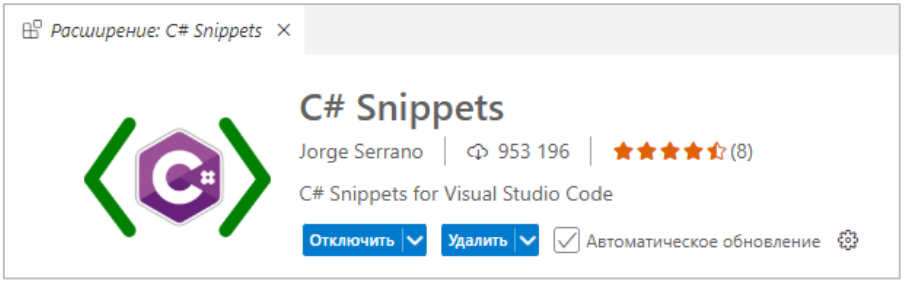


Рис. 1.45. Расширение Snippets для вставки фрагментов кода

Это полезно знать!

Сниппет *– это небольшой фрагмент кода, который можно использовать повторно в разных частях программы или в разных проектах. Обычно сниппеты создаются именно для механизма автоматизированных подсказок кода.*

В первую очередь нам потребуется сниппет, вставляющий полный («классический») стартовый шаблон консольного приложения.

Для этого введите `#` и выберете из подсказки сниппет `#helloworld`. Далее нажмите *Tab*, шаблон стартового приложения будет вставлен в редактор кода (рис. 1.46).

Можно заметить, что Visual Studio Code подчеркивает пунктирными точками название пространства имен *ConsoleApp*. Согласно рекомендациям к правилам оформления кода основное пространство должно называться по аналогии с каталогом проекта (но это не является ошибкой). Поэтому щелкните *ЛКМ* по названию *FirstProject*, далее на появившийся символ лампочки и выберете опцию *Изменить пространство имен на FirstProject* (рис. 1.47).

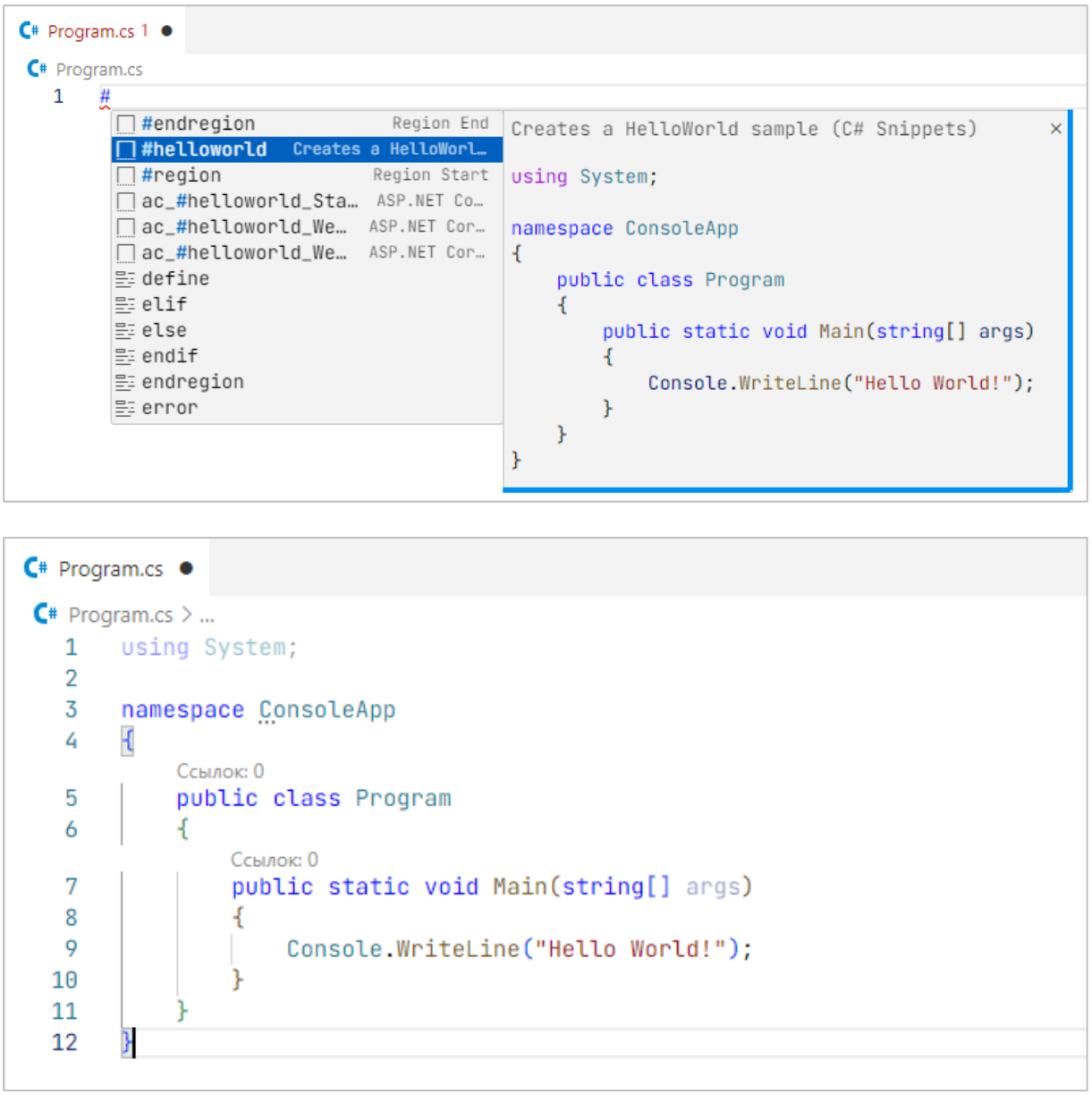


Рис. 1.46. Стартовый «классический» шаблон консольного приложения

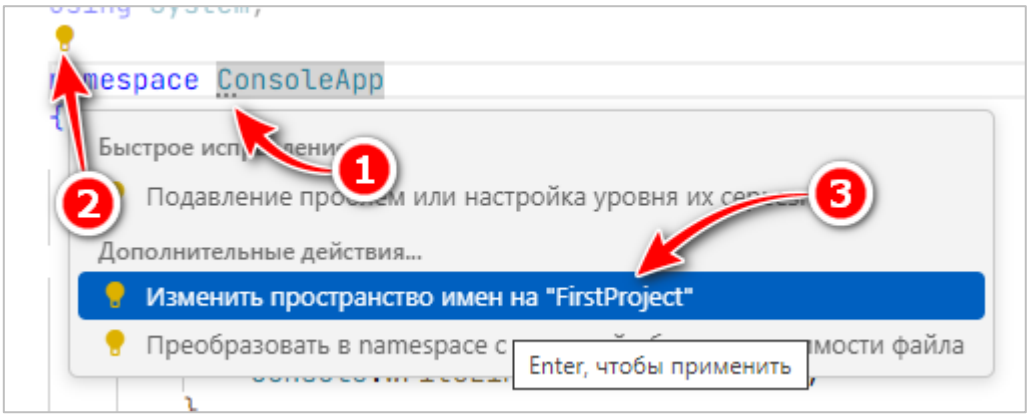


Рис. 1.47. Использование рекомендации по исправлению кода

Будем отталкиваться от следующего шаблона (внесите небольшие исправления в тело функции `Main`):

Глава 1\FirstProject\Program.cs

```
using System;

namespace FirstProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Привет, мир!");
            Console.ReadKey();
        }
    }
}
```

1.4.2. Основные компоненты в структуре консольного приложения

Общая структура

Язык C# хорошо структурирован.

«Пространство» программного кода разделено на **пространства имен**. Внутри каждого пространства имен содержатся не менее одного **класса** или иного компонента. А каждый класс также содержит определенные компоненты.

Рис. 1.48. Деление кода на логические разделы

Пространство имен

Пространство имен является именованным контейнером, объединяющим классы, структуры, интерфейсы и другие типы данных единой природы с логически связанным функционалом.

В программе пространство имен определяется командой **`namespace`**. Название можно изменить; по умолчанию оно задается как название проекта: `FirstProject`.

Пространства имен ограничивают в себе область видимости классов, что позволяет использовать классы с одинаковым названием, но разной реализацией.

```
using System;

namespace FirstProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Привет, мир!");
            Console.ReadKey();
        }
    }
}
```

Фигурные скобки

Пары фигурных скобок **`{ }`** задают **тело блока**. Их обязательно указывают, например, для пространства имен, класса, метода; также их используют условные и циклические инструкции, другие операторы.

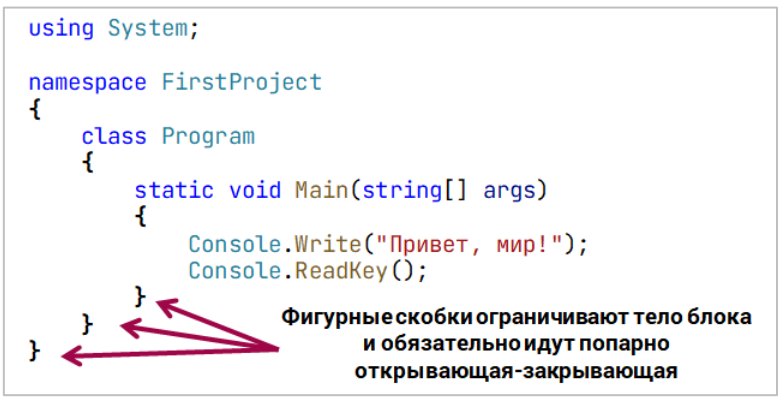


Рис. 1.49. Фигурные скобки определяют тело некоторой конструкции

Если пропустить парную скобку, то компилятор выдаст ошибку.

Конец инструкции

Оператор **;** обозначает конец инструкции. Он не ставится лишь после фигурных скобок блока.

Благодаря этому оператору можно располагать несколько инструкций в одной строке, но это считается плохим стилем оформления кода.

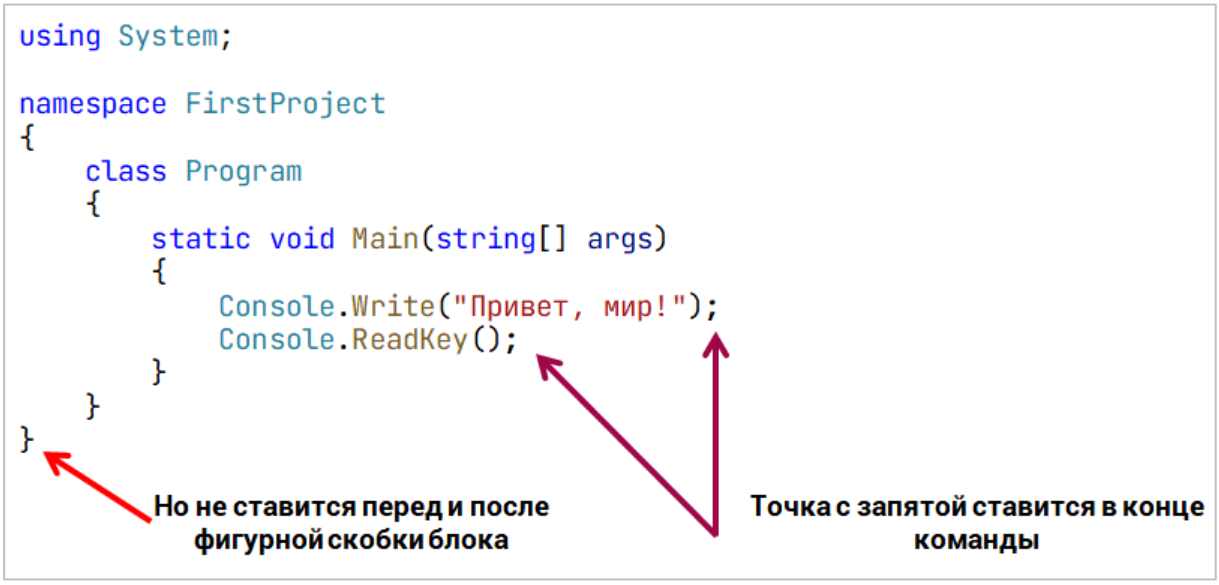


Рис. 1.50. Оператор завершения инструкции

Классы

Внутри пространств имен описываются **классы** и другие типы данных. В любой программе на C# есть хотя бы один класс. Классы – важнейшие компоненты C#, поскольку в них описаны методы для обработки данных.

Классы библиотеки .NET предоставляют уже готовые возможности для реализации ваших идей. Однако разработчик сам может создавать классы для собственных проектов.

Классы задаются командой **`class`**. В нашем примере лишь один класс; имя мы также вправе определить самостоятельно (`Program`).

```
using System;

namespace FirstProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Привет, мир!");
            Console.ReadKey();
        }
    }
}
```

Важное замечание!

C# чувствителен к регистру букв! Например, если название класса задать с маленькой буквы, то это уже будет другой класс.

Методы

В C# понятие функции и процедуры объединяется термином «**метод**». Метод всегда является частью класса, он оперирует данными и описывают некоторую логику работы. В круглых скобках указываются параметры метода, их нельзя опускать, даже если параметры не передаются.

Рис. 1.51. Методы в классе

Любая программа должна содержать особый метод **Main** – это точка входа в приложение и его название менять запрещено.

Для вывода текста на экран консоли используется метод `Write`, который описан в классе `Console`.

Метод `ReadKey` ожидает нажатия клавиши в конце программы. Это необходимо, поскольку окно консоли закрывается после выполнения всех команд (при работе с терминалом в редакторе он не требуется). Метод также описан в классе `Console`.

```
using System;

namespace FirstProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Привет, мир!");
            Console.ReadKey();
        }
    }
}
```

Методы описываются внутри классов: это отдельные модули, задача которых обработать входные данные, осуществить какое-либо преобразование или возвратить определенный результат. Методы привязаны либо к классу (как в случае с `Console`), либо к объекту, который создается на базе этого класса.

Методы можно понимать как черный ящик. Разработчику необязательно знать реализацию кода метода. Достаточно знать, что делает метод и какие параметры (данные) ему нужны для работы.

Разумеется, если вы создаете свой класс и реализуете в нем методы, то их программная реализация вам будет известна.

Рис. 1.52. Вызов метода в контексте объектно-ориентированного программирования

Оператор точка

Разграничить иерархию при вызове компонент позволяет **оператор точка**. Слева от точки находится класс или объект класса, а справа – его компонента. В одной записи может присутствовать несколько операторов точек, все зависит от иерархии пространств имен, классов и их компонент.

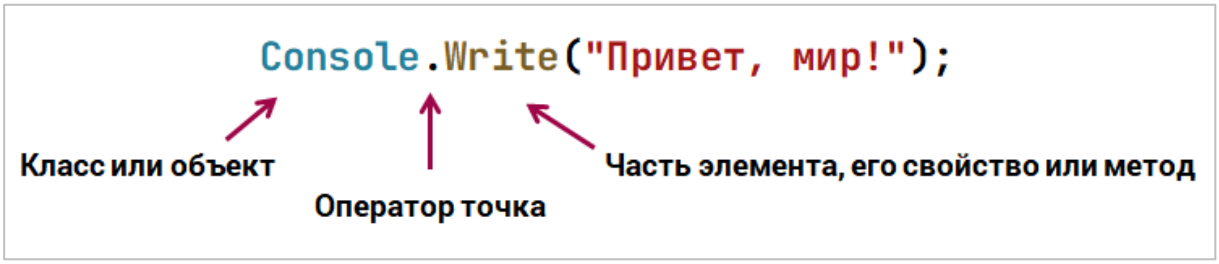


Рис. 1.53. Оператор-точка отделяет компоненты структурированных сущностей

Комментарии в коде

В C# поддерживаются два типа **комментариев**: однострочные и многострочные комментарии.

Однострочный комментарий начинается с двойной косой черты **//**. Все что после него в строке, читается комментарием и не влияет на выполнение программы

Многострочный заключается в следующих скобках **/* */**.

```
/*
   Моя первая программа
   на языке C#
*/
using System;   // Подключение пространства имен System

namespace FirstProject      // Пространство имен
{
    class Program           // Основной класс
    {
        // Точка входа
        static void Main(string[] args)
        {
            // Вывод сообщения
            Console.Write("Привет, мир!");

            // Ожидание нажатия клавиши
            Console.ReadKey();
```

```
        }
    }
}
```

Это полезно знать!

*В Visual Studio Code закомментировать фрагмент выделенного кода можно нажав комбинацию клавиш **Ctrl** + /.*
Повторная операция снимает комментарии.

Директива using

Директива **using** указывает пространство имен, к классам и компонентам которого мы хотим обращаться в короткой форме.

В примере задействуется пространство имен `System` стандартной библиотеки .NET: в нем описан класс `Console`.

```
using System;
```

По мере необходимости «подключаются» и другие пространства имен.

Использовать `using` необязательно. Однако в этом случае потребуется явно указывать, из какого пространства имен взят тот или иной класс (но иногда это необходимо).

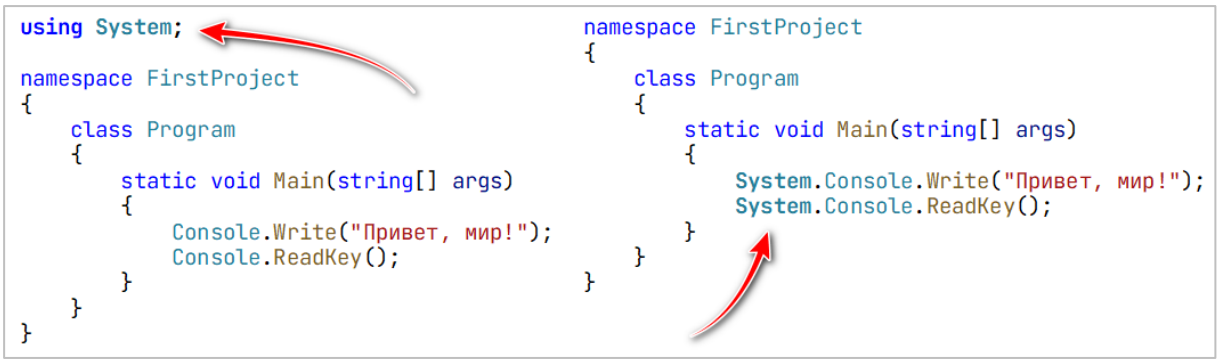


Рис. 1.54. Обе реализации равнозначны

Есть ситуации, когда пространство имен необходимо указывать обязательно. Например, в программе используется встроенный класс `Math` и написанный пользователем - `MyLib`. В этом случае компилятор не может определить, к какому из классов вы пытаетесь обратиться, возникнет ошибка:

```
Math.Abs(-2025);
Math.Abs(-2025);
```

При явном указании пространства имен обращение идет к соответствующему:

```
System.Math.Abs(-2025);
MyLib.Math.Abs(-2025);
```

Стиль оформления кода

Компилятору безразлично, как набран код: пусть даже в одну строку. Однако с помощью отступов и дополнительных пустых строк код становится более читабельным, отчетливее видна его структура, иерархия и связь элементов. Хороший стиль оформления кода формируется с опытом.

Visual Studio Code в процессе набора старается автоматически форматировать код согласно принятым традициям.

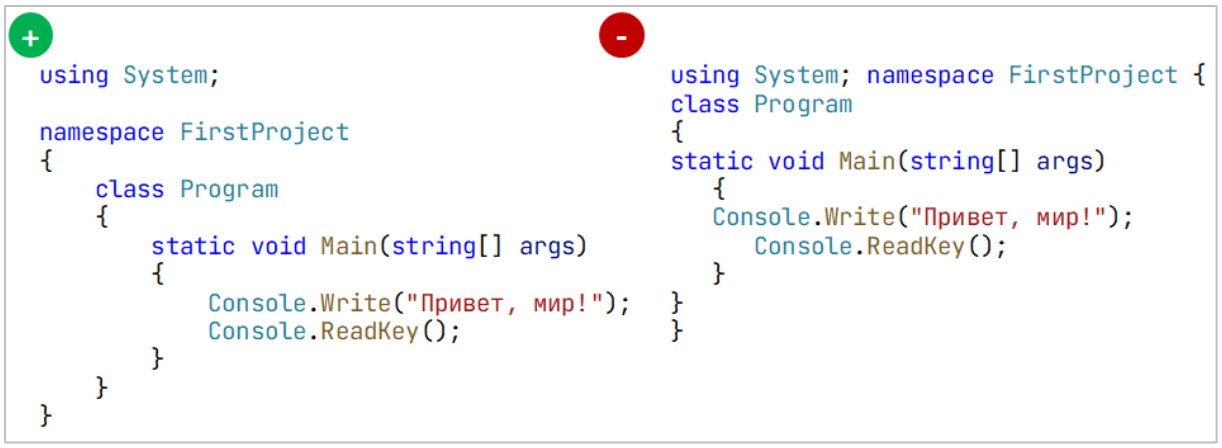


Рис. 1.55. Корректное и некорректное форматирование кода

Это полезно знать!

*Если форматирование все же «поехало», выделите весь код и воспользуйтесь опцией **ПКМ / Форматировать документ**.*

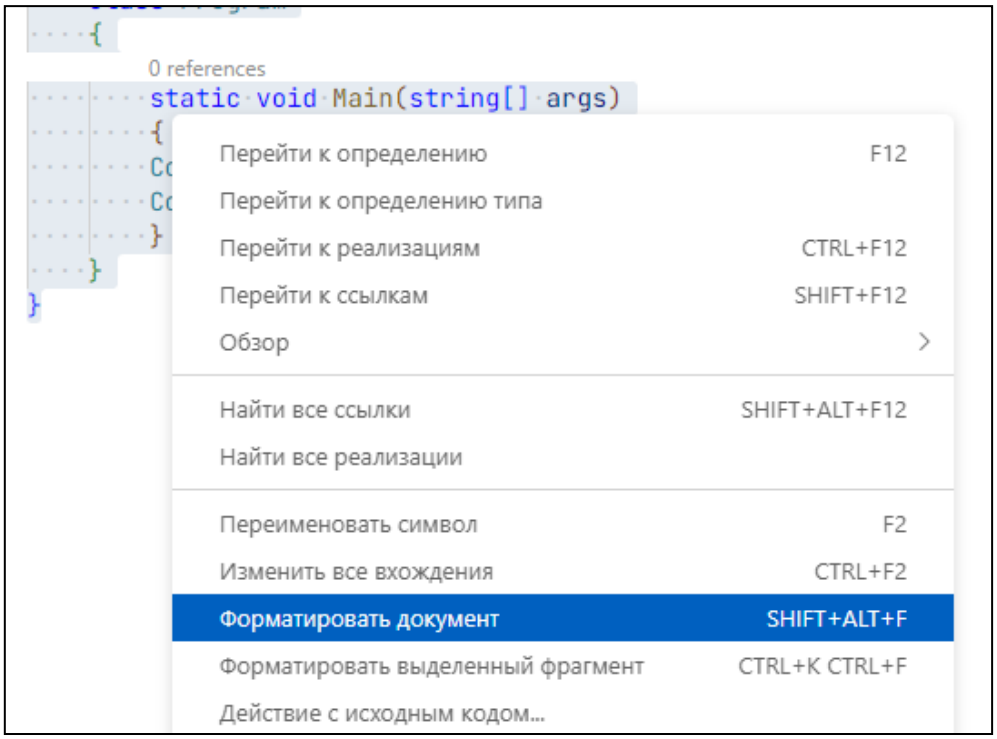


Рис. 1.56. Форматирование кода программы средствами редактора VSC

1.4.3. Рекомендации учителю

Начало курса по программированию

- Перед началом курса программирования кратко объясните учащимся, в чем отличие между программой компилятором и интерпретатором, приведите примеры.
- Объясните, почему современный разработчик должен изучать не только языки программирования, но и уметь использовать различные библиотеки и фреймворки.
- Выделите преимущества .NET и языка C#. Приведите примеры известных проектов, которые реализованы на их основе. Если это не первый изучаемый школьниками язык программирования, сравните его с изученными.
- Предоставьте учащимся краткий план (опорную карту), какое ПО им потребуется для изучения курса (с ссылками на официальный дистрибутив). Опишите назначение каждого.
- Продемонстрируйте основные операции с редактором Visual Studio Code: запуск, установка расширений, работа с

меню, настройка внешнего вида, управление вкладками и разделение области редактора на колонки.
- Уточните роль командной строки (терминала).
- Детально разберите алгоритм создания нового проекта. Приведите список основных команд для работы с компилятором dotnet (получение справки по командам, создание нового проекта, компиляция без запуска и с запуском приложения).
- Повторите процедуру еще несколько раз (с вызовом учащихся к доске).
- Продемонстрируйте принцип работы режима автодополнения кода.

Начало знакомства с языком

- Предоставьте учащимся шаблон консольного приложения в классической полной форме.
- Обратите внимание на иерархию компонент в приложении:

```
пространство имен >
    класс(ы) >
        содержимое класса(ов).
```

- Обсудите с учениками, в чем достоинства такой модели структуры кода.
- Заострите внимание на объектно-ориентированной основе используемых сущностей, наличие у объектов свойств и методов.
- Объясните, почему важно следовать принятым правилам оформления программного кода.

Вопросы для самопроверки

1. Перечислите основные компоненты в иерархии приложения на языке C#?
2. Какую роль играют классы в структуре программы?
3. Почему подпрограммы в C# называют методами?
4. Для чего используется команда `using`?

5. Почему важно соблюдать принятую стилистику оформления кода?

Практикум

1. Создайте новый проект под названием *SecondProject*.
2. Подготовьте его код, используя стартовый сниппет.
3. Замените название пространства имен с шаблонного *ConsoleApp* на название проекта (используйте рекомендации редактора VSC).
4. Скомпилируйте и запустите программу на выполнение.
5. Найдите любой онлайн-компилятор, способный тестировать работу простых консольных программ на языке C#. Например:
   - https://www.programiz.com/;
   - http://rextester.com;
   - https://onecompiler.com/.
6. Сравните возможности онлайн сервисов с возможностями редакторов кода.

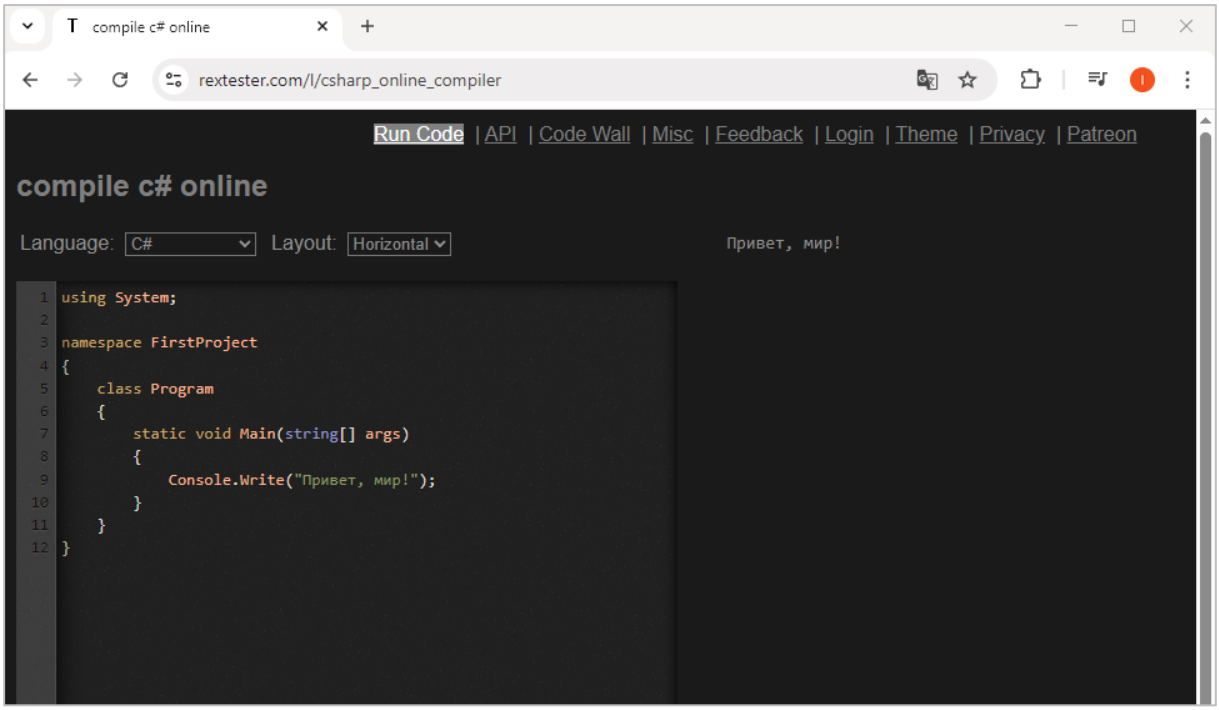


Рис. 1.57. Использование сервиса rexter.com для проверки работы простых приложений на языке C#

Глава 2. ОСНОВЫ СИНТАКСИСА ЯЗЫКА C#

2.1. Работа с данными

2.1.1. Типы данных .NET

Строгая типизация C#

C# является **строго типизированным** языком. Это значит, что любой переменной при описании должен быть задан **тип данных**.

Тип данных определяет:

- внутреннюю форму представления переменной в оперативной памяти;
- диапазон значений, которые может принимать переменная;
- допустимые операции над переменными заданного типа.

Строгая типизация позволяет безопасно оперировать переменными, поскольку тип каждой переменной, параметра или возвращаемого значения известен заранее и проверяется на этапе компиляции. Например, операция сложения строки и числа недопустима, если нет явного преобразования типов.

Типы данных .NET

Дерево типов

Типы данных C# определяются платформой .NET, в которой все типы можно разделить на две основные категории (далее подробнее):

- значимые типы;
- ссылочные типы.

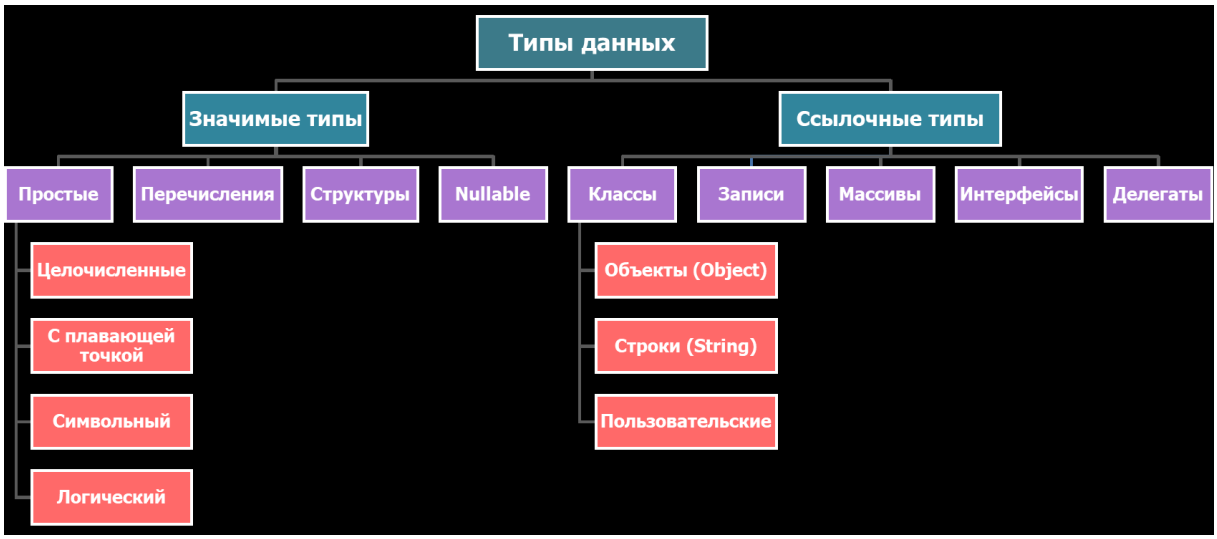


Рис. 2.1. Типы данных .NET

Значимые и ссылочные типы

Согласно представленной схеме на рис. 2.1 выделяют две основные категории типов данных, которые имею важные отличия:

1. Переменная **значимого типа** (**Value Type**) хранит само значение. Обычно данные значимого типа хранятся в **стеке**. При копировании таких переменных создаются независимые копии. Например, все числовые типы значимые.
2. Переменная **ссылочного типа** (**Reference Type**) хранит не значение, а ссылку на него (указатель). Обычно они хранятся в **куче**. При копировании таких переменных копируются не их значения, а их ссылки, т.е. между объектами, связанными с этими переменными, сохраняется связь. Примером ссылочного типа данных являются любые классы.

Кроме того, отличие между значимыми и ссылочными типами кроется в работе с памятью. Так, для переменных значимого типа память резервируется еще на этапе компиляции приложения. А для ссылочных типов память выделяется непосредственно во время работы приложения и автоматически очищается по мере необходимости с помощью специального механизма .NET – **сборщика мусора**.

Благодаря такому подходу удается оптимизировать процесс выделения оперативной памяти под данные приложения.

Важно понимать разницу между значимыми и ссылочными типами. Мы еще вернемся к этим особенностям при изучении классов.

Значимые типы

1. Целочисленные типы

Тип	Разрядность в байтах	Диапазон возможных значений переменной
byte		
sbyte	1	-128..127
short		
ushort	2	0..65535
int		
uint	4	0..4294967295
long		
ulong	8	0..18446744073709551615

Рис. 2.2. Целочисленные типы данных

В C# представлено 8 целочисленных типов данных, которые попарно делятся на знаковые и беззнаковые.

Чаще всего используется тип **int**: его диапазон достаточно велик для реализации самых разнообразных задач с целыми числами.

2. Типы с плавающей точкой

Тип	Разрядность в байтах	Диапазон возможных значений переменной
float		
double	8	$\pm 5 \cdot 10^{-324}$ .. $\pm 1{,}7 \cdot 10^{308}$

Рис. 2.3. Числа с представлением дробной части

Для операций с числами, которые могут иметь дробную часть, доступны два типа. Большинство встроенных методов классов ориентированы на тип **double**, поэтому в дальнейшем мы будем использовать его в качестве основного вещественного.

3. Десятичный тип

Тип	Разрядность в байтах	Диапазон возможных значений переменной
`decimal`		

Рис. 2.4. Дробные числа в десятичном представлении

Особый тип представления чисел с плавающей точкой (по существу – третий тип для описания вещественных чисел). На хранение переменной этого типа отводится 16 байт. Его диапазон меньше, чем у типа `double`, однако выше точность представления знаков в дробной части.

Обычно **`decimal`** используется для реализации задач с финансовыми вычислениями, где крайне важна точность при округлении.

4. Логический тип

Тип	Разрядность в байтах	Диапазон возможных значений переменной
`bool`		`true` `false`

Рис. 2.5. Логический тип данных

Используется в любой логической операции (сравнение, условные проверки и т.п.). Переменная логического типа может принимать одно из двух значений: **`true`** (истина) либо **`false`** (ложь).

5. Символьный тип

Тип	Разрядность в байтах	Диапазон возможных значений переменной
`char`		

Рис. 2.6. Тип для хранения символов

Символы .NET кодируются по стандарту Unicode, т.е. отводится два байта на хранение одного символа.

На самом деле переменные символьного типа **`char`** хранят числовой код одного символа (согласно таблице кодировки Unicode, например, `'f'` или в форме юникода – `'\u0066'`).

6. Другие типы

Структуры

Составной тип данных, близкий по строению к классам. Обычно они группируют связанные данные и в целом могут иметь такие же компоненты, как и классы. Однако структуры нельзя использовать для организации наследования.

Перечисления

Представляют собой набор именованных констант в рамках одной сущности. Переменные типа перечисления могут принимать только ограниченное количество значений, что повышает безопасность, абстрактность и читаемость кода.

Nullable

Специальный тип для обозначения **`null`** в значимом типе. Этот тип был введен, чтобы иметь аналогию отсутствия значения в переменной значимого типа, как это реализовано в классах.

Ссылочные типы

1. Строковый

Тип	Разрядность в байтах	Диапазон возможных значений переменной
`string`		

Рис. 2.7. Тип для представления текстовых значений

Строковый тип **`string`** не относится к простым типам данных. Строки в .NET описаны классами **`String`** и **`StringBuilder`** и имеют множество полезных свойств и методов.

Также у строковых переменных есть одна важная особенность: после создания строки ее содержимое не может быть изменено. А любые операции преобразования над строкой на самом деле создают новую строку и возвращают её.

2. Тип object

Тип	Разрядность в байтах	Диапазон возможных значений переменной
`object`		

Рис. 2.8. Единый тип для описания объектов

Тип **`object`** может хранить значение любого типа данных. Для 32-битных операционных систем он занимает 4 байта, для 64-битных – 8 байт.

Это базовый тип для всех типов данных в .NET. Его наследуют все классы.

Полные и сокращенные названия типов в .NET

Важно заметить, что даже простые типы данных в C# имеют объектно-ориентированную основу.

Например, встроенный целочисленный тип `int` представляет собой структуру `Int32`:

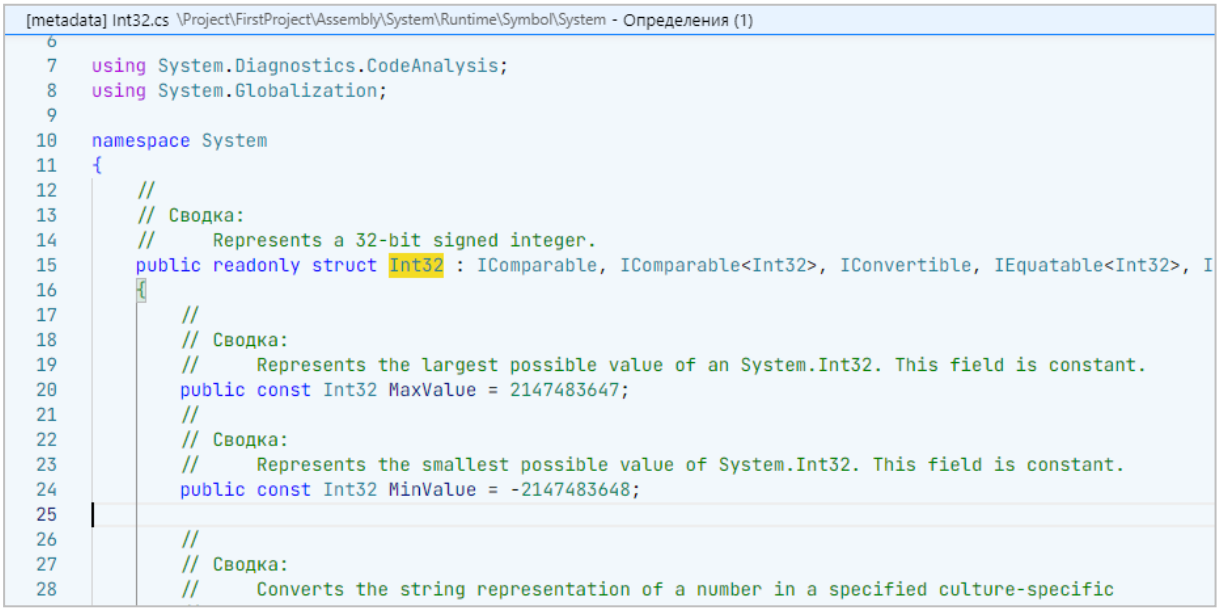


Рис. 2.9. Реализация системного типа `Int32`, соответствующего типу `int`

Внимательный читатель может заметить, что для ряда типов данных названия (в всплывающей подсказке) задаются как с маленькой, так и заглавной буквы. Например, есть тип `Double` и `double`.

На самом деле это одни и те же типы. Такой дуализм связан с тем, что в .NET используется **полное название** типов, а рассмотренные выше названия представляют собой **сокращенные псевдонимы**, используемые в пространстве имен `System`.

На самом деле мы можем создавать собственные псевдонимы, используя команду **`using`**:

```
using dbl = System.Double;
using integer = System.Int32;
```

```
using lng = System.Int64;
using obj = System.Object;
```

и т.п.

Иными словами, тип переменных можно описывать и с их полным названием в .NET.

2.1.2. Описание переменных и констант

Описание переменных

Объявление переменной

Правило

*Перед использованием **переменную** необходимо **описать (объявить)**, т.е. задать ей **тип**. Переменная описывается следующим образом:*

```
тип название_переменной;
```

Рис. 2.10. Описание переменной

В C# переменную можно описывать в любом месте внутри метода, там, где она понадобилась, но только один раз. Область видимости и роль переменной зависит от ее расположения внутри класса (это будет конкретизироваться в курсе).

```
int number;          // Целочисленная переменная
double coeff;        // Вещественная переменная
string text;         // Строковая переменная
char letter;         // Символьная переменная
```

Можно объявить несколько переменных одного типа в одной строке, разделяя их имена запятыми:

```
double dx, dy, dz;   // 3 вещественных переменных
```

Присваивание и инициализация значения

Для работы с переменной недостаточно описать только ее тип и имя: ей также требуется задать значение (**инициализировать**).

Оператором **присваивания** в C# является знак равно `=`.

Так, если предположить, что объявлены вышеуказанные переменные, то их можно инициализировать значением:

```
number = 2025;
coeff = 20.25;
text = "Привет, Денис!";
letter = 'Д';
dx = 0.2;
dy = 0.25;
dz = 0.1;
```

Обратите внимание, что для типа `char` символ отмечается в одинарных кавычках, а для типа `string` – в двойных.

Важное замечание!

В C# переменная должна быть инициализирована перед использованием. Если попытаться использовать неинициализированной переменной, ошибка выдается на этапе компиляции.

Рекомендуется описывать переменную и сразу ее инициализировать. Даже если в дальнейшем переменная будет менять значение, следует заранее определить некоторое стартовое значение:

```
int number = 2025;
double coeff = 20.25;
string text = "Привет, Денис!";
char letter = 'Д';
double dx = 0.2, dy = 0.25, dz = 0.1;
```

Для трех последних переменных показана инициализация в одну команду (однако их можно было разделить).

```
double dx = 0.2;
double dy = 0.25;
double dz = 0.1;
```

Это полезно знать!

Visual Studio Code выделяет переменные, которые описаны, но нигде не используются (как потенциально ненужные). Любое задействование переменной убирает эту подсказку и предупреждение.

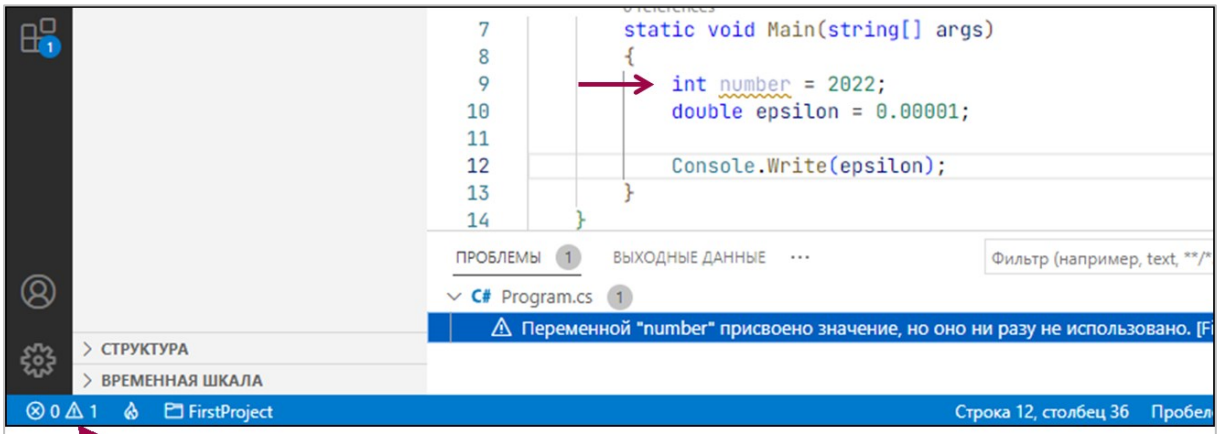


Рис. 2.11. Предупреждение редактора о неиспользуемых переменных

Описание констант

Правило

Константа *является неизменной и инициализируется при описании:*

```
const тип имя_константы = значение;
```

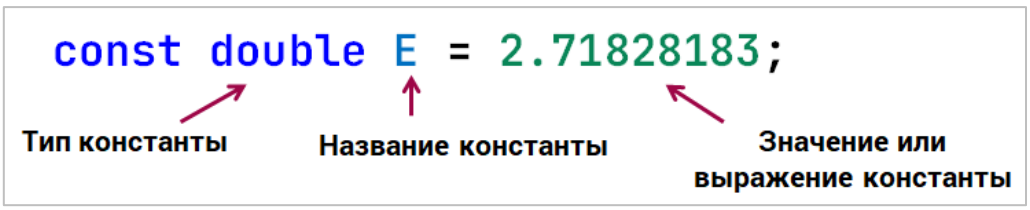


Рис. 2.12. Описание константы

Константу требуется сразу инициализировать значением и в дальнейшем его нельзя изменять, иначе такой код не скомпилируется:

```
const double E = 2.71828183;
```

Также константе можно присвоить некоторое выражение, состоящее из других констант:

```
const double Coeff = 2 * E;
```

Константы упрощают код и предотвращают случайное изменение значений.

Правила именования переменных и констант

Общепринятые нотации

В языке C# используются следующие стили именования переменных и констант (**нотации**):

- *Camel Case* (`camelCase`): каждое слово, за исключением первого, начинается с заглавной буквы. Используется для обозначения имен переменных.
- *Pascal Case* (`PascalCase`): каждое слово начинается с заглавной буквы. Используется для названия пространств имен, классов, структур, интерфейсов, методов, свойств и событий.
- *Upper Case* (`UPPER_CASE`): все буквы заглавные, слова разделяются символом подчеркивания. Используется для названий констант.

1. Именование переменных

Имя создаваемой переменной не должно совпадать с ключевыми командами C# или содержать пробелов. Оно должно быть информативным и максимально соотвествовать содержимому переменной.

Таблица 2.1. Именование переменных в «верблюжьей» и «змеиной» нотации

Нотация camelCase	Нотация snake_case
`string userPassword;` `DateTime dayOfWeek;`	`string user_password;` `DateTime day_of_week;`

2. Именование констант

Заранее известные константы именуется заглавными буквами. Что касается констант, вычисляемых из других констант, то их принято именовать в стиле CamelCase.

Таблица 2.2. Именование констант

Заранее определенные константы	Вычисляемые константы

`const double PI = 3.1459;`
 	
`const string RED = "FF0000";`
	`const double RotationAngle =`
	`  PI / 4;`
	`const string FontColor =`
	`  "#" + RED;`

Это полезно знать!

Задавайте переменным осмысленные имена. Имя переменной должно подразумевать, ЧТО в ней записано.
Часто название составляют из нескольких слов, например

```
int files_counter = 0;
```

является счетчиком файлов,

```
string document_title = "Урок 3. Описание переменных";
```

хранит заголовок документа, а в

```
DriveInfo[] all_drives = DriveInfo.GetDrives();
```

записываются данные о всех дисках на ПК (массив).
Хорошее название косвенно также отражает и тип переменной.

Числовые литералы

В ряде случаев C# требует уточнения типа значения константы (**литерала**), которое записывается в переменную.

Например, следующий код вызовет ошибку:

```
float dx = 0.01;
```

т.к. по умолчанию любой вещественный литерал рассматривается как значения типа `double`.

Уточнить тип литерала помогают специальные **суффиксы**.

Так для типа `float` возможны два варианта:

```
float dx = 0.01f;
float dx = 0.01F;
```

Для числовых литералов используются следующие суффиксы:

```
Console.Write(0.01);      // литерал типа double

Console.Write(0.01f);     // литерал типа float
Console.Write(0.01F);     // литерал типа float

Console.Write(25);        // литерал типа int

Console.Write(25u);       // литерал типа uint
Console.Write(25U);       // литерал типа uint

Console.Write(25l);       // литерал типа long
Console.Write(25L);       // литерал типа long

Console.Write(25ul);      // литерал типа ulong
Console.Write(25UL);      // литерал типа ulong
```

Неявная типизация

В C# поддерживается **неявная типизация** с помощью команды **var**. В этом случае мы предоставляем компилятору право самостоятельно вывести тип переменной по значению, поэтому переменная обязательно должна быть сразу инициализирована определенным значением.

В дальнейшем неявно типизированная переменная может принимать значения только того типа, который компилятор вывел изначально:

```
var number = 20.5;          // Определяет как тип double
number = number + 45;          // Операция допустима
number = number + "Hello";  // Ошибка!
                               // Не соответствуют типы
```

Однако в общем случае так описывать переменные не следует! Для неявной типизации есть ряд случаев, когда она помогает укоротить код.

Например, она может уменьшить код при описании переменных типа класса:

```
Collections.Generic.Stack<int> stack =
    new Collections.Generic.Stack<int>(10);
```

Левую часть рационально сократить (тип выводится по правой части):

```
var stack = new Collections.Generic.Stack<int>(10);
```

Вопросы для самопроверки

1. Какими преимуществами обладает строгая явная типизация данных?
2. В чем отличие между значимыми и ссылочными типами?
3. Перечислите значимые типы данных в C#.
4. Опишите основные ссылочные типы данных в C#.
5. Каким образом описываются переменные и константы?
6. В каком случае инициализация переменной является обязательной?
7. Перечислите основные правила и рекомендации по именованию переменных и констант.

Практикум

Задание 1

1. Создайте новый проект *Variables*.
2. Скопируйте в него код ниже.
3. Последовательно опишите следующие переменные:
   a. целочисленную `counter` со значением 10;
   b. вещественные `coordX` и `coordY` со значениями -4.5 и 5.7;
   c. строковую переменную `user` с вашими ФИО;
   d. логическую переменную `fileIsOpen` со значением `true`;
   e. опишите постоянную Планка `PLANK`, используя экспоненциальную запись числа.
4. Вывести значение переменных на экран (в терминал) с названием переменных (рис. 2.13).

Глава 2\Variables\Program.cs

```
using System;

namespace Variables
{
    class Program
    {
        static void Main(string[] args)
        {
            int counter = 10;
            // Дописать

            Console.WriteLine("counter = " + counter);
            // Дописать

        }
    }
}
```

```
PS D:\Documents\Programming\NET\FirstProject> dotnet run
counter = 10
coordX = -4,5
coordY = 5,7
user = Якубович Денис Андреевич
fileIsOpen = True
h = 6,62607015E-34
```

Рис. 2.13. Результат работы программы из задания 1

Задание 2

1. Создайте новый проект *ArithmeticalMean*.
2. Опишите три вещественные переменные `a`, `b`, `c`. В переменную `mean` записать формулу, вычисляющую среднее арифметическое этих чисел.
3. Вывести в консоль значения переменных с комментариями.

```
PS D:\Documents\Programming\NET\Program> dotnet run
a = 2,4
b = 5,1
c = 34
Среднее = 13,833333333333334
```

Рис. 2.14. Результат работы программы из задания 2

2.2. Арифметические операции и математические функции

2.2.1. Арифметические операторы

1. Основные операторы

Арифметические операции в C# реализованы в виде операторов сложения, разности, умножения, деления, вычисления остатка от деления и скобок, задающих приоритет операций:

Оператор	Операция
+	
-	Разность (или знак перед числом)
*	
/	Деление (неполное частное для целых чисел)
%	
( )	Скобки (порядок вычисления)

Рис. 2.15. Арифметические операторы C#

Пример:

```
int a = 10;
int b = 20;
int c = (a + b) * 3;
a = a % 4;
b = b / 3;
```

В результате вычислений:

- в `c` будет записано 90 $((10 + 20) * 3)$;
- значение `a` перезаписано 2 (остаток от деления 10 на 4, т.к. $10 = 2 * 4 + 2$);
- значение `b` перезаписано 6 (т.к. `b` описано как целое, то дробная часть игнорируется).

Операторы деления и поиска остатка от деления для целочисленных операндов дают целый результат. А вот для чисел с плавающей точкой учитывается и дробная часть:

```
double x = 5.2;
double y = 2.0;

// z <- 2.6 (здесь дробная часть уже не отбрасывается)
double z = x / y;

// v <- 1.2 (5.2 = 2.0*2 + 1.2)
double v = x % y;
```

Однако при делении важно учитывать типы литералов, входящих в выражение. Например:

```
double result1 = 20 / 3;
double result2 = 20.0 / 3;
double result3 = 20 / 3.0;
double result4 = 20.0 / 3.0;

Console.WriteLine(result1);
Console.WriteLine(result2);
Console.WriteLine(result3);
Console.WriteLine(result4);
```

```
PS D:\Documents\Programming\NET\FirstProject> dotnet run
6
6,666666666666667
6,666666666666667
6,666666666666667
```

Рис. 2.16. Особенность деления с литералами

Несмотря на то, что переменная `result1` описана числом с плавающей точкой, литералы 20 и 3 являются целочисленными, поэтому вычисляется неполное частное.

Чтобы этого избежать, хотя-бы один из литералов нужно записать в формате вещественного числа.

2. Укороченные операторы

Кроме основных арифметических операторов, реализованы укороченные операторы, являющиеся вариацией оператора. Они позволяет упрощать инструкцию присваивания, когда необходимо изменить величину переменной.

Рис. 2.17. Укороченные арифметические операторы как вариация оператора присваивания

Укороченный оператор убирает в правой части дублирование изменяемой переменной:

```
double x = 4;
double y = 5;
x += y;           // x <- 9  (аналог: x = x + y)
y++;              // y <- 6  (аналог: y = y + 1)
y *= (y + 1);     // y <- 60 (аналог: y = y * (x+1)
```

Оператор **инкремента** / **декремента** может быть **префиксный** и **постфиксный**. Для изменения одной переменной можно применять один из двух вариантов:

```
int k = 0;
k++;          // k <- 1 (постфиксный)
++k;          // k <- 2 (префиксный)
```

Однако у префиксного оператора приоритет выше, поэтому в случае нескольких переменных порядок операций важно учитывать:

```
int n = 0;
int a = n++;    // сначала a <- 0, далее n <- 1

int n = 0;
int a = ++n;    // сначала n <- 1, далее a <- 1
```

Важное замечание!

Операторы инкремента и декремента не рекомендуется использовать в составных выражениях, только в одиночку с переменной (верхний пример).

2.2.2. Преобразование типов

Неявные преобразования

В процессе работы с переменными разного типа возникает необходимость преобразований данных из одного типа в другой.

Неявные преобразования производятся компилятором автоматически, когда преобразование безопасно и не приводит к потере данных. В частности – при преобразовании от типа с меньшим диапазоном к типу с большим диапазоном (`int → long`, `int → float`, `float → double` и др.).

Так в приведенном ниже примере данные не теряются, поэтому компилятор выполняет преобразование автоматически (неявно):

```
int day_temperature = 26;

// Неявное преобразование int в double
double precise_day_temperature = day_temperature;

Console.WriteLine(day_temperature);              // 26
Console.WriteLine(precise_day_temperature);      // 26
```

Неявные преобразования возникают в следующих случаях:

- при расширяющем преобразовании (`int → double`);
- при безопасном преобразовании и отсутствии потери данных;
- при преобразовании производного класса к базовому.

Явные преобразования

Явные преобразования необходимы, когда необходим переход от одного типа к другому, при этом преобразование может привести к потере данных или небезопасно. В частности – при преобразовании от

типа с более широким диапозоном к типу с менее широким диапазоном (`double → int`, `long → short`, `float → int`).

Для осуществления явного преобразования необходимо указать целевой тип в скобках перед значением:

```
double precise_day_temperature = 26.8;

// Явное преобразование double в int
int day_temperature = (int) precise_day_temperature;

Console.WriteLine(precise_day_temperature);     // 26.8
Console.WriteLine(day_temperature);             // 26
```

В этом примере дробная часть числа теряется, поскольку происходит сужающее преобразование.

Явные преобразования необходимы в следующих случаях:

- при сужающем преобразовании (`double → int`).
- при потенциально небезопасном преобразовании, допускающем потерю точности или данных;
- при работе с несовместимыми типами (например, преобразование объекта базового класса к производному).

Методы преобразования

Кроме оператора явного преобразования типа в C# также допускается использовать методы преобразования данных. Например, класс `Convert` реализует ряд методов преобразования данных к простым типам и объектам.

В следующем примере осуществляется перевод строковой записи в числовой тип:

```
string text_coeff = "10,4";
double coeff = Convert.ToDouble(text_coeff);
```

2.2.3. Класс Math

Для более сложных вычислений с применением математических функций, в т.ч. трансцендентных, используется класс **`Math`** из пространства имен `System`. В нем реализован набор функций (в качестве методов) и несколько констант (в качестве свойств). Они являются статическими и вызываются по обращении к имени класса.

Свойство	Описание
E	
PI	Число π = 3,14159 ...
Tau	

Рис. 2.18. Свойства класса Math

Метод	Описание
Abs(x)	
Sin(x)	Синус (аргумент задается в радианах).
Cos(x)	
Atan(x)	Арктангенс (аргумент задается в радианах).
Log(x)	
Exp(x)	Экспонента
Pow(x,y)	
Sqrt(x)	Квадратный корень.
Truncate(x)	
Round(x)	Округление.

Рис. 2.19. Некоторые методы класса Math

Примеры использования:

```
Console.WriteLine("Число Пи: " + Math.PI);
Console.WriteLine("Число e: " + Math.E);

double alpha = Math.PI / 2;
double sinA = Math.Sin(alpha);
double cosA = Math.Cos(alpha);
double max = Math.Max(sinA, cosA);

Console.WriteLine("sin(alpha) = " + sinA);
Console.WriteLine("cos(alpha) = " + cosA);
Console.WriteLine("Max = " + max);
```

В контексте метода WriteLine оператор + склеивает текстовые комментарии и значения переменных в единую строку.

Ниже представлен вывод результата:

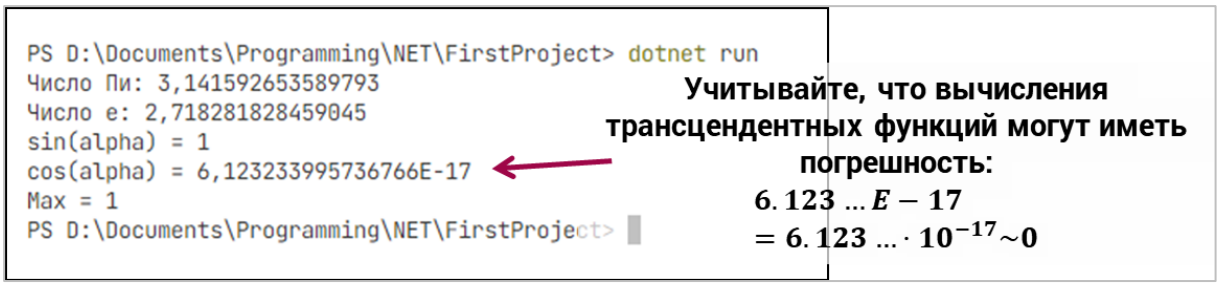


Рис. 2.20. Использование класса `Math`

2.2.4. Класс Random

Класс **`Random`** из пространства имен `System` позволяет генерировать **псевдослучайные числа**. Для этого необходимо создать **экземпляр класса**:

```
Random rnd = new Random();
```

и далее воспользоваться одним из его методов.

Генерация целых чисел

Для генерации целых чисел используется метод **`Next()`**, который имеет три вариации в зависимости от числа параметров:

```
Console.WriteLine(rnd.Next());  // [0, Int32.MaxValue)
Console.WriteLine(rnd.Next(10));      // [0, 10)
Console.WriteLine(rnd.Next(3, 10));   // [3, 10)
```

Важно заметить, что правая граница не включается в диапазон, поэтому для ее включения необходимо добавить 1:

```
Console.WriteLine(rnd.Next(3, 11));   // [3, 10]
```

```
PS D:\Documents\Programming\NET\FirstProject> dotnet run
298046498
0
4
```

Рис. 2.21. Генерация целых чисел

Генерация чисел с плавающей точкой

Для генерации вещественных чисел используют метод **`NextDouble()`**, который генерирует и возвращает случайное вещественное число в диапазоне $[0.0, 1.0)$. Чтобы получить число в произвольном интервале $[a, b)$, используется масштабирование и смещение интервала:

```
// [0, 1)
Console.WriteLine(rnd.NextDouble());

// [0, 1) -> [a, b)
double a = -4;
double b = 8.7;
double number = (b - a) * rnd.NextDouble() + a;
Console.WriteLine(number);
```

```
PS D:\Documents\Programming\NET\FirstProject> dotnet run
0,2372227208740505
-0,3555348634122186
PS D:\Documents\Programming\NET\FirstProject>
```

Рис. 2.22. Генерация вещественных чисел

2.2.5. Рекомендации учителю

- Обсудите с учениками, в чем важность типизации данных в программе и какие задачи она решает.
- Систематизируйте типы .NET по категориям. Объясните, почему возникла необходимость делить типы на значимые и ссылочные, в чем достоинства и недостатки каждого из них.
- Продемонстрируйте варианты описания переменных и констант. На конкретных примерах разберите, в каких ситуациях рационально описывать данные в форме переменной или константы.
- Объясните важность осмысленного именования переменных и констант. Сформируйте список рекомендации по их именованию.
- Разберите примеры программ, в которых требуется скорректировать имена переменных, чтобы программный код стал проще для анализа.

- Разберите задачи с использованием арифметических операторов. Особое внимание обратите на ситуации, в которых возникает необходимость преобразований между целочисленной арифметикой и арифметикой с плавающей точкой.
- Продемонстрируйте механизм работы с классами `Math` и `Random`. Уделите внимание функциям с составным аргументом (т.е. вызову сложных функций), порядку их вычисления.
- Закрепите материал разбором расчетных задач по заданным формулам из математики, физики, информатики, где требуется работа с математическими функциями.

Вопросы для самопроверки

1. Какие особенности арифметических операторов важно учитывать при работе с целыми числами и числами с плавающей точкой?
2. Опишите принцип работы укороченных операторов. Почему они являются вариацией оператора присваивания?
3. В каких ситуациях осуществляются неявное преобразование типов?
4. Каким образом можно осуществлять явное преобразование типов?
5. Перечислите часто используемые методы и свойства класса `Math`.
6. Опишите варианты генерации целых и вещественных чисел.

Практикум

1. Арифметические вычисления

Задание 1

1. Создайте новый проект *Calculation*.
2. Вычислить при заданных целых $a = 1$ и $b = 4$ (используйте только арифметические операции):

$$z = 5x^2 + 4y^2,$$

где

$$x = (a + 1)^2,$$
$$y = 3a(b - 2).$$

3. Вывести результат в терминал.

```
PS D:\Documents\Programming\NET\FirstProject> dotnet run
x = 4
y = 6
z = 224
```

Рис. 2.23. Результат работы программы из задания 1

Задание 2

1. Создайте новый проект *Ring*.
2. Скопируйте код заготовки ниже.
3. Вычислить площадь кольца по заданным внешнему и внутреннему радиусам.
4. Вывод оформить, как изображено на рис. 2.24.

Глава 2\Ring\Program.cs

```
using System;

namespace Ring
{
    class Program
    {
        static void Main(string[] args)
        {
            // Задаем Пи, внешний и внутренний радиус кольца
            const double PI = 3.14159265359;
            double outherRadius = 10.0;
            double innerRadius = 5.6;

            // Записать формулу для вычисления пощади в
            // переменную ringSquare и вывести данные на экран
        }
    }
}
```

```
PS D:\Documents\Programming\NET\FirstProject> dotnet run
Внешний радиус: 10
Внутренний радиус: 5,6
Площадь кольца: 215,6389197424176
```

Рис. 2.24. Результат работы программы из задания 2

Задание 3

1. Создайте новый проект *Geometry*.
2. Скопируйте код заготовки ниже.
3. Задан радиус окружности R. Написать программу, выводящую на экран:
   - длину окружности ($L = 2\pi R$);
   - площадь круга ($S = \pi R^2$);
   - площадь поверхности сферы ($\Gamma = 4\pi R^2$)
   - объем шара ($V = \frac{4}{3}\pi R^3$).
4. Часть кода уже реализована. Используйте его в качестве основы.
5. Результат вывести на экран, как изображено на рис. 2.25.

Глава 2\Geometry\Program.cs

```
using System;

namespace Geometry
{
    class Program
    {
        static void Main(string[] args)
        {
            double R = 10.3;
            const double PI = Math.PI;

            // Здесь описать формулы для вычисления
            // Расчет длины окружности
            double circle_length = 2 * PI * R;


            // Вывод информации
            Console.WriteLine("Радиус: " + R);
            Console.WriteLine();
            Console.WriteLine("Длина окружности: " +
                circle_length + " ед^2");

        }
    }
}
```

```
PS D:\Documents\Programming\NET\FirstProject> dotnet run
Радиус: 10,3

Длина окружности: 64,71680866394975 ед^2
Площадь круга: 333,2915646193412 ед^2
Площадь сферы: 1333,1662584773649 ед^2
Объем шара: 3432,903115579215 ед^2
```

Рис. 2.25. Результат работы программы из задания 3

2. Вычисления с классами Math и Random

Задание 1

1. Создайте новый проект *Formula*.
2. Написать программу, вычисляющую значение H по формуле:

$$H = \max(a, b) \cdot \frac{\ln(1 + a^2) + \sin(2\cos(b))}{\sqrt{a^2 + b^2} + 1},$$

где a, b являются произвольно заданными действительными числами. Используйте методы из класса `Math`.

```
a = 3,7
b = 2,08
H = 1,3118504
PS D:\Documents\Programming\NET\FirstProject>
```

Рис. 2.26. Результат работы программы из задания 1

Задание 2

1. Создайте новый проект *RandomNumber*.
2. Написать программу, которая при заданных вещественных a и b ($a < b$) возвращает случайное x из отрезка $[a; b]$.
3. Использовать формулу:

$$x = (b - a) \cdot r + a,$$

где $r \in [0; 1]$.

2.3. Ввод данных с клавиатуры

2.3.1. Ввод данных

Метод ReadLine

Для ввода текстовых данных с клавиатуры используется метод `ReadLine()` из класса `Console`. Метод считывает введенную текстовую запись до нажатия клавиши *Enter* и возвращает её в виде строки.

Важно понимать, что этот метод возвращает поток символов, которые вводятся с клавиатуры, т.е. данные типа `string`. Поэтому результат работы метода можно записать в строковую переменную:

```
string text = Console.ReadLine();
```

Обычно ввод должен сопровождаться выводом подсказки, иначе пользователь лишь увидит мигающий курсор:

```
Console.Write("Введите текст: ");
string text = Console.ReadLine();
```

Nullable-типы и тип string?

Начиная с C# 8.0 при попытке присвоить строковой переменной результат работы метода `ReadLine()` .NET-компилятор выдает предупреждение о том, что строка не может принимать значение `null`:

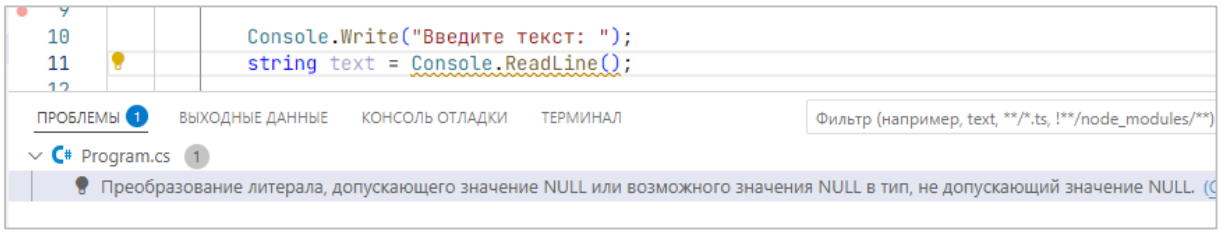


Рис. 2.27. Предупреждение о недопустимости значения null

Подобное решение призвано повысить безопасность операций со значениями, которые вводятся с клавиатуры.

Тем не менее ситуация, когда значение в переменной после выполнения ряда операций может отсутствовать, не редкость. Например, информация считывалась из базы данных и ячейка таблицы оказалась пуста.

Чтобы иметь возможность помечать переменные значимого типа значением `null`, как это реализовано для ссылочных типов, в .NET 5 (C# 8.0) ввели специальные **Nullable-типы**.

```
int n = null;     // Ошибка!
int? m = null;    // Все хорошо
```

Так тип `int` как значимый не может иметь значение `null`. Однако тип **`int?`** уже допускает такую ситуацию. Подобным образом можно помечать переменные, значения которых на данных момент неизвестны, но их уже можно использоваться в операциях.

Несмотря на то, что строки в C# относятся к классам, они хранятся по значению, поэтому изначально не допускают значения `null`. Nullable-тип **`string?`** устраняет это ограничение:

```
string? text = Console.ReadLine();
```

Важное замечание!

Далее при работе со строками мы будем использовать Nullable-допустимый тип ***`string?`***.

Обработка пустого ввода

Если вместо ввода символов (в т.ч.) пробелов пользователь сразу нажмёт *Enter*, метод `ReadLine()` возвратит пустую строку `""` или значение `null` (если ввод перенаправлен).

Чтобы избежать проблем в дальнейших операциях, можно предварительно проверить содержимое строки:

```
Console.Write("Введите текст: ");
string? input = Console.ReadLine();

if (string.IsNullOrEmpty(input))
{
    input = "Текст по умолчанию";
}
```

Метод **`IsNullOrEmpty()`** класса `string` возвращает истину, когда `input` содержит `null` или является пустой строкой. В этом случае значение переменной рационально установить некоторой записью по умолчанию (условные операторы подробно рассматриваются в следующих темах).

Операторы ?. и ??

Поскольку ситуации, когда переменные могут принимать значение `null` в процессе выполнения программы не редкость, то возникает необходимость предварительной проверки значений.

Оператор нулевого слияния

Например, в следующем фрагменте, если строка `text` содержит ссылку `null`, то значение заменится на пустую строку (которую уже можно обрабатывать в операциях):

```
if (text == null)
{
    text = "";
}
```

Чтобы укоротить подобные проверки, в C# ввели **оператор нулевого слияния `??`**:

```
text = text ?? "";
```

Оператор работает следующим образом: если `text` не содержит `null`, то в `text` запишется то же самое значение, иначе переменная содержит `null` и будет записана пустой строкой.

Более того, в этом случае можно использовать укороченный оператор присваивания **`??=`**:

```
text ??= "";
```

Оператор условного null

Другая возможная ситуация – попытка вызвать свойство или метод объекта со значением `null`. В этом случае во время работы программы выбрасывается исключение `NullReferenceException`.

Чтобы этого не происходило, можно также осуществить предварительную проверку. Например, выведем количество символов в строке, предварительно убедившись, что строка не `null`:

```
if (text != null)
{
    Console.WriteLine(text.Length);
}
```

Подобные проверки можно укоротить, используя **оператор условного null – `?.`**.

Код примера допускается сократить до следующей формы:

```
Console.WriteLine(text?.Length);
```

Иными словами, если `text` не `null`, то выведется количеств символов строки (для пустой строки – нуль), иначе – ничего.

Ввод нескольких значений

C# не предусматривает возможность ввода нескольких переменных и их преобразование в требуемый тип в одну команду (как, например – в Pascal). Предполагается, что введенная текстовая строка сначала разбивается на части и с каждой частью будет осуществлено надлежащее преобразование.

Впрочем осуществить ввод данных одного типа достаточно просто, если использовать массивы.

В следующем примере введенная строка разбивается на массив чисел `numbers`:

```
string input_text = Console.ReadLine() ?? "";

int[] numbers = Array.ConvertAll(
    input_text.Split(' '),
    int.Parse
);

Console.WriteLine(numbers[0]);
Console.WriteLine(numbers[1]);
// ...
```

В примере:

- метод **`Split(' ')`** разбивает строку по символу пробела между числами и возвращает массив строк (символ разделителя может быть и любы другим);
- метод **`ConvertAll()`** класса **`Array`** принимает массив строк и преобразует каждый элемент в целое число с помощью **`int.Parse()`**.

Далее каждое из чисел массива `numbers` можно записать в отдельную переменную, обращаясь к нему по индексу, или обработать все числа массива в цикле (массивам также посвящены отдельные параграфы).

2.3.2. Конвертирование введенных данных

Метод Parse

Если введенный текст необходимо записать в числовую, логическую или символьную переменную, потребуется операция преобразования типа.

Первый подход заключается в использовании метода **`Parse()`**, который привязан как статический метод к каждому простому типу данных. В качестве параметра методу передается результат ввода или текстовая переменная. `Parse()` пытается преобразовать текст в значение искомого типа, и если это возможно, то значение записывается в переменную, иначе генерируется ошибка выполнения программы **`FormatException`**.

В следующем примере показано, как можно сконвертировать введенный с клавиатуры текст в переменные соответствующих типов:

```
int number = int.Parse(Console.ReadLine() ?? "");
double epsilon = double.Parse(Console.ReadLine() ?? "");
bool file_exists = bool.Parse(Console.ReadLine() ?? "");
```

Здесь мы также используем оператор `??`, чтобы убрать предупреждение о недопустимости `null`. Однако в случае «пустого» ввода метод `ReadLine()` и так возвратит пустую строку, которую метод `Parse()` не может конвертировать в какое либо значение, поэтому выскочит исключение. Обычно ввод обрабатывается более детально. Например, в начале данные записываются в текстовую строку, а далее эта строка детально анализируется:

```
string? buffer = Console.ReadLine();

// Далее делается анализ строки buffer, разбиение
// ее на части и дальнейшее конвертирование каждой части
// в соответствующий тип
```

Класс Convert

Другой подход к преобразованию текста с клавиатуры заключается в использовании класса **`Convert`**, который содержит методы для перевода в соответствующие типы данных.

```
int number = Convert.ToInt32(
    Console.ReadLine() ?? ""
);

double epsilon = Convert.ToDouble(
    Console.ReadLine() ?? ""
);

bool file_exists = Convert.ToBoolean(
    Console.ReadLine() ?? ""
);
```

У методов класса `Convert` есть важное преимущество по сравнению с методом `Parse()`. Например, метод `ToInt32()` преобразует значение `null` в 0, в то время как `Parse()` выбрасывает исключение.

Сравнение подходов к преобразованию

В простых задачах нет особой разницы, как конвертировать введенный с клавиатуры текст: с помощью метода `ReadLine()` или методов класса `Convert`.

Однако класс `Convert` в плане конвертирования более универсален. Каждый его метод может принимать не только строку, но и объект класса и реализовывать попытку его преобразования к базовому типу. Такое может потребоваться в различных задачах.

Вопросы для самопроверки

1. Каким образом осуществляется ввод данных в консоль?
2. Для каких целей используются Nullable-типы и в чем необходимость использования типа `string?`?
3. Каким образом можно предусмотреть безопасное поведение программы в случае пустого ввода?
4. Для каких целей используются операторы нулевого слияния и условного `null`?
5. Опишите, каким образом можно ввести сразу несколько значений вещественных чисел?
6. Почему необходимо осуществлять преобразования введенного текста?
7. В чем разница между методом `Parse()` и методами конвертирования типа класса `Convert`.

Практикум

Задание 1

1. Создайте новый проект *ConsoleInput*.
2. Скопируйте в него код проекта *Formula* (из п. 2.2).
3. Реализуйте ввод a и b с клавиатуры (сопровождайте ввод подсказками).

```
Вычисление выражения
a = 3,7
b = 2,08
H(3,7;2,08) = 1,3118504012987178
```

Рис. 2.28. Результат работы программы из задания 1

Задание 2

1. Скопируйте проект *Ring* и назовите его *RingRadiusInput*.
2. Дополните программу таким образом, чтобы радиусы кольца запрашивались с клавиатуры.

```
Внешний радиус: 10
Внутренний радиус: 5,6
Площадь кольца: 215,6389197424176
```

Рис. 2.29. Результат работы программы из задания 2

Задание 3

1. Создайте новый проект *ModelParamsInput*.
2. Написать программу, в которой с клавиатуры вводятся параметры некоторой математической модели. При этом в первый запрос задается 4 параметра, а во второй 2. Ввод элементов организовывается через пробел.
3. Сопроводите ввод пояснениями.

```
Введите параметры модели:
Стороны прямоугольника [(a, b); (c, d)]: -3,2 6 0 5
Координаты точки симметрии (x, y): 10,04 8

Модель имеет вид: [(-3,2, 6); (0, 5)] с точкой симметрии (10,04, 8)
```

Рис. 2.30. Результат работы программы из задания 3

2.4. Вывод данных в окно консоли

2.4.1. Вывод данных

Методы **`Write()`** и **`WriteLine()`** из класса `Console` реализуют вывод строкового представления данных в окно консоли. `WriteLine()` дополнительно переводит каретку на новую строку и может быть указан без параметров.

Чаще всего оба метода используют для вывода строковых и числовых данных: их можно передавать через литералы, переменные и их комбинации. Однако методам можно передавать данные любого типа (без предварительной реализации выводится только название типа).

```
Console.Write(значение);
Console.WriteLine();
Console.WriteLine(значение);
```

Приведем примеры вывода значений разных типов данных:

```
int year = 2025;
string text = "Привет, Денис!";

Console.WriteLine(year);
Console.WriteLine(text);
Console.WriteLine(text + " На дворе " + year + " год");

Console.WriteLine(DateTime.Today.Date);

Student denis = new Student();
Console.WriteLine(denis);
```

```
PS D:\Documents\Programming\NET\FirstProject> dotnet run
2025
Привет, Денис!
Привет, Денис! На дворе 2025 год
27.04.2025 0:00:00
Formula.Student
```

Рис. 2.31. Вывод данных разного типа

Для объекта `denis` методы `Write()` и `WriteLine()` неявно вызывают метод `ToString()`, строковое представление которого нужно получить. Этот метод наследуется всеми классами из базового `Object`. Многие классы содержат его собственную реализацию.

Если код метода `ToString()` не переопределен в классе, то возвращается только название пространства имен и класса.

2.4.2. Интерполяция строк

Шаблонные строки

Начиная с C# 6-й версии в язык была внедрена возможность работы с **интерполированными** или **шаблонными строками**. Нововведение заключается в том, что значения переменных и констант можно подставлять прямо в строку, не используя при этом оператор склеивания `+`, преобразование к типу `string` / `string?` или более старый синтаксис (см. далее).

Шаблонная строка должна начинаться с символа **`$`** перед кавычками. Далее подставляемые значения указываются внутри **`{ }`**. При этом шаблонная строка сохраняет все возможности форматирования значений.

```
string goods = "Наушники";
double cost = 5999;

Console.WriteLine(
    $"Стоимость товара '{goods}' составляет {cost:f2}р"
);
```

В приведенном примере значение переменных `goods` и `cost` подставляются внутрь строки.

```
PS D:\Documents\Programming\NET\FirstProject> dotnet run
Стоимость товара 'Наушники' составляет 5999,00р
```

Рис. 2.32. Вывод шаблонной строки

Старый синтаксис форматирования строк

До появления интерполяции строк использовался менее удобный синтаксис. Чтобы подставить некоторое значение в указанное место строки, методу `Write()` или `WriteLine()` первым параметром необходимо указать строку с фигурными скобками и индексом элемента, а подставляемые элементы перечислялись далее через запятую (индексация с нуля):

```
Console.WriteLine("Стоимость товара '{0}' составляет
    {1:f2}р", goods, cost);
```

Также можно использовать метод `Format()`:

```
Console.WriteLine("Стоимость товара '" + goods +
    "' составляет " + string.Format("{0:f2}р", cost));
```

Очевидно, что синтаксис шаблонной строки удобнее, поскольку:

- запись получается короче;
- проще анализировать содержимое строки, когда вместо индексов стоят переменные;
- минимизирует ошибки при склеивании строк;
- позволяют указывать форматирование подставляемого значения непосредственно в строке;
- Visual Studio Code подсвечивает переменные и их формат в строке.

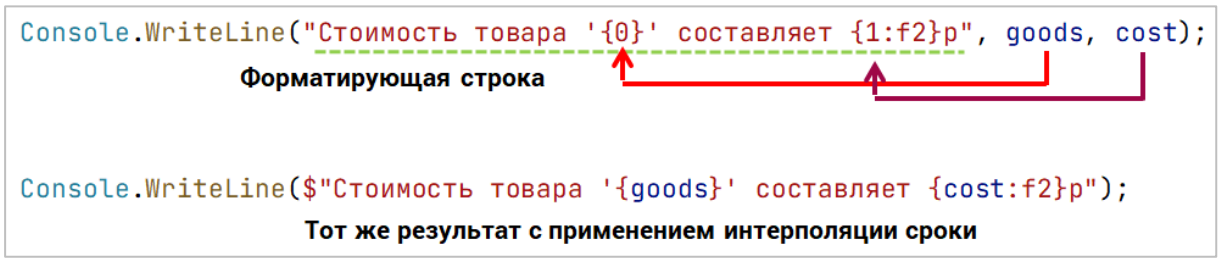


Рис. 2.33. Сравнение стандартного форматирования и шаблонной строки

Важное замечание!

Отдавайте предпочтение шаблонным строкам.
Старый синтаксис также поддерживается и его можно найти во множестве ранее реализованных проектов, в т.ч. библиотеке .NET.

2.4.3. Форматирование строк

Специальные символы

Расширять возможности форматирования позволяют **форматирующие символы**: они не отображаются при выводе, но позволяют внести изменения в оформление строки.

Чаще всего используются два форматирующих символа:

- `\n` переводит каретку на новую строку;
- `\t` табулирует данные, выводимые в разных строках.

Символ `\n` может быть альтернативой вызова нескольких методов `WriteLine()` подряд. А символ `\t` позволяет подравнять данные, которые следует вывести в колонки.

```
Console.WriteLine("Первая строка\nВторая строка\n\n");
Console.WriteLine("один\tдва\tтри");
Console.WriteLine("четыре\tпять\tшесть");
```

```
PS D:\Documents\Programming\NET\FirstProject> dotnet run
Первая строка
Вторая строка


один    два     три
четыре  пять    шесть
```

Рис. 2.34. Использование форматирующих символов в строке

Форматирование чисел

При форматирование чисел в C# можно настраивать количество знаков дробной части, добавлять разделители тысяч, отображать символы валют и многое другое.

Форматирование дробной части

- Заполнитель цифры `#` указывает, что если в числе на этой позиции есть цифра, она будет отображена; в противном случае ничего не будет показано.
- Заполнитель цифры `0` указывает, что если в числе на этой позиции есть цифра, она будет отображена; в противном случае она замещается нулем.
- Символ `.` и `,` отвечают за десятичный разделитель и разделитель тысяч соответственно.

Пример комбинированного использования:

```
const double PI = Math.PI;

Console.WriteLine($"Число: {PI}");
Console.WriteLine($"Число: {PI:#.#}");
Console.WriteLine($"Число: {PI:#.###}");

double number = 50345.29;

Console.WriteLine(number.ToString("#,##0.000"));
Console.WriteLine($"{number:#,##0}");
```

```
Число: 3,141592653589793
Число: 3,1
Число: 3,142
50 345,290
50 345
```

Рис. 2.35. Форматирование дробной части

Различные форматы представления

Для форматирования чисел доступен ряд специальных форматов. В следующем примере в первом случае под каждое число резервируется 15 позиций, а во втором указывается формат вывода дробной части.

```
const double PI = Math.PI;
const int Year = 2025;

// Целочисленный
Console.WriteLine($"D формат:{Year,15:D} \t{Year:D8}");

// Дробные числа
Console.WriteLine($"F формат:{PI,15:F} \t{PI:F2}");

// Экспоненциальная форма
Console.WriteLine($"E формат:{PI,15:E} \t{PI:E5}");

// Универсальный формат
Console.WriteLine($"G формат:{PI,15:G} \t{PI:G5}");

// С разделителями тысяч
Console.WriteLine($"N формат:{Year,15:N} \t{Year:N5}");

// Денежный формат
```

```
Console.WriteLine($"C формат:{PI,15:C} \t{PI:C3}");

// 16-чный формат
Console.WriteLine($"X формат:{Year,15:X}");

// Процентный
Console.WriteLine($"P формат:{PI,15:P}");
```

Отметим, что формат вывода дробного числа **F** будет активно использоваться далее, чтобы отсекать вывод лишней дробной части.

```
PS D:\Documents\Programming\NET\FirstProject> dotnet run
D формат:            2025        00002025
F формат:            3,14        3,14
E формат:  3,141593E+000        3,14159E+000
G формат:3,141592653589793      3,1416
N формат:       2 025,00        2 025,00000
C формат:         3,14 ₽        3,142 ₽
X формат:             7E9
P формат:       314,16 %
```

Рис. 2.36. Форматы представления чисел

2.4.4. Метод String.Format()

До появления синтаксиса интерполяции строк часто использовался метод **String.Format()**, который преобразует значения объектов в строку на основе указанных форматов. Метод собирает параметры в строке, как это делают `Write()` и `WriteLine()`.

```
string goods = "Наушники";
double cost = 5999;

string message = String.Format(
    "Стоимость товара '{0}' составляет {1:f2}р",
    goods, cost
);
Console.WriteLine(message);
```

```
Стоимость товара 'Наушники' составляет 5999,00р
```

Рис. 2.37. Форматирование строки с помощью `String.Format()`

Интерполяция строки существенно упрощает работу: теперь вызов `String.Format()` больше не требуется:

```
string goods = "Наушники";
double cost = 5999;

string message = $"Стоимость товара '{goods}' составляет
    {cost:f2}р";
Console.WriteLine(message);
```

Это полезно знать!

Visual Studio Code позволяет преобразовывать старый синтаксис форматирования строк в новый. Для этого необходимо поставить курсор на фрагмент строки и щелкнуть на символ лампочки. Далее в выпадающем списке выбрать пункт ***Преобразовать в интерполированную строку****.*

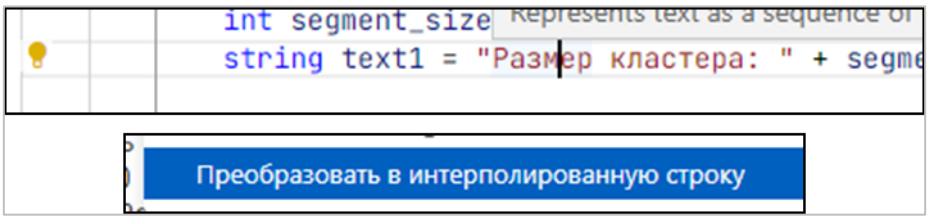


Рис. 2.38. Преобразование в шаблонную строку

2.4.5. Решение практических задач

Задача 1

Требуется написать программу, которая вычисляет ежемесячный платеж клиента банка по кредитному займу. В качестве искомых данных выступают:

- S – сумма займа;
- P – процентная ставка (годовая);
- M – срок займа (лет).

Считать, что банк использует аннуитетный график начисления процента. Кроме того, банк выдает кредит без требования первоначального взноса и разбивает его погашение на равные доли по каждому месяцу.

Решение

При аннуитетном способе расчета основной долг по платежу разбивается на неравные части: самая малая сумма приходится на начало срока, наибольшая – на конец.

В отличие от дифференцированного графика платежа аннуитетный способ более понятен заемщику и чаще используется банками:

- его формулы проще рассчитать;
- заемщик ежемесячно платит одну и ту же сумму.

Ежемесячный платеж определяется по формуле:

$$X = S \cdot \left(p + \frac{p}{(1+p)^N - 1}\right),$$

где

- S – сумма займа;
- p – доля процентной ставки в месяц;
- N – срок кредитования в месяцах.

Например, заемщик планирует взять кредит в 800 000 руб. под 12% годовых сроком на 3 года.

Тогда доля процентной ставки в месяц составит:

$$p = P/100/12 = 12/100/12 = 0.01.$$

Срок в месяцах: $3 \cdot 12 = 36$.

Ежемесячный платеж составит:

$$X = 800\,000 \cdot \left(0.01 + \frac{0.01}{(1+0.01)^{36} - 1}\right) \approx 26\,571\text{р.}$$

На следующем графике показано, как меняется ежемесячный платеж при сохранении суммы займа и процентной ставки:

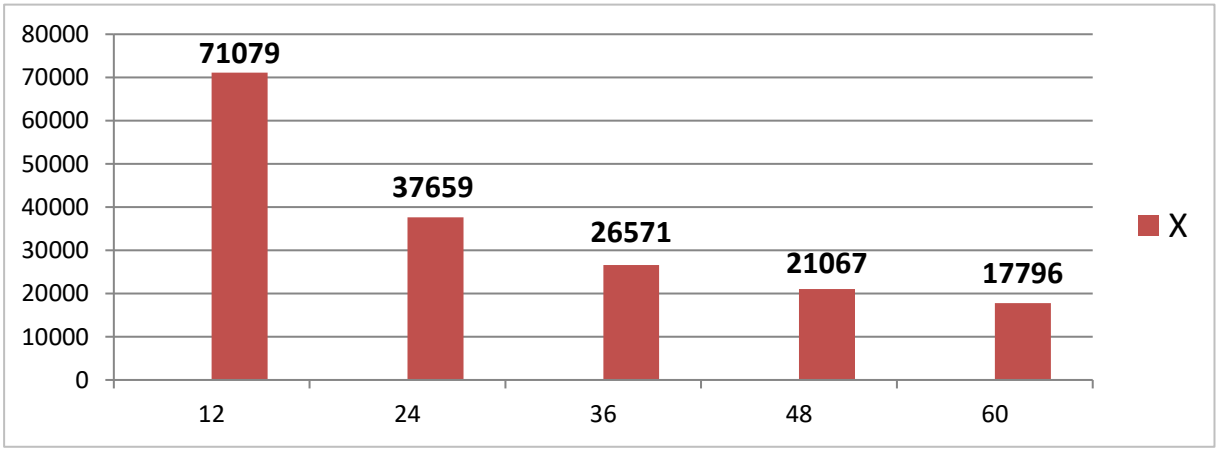


Рис. 2.39. График изменения ежемесячного платежа

Глава 2\CreditLoan\Program.cs

```
using System;

namespace CreditLoan
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Расчет кредитного займа");

            Console.Write("Сумма кредита: ");
            double loan_sum = double.Parse(
                Console.ReadLine() ?? "0"
            );

            Console.Write("Ставка по кредиту (годовая): ");
            double loan_percent = double.Parse(
                Console.ReadLine() ?? "0"
            );

            Console.Write("Срок кредитования (лет): ");
            int loan_term = int.Parse(
                Console.ReadLine() ?? "0"
            );

            double p = loan_percent / 100 / 12;
            int months = loan_term * 12;

            double monthly_fee =
                loan_sum * p *
                (1 + 1 /(Math.Pow(1 + p, months) - 1));

            Console.WriteLine("---------------");
            Console.WriteLine(
                $"Ежемесячный взнос составляет
                {monthly_fee:F2}р"
            );
        }
    }
}
```

```
Расчет кредитного займа
Сумма кредита: 800000
Ставка по кредиту (годовая): 12
Срок кредитования (лет): 3
--------------
Ежемесячный взнос составляет 26571,45р
```

Рис. 2.40. Результат работы программы

Задача 2

С помощью метода `TryParse()` реализуйте более безопасный подход к вводу и конвертированию данных на примере вещественного числа.

Решение

Метод `Parse()` способен преобразовывать строку в указанный тип (относительно которого вызывается). Однако если строка пустая или содержит нечисловые символы, то во время выполнения программы возникает исключение `FormatException`. Для его обработки потребуется использовать блоки `try-catch`, что несколько затрудняет чтение кода, особенно в случае многократных подобных проверок.

В C# доступен еще один метод – **`TryParse()`**, который в случае недопустимого для перевода значения не вызывает исключение, а возвращает логическое значение `true` в случае успешного либо `false` в случае неуспешного преобразования.

```
Console.Write("Введите число: ");
string? text_input = Console.ReadLine() ?? "";

if (double.TryParse(text_input, out double number))
{
    Console.WriteLine($"Введено число {number}");
}
else
{
    Console.WriteLine($"Некорректный ввод");
}
```

В качестве первого параметра методу `TryParse()` передается строка, которую он пытается преобразовать. А в качестве второго – переменная, в которую записывается преобразованный результат. Здесь мы также демонстрируем работу с оператором `if` (подробнее в следующем параграфе).

Здесь важно заметить два момента:

- Переменная описывается с модификатором **out**, т.е. передается в метод *по ссылке*. Он гарантирует, что после завершения работы метода переменная будет инициализирована некоторым значением.
- Переменная с модификатором out не обязана иметь установленное заранее значение. Это значение присваивается внутри метода. Поэтому начиная с C# 7.0 out-переменные можно описывать в момент вызова метода.

Таким образом полная процедура обработки ввода может выглядеть, например, следующим образом:

Глава 2\TryParseInput\Program.cs

```
using System;

namespace TryParseInput
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.Write("Введите число: ");
            string? text_input = Console.ReadLine() ?? "";

            if (double.TryParse(text_input,
                                out double number))
            {
                // Преобразование допустимо.
                // В number запишется корректное число
                // Далее остальные операции, если необходимы
            }
            else
            {
                // Некорректный ввод. Возьмем по умолчанию 0
                number = 0;
            }

            // Далее другие операции...
        }
    }
}
```

2.4.6. Рекомендации учителю

- Продемонстрируйте практику оформления ввода данных с текстовым комментарием.
- Покажите, каким образом введенный текст можно преобразовать в число, а когда такие операции невозможны. Отдельно проработайте ситуацию, как можно обработать пустой ввод.
- Разберите примеры с конкатенацией строк.
- Обучите школьников работе с форматирующими символами при оформлении результатов вывода в консоль.
- Проведите сравнительный анализ между старым и новым синтаксисом форматирования строк. Используйте конкретные примеры.

Вопросы для самопроверки

1. Опишите особенности использования методов `Write()` и `WriteLine()`.
2. Какие строки называют шаблонными и какое у них преимущество по сравнению с обычным форматированием?
3. Какие форматирующие символы помогаю оформлять текст?
4. Перечислите форматы представления целых и десятичных чисел.
5. Почему метод `String.Format()` в современном C# используется редко?

Практикум

Задание 1

1. Создайте новый проект *RomulaFormat*.
2. Скопируйте в него код проекта *ConsoleInput* (из п. 2.3).
3. Отформатировать ввод и результаты вывода следующим образом:

```
PS D:\Documents\Programming\NET\Program> dotnet run
a = 3,7
b = 2,08
H(3,7;2,08) = 1,3118504 (с точностью до 7 знаков)
```

Рис. 2.41. Результат работы программы из задания 1

Задание 2

1. Создайте новый проект *StringFormat*.
2. Скопируйте код ниже.
3. Исправьте форматирование строк `text1`, `text2` и `text3`, заменив его на синтаксис интерполяции строки.

Глава 2\StringFormat\Program.cs

```
using System;

namespace StringFormat
{
    class Program
    {
        static void Main(string[] args)
        {
            int segment_size = 4096;
            string text1 =
               "Размер кластера: " + segment_size + "Кб";

            double side = 4;
            double value = side*side;
            string text2 =
                "Площадь квадрата со стороной " +
                side + " равна " + value;

            double x = 3.05;
            double f = Math.Sin(x*x + 1);
            string text3 = String.Format(
                "f({0}) = {1:F5}", x, f
            );

            Console.WriteLine(text1);
            Console.WriteLine(text2);
            Console.WriteLine(text3);
        }
    }
}
```

```
Размер кластера: 4096Кб
Площадь квадрата со стороной 4 равна 16
f(3,05) = -0,76929
```

Рис. 2.42. Результат работы программы из задания 2

Задание 3

1. Доработайте проект *CreditLoan*.
2. Пусть банк выдает кредит без требования первоначального взноса и разбивает его погашение на равные доли по каждому месяцу.
3. Дополните код задачи: необходимо также вывести на экран информацию о том, какую сумму по итогу погашения кредита выплатит клиент и какую выручку получит банк.

```
Расчет кредитного займа
Сумма кредита: 800000
Ставка по кредиту (годовая): 12
Срок кредитования (лет): 3
---------------
Ежемесячный взнос составляет 26571,45р
Клиент по истечению срока выплатит 956572,12р
Прибыль банка составит 156572,12р
```

Рис. 2.43. Результат работы программы из задания 3

Задание 3

1. Создайте проект *Wages*.
2. С клавиатуры запрашивается оклад учителя и количество его ставок (коэффициент 1.0 соответствует одному окладу).
3. Вывести на экран его зарплату с точностью до двух знаков, если ставка налога составляет 13%.
4. Для обработки ввода использовать метод `TryParse()`. В случае некорректного ввода оклад установить в размере 24 700р и полную ставку соответственно.

```
Введите оклад педагога: 32700
Введите число ставок: 1,3
Оклад: 32700₽
Число ставок: 1,3
Зарплата (с вычетом налога): 36983,70₽
```

Рис. 2.44. Результат работы программы из задания 4

2.5. Условный выбор if-else

2.5.1. Операторы отношения

Операторы их применение

Определение

Операторы отношения *используются для сравнения объектов. Результат является логическим значением* ***true*** *или* ***false****. Обычно используются в условных операциях.*

Операторы отношения связывают два операнда. Результат сравнения выражается значением логического типа, поэтому может использоваться не только в условных и циклических операторах, но и присваиваться в переменные.

В C# реализованы шесть операторов отношения (рис):

Оператор	Действие
==	
!=	Не равно
>	
<	Меньше
>=	
<=	Меньше или равно

Рис. 2.45. Операторы отношения

Необходимо обратить внимание на операторы равенства и неравенства: они комбинируются из двух знаков, чтобы отличаться от оператора присваивания.

Примеры:

```
int x = 10;
int y = 7;
bool r1 = x == y;   // false
bool r2 = x != y;   // true
bool r3 = x > y;    // true
bool r4 = x >= y;   // true
```

```
bool r5 = x < y;    // false
bool r6 = x <= y;   // false
```

Необходимо учитывать, что операторы сравнения **<**, **>**, **<=** и **>=** могут применяться только для сравнения тех типов данных, которые поддерживают **отношение порядка**. В частности – это все числовые типы данных.

А вот значения типа `bool` можно сравнивать только на равенство или неравенство, другие операторы отношения неприменимы.

```
bool result1 = true == false;   // false
bool result2 = true > false;    // Ошибка!
```

Особенности сравнения вещественных чисел

При сравнении чисел с плавающей точкой на равенство / неравенство можно получить неожиданные результаты.

Приведем простой пример:

```
double a = 0.1;
double b = 0.2;
double sum = a + b;

Console.WriteLine(sum == 0.3);    // false
Console.WriteLine(sum);
```

```
False
0,30000000000000004
```

Рис. 2.46. При сложении вещественных чисел возникла погрешность

Несмотря на то, что с математической точки зрения ожидался положительный результат сравнения суммы с числом, проверка дала ложный ответ.

Возникшая проблема связана с особенностями представления чисел с плавающей точкой в ограниченной по точности компьютерной арифметике. В двоичном коде конечную форму имеют только те десятичные дроби, которые можно представить степенью или линейной комбинацией степеней двойки: $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{3}{4} = \frac{1}{2} + \frac{1}{4}$ и т.д. А числа 0.1 и 0.2 в двоичной форме представлены бесконечными периодическими дробями, поэтому кодируются с погрешностью.

Эта особенность кодирования дробных чисел присуща и многим другим языкам программирования.

Чтобы обойти эту проблему, вместо прямого сравнения вещественных чисел используют сравнение через неравенство.

Рассмотрим один из наиболее простых приемов – допустимая точность. Для сравнения двух чисел на равенство проверяют абсолютное значение разницы между ними и сравнивают с некоторым достаточно малым положительным числом (эпсилоном), называемым **допуском** или **допустимой точностью**:

$$|n - m| < \varepsilon.$$

В этом случае числа n и m считают равными.

Так, в примере можно задать константу `epsilon`, значение которой достаточно мало, чтобы сумму можно было считать равной 0.3:

```
const double epsilon = 1e-15;    // 0.000000000000001

double a = 0.1;
double b = 0.2;
double sum = a + b;              // 0.30000000000000004

Console.WriteLine(Math.Abs(sum - 0.3) < epsilon); // true
```

Выбор значения `epsilon` зависит от контекста задачи и допустимой точности. А вот работать со свойствами `float.Epsilon` и `double.Epsilon` как пределами точности типов `float` и `double` небезопасно в силу того, что модуль разности их будет превышать в силу погрешности. Поэтому в примере `epsilon` взято на порядок меньше, чем предельная точность дробной части у типа `double`.

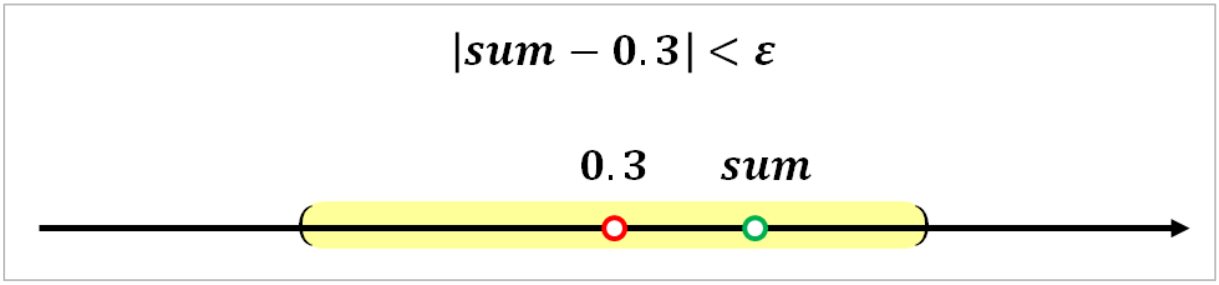


Рис. 2.47. Сумма лежит в окрестности 0.3, поэтому можно считать $sum = 0.3$

Важное замечание!

Такое сравнение необходимо только для чисел с плавающей точкой.

2.5.2. Логические операторы

Основные операторы

Определение

Логические операторы *определяют булевы операции (логические союзы). Результат является логическим значением* **`true`** *или* **`false`**. *Обычно используются в условных операциях.*

Логические операторы выступают в роли логических союзов и могут объединяться в составные выражения. Между ними действуют хорошо известные правила математической логики.

Оператор	Союз	Действие
&		
|	ИЛИ	Дает истину в случае истинности хотя-бы одного из условий.
!		
^	Исключающее ИЛИ (ЛИБО)	Дает истину, когда истина – либо первое, либо второе, но не оба вместе.

Рис. 2.48. Логические операторы

Каждый из операторов, кроме отрицания, связывает два логических выражения. С помощью круглых скобок можно контролировать приоритет операций в составных выражениях. Операторы могут использоваться в одном выражении многократно.

Перечисленные операторы имеют приоритет:

1. отрицание;
2. союз И (логическое умножение);
3. союзы ИЛИ и ЛИБО (логическое сложение).

Союз ЛИБО можно получить из первых трех операторов (см. правила математической логики).

```
double x = 3.6;
bool result1 = (1 <= x) & (x <= 3);   // false
bool result2 = (x < 0) | (x > 2);     // true
bool result3 = !(x > 0);              // false
bool result4 = (1 < x) ^ (x == 3.6);  // false
```

Укороченные операторы

Помимо основных операторов для союзов И и ИЛИ реализованы укороченные аналоги.

Оператор	Союз	Действие
&&		
||	ИЛИ	Дает истину в случае истинности хотя-бы одного из условий. Если первое условие истинно, то второе не проверяется.

Рис. 2.49. Укороченные логические союзы

Укороченные операторы **&&** и **||** используют «**ленивое вычисление**». В отличие от обычных операторов `&` и `||`, которые всегда проверяет как левую, так и правую часть условий, укороченные работают более рационально и останавливают проверку, когда значение первого операнда уже посчитано (а второй операнд не влияет на общее значение выражения).

Например,

```
double x = 3.6;

bool result1 = (4 <= x) && (x <= 7);  // false

bool result2 = (1 <= x) || (x > 0);   // true
```

Для `result1` первое условие ложно, поэтому второе можно не проверять, результат в любом случае получается `false`.

Для `result2` первое условие истинно, второе можно не проверять, результат в любом случае получается `true`.

Что касается оператора `^`, то он не обладает укороченной формой, поскольку результат выражения требует вычисления обоих операндов.

Применение основных и укороченных союзов

В подавляющем большинстве случаев используются укороченные союзы И и ИЛИ: они позволяют избежать лишних проверок. Более того, зачастую укороченные операторы позволяют избежать и исключений во время работы.

Так в следующем примере в случае `text == null` второе условие вызовет исключение `NullReferenceException` и программа прервет выполнение:

```
if (text != null & text.Length > 0)
{
    Console.WriteLine("Текст не пустой.");
}
```

В случае укороченного оператора достаточно проверить только первое условие (дает ложный ответ), второе условие уже не проверяется и программа продолжит работу:

```
if (text != null && text.Length > 0)
{
    Console.WriteLine("Текст не пустой.");
}
```

Таким образом:

- Укороченные операторы лучше использовать, когда нужно избежать лишних проверок. Например, в ситуациях, когда вычисление второго операнда ресурсоемкое.
- Укороченные операторы могут предотвратить ошибки, когда операция недопустима.

Однако это не означает, что укороченный оператор следует всегда применять вместо обычного. Обычные операторы И и ИЛИ часто находят применение в задачах с поразрядными операциями над числами (например, двоичной арифметике).

Это полезно знать!

Работая с укороченными операторами старайтесь выносить в начало наиболее «важные» условия из выражения. В частности те, от которых зависят или даже будут логически корректны следующие ограничения.

2.5.3. Условный выбор. Оператор if и его вариации

Оператор if в полной форме

Определение

Инструкция ***if*** *(«**если**») в зависимости от истинности условия выполняет соответствующий блок кода. Условие является логическим выражением (типа* `bool`*).*

Оператор `if` реализует разветвленный алгоритм выполнения последовательности команд, где выбор действий зависит от истинности условия.

В **полной форме** оператор имеет вид:

```
if (условие)
{
    // делаем, когда условие истинно
}
else
{
    // делаем, когда условие ложно
}
```

где условие выступает выражением логического типа `bool`.

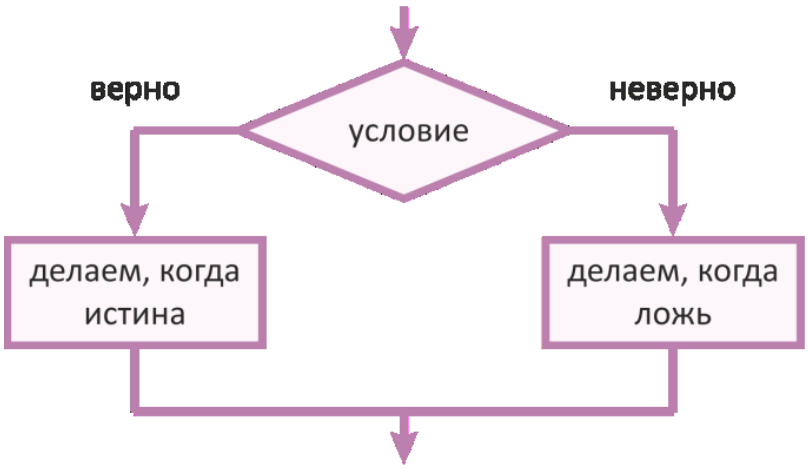


Рис. 2.50. Блок-схема оператора `if` в полной форме

Круглые скобки для условия являются частью оператора `if`, их необходимо указывать в обязательном порядке! А фигурные скобки тела блоков можно не писать, если в блоке одна операция.

Например, в следующем фрагменте кода проверяется, допущен ли студент до сдачи экзамена (суммарный рейтинговый балл не менее указанного):

```
const int Passing_Points = 20;
int rating_points = 45;

if (rating_points >= Passing_Points)
{
    Console.WriteLine("Студент допущен к экзамену");
}
else
{
    Console.WriteLine("Студент не допущен к экзамену");
}
```

Обе ветви оператора содержат по одной инструкции, поэтому фигурные скобки блоков можно опустить:

```
const int Passing_Points = 20;
int rating_points = 45;

if (rating_points >= Passing_Points)
    Console.WriteLine("Студент допущен к экзамену");
else
    Console.WriteLine("Студент не допущен к экзамену");
```

Разумеется, в условиях могут стоять и составные выражения. Например, проверим, можно ли осуществить запись данных на флеш-карту:

```
double flash_memory = 16.00;    // размер флешки (Гб)
double file_size = 1.78;        // размер файла (Гб)
bool write_access = true;       // разрешена ли запись?

if (write_access && (file_size < flash_memory))
    Console.WriteLine("Файл доступен для записи");
else
    Console.WriteLine("Запись невозможна");
```

Обратите внимание: логическую переменную `write_access` в условии необязательно явно сравнивать со значением `true`.

Это полезно знать!

Все логические переменные в условии автоматически сравниваются со значением ***true****.*

Оператор if в неполной форме

Блок **else** в операторе if является необязательным. Если его опустить, то реализуется **неполная форма** оператора if:

```
if (условие)
{
    // делаем, когда условие истинно
}
```

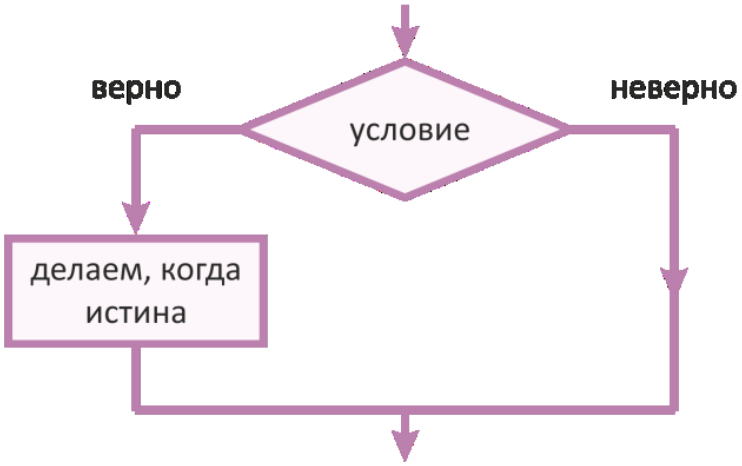


Рис. 2.51. Блок-схема оператора if в неполной форме

Пример:

```
string? today_day = "суббота";

if (today_day == "суббота" || today_day == "воскресенье")
{
    Console.WriteLine("Сегодня выходной");
}
```

или в более упрощенной форме:

```
string? today_day = "суббота";

if (today_day == "суббота" || today_day == "воскресенье")
    Console.WriteLine("Сегодня выходной");
```

2.5.4. Решение практических задач с if-else

Задача 1

Требуется определить, является ли текущий год високосным. Год должен получаться автоматически. Также вывести текущую дату.

Решение

Год является високосным, если он кратен 4.

Чтобы автоматически извлекать значение даты, воспользуемся классом **DateTime**.

Глава 2\CurrentYear\Program.cs

```
using System;

namespace CurrentYear
{
    public class Program
    {
        public static void Main(string[] args)
        {
            DateTime current_date_time = DateTime.Now;
            int current_year = current_date_time.Year;

            Console.Write($"На дворе {current_year} год ");

            if (current_year % 4 == 0)
                Console.WriteLine("(високосный год).");
            else
                Console.WriteLine("(обычный год).");

            Console.WriteLine(
                $"Дата: {current_date_time:dd.MM.yyyy}"
            );
        }
    }
}
```

```
На дворе 2025 год (обычный год).
Дата: 08.05.2025
```

Рис. 2.52. Результат работы задачи 1

В приведенном листинге в переменную `current_date_time` извлекается текущие системные дата и время в момент выполнения программы (в форме объекта типа `DataTime`). В переменную `current_year` записывается текущий год. Для вывода даты используются форматирующие символы.

Задача 2

В конкурсе на получение гранта «Молодой ученый» могут принять участие совершеннолетние студенты бакалавриата или специалитета не старше 30 лет, написавшие не менее 2-х ВАК статей и не состоящие в других грантах. В случае наличия не менее 3-х ВАК статей студенту также полагается дополнительная надбавка к стипендии.

Определить, может ли участвовать в конкурсе студент, если известны перечисленные данные, а также получить дополнительную надбавку.

Решение

В задаче требуется проверка условия, которое зависит от 4-х характеристик.

Глава 2\Grant\Program.cs

```
using System;

namespace Grant
{
    class Program
    {
        static void Main(string[] args)
        {
            int age = 21;
            string education_level = "специалитет";
            int articles = 2;
            bool has_grant = true;

            if (
                (18 <= age) && (age <= 30) &&
                (education_level == "бакалавриат" ||
                 education_level == "специалитет") &&
                (articles >= 2) &&
                !has_grant
            )
            {
```

```
                    Console.WriteLine("Рекомендовать к участию в
                        конкурсе на получение гранта");

                    if (articles >= 3)
                        Console.WriteLine("Студенту полагается
                            надбавка к стипендии");
                }
                else
                {
                    Console.WriteLine("К сожалению, студент не
                        может принять участие в конкурсе");
                }
            }
        }
}
```

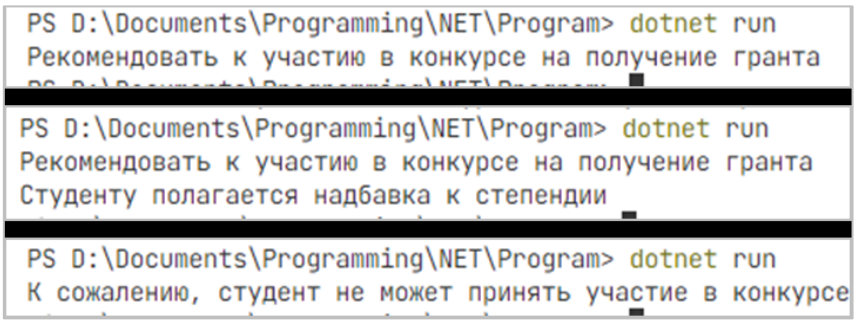


Рис. 2.53. Результат выполнения задачи 2 при разных стартовых данных

Задача 3

Реализовать алгоритм принятия решений согласно шуточной «памятке инженера». Исходные данные – текстовые ответы ДА/НЕТ на соответствующие вопросы.

Рис. 2.54. Схема для задачи

Решение

Согласно схеме потребуется три условные проверки: первая относительно ответа на вопрос «Оно двигается?», а вторая и третья «Оно должно двигаться?». При этом вторая и третья проверки будут вложенными в соответствующие блоки первой конструкции `if`.

Рис. 2.55. Наличие повторяющейся проверки

Глава 2\EngineeringScheme_1\Program.cs

```
using System;
using System.Text;

namespace EngineeringScheme
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.OutputEncoding =
                Encoding.GetEncoding("utf-16");
            Console.InputEncoding =
                Encoding.GetEncoding("utf-16");

            Console.Write("Оно двигается? ");
            string? answer1 = Console.ReadLine() ?? "";

            Console.Write("Оно должно двигаться? ");
            string? answer2 = Console.ReadLine() ?? "";

            if (answer1 == "да")
            {
                if (answer2 == "да")
```

```
                    Console.WriteLine("Все хорошо!");
                else
                    Console.WriteLine("Используй скотч");
            }
            else
            {
                if (answer2 == "да")
                    Console.WriteLine("Используй WD-40");
                else
                    Console.WriteLine("Все хорошо!");
            }

            Console.WriteLine(answer1);
            Console.WriteLine(answer2);

            Console.ReadKey();
        }
    }
}
```

```
Оно двигается? да
Оно должно двигаться? нет
```

Рис. 2.56. Результат выполнения задачи 3

Прежде всего отметим, что для корректного вывода и вывода русских символов через переменные в терминале Visual Studio Code может потребоваться установить стандарт кодировки UTF-16 (в случае, если программа некорректно работает или значения в переменных `answer1` и `answer2` не выводятся в терминал):

```
Console.OutputEncoding = Encoding.GetEncoding("utf-16");
Console.InputEncoding = Encoding.GetEncoding("utf-16");
```

Для использования указанных команд необходимо подключить короткий доступ к пространству имеен **`Text`**:

```
using System.Text;
```

Впрочем, для скомпилированного Exe-файла приложения перечисленные команды не требуются.

В начале задаем ответы на вопросы. Работать будем с текстовыми переменными `answer1` и `answer2`. Внешнее условие отвечает за первый вопрос, а вложенные в обе ветки – за второй.

Оптимизация решения

Однако возможности программы сейчас существенно ограничены: любой ответ на вопрос кроме «да» будет распознаваться как ложь, т.е. ответ «нет». Это связано с тем, что ответ «да» будет засчитываться как истина лишь для прописных букв, однако пользователь может, например, написать его заглавными или даже буквами с разным регистром и в этом случае программа сработает логически некорректно.

Одно из решений – расширить проверку условий дополнительными ограничениями. Поскольку нам нужен один из ответов, то связываем их логическим союзом ИЛИ.

Кроме того, громоздкие условия в операторе if – плохой стиль. В таких случаях до начала проверки условия можно поместить в логические переменные. Тем более, что условие для второго вопроса повторяется дважды.

Глава 2\EngineeringScheme_2\Program.cs

```
using System;
using System.Text;

namespace EngineeringScheme
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.OutputEncoding =
                Encoding.GetEncoding("utf-16");
            Console.InputEncoding =
                Encoding.GetEncoding("utf-16");

            Console.Write("Оно двигается? ");
            string? answer1 = Console.ReadLine() ?? "";

            Console.Write("Оно должно двигаться? ");
            string? answer2 = Console.ReadLine() ?? "";

            bool is_working =
                (answer1 == "да") ||
                (answer1 == "Да") ||
                (answer1 == "ДА");

            bool should_it_work =
```

```
                (answer2 == "да") ||
                (answer2 == "Да") ||
                (answer2 == "ДА");

            if (is_working)
            {
                if (should_it_work)
                    Console.WriteLine("Все хорошо!");
                else
                    Console.WriteLine("Используй скотч");
            }
            else
            {
                if (should_it_work)
                    Console.WriteLine("Используй WD-40");
                else
                    Console.WriteLine("Все хорошо!");
            }
        }
    }
}
```

```
Оно двигается? Нет
Оно должно двигаться? ДА
Используй WD-40
```

Рис. 2.57. Результат выполнения модификации задачи 3

Строки являются объектами и обладают полезными встроенными методами. В частности, метод **`ToLower()`** возвращает новую текстовую строку, преобразуя все буквы строки к нижнему регистру. Нам этот метод позволит свернуть проверку трех условий в одну.

Кроме того, предусмотрим еще ответы на английском языке и форме символа «+».

Глава 2\EngineeringScheme_3\Program.cs

```
using System;
using System.Text;

namespace EngineeringScheme
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            Console.OutputEncoding =
                Encoding.GetEncoding("utf-16");
            Console.InputEncoding =
                Encoding.GetEncoding("utf-16");

            Console.Write("Оно двигается? ");
            string? answer1 = Console.ReadLine() ?? "";

            Console.Write("Оно должно двигаться? ");
            string? answer2 = Console.ReadLine() ?? "";

            bool is_working =
                (answer1.ToLower() == "да") ||
                (answer1.ToLower() == "yes") ||
                (answer1 == "+");

            bool should_it_work =
                (answer2.ToLower() == "да") ||
                (answer2.ToLower() == "yes") ||
                (answer2 == "+");

            if (is_working)
            {
                if (should_it_work)
                    Console.WriteLine("Все хорошо!");
                else
                    Console.WriteLine("Используй скотч");
            }
            else
            {
                if (should_it_work)
                    Console.WriteLine("Используй WD-40");
                else
                    Console.WriteLine("Все хорошо!");
            }
        }
    }
}
```

```
Оно двигается?
Оно должно двигаться? да
Используй WD-40
```

Рис. 2.58. Результат выполнения модификации задачи 3

2.5.5. Оператор if в расширенной форме

Когда необходимо организовать последовательную проверку условия, где возможны более двух вариантов действий, используют **расширенную форму** оператора `if`.

```
if (условие_1)
{
    // условие_1 истинно
}
else if (условие_2)
{
    // условие_2 истинно
}
. . .
else if (условие_N)
{
    // условие_N истинно
}
else
{
    // все условия ложны
}
```

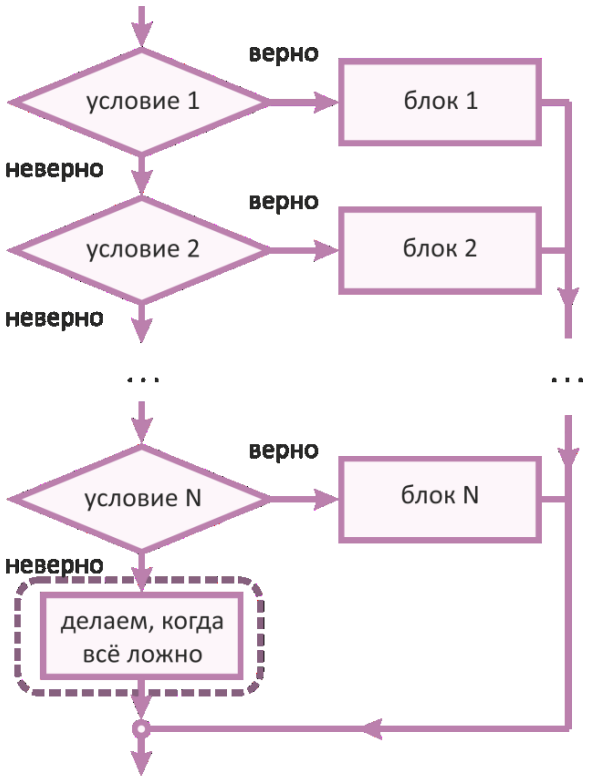


Рис. 2.59. Блок-схема расширенного оператора `if`

Расширенная конструкция `if` работает последовательно:

- если первое условие истинно, то выполняется соответствующий блок кода и осуществляется выход из конструкции;
- иначе осуществляется проверка второго условия; если оно истинно, то выполняется соответствующий блок кода и осуществляется выход из конструкции;
- и т.д., в противном случае выполняется блок `else` (который является необязательным).

На самом деле расширенный оператор лишь дополняет конструкцию `if-else` в ситуациях с нескольким выбором. Расширенная форма оптимальна для организации последовательных проверок и зачастую позволяет избежать многократного вложения условий.

Например, в следующем фрагменте определяется тип числа:

```
double number = -56;

if (number > 0)
{
    Console.WriteLine("Положительное");
}
else
{
    if (number == 0)
        Console.WriteLine("Нуль");
    else
        Console.WriteLine("Отрицательное");
}
```

Расширенный оператор позволяет избавиться от вложенной проверки, выстроив условия на одном уровне:

```
double number = -56;

if (number > 0)
    Console.WriteLine("Положительное");
else if (number == 0)
    Console.WriteLine("Нуль");
else
    Console.WriteLine("Отрицательное");
```

Однако расширенная форма не всегда может заменить вложенные проверки (см пример задачи 3 и из п. 2.5.4).

Рассмотрим еще один пример: требуется определить оценку учащегося, если известен его итоговый рейтинговый балл. Критерии оценивания указаны на изображении:

Рис. 2.60. Критерии выставления итоговой оценки

Рационально организовать последовательную проверку условий по возрастанию или убыванию баллов. Это позволяет избежать лишних условий при проверке. Здесь подойдет расширенный оператор `if`, который уберет многократные вложения условий.

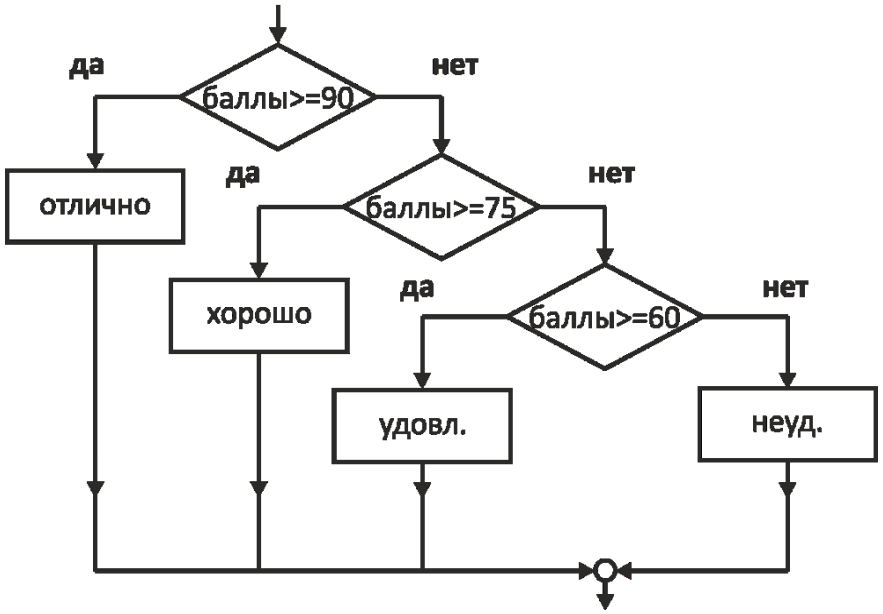


Рис. 2.61. Оптимальная блок-схема определения оценки

Глава 2\StudentMark\Program.cs

```
using System;

namespace StudentMark
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Баллы студента: ");
```

```
            string? text_input = Console.ReadLine() ?? "";

            if (int.TryParse(text_input, out int points))
            {
                string? mark;

                if (points >= 90)
                    mark = "отлично";
                else if (points >= 75)
                    mark = "хорошо";
                else if (points >= 60)
                    mark = "удовлетворительно";
                else
                    mark = "неудовлетворительно";

                Console.WriteLine($"Баллы: {points}");
                Console.WriteLine($"Оценка: {mark}");
            }
            else
            {
                Console.Write("Некорректное значение");
            }
        }
    }
}
```

2.5.6. Тернарный оператор

Синтаксис тернарного оператора

Определение

Тернарная *условная операция – оператор вида*

`Условие ? Выражение1 : Выражение2`

возвращающий ***Выражение1*** *при истинном условии, либо* ***Выражение2*** *при ложном условии.*

Название «тернарный» связано с тем, что этот оператор имеет три операнда. **Условие** является выражением логического типа, символ **?** отделяет условие от двух возможных значений, а **:** разделяет два возможных значения для истинного или ложного условия.

Тернарный оператор удобен в том случае, когда необходимо выбрать одно из двух значений в зависимости от условия. Обычно тернарный оператор используется для записи в переменную значения, выбираемого относительно истинности условия.

```
переменная = условие ? выражение_истина : выражение_ложь;
```

В подобной форме он аналогичен проверке с помощью оператора `if-else`:

```
if (условие)
    переменная = выражение_истина;
else
    переменная = выражение_ложь;
```

Например, поиск наибольшего значения из двух заданных с помощью `if-else` имеет вид (в полной форме):

```
double m = 12;
double n = 23;
double max;

if (m > n)
    max = m;
else
    max = n;
```

Для тернарного оператора выражение можно записать непосредственно в переменную `max`:

```
double m = 12;
double n = 23;
double max = m > n ? m : n;
```

По существу тернарный оператор представляет собой **условное выражение**. Благодаря этому его удобно использовать в вычислениях, чтобы избежать разветвленных действий и скомпоновать операцию с условным выбором в одну команду.

В следующем примере вычисляется зарплата работника на основе базового оклада и бонуса за эффективную работу берется как доля от оклада). Если работник имеет бонус, то тернарный оператор возвратит его и подставит в формулу вычисления зарплаты, иначе возвратит нуль (т.е. работник получит только оклад):

```
double base_salary = 48600;
bool has_bonus = true;
double quarterly_bonus_coeff = 0.35;

double employee_salary =
    base_salary *
    ( 1 + (has_bonus ? quarterly_bonus_coeff : 0) );

Console.WriteLine(
    $"Зарплата работника: {employee_salary} ₽."
);
```

Вложение тернарного оператора

Тернарный оператор допускает вложение, что позволяет реализовать расширенную форму оператора `if`.

Так, выбор типа числа из следующего фрагмента кода

```
double number = -56;

string? number_type;
if (number > 0)
    number_type = "положительное";
else if (number == 0)
    number_type = "нуль";
else
    number_type = "отрицательное";

Console.WriteLine($"Число: {number_type}");
```

можно свернуть с помощью тернарного оператора следующим образом:

```
double number = -56;

string? number_type =
    (number > 0)
    ? "положительное"
    : (number == 0)
        ? "нуль"
        : "отрицательное";

Console.WriteLine($"Число: {number_type}");
```

(здесь мы намеренно не пишем оператор в одну строку, чтобы его было проще ассоциировать с условным выбором).

2.5.7. Вопросы для самопроверки

1. Перечислите операторы отношения, поддерживаемые в языке C#.
2. Почему не рекомендуется сравнивать вещественные числа на равенство с помощью оператора ==? Опишите один из методов сравнения чисел с дробной частью.
3. В чем разница между обычными и укороченными логическими союзами?
4. Перечислите формы оператора `if`. Приведите примеры использования каждой из форм.
5. Опишите рекомендации, когда стоит и не стоит использовать тернарный оператор.

Практикум

1. Конструкция if-else

Задание 1

1. Создайте новый проект *Maximum*.
2. С клавиатуры вводятся два числа. Вывести на экран наибольшее из них и указать, какое оно по счету (если оба числа равны, наибольшим считать первое).

```
Введите первое число: 34,2
Введите второе число: 50
Второе больше. Оно равно 50
```

```
Введите первое число: 5
Введите второе число: 0,4
Первое больше. Оно равно 5
```

Рис. 2.62. Результат работы программы из задания 1

Задание 2

1. Создайте проект *Sector*.
2. Определить, попадает ли точка в сектор, если известны ее координаты (задаются с клавиатуры).
3. Сектор представляет собой замкнутую область между прямой и окружность (рис. 2.63).
4. Вывод оформить согласно рис. 2.64.

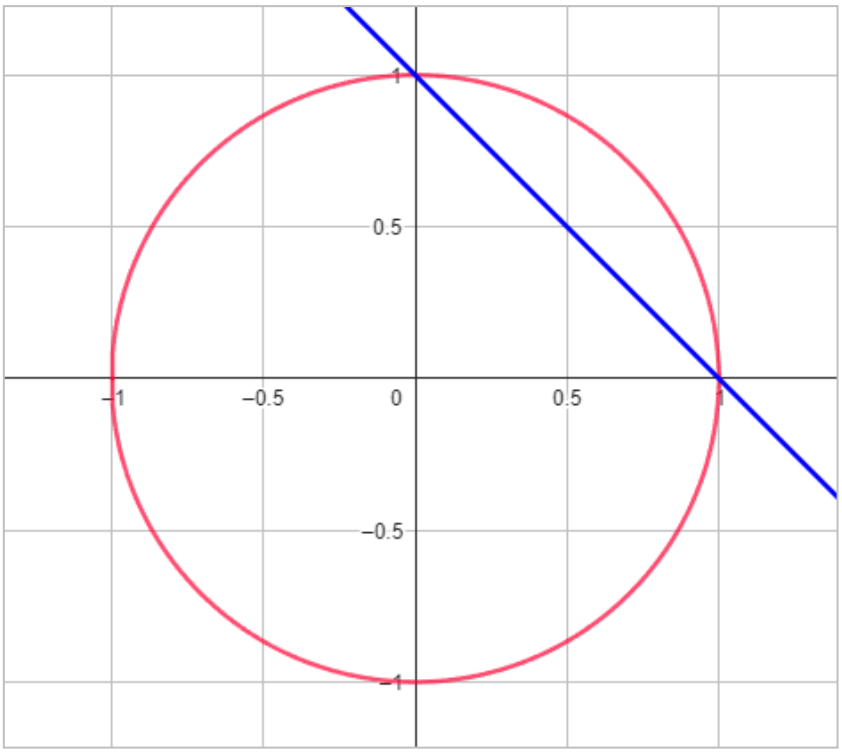


Рис. 2.63. Искомый сектор для задания 2

```
Введите координаты точки:
x = 0,52
y = 0,5
Точка M(0,52,0,5) попадает в искомую область
```

```
Введите координаты точки:
x = 0,49
y = 0,4
Точка M(0,49,0,4) не попадает в искомую область
```

Рис. 2.64. Результат работы программы из задания 2

Задание 3

1. Создайте проект *Function*.
2. С клавиатуры запрашивается значение x. В зависимости от его величины вычислить значение функции:

$$f(x) = \begin{cases} |x|\sin(x), & x < -2, \\ e^{2x}, & -2 \leqslant x < 0, \\ 1, & x = 0, \\ \ln(x^2+1), & x > 0 \end{cases}$$

3. Используйте заданный ниже код.
4. Ввод и вывод оформить следующим образом:

```
x = -1,3
f(-1,3) = 0,0742736
```

Рис. 2.65. Результат работы программы из задания 3

Глава 2\Function\Program.cs

```
using System;

namespace Function
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("x = ");
            string? input_text = Console.ReadLine() ?? "";

            if (!double.TryParse(input_text, out double x))
            {
                x = 0;
            }

            // Далее с помощью расширенного if вычислить
            // значение f
            double f;

        }
    }
}
```

Задание 4

1. Создайте проект *Quadratic_Equation*.
2. Написать программу, вычисляющую корни квадратного уравнения
$$ax^2 + bx + c = 0$$
при заданных коэффициентах $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$. Предусмотреть всевозможные случаи коэффициентов, в том числе $a = 0$ и расчет комплексных корней.
3. Протестируйте работу приложения на разных комбинациях коэффициентов (рис. 2.66).

```
PS D:\Documents\Programming\NET\Program> dotnet run
Введите коэффициенты уравнения
a = 0
c = 2,8
Уравнение вырождается в линейное
x = -0,93333
PS D:\Documents\Programming\NET\Program> dotnet run
Введите коэффициенты уравнения
a = 0
b = 5
c = 0
Уравнение вырождается в линейное
x = -0,00000
PS D:\Documents\Programming\NET\Program> dotnet run
Введите коэффициенты уравнения
a = 0
b = 0
c = -5
Уравнение вырождается в линейное
Корней нет
PS D:\Documents\Programming\NET\Program> dotnet run
Введите коэффициенты уравнения
a = 0
b = 0
c = 0
Уравнение вырождается в линейное
x - любое вещественное
```

```
PS D:\Documents\Programming\NET\Program> dotnet run
Введите коэффициенты уравнения
a = 2
b = 10,5
c = 3
Уравнение является квадратным
x1 = -4,94677
x2 = -0,30323
PS D:\Documents\Programming\NET\Program> dotnet run
Введите коэффициенты уравнения
a = 1
b = -3
c = 4
Уравнение является квадратным
x1 = (1,50000) - (1,32288)i
x2 = (1,50000) + (1,32288)i
PS D:\Documents\Programming\NET\Program> dotnet run
Введите коэффициенты уравнения
a = 1
b = 6
c = 9
Уравнение является квадратным
x = -3,00000
PS D:\Documents\Programming\NET\Program> dotnet run
Введите коэффициенты уравнения
a = 1
b = 0
c = 1
Уравнение является квадратным
x1 = (-0,00000) - (1,00000)i
x2 = (-0,00000) + (1,00000)i
```

Рис. 2.66. Результат работы программы из задания 4

Задание 5

1. Создайте проект *Quadratic_Equation*.
2. Пользователь с клавиатуры задает коэффициенты ***a***, ***b*** и один из знаков: $\geq$, $=$, $\leq$.
3. Написать программу, решающую соответствующее неравенство (уравнение):

$$ax + b \blacksquare 0$$

в зависимости от знака и коэффициентов.

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите коэффициент a: 0
Введите коэффициент b: 0
Уравнение выполняется для всех x.
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите коэффициент a: 3
Введите коэффициент b: 0
Введите знак (>=, =, <=): =
x = -0,00
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите коэффициент a: 0
Введите коэффициент b: 3
Введите знак (>=, =, <=): =
Уравнение не имеет решений.
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите коэффициент a: 4
Введите коэффициент b: -3
Введите знак (>=, =, <=): =
x = 0,75
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите коэффициент a: 4
Введите коэффициент b: 5
Введите знак (>=, =, <=): >=
x >= -1,25
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите коэффициент a: 0
Введите коэффициент b: -5
Введите знак (>=, =, <=): <=
Уравнение выполняется для всех x.
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите коэффициент a: -1,3
Введите коэффициент b: 2
Введите знак (>=, =, <=): >=
x >= 1,54
```

Рис. 2.67. Результат работы программы из задания 5

2. Тернарный оператор

Задание 1

1. Создайте проект *FunctionTernary*.
2. Используя возможности тернарного оператора, запишите результаты выбора в следующие переменные:
   a. Из заданных с клавиатуры $\boldsymbol{a}$ и $\boldsymbol{b}$ в переменную `min` записать наименьшее значение.
   b. Далее в логическую `is_even_positive` записать `true`, если `min` является четным положительным, иначе – `false`.
   c. Для заданной с клавиатуры $\boldsymbol{n}$ в переменную `result` записать $2n$, если n четное положительное, иначе записать $-2n$.
3. Оформите вывод, как изображено на рис. 2.68.

```
a = 56
b = 23
Введите целое число n: 89

min = 23
min четное положительное? = False
result = -178
```

Рис. 2.68. Результат работы программы из задания 1

Задание 2

1. Создайте проект *StudentMarkTernary*.
2. Используя разобранный в занятии пример, перепишите условный выбор через вложение тернарных операторов.

```
Баллы студента: 86

Баллы: 86
Оценка: хорошо
```

Рис. 2.69. Результат работы программы из задания 2

Задание 3

1. Создайте проект *FunctionTernary*.
2. Используйте код ранее выполненной задачи *Function*.
3. Перепишите выпор значения функции по ее аргументу, используя тернарные операторы.

2.6. Конструкция выбора switch

2.6.1. Синтаксис switch

Синтаксис оператора

Определение

Оператор **`switch`** *(«выбор», «**переключение**») в зависимости от значения выражения осуществляет выполнение соответствующего блока кода. Блок* **`default`** *выполняется, если ни одно из условий не дало истины; является необязательным.*

Конструкция **`switch-case`** позволяет выбрать определенный набор действий в зависимости от значения некоторого выражения, относительно которого осуществляется выбор.

Конструкция `switch` имеет следующий синтаксис:

```
switch(выражение)
{
    case значение_1:
        // блок операторов 1
        break;
    case значение_2:
        // блок операторов 2
        break;
    . . .
    default:
        // блок операторов N
        break;
}
```

Обычно **`выражение`** – это переменная, относительно которой осуществляется выбор дальнейшего **кейса**. В качестве **`значение_1`**, **`значение_2`** и т.д. выступают значения, которое может принять выражение. Конструкция последовательно сравнивает выражение со значением: если они совпадают, то выполняется соответствующий блок **`case`**.

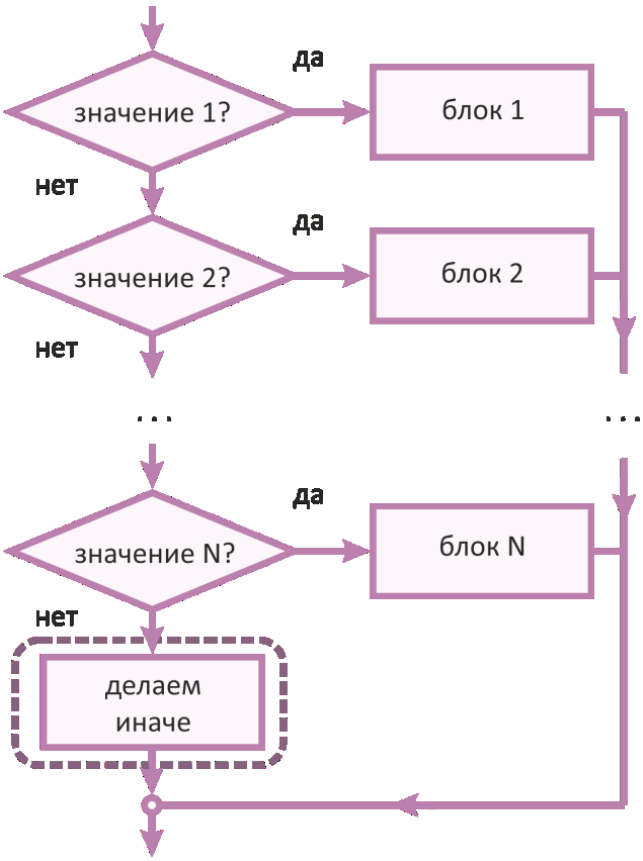


Рис. 2.70. Блок-схема конструкции `switch` в полной форме

Обычно каждый case-блок завершается оператором **`break`**. Кроме того можно использовать команды:

- **`goto case`** – переход на соответствующий кейс;
- **`return`** – возврат из метода (в т.ч. со значением);
- **`throw`** – искусственная генерация исключения.

В следующем примере в зависимости от размера ячейки в байтах выдается ее название:

```
int memory_cell = 2;

switch (memory_cell)
{
    case 1:
        Console.WriteLine("BYTE");
        break;
    case 2:
        Console.WriteLine("WORD");
        break;
    case 4:
        Console.WriteLine("DWORD");
```

```
            break;
        default:
            Console.WriteLine("ERROR");
            break;
    }
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
WORD
```

Рис. 2.71. Выбор соотвествующего кейса

Типы выражения в switch

Выражение в операторе `switch` должно иметь тип, который поддерживает операцию сравнения на равенство, либо тип данных, который можно к нему преобразовать.

Изначально `switch` мог работать с целочисленными выражениями (типами `char`, `byte`, `short`, `int` и др.), строками и перечислениями (далее). А выражения других типов, например с плавающей точкой, в операторе `switch` не допустимы.

Начиная с C# 7.0 `switch` стал поддерживать работу с более сложными шаблонами и конструкциями. Тем не менее `switch` отдают предпочтение в том случае, когда выражение имеет однородный тип и множество констант в качестве возможных значений.

Значения констант выбора должны быть совместимы с типом выражения.

Примеры использования

Пример 1

Пусть светофор закодирован тремя состояниями: 1 – красный, 2 – желтый, 3 – зеленый. В зависимости от состояния требуется определить цвет светофора.

Глава 2\TrafficLight_1\Program.cs

```
using System;

namespace TrafficLight
{
    class Progam
    {
        static void Main(string[] args)
        {
```

```
            byte color = 2;

            switch (color)
            {
                case 1:
                    Console.WriteLine("Красный");
                    break;
                case 2:
                    Console.WriteLine("Желтый");
                    break;
                case 3:
                    Console.WriteLine("Зеленый");
                    break;
                default:
                    Console.WriteLine("Светофор не
                        работает!");
                    break;
            }
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Желтый
```

Рис. 2.72. Результат работы программы

Пример 2

Пусть задан день недели на английском языке.

С помощью `switch` требуется записать в отдельную переменную значение на русском.

Глава 2\WeekdayNames_1\Program.cs

```
Апрпа
using System;

namespace WeekdayNames
{
    class Progam
    {
        static void Main(string[] args)
        {
            string? day_of_week_en = "Wednesday";
            string? day_of_week_ru = "";
```

```
            switch (day_of_week_en)
            {
                case "Monday":
                    day_of_week_ru = "Понедельник";
                    break;
                case "Tuesday":
                    day_of_week_ru = "Вторник";
                    break;
                case "Wednesday":
                    day_of_week_ru = "Среда";
                    break;
                case "Thursday":
                    day_of_week_ru = "Четверг";
                    break;
                case "Friday":
                    day_of_week_ru = "Пятница";
                    break;
                case "Saturday":
                    day_of_week_ru = "Суббота";
                    break;
                case "Sunday":
                    day_of_week_ru = "Воскресенье";
                    break;
                default:
                    day_of_week_ru = "Не определено";
                    break;
            }

            Console.WriteLine($"День недели (на английском):\t
                {day_of_week_en}");
            Console.WriteLine($"День недели (на русском):\t
                {day_of_week_ru}");
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
День недели (на английском):    Wednesday
День недели (на русском):       Среда
```

Рис. 2.73. Результат работы программы

Проваливание

Оператор `switch` допускает побочный эффект – т.н. «**проваливание**». Если убрать команду `break` в блоке `case`, то после выполнения его инструкций происходит переход к следующему ниже блоку `case`. При этом соответствие литералу уже не проверяется, выполняется соответствующий код этого блока.

Так, если в примере выше убрать тело блоков 1 и 2, то все три значение выражения `color` можно рассматривать как рабочее состояние светофора.

Глава 2\TrafficLight_2\Program.cs

```
using System;

namespace TrafficLight
{
    class Progam
    {
        static void Main(string[] args)
        {
            byte color = 2;

            switch (color)
            {
                case 1:
                case 2:
                case 3:
                    Console.WriteLine("Светофор работает");
                    break;
                default:
                    Console.WriteLine("Светофор не
                        работает!");
                    break;
            }
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Светофор работает
```

Рис. 2.74. Результат работы программы

Выражение сопоставления шаблонов switch

В версии C# 8.0 появилась возможность модифицировать конструкцию переключения `switch` в укороченное **выражение сопоставления шаблонов**. Его можно использовать в случаях, когда требуется сопоставить определенным значениям одного выражения значения другого.

Преобразуем код примера *WeekdayNames_1*:

Глава 2\WeekdayNames_2\Program.cs

```
using System;

namespace WeekdayNames
{
    class Progam
    {
        static void Main(string[] args)
        {
            string? day_of_week_en = "Wednesday";

            string? day_of_week_ru = day_of_week_en switch
            {
                "Monday" => "Понедельник",
                "Tuesday" => "Вторник",
                "Wednesday" => "Среда",
                "Thursday" => "Четверг",
                "Friday" => "Пятница",
                "Saturday" => "Суббота",
                "Sunday" => "Воскресенье",
                _ => "Не определено",
            };

            Console.WriteLine($"День недели (на английском):\t
                {day_of_week_en}");
            Console.WriteLine($"День недели (на русском):\t
                {day_of_week_ru}");
        }
    }
}
```

Сопоставление ведется относительно переменной `day_of_week_en`. Значению константы из перечисления сопоставляется текстовое значение и записывается в другую переменную `day_of_week_ru`. Оператор **_** («**дискарт**») – это блок **`default`**.

2.6.2. Перечисления

Перечисления как именованные константы

C# позволяет объединять логически связанные числовые константы в **перечисления**. Это отдельный тип данных, который скрывает группу целых чисел под осмысленными названиями. За каждым значением перечисления закрепляется число, начиная с нуля.

Описание перечисления начинается с команды **enum**. Далее в фигурных скобках через запятую перечисляются названия числовых констант:

```
enum Color { Red, Yellow, Green };
```

Рис. 2.75. Описание перечислимого типа (каждая константа имеет номер согласно порядку перечисления)

Перечисление базируется на целочисленном типе `int`, но при необходимости его можно поменять, указав после двоеточия: `byte`, `short`, или `long`.

```
enum Color : byte { Red, Yellow, Green };
```

Значения констант в перечислениях не обязаны идти непрерывно. Например:

```
enum MemoryCell
{
    Empty = 0,
    Byte= 1,
    Word = 2,
    Dword = 4
}
```

Перечисления в улучшении читаемости кода

Главная цель использования перечислений – повысить абстрактность и улучшить читаемость кода. Они позволяют избавиться от «магических» чисел – это константы в коде, смысл которых трудно понять без предварительного описания или комментария (как в задаче со светофором).

Особенно часто перечисления используются в конструкции переключения `switch`.

Так в примерах *TrafficLight_1* и *TrafficLight_2* нельзя однозначно понять, что означает значение 2 и почему именно оно: необходимо искать описание условий задачи:

```
byte color = 2;
```

Описание и использование перечисления позволит внести ясность в код:

Глава 2\TrafficLight_3\Program.cs

```
using System;

namespace TrafficLight
{
    class Progam
    {
        enum TrafficColor { Red, Yellow, Green, Error };

        static void Main(string[] args)
        {
            TrafficColor traffic_light = TrafficColor.Yellow;

            switch (traffic_light)
            {
                case TrafficColor.Green:
                    Console.WriteLine("Красный");
                    break;
                case TrafficColor.Yellow:
                    Console.WriteLine("Желтый");
                    break;
                case TrafficColor.Red:
                    Console.WriteLine("Красный");
                    break;
                default:
                    Console.WriteLine("Светофор не
                        работает!");
                    break;
            }
        }
    }
}
```

Описание перечисления выносится как отдельный член класса `Program`: так он будет доступен всем методам внутри класса. Переменная `traffic_light` может принять только одно из 4-х значений. По их названию нетрудно понять, что это за состояния светофора.

Также магические константы убраны и в процессе выбора оператором `switch`.

Перечисления решают следующие задачи:

- Качественно абстрагируют код, убирая магические константы под оболочку именованных констант.
- Замыкают возможный диапазон значений, которые могут принимать переменные.
- Упрощают набор кода, активируя режим интеллектуальных подсказок в среде разработки / редакторе кода.
- Часто применяются в операторе `switch`, делая его реализацию безопасной.

Это полезно знать!

Редактор Visual Studio выдает выпадающую подсказку, предлагая выбрать доступные значения переменной из перечисления.

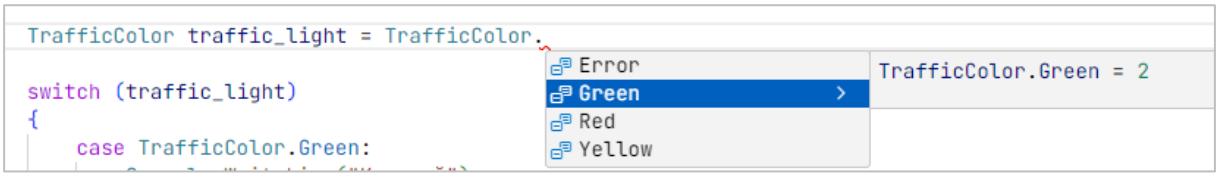


Рис. 2.76. Замыкание возможных значений перечисления

Работа с перечислениями

Само по себе перечисление лишь скрывает числовую константу.

При обращении к переменной типа перечисления возвращается название константы.

```
TrafficColor traffic_light = TrafficColor.Yellow;
Console.WriteLine(traffic_light);          // Yellow
```

А чтобы узнать ее числовое значение, нужно осуществить явное преобразование типа переменной к типу `int`.

```
Console.WriteLine((int) traffic_light);    // 1
```

Перечисления очень удобны в проверке диапазонов.

```
DayOfWeek today = DateTime.Today.DayOfWeek;

if (
    (DayOfWeek.Monday <= today) &&
    (today <= DayOfWeek.Friday)
)
{
    Console.WriteLine("Рабочий день");
}
```

Встроенное в пространство имен `System` перечисление `DayOfWeek` включает 7 дней недели, с воскресенья по субботу. С помощью класса `DateTime` получаем сегодняшний день недели и записываю в переменную `today`.

Дни недели в перечислении `DayOfWeek` – это числовые константы, пронумерованные с нуля, что позволяет вместо сравнения чисел сравнивать абстрактные названия дней недели.

Это полезно знать!

*Все, что способен реализовать оператор **switch**, можно достичь и с помощью комбинаций конструкции **if-else**.*

*Однако оператор **switch** удобен для тех случаев, когда выбор ведется среди большого набора связанных значений. Разработчики также отмечают, что в этом случае **switch** работает несколько быстрее, чем конструкции **if**. В остальных следует отдавать предпочтение **if**.*

*Впрочем, начиная с версии C# 7.0 в конструкцию **switch** были внесены ряд изменений, несколько расширяющих ее гибкость и возможности. Однако в текущем курсе они не столь принципиальны.*

*Вообще говоря, часто оператор **switch** используется именно совместно с перечислениями.*

2.6.3. Решение практических задач

Задача 1

Пользователю выдается оформленное в виде текста меню, которое запрашивает код дальнейшей операции. Требуется подготовить каркас этого приложения, не реализуя сами пункты меню.

Решение

Глава 2\ConsoleMenu\Program.cs

```
using System;

namespace ConsoleMenu
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.Clear();
            Console.WriteLine("----------------------------");
            Console.WriteLine("МЕНЮ:");
            Console.WriteLine("----------------------------");
            Console.WriteLine("1. Открыть директории с
                файлами");
            Console.WriteLine("2. Получить информацию о
                файлах");
            Console.WriteLine("3. Записать новые файлы");
            Console.WriteLine("4. Сделать резервную копию");
            Console.WriteLine("5. Выход");
            Console.WriteLine("----------------------------");
            Console.Write("Введите номер пункта меню: ");

            if (int.TryParse(
                    Console.ReadLine(), out int user_case)
            )
            {
                switch (user_case)
                {
                    case 1:
                        Console.WriteLine("Загрузка
                            директорий...");
                        // Реализация кода
                        break;
                    case 2:
```

```
                        Console.WriteLine("Сбор
                            информации...");
                        // Реализация кода
                        break;
                    case 3:
                        Console.WriteLine("Подготовка к
                            записи...");
                        // Реализация кода
                        break;
                    case 4:
                        Console.WriteLine("Создание резервной
                            копии...");
                        // Реализация кода
                        break;
                    case 5:
                        Console.WriteLine("Завершение
                            сеанса");
                        // Реализация кода
                        break;
                    default:
                        Console.WriteLine("Номер пункта
                            некорректен");
                        break;
                }
            }
            else
            {
                Console.WriteLine("Некорректный ввод");
            }
        }
    }
}
```

```
------------------------------
МЕНЮ:
------------------------------
1. Открыть директории с файлами
2. Получить информацию о файлах
3. Записать новые файлы
4. Сделать резервную копию
5. Выход
------------------------------
Введите номер пункта меню: 3
Подготовка к записи...
```

Рис. 2.77. Пример реализации меню

Задача 2

Задано перечисление, в котором описаны должности в компании. С помощью оператора `switch` требуется сопоставить должность и оклад каждого типа работника.

Решение

Глава 2\Company\Program.cs

```
using System;

namespace Company
{
    public class Program
    {
        enum JobPositions
        {
            Director = 1,
            Manager,
            Administrator,
            DepartmentHead,
            Designer,
            Developer,
            Error
        }

        public static void Main(string[] args)
        {
            Console.Write("Введите код интересующей вас
                должности: ");
            string? console_input = Console.ReadLine() ?? "";

            JobPositions user_case;

            if (int.TryParse(console_input,
                out int user_case_code))
            {
                // Переводим числовой код в перечислимое
                // значение
                user_case = user_case_code switch
                {
                    1 => JobPositions.Director,
                    2 => JobPositions.Manager,
                    3 => JobPositions.Administrator,
                    4 => JobPositions.DepartmentHead,
```

```
                    5 => JobPositions.Designer,
                    6 => JobPositions.Developer,
                    _ => JobPositions.Error
                };

                // Переход от перечислимого значения к строке
                string? user_case_name = user_case switch
                {
                    JobPositions.Director => "Директор",
                    JobPositions.Manager => "Менеджер",
                    JobPositions.Administrator =>
                        "Администратор",
                    JobPositions.DepartmentHead =>
                        "Начальник отдела",
                    JobPositions.Designer => "Дизайнер",
                    JobPositions.Developer => "Разработчик",
                    _ => "Нет должности с указанным кодом"
                };

                // Сопоставляем оклад
                decimal salary_size = user_case switch
                {
                    JobPositions.Director => 230_000,
                    JobPositions.Manager => 148_000,
                    JobPositions.Administrator => 69_000,
                    JobPositions.DepartmentHead => 82_000,
                    JobPositions.Designer => 50_000,
                    JobPositions.Developer => 48_000,
                    _ => 0
                };

                Console.WriteLine($"Вы выбрали должность:
                    {user_case_name}");
                Console.WriteLine($"Шифр: {user_case}");
                Console.WriteLine($"Оклад: {salary_size:F2}
                    р.");
            }
            else
            {
                Console.WriteLine("Некорректный ввод");
            }
        }
    }
}
```

В приведенном примере оператор `switch` для краткости сворачивается в более компактные выражения шаблонов сопоставления. В начале числовой код переводится в абстрактную величину `user_case` перечислимого типа `JobPositions`, далее выбирается соответствующее название должности, и, наконец – соответствующий оклад сотрудника.

2.6.4. Рекомендации учителю

- При обучении школьников операциям отношения и сравнения начните с простых отношений. Далее переходите к разбору задач с составными условиями и логическими союзами, постепенно усложняя условные зависимости.
- Комбинируйте в условиях операции с переменными разного типа.
- Рационально также задействовать правила математической логики, которые позволяют упрощать составные условия, когда это возможно.
- Разберите задачи на разные вариации оператора `if`. Покажите на примерах, когда расширенная форма оператора может заменить вложенную проверку, а когда – нет.
- Разберите задачи, позволяющие задействовать оператор переключения `switch`. Проведите сравнительный анализ его достоинств и недостатков с `if`.
- Покажите, в чем важность избавления от «магических» чисел в коде и как в этом помогают перечисления.

Вопросы для самопроверки

1. Опишите синтаксис оператора `switch`. Почему его называют оператором переключения?
2. Что общего у конструкции `switch` и расширенной `if-else`?
3. Приведите примеры практического использования эффекта «проваливания» в `switch`.
4. Что такое выражение сопоставления шаблонов и как оно упрощает код?
5. Для каких целей используют перечисления?

Практикум

Задание 1

1. Создайте новые проект *Robot*.
2. Направление движения робота кодируются следующим образом:
   - 1 – двигаться на север;
   - 2 – двигаться на восток;
   - 3 – двигаться на юг;
   - 4 – двигаться на запад;
   - 0 – поломка робота (или любое другое число).
3. В зависимости от заданного кода операции определить направление движение робота. Использовать оператор `switch`.
4. В начале программы с клавиатуры запрашивается код инструкции (число).

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите код инструкции для робота: 2
Восток
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите код инструкции для робота: 0
Поломка!!!
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите код инструкции для робота: 4
Запад
```

Рис. 2.78. Результат работы программы из задания 1

Задание 2

1. Создайте новые проект *RobotEnum*.
2. Перепишите предыдущую задачу, задав вместо кодов перечисление `Direction` (с указанием именных идентификаторов каждого направления и ошибки).

Задание 3

1. Создайте новые проект *FileMenu*.
2. По аналогии с рассмотренным в текущем занятии примером с меню самостоятельно реализуйте шаблон консольного меню, которое запрашивает от пользователя некоторые операции с файлом (сами операции реализовать не нужно).

2.7. Циклические конструкции while и do-while

2.7.1. Циклы в C#

В языке C# реализовано четыре оператора для организации циклов.

1. Цикл while

Цикл **`while`** относится к циклам с **предусловием**: этот цикл выполняется до тех пор, пока условие истинно, а проверка осуществляется перед каждой **итерацией** (витком цикла). Поэтому команды в теле цикла могут ни разу не выполниться.

2. Цикл do-while

Цикл **`do-while`** относится к циклам с **постусловием**: этот цикл также выполняется до тех пор, пока условие истинно, но проверка осуществляется после осуществления итерации. Поэтому команды в теле цикла выполняются хотя-бы один раз (в начале).

3. Цикл for

Цикл **`for`** является циклом с предусловием и обычно используется в том случае, когда число итераций известно заранее. В цикле можно определить начальное значение некоторой переменной, выступающей в роли счетчика, условие выполнения и правило изменения счетчика после каждого витка цикла.

Впрочем цикл for является универсальным циклом и, как будет показано далее, может быть альтернативной для любых других.

4. Цикл foreach

Цикл **`foreach`** реализован для более удобной обработки элементов массивов или коллекций. Он упрощает синтаксис, однако работает только на обход и чтение элементов.

2.7.2. Цикл с предусловием while

Синтаксис цикла

Определение

Оператор **`while`** *реализует цикл с предусловием.*
Условие является выражением логического типа.
Если в теле цикла один оператор, фигурные скобки можно опустить.

Для цикла `while` проверка условия осуществляется в начале: если оно ложно изначально, то тело цикла не выполняется ни разу.

Цикл имеет следующий синтаксис:

```
while (условие)
{
    // тело цикла
}
```

Цикл продолжает работу до тех пор, пока условие истинно.

Круглые скобки для условия обязательны – это часть оператора.

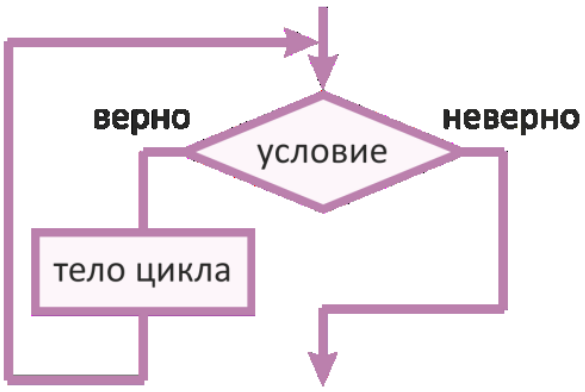


Рис. 2.79. Блок-схема конструкции `while`

Цикл `while` рационально использовать в ситуациях, когда заранее неизвестно количество итераций и оно зависит от выполнения некоторого простого или составного условия. Например, для задач обработки ввода и вывода, выполнении команд до достижения определенных условий, создании бесконечных циклов (в этом случае выход осуществляется посредством внутренних проверок и команд прерывания цикла).

Приведем простой пример использования цикла `while` – вывод целых чисел от 1 до 100:

```
int n = 100;
int counter = 1;

while (counter <= n)
{
    Console.WriteLine("Число: " + counter);
    counter++;
}
```

В данном случае условие цикла зависит от переменной `counter` и `n`: первая выполняет роль счетчика, который наращивается на каждом витке цикла, а вторая является ограничением. Цикл повторится ровно 100 раз, а на 101 осуществится выход из цикла

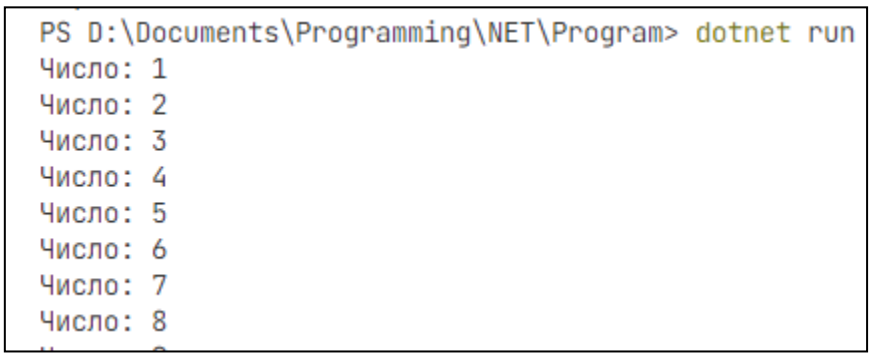


Рис. 2.80. Вывод чисел в порядке возрастания с помощью цикла `while`

Составные условия

Условие цикла мажет быть составным и зависеть от нескольких переменных и ограничений.

В следующем примере условие работы цикла определяется переменными `i` и `j`: пока `i` не превосходит `j` и их произведение не более 2025, выводим числа на экран, увеличиваем `i` и уменьшаем `j` на1:

```
int i = 0;
int j = 100;

while (i <= j && i * j <= 2025)
{
    Console.WriteLine($"{i}, {j}");
    i++;
    j--;
}
```

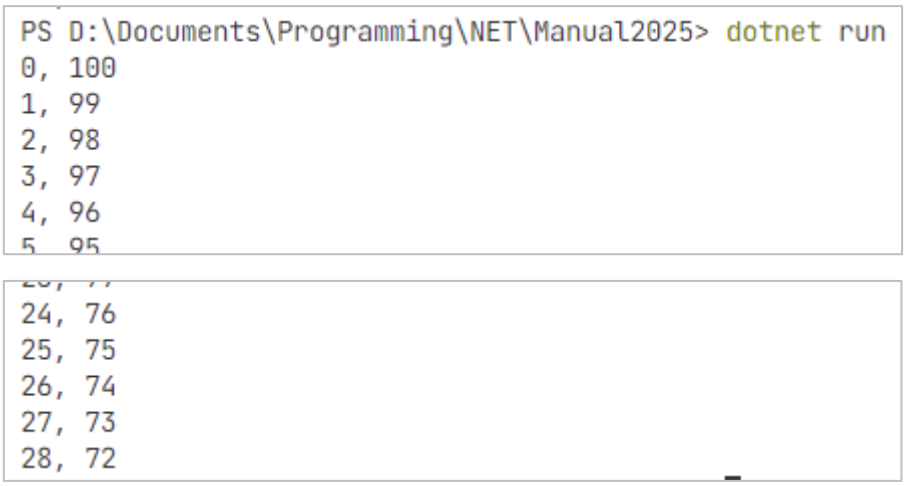


Рис. 2.81. Контроль нескольких переменных в условии

2.7.3. Цикл с постусловием do-while

Синтаксис цикла

Определение

Оператор ***do-while*** *реализует цикл с постусловием.*
Условие является выражением логического типа.
Если в теле цикла один оператор, фигурные скобки можно опустить

Цикл `do-while` проверяет условие в конце. Поэтому тело цикла в любом случае выполняется хотя бы один раз. Если условие истинно, осуществляется повторная итерация и т.д. до тех пор, пока условие не даст ложный результат.

Синтаксис цикла:

```
do
{
    // тело цикла
} while (условие);
```

Цикл `do-while` рационален в тех случаях, когда последовательность операций в теле цикла должна быть осуществлена в любом случае и хотя бы один раз.

Перепишем код задачи вывода чисел от 1 до 100, используя цикл `do-while`:

```
int n = 100;
int counter = 1;

do
{
    Console.WriteLine("Число: " + counter);
    counter++;
} while (counter <= n);
```

В этом случае нужно учитывать, что обе реализации с циклами `while` и `do-while` работают одинаково при $n \geq 1$. Если же задать $n = 0$, то цикл `do-while` сработает некорректно и выведет 1, потому что тело этого цикла сработает до проверки условия.

Работа цикла на примере обработки ввода с клавиатуры

Цикл `do-while` позволяет в более удобной и очевидной форме реализовать циклическую проверку ввода текста с клавиатуры, до тех пор, пока не будут достигнуты требуемые ограничения.

Например, в следующей программе запрашивается ввод текста до тех пор, пока он не совпадет с ключевой фразой («паролем»).

Глава 2\Password\Program.cs

```
using System;

namespace Password
{
    class Program
    {
        static void Main(string[] args)
        {
            const string? password = "Denis25";

            do
            {
                Console.Clear();
                Console.Write("Введите пароль: ");
            } while (Console.ReadLine() != password);

            Console.WriteLine("Вход осуществлен");
        }
    }
}
```

В реализованной программе в начале очищаем окно консоли, далее запрашивается ввод текста и если он не совпадает с ключевым словом, цикл повторяется.

2.7.4. Решение практических задач

Задача 1

Осуществить табулирование функций

$$F(x) = \sin^2 x + \sin 3x$$
$$G(x) = \cos^2 x + \cos 3x$$

на отрезке [a, b] с шагом h.

Решение

Табулирование функции – это вычисление ее значений в узловых точках. Для простоты берем равномерный шаг h.

Алгоритм вычисления заключается в циклическом переборе всех узловых точек, начиная от $x = a$ и до тех пор, пока очередная точка $x \leq b$. На каждом шаге вычисляем значение функции, выводим на экран аршумент и значение функции, наращиваем x.

Глава 2\Tabulation\Program.cs

```
using System;

namespace Tabulation
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Введите концы отрезка [a,b]
                и шаг h");
            Console.Write("a = ");
            double a = GetNumberInput();
            Console.Write("b = ");
            double b = GetNumberInput();
            Console.Write("h = ");
            double h = GetNumberInput();

            const double error = 1e-14;

            double x = a;
```

```
            while (x <= b + error)
            {
                double f = F(x);
                double g = G(x);

                Console.WriteLine(
                    $"{x,8:F3}{f,12:F5}{g,12:F5}"
                );

                x += h;
            }
        }

        static double GetNumberInput()
        {
            string? text_input = Console.ReadLine();

            return double.TryParse(
                text_input, out double number) ? number : 0;
        }

        static double F(double x)
        {
            return Math.Sin(x) * Math.Sin(x) +
                Math.Sin(3 * x);
        }

        static double G(double x)
        {
            return Math.Cos(x) * Math.Cos(x) +
               Math.Cos(3 * x);
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите концы отрезка [a,b] и шаг h
a = 3
b = 6
h = 0,2
   3,000     0,43203     0,06895
   3,200    -0,17092     0,01190
   3,400    -0,63457     0,22043
   3,600    -0,78511     0,60985
   3,800    -0,54496     1,01912
   4,000     0,03618     1,27110
   4,200     0,79327     1,23979
   4,400     1,49762     0,90034
   4,600     1,93112     0,34339
   4,800     1,95800    -0,25216
   5,000     1,56982    -0,67922
   5,200     0,88825    -0,77467
   5,400     0,12474    -0,47854
   5,600    -0,48907     0,14082
   5,800    -0,77680     0,90509
   6,000    -0,67291     1,58224
```

Рис. 2.82. Таблица значений функций $F(x)$ и $G(x)$

Комментарии к программе

Здесь мы впервые используем описание функций, чтобы упростить код (подробнее о функциях будет рассказано далее).

- Функция `GetNumberInput()` запрашивает ввод текста и возвращает введенное вещественное число, иначе нуль. Вместо условия `if` используем тернарный оператор.
- Функции `F()` и `G()` возвращают значение функции в заданной точке x (передается в качестве аргумента).

Кроме того, в условии цикла используется добавочная величина `epsilon`:

```
while (x <= b + error)
```

Ранее было отмечено, что работа с числами с плавающей точкой приводит к ошибкам округления. Поэтому даже если шаг h будет выбран так, что последняя точка в разбиении $x = b$, в силу ошибки может получиться $x > b$. Добавочная малая величина `epsilon`, заведомо меньшая h, решает эту проблему.

Задача 2

Для заданного натурального $n \geq 2$ осуществить каноническое разложение на простые множители, получив степень каждого.

Например:

$$12566840000 = 2^6 \cdot 5^4 \cdot 11^1 \cdot 13^4.$$

Решение

Согласно основной теореме арифметики, любое натуральное число ≥ 2 можно разложить на произведение простых чисел.

Для поиска делителей воспользуемся простым алгоритмом перебора возможных делителей (рис. 2.83).

Будем делить число n на предполагаемые делители до тех пор, пока оно не станет равным 1. В качестве первого делителя возьмем минимальное простое число – 2. Перебор делителей организуем с помощью цикла.

Поскольку, вообще говоря, делитель может входит в число в некоторой степени (рис. 2.83), то имеет смысл делить число n на текущий делитель до тех пор, пока это возможно. Деление также организуем в цикле (внутри основного, который перебирает возможные делители). В этом же цикле переменная счетчик будет подсчитывать степень делителя.

Наконец, если счетчик степени ненулевой (т.е. предполагаемый делитель является таковым), то выводим его вместе со степенью.

Далее переходим к проверке следующего кандидата в делители числа.

Заметим, что любые составные числа уже не будут являться делителями, потому что к моменту их проверки искомое число уже сокращается на их простые множители.

Так в примере к моменту проверки делимости на 4 искомое число уже сократилось на двойки (4 = 2 * 2), поэтому оно не делится на 4, осуществляется переход к следующему возможному делителю 5.

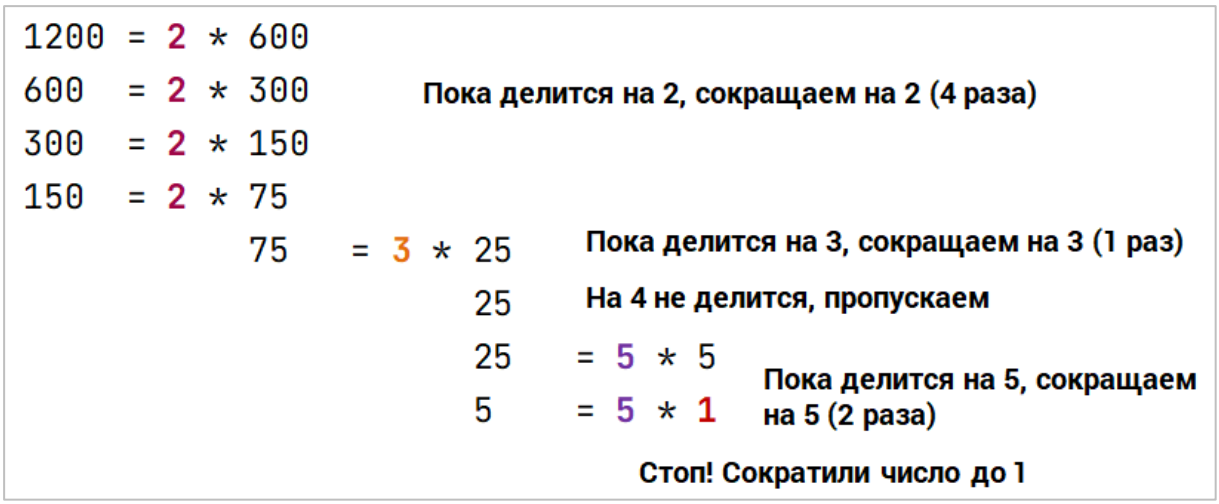


Рис. 2.83. Визуальная схема работы алгоритма на примере разложения числа 1200

Примечательно, что несмотря на простой циклический перебор предполагаемых делителей, предложенный алгоритм является достаточно эффективным. Это связано с тем, что в процессе число делится на делители, тем самым уменьшается количество возможных шагов.

Наиболее «плохой» случай – это когда n само является простым. В этом случае цикл повторится максимальное число раз (т.к. единственный делитель здесь – это само число). Однако с ростом n уменьшается и плотность расположения простых чисел.

Глава 2\CanonicalDecomposition\Program.cs

```
using System;

namespace CanonicalDecomposition
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Введите натуральное n: ");
            string? text_input = Console.ReadLine();

            if (!long.TryParse(text_input, out long n) ||
                n < 2)
            {
                n = 2;
            }
```

```
            long divider = 2;

            while (n > 1)
            {
                int degree = 0;

                while (n % divider == 0)
                {
                    n /= divider;
                    degree++;
                }

                if (degree > 0)
                    Console.WriteLine($"{divider}^{degree}");

                divider++;
            }
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите натуральное n: 12566840000
2^6
5^4
11^1
13^4
```

Рис. 2.84. Результат канонического разложения натурального числа

Комментарии к программе

В начале запрашивается число для разложения n. Если будет введено нечисловое значение, то число берется равным 2.

Начальный возможный простой делитель `divider` берем равным 2. Внешний цикл следит, не сокращено ли число до 1.

Далее на каждом шаге устанавливаем счетчик степени простого делителя `counter` (изначально считаем, что пока не является таковым). Внутренний цикл сокращает число до тех пор, пока оно делится на предполагаемый делитель `divider`, наращиваем счетчик степени делителя. Если получили ненулевую степень, значит `divider` является делителем, выводим его вместе со степенью и в любом случае переходим к следующему возможному делителю.

Вопросы для самопроверки

1. Перечислите и опишите виды циклов, которые реализованы в языке C#.
2. Опишите синтаксис и приведите примеры задач, в которых можно использовать цикл `while`.
3. Опишите синтаксис и приведите примеры задач, в которых можно использовать цикл `do-while`.
4. В каких случаях рационально использовать цикл с предусловием, а в каких – с постусловием?
5. Почему реализовывать циклы с помощью оператора `goto` не рекомендуется?

Практикум

Задание 1

1. Создайте проект *MaximumValue*.
2. Скопируйте и изучите код, приведенный ниже.
3. На отрезке $[a, b]$ с шагом h осуществите табулирование функций
$$f(x) = \ln\left(\frac{\sin x^2 + 2}{x^2 + 2}\right) + 3$$
и среди узловых точек найдите x_{max}, в которой функция достигает наибольшего значения на этом отрезке.
4. Также вычислите y_{max}.
5. Для вычисления функции $f(x)$ используйте описанную в файле функцию `F()`.
6. Результат оформит, как изображено на рис. 2.85.

Глава 2\MaximumValue\Program.cs

```
using System;

namespace MaximumValue
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            Console.WriteLine("Введите концы отрезка [a,b]
                и шаг h");
            Console.Write("a = ");
            double a = GetNumberInput();
            Console.Write("b = ");
            double b = GetNumberInput();
            Console.Write("h = ");
            double h = GetNumberInput();


            const double epsilon = 1e-14;

            double x = a;
            double xmax = x;

            // Здесь реализовать поиск xmax и ymax


            // Вывод xmax и ymax

        }

        static double GetNumberInput()
        {
            string? text_input = Console.ReadLine();

            return double.TryParse(text_input,
                out double number) ? number : 0;
        }

        static double F(double x)
        {
            return Math.Log((Math.Sin(x*x) + 2) / (x*x + 2))
                + 3;
        }
    }
}
```

Задание 2

1. Создайте проект *CanonicalDecompositionModification*.
2. Используйте код из примера *CanonicalDecomposition*.
3. В каноническом разложении числа дополнительно выводить парный множитель, как изображено на рис. 2.86.

Задание 3

1. Создайте проект *InputCorrection*.
2. Реализуйте с помощью `do-while` следующий алгоритм. С клавиатуры запрашивается ввод неотрицательного коэффициента. Если значение некорректно, то выдается предупреждение и запрашивается повторный ввод (рис. 2.87).

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите концы отрезка [a,b] и шаг h
a = -3
b = 1
h = 0,5
  -3,000     1,48261
  -2,500     1,56621
  -2,000     1,42593
  -1,500     2,57484
  -1,000     2,94571
  -0,500     2,99885
   0,000     3,00000
   0,500     2,99885
   1,000     2,94571
Xmax = 0,00000
Ymax = 3,00000
```

Рис. 2.85. Результат работы программы из задания 1

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите натуральное n: 457100
114275 * 2^2
18284 * 5^2
65300 * 7^1
700 * 653^1
```

Рис. 2.86. Результат работы программы из задания 2

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите неотрицательное k: ghj
Ввод некорректен. Повторите еще раз
Введите неотрицательное k: -6
Ввод некорректен. Повторите еще раз
Введите неотрицательное k: 12

Введен коэффициент: 12
```

Рис. 2.87. Результат работы программы из задания 3

2.8. Циклические конструкции for и foreach

2.8.1. Цикл for

Синтаксис цикла

Определение

Оператор **`for`** *реализует цикл с предусловием.*
Условие является выражением логического типа.
Если в теле цикла один оператор, фигурные скобки можно опустить.

Цикл `for` обычно используется для организации циклической обработки, когда количество итераций заранее известно. Он предполагает инициализацию переменно счетчика, проверку условия и обновление счетчика после каждого витка цикла.

Синтаксис цикла:

```
for (нач_знач; условие; итерация)
{
    // Тело цикла
}
```

Цикл `for` в заголовке реализует три части:

- **`нач_знач`** – начальное значение счетчика;
- **`условие`** – логическое выражение, указывающее условие работы цикла (не обязательно зависит от счетчика);
- **`итерация`** – увеличение / уменьшение счетчика.

Однако цикл `for` необязательно должен быть ограничен исключительно переменной счетчиком. В условии могут участвовать и другие переменные, значение которых устанавливается до входа в цикл или в его теле.

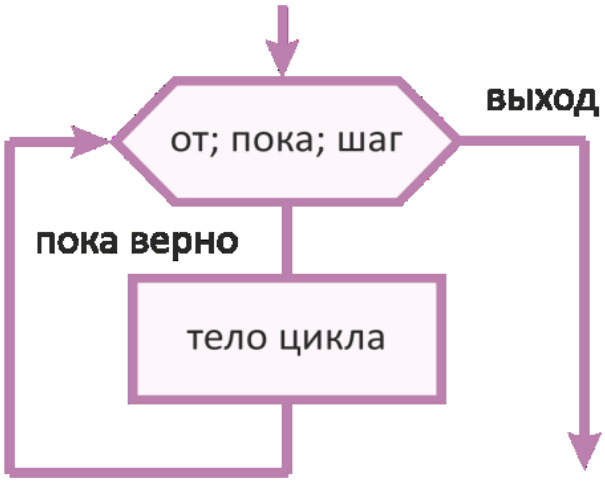


Рис. 2.88. Блок схема цикла `for`

В следующем примере осуществляется вывод чисел от 0 до 1000. После каждого витка выполняется наращивание счетчика `i++`. Также можно использовать префиксный инкремент `++i`, здесь приоритет инкремента не принципиален.

```
int i;

for (i = 0; i <= 1000; i++)
{
    Console.WriteLine(i);
}
```

В теле цикла одна команда, скобки блока можно опустить:

```
for (i = 0; i <= 1000; i++)
    Console.WriteLine(i);
```

```
PS D:\Documents\Programming\NET\Program> dotnet run
0
1
2
3
4
5
6
7
8
9
10
```

Рис. 2.89. Вывод чисел с помощью цикла `for`

Шаг цикла может быть произвольным. В следующем примере вычисляется сумма чисел, кратных 5, в пределах от 0 до 1000. Здесь счетчик `i` увеличивается на 5 после очередного витка цикла:

```
int i;
int total = 0;

for (i = 0; i <= 1000; i += 5)
{
    total += i;
}

Console.Write("Сумма равна " + total);
```

```
PS D:\Documents\Programming\NET\Program> dotnet run
Сумма равна 100500
```

Рис. 2.90. Подсчет суммы

Вариации for

1. Цикл без изменения шага

Если убрать операцию изменения счетчика, то `for` не будет автоматически менять его при каждом новом витке. В этом случае нужно будет делать это вручную, в теле цикла.

```
int i;
int total = 0;

for (i = 0; i <= 1000; )
{
    total += i;
    i += 5;
}
```

2. Цикл без инициализации

Если убрать инициализацию, то начальное значение переменной счетчика нужно устанавливать до входа в цикл.

```
int i = 0;
int total = 0;

for ( ; i <= 1000; i += 5)
{
    total += i;
}
```

3. Цикл без условия

Можно убрать условие из заголовка: в этом случае выход из цикла должен контролироваться в его теле проверкой условия и принудительным выходом из цикла.

```
int i;
int total = 0;

for (i = 0; ; i += 5)
{
    if (i > 1000)
        break;

    total += i;
}
```

4. Цикл без инициализации и изменения шага

Если убрать инициализацию и изменение счетчика, то получаем аналог цикла `while`: условие зависит от переменной, которая задается до входа в цикл, а шаг меняется в теле цикла.

```
int i = 0;
int total = 0;

for ( ; i <= 1000; )
{
    total += i;
    i += 5;
}
```

Аналогичная реализация (и более очевидная) в цикле `while`:

```
int i = 0;
int total = 0;

while (i <= 1000)
{
    total += i;
    i += 5;
}
```

5. Бесконечный цикл

Если убрать все выражения из заголовка, цикл вырождается в бесконечный. Для выхода из него в теле цикла должна быть организована проверка некоторого условия, при котором осуществляется принудительное прерывание цикла.

```
for ( ; ; )
{

}
```

Аналогичную циклическую конструкцию организует следующий цикл `while`:

```
while (true)
{

}
```

Подобные циклы могут использоваться в системах реального времени, опрашивая состояние системы или оборудования.

6. Несколько счетчиков или переменных в условии

В описание цикла можно вносить и другие переменные, которые могут участвовать в инициализации, условии и изменении счетчика.

```
int i;
int j;

for (i = 0, j = 50; (i < 100) && (j > 20); i++, j -= 3)
{
    Console.WriteLine(i + "\t" + j);
}
```

```
PS D:\Documents\Programming\NET\Program> dotnet run
0       50
1       47
2       44
3       41
4       38
5       35
6       32
7       29
8       26
9       23
```

Рис. 2.91. Цикл `for` может контролировать работу с несколькими переменными, выполняющими роль счетчика или участвующими в условии

Локальная область видимости

В C# описывать переменные можно на разных уровнях класса и они приобретают свою область видимости. Кроме того, блоки тела для условных конструкций и циклов также замыкают область видимости описанных в них переменных.

На уровне методов выделяют **локальную** и **глобальную** область видимости.

1. ***Глобальная видимость***

В следующем примере переменная `global` доступна как на уровне метода (где описана), так и для внутренних блоков инструкций:

```
int global = 1;

if (true)
{
    Console.WriteLine(global);
}

Console.WriteLine(global);
```

2. ***Локальная видимость***

В этом примере переменная `local` уже доступна только внутри блока `if`. Внешнее обращение к ней невозможно и вызовет ошибку компиляции:

```
if (true)
{
    int local = 2;
    Console.WriteLine(local);
}

Console.WriteLine(local);
```

Локальное описание счетчика цикла

Наилучшей практикой при работе с циклом `for` является описывать переменную счетчик в заголовке цикла локально. Это позволяет закрыть область видимости переменной от внешних операций и позволяет описывать локальную переменную с таким же именем в других циклах.

```
int total = 0;
```

```
for (int i = 0; i <= 1000; i += 5)
{
    total += i;
}
```

Прерывание цикла

В C# реализованы две команды, которые позволяют прервать цикл или текущую итерацию.

1. Оператор break

Команда **`break`** прерывает выполнение цикла и переходит к следующей операции после цикла.

В следующем фрагменте при достижении некоторого условия осуществляется прерывание выполнения цикла переход к следующим после него операциям:

```
for (int i = 0; i < 10; i++)
{
    // Некоторые операции
    if (условие)
        break;
    // Иначе некоторые операции
}
```

В случае вложенных циклов `for` оператор `break` прерывает внутренний цикл и переходит в область тела внешнего цикла.

2. Оператор continue

Оператор **`continue`** прерывает выполнение текущей итерации и переходит к следующей.

В следующем фрагменте при достижении некоторого условия осуществляется прерывание итерации и переход к следующему витку:

```
for (int i = 0; i < 10; i++)
{
    // Некоторые операции
    if (условие)
        continue;
    // Иначе некоторые операции
}
```

Замечания по break и continue

На практике оператор `break` применяется чаще, чем `continue`.

С одной стороны это связано с тем, что прерывать выполнение цикла `for` приходится чаще, поскольку достигается нужное условие и больше нет необходимости циклической обработки. Например – в задачах поиска элемента, удовлетворяющего критериям.

С другой стороны оператор `continue` не столь востребован, поскольку вместо него можно организовать условную проверку вида:

```
for (int i = 0; i < 10; i++)
{
    // Некоторые операции

    if (условие)
    {
        // Некоторые операции, иначе переход к
        // следующей итерации
    }
}
```

Однако `continue` зачастую позволяет уменьшить число вложенных проверок и кода внутри цикла, что упрощает его чтение.

Это полезно знать!

*Если в цикле многократно требуется обращаться к операторам **`break`** и **`continue`**, то следует переработать алгоритм: возможно вместо цикла **`for`** будет более рациональным заменить его циклом **`while`** или **`do-while`**.*

2.8.2. Цикл foreach

Синтаксис цикла

Определение

*Конструкция цикла **`foreach`** организует перебор элементов из **коллекции**. Простым примером коллекции являются массивы.*

Цикл `foreach` способен автоматически перебирать элементы коллекций, массивов, списков, словарей и других итерируемых структур данных, как элементы некоторого множества. Важное его достоинство состоит в том, что для обращения к каждому последующему элементу не требуется самостоятельно управлять индексами или счетчиками цикла.

Синтаксис цикла:

```
foreach (var элемент in коллекция)
{
    // Тело цикла
}
```

Вместо `var` можно указать конкретный тип элемента, совпадающий с типом перебираемых элементов коллекции.

Важное замечание!

*Цикл **`foreach`** способен только считывать значения из коллекции, но не имеет прав его менять.*

Простые примеры использования

Цикл `foreach` удобно использовать, когда требуется обойти все элементы заданной коллекции и выполнить с каждым определённые действия.

Например, в следующем фрагменте осуществляется перебор и вывод в консоль элементов массива:

```
double[] array = { 1, 5.3, 2, 0, 4.8 };

foreach (var number in array)
{
    Console.Write($"{number}  ");
}
```

```
PS D:\Documents\Programming\NET\Program> dotnet run
1  5,3  2  0  4,8
```

Рис. 2.92. Вывод элементов массива с помощью цикла `foreach`

В отличие от цикла `for` здесь не требуется обращаться к элементам массива по индексу. А вместо `var` можно указать `double` (поскольку массив состоит из чисел типа `double`).

В следующем примере выводим построчно символы строки, кроме пробелов (строки тоже можно считать коллекциями):

```
string? text = "Циклические операторы языка C#";

foreach (var simbol in text)
{
    if (simbol != ' ')
        Console.WriteLine(simbol);
}
```

Реализация цикла for в качестве цикла foreach

При работе с индексируемыми коллекциями (например – массивами) цикл `for` может быть упрощен и реализован как аналог цикла `foreach`. А именно, если в цикле необходимо работать с самими элементами коллекции, а явное обращение к их индексам не требуется, то на каждой итерации в начале рационально записать элемент в локальную переменную с говорящим именем:

```
for (int i = 0; i < array.Length; i++)
{
    double number = array[i];

    // Далее операции с i-м элементом через number
}
```

Приведенный пример абсолютно аналогичен реализации с циклом `foreach`. Однако внутри тела цикла нет необходимость обращаться к элементу по индексу, а можно работать с более понятной мнемонической переменной `number`.

Это полезно знать!

Однако не стоит упускать из виду, что изменение ***number*** *не меняет соответствующий элемент коллекции. Т.е. такой прием работает только в случаях, когда перебор элементов осуществляется только для чтения.*

2.8.3. Решение практических задач

Задача 1

С клавиатуры запрашивается натуральное число n.

Для целых чисел от 0 до n (где $n \geq 1000$) найти сумму квадратных корней только тех чисел, которые кратны 2, 3 и 5 (одновременно). Реализуйте вычисления с помощью цикла `for`.

Решение

Проверку кратности одновременно на 2, 3 и 5 делать не требуется, поскольку согласно правилам алгебры минимальным кратным будет НОК(2,3,5), т.е. число 30.

Более того, можно начать с нуля и наращивать шаг на 30, тем самым убирая проверку на кратность и исключая лишние числа.

Глава 2\SumOfSquareRoots\Program.cs

```
using System;
namespace SumOfSquareRoots
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Введите неотрицательное n: ");
            string? text_input = Console.ReadLine();

            if (!int.TryParse(text_input, out int n) || n < 0)
            {
                Console.WriteLine("Некорректное значение.
                    Автоматически исправлено на 1000");
                n = 1000;
            }

            double total = 0;

            for (int i = 0; i <= n; i += 30)
                total += Math.Sqrt(i);

            Console.WriteLine($"sum = {total:F5}");
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите неотрицательное n: 2000
sum = 1979,01445
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите неотрицательное n: -3400
Некорректное значение. Автоматически исправлено на 1000
sum = 706,84706
```

Рис. 2.93. Результат работы программы

Задача 2

С клавиатуры запрашивается натуральное число $\boldsymbol{n} \geq 10$ и вещественное $\boldsymbol{x}$.

Вычислите приближенное значение e^x в заданной точке x по разложению функции в ряд Тейлора:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} + \cdots.$$

Сравните результат с работой встроенного метода `Exp()` из класса `Math`.

Решение

Ряд Тейлора – это представление бесконечно дифференцируемой функции в виде бесконечной суммы слагаемых, каждое из которых зависит от производных функции в определенной точке. Разложение в ряд Тейлора позволяет вычислять приближенные значения трансцендентных функций с наперед заданной точностью, т.е. функций, которые нельзя представить через комбинации арифметических операций.

Например, разложение в ряд Тейлора функции e^x представлено следующей формулой:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} + \cdots$$

При этом чем больше слагаемых взять из разложения, тем более точный результат.

Это разложение позволяет вычислить приближенно такие выражения, как например $e^{\sqrt{2}}$: возведение в иррациональную степень распадается на обычные арифметические операции.

Обычно, чтобы избежать быстрого роста чисел и как следствие – ошибок округления, каждый k-й член ряда считают не напрямую, а через значение предыдущего $(k-1)$-го члена.

Его нетрудно найти:

$$\frac{a_{k+1}}{a_k} = \frac{x}{k+1} \rightarrow a_{k+1} = a_k \cdot \frac{x}{k+1}$$

где начальный член ряда значение $a_0 = 1$.

Глава 2\TeylorExp\Program.cs

```
using System;

namespace TeylorExp
{
    class Program
    {
        static void Main(string[] args)
        {
            string? text_input;

            Console.Write("Введите натуральное число
                n >= 1: ");
            text_input = Console.ReadLine();

            if (!int.TryParse(text_input, out int n) || n < 1)
            {
                Console.WriteLine("Некорректное значение.
                    Исправлено на 1");
                n = 1;
            }

            Console.Write("Введите x: ");
            text_input = Console.ReadLine();

            if (!double.TryParse(text_input, out double x))
            {
                Console.WriteLine("Некорректное значение.
                    Исправлено на 0");
                x = 0;
            }

            double exp_sum = 0;
            double a_n = 1;

            for (int i = 0; i <= n; i++)
            {
                exp_sum += a_n;
                a_n *= x / (i + 1);
```

```
            }

            Console.WriteLine("------------------");
            Console.WriteLine($"Наш алгоритм = {exp_sum:F5}");
            Console.WriteLine($"Встроенный метод =
                {Math.Exp(x):F5}");
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите натуральное число n >= 1: 40
Введите x: 1,504
------------------
Наш алгоритм = 4,49965
Встроенный метод = 4,49965
```

Рис. 2.94. Результат работы программы

Задача 3

Реализовать таблицу умножения n на m.

Решение

Реализация программы потребует вложения циклов:

- внешний цикл отвечает за номер строки;
- внутренний цикл перебирает колонки в строке.

Для упрощения ввода коэффициентов создадим единую функцию `GetInputNuber()`.

Глава 2\MulTable\Program.cs

```
using System;

namespace MulTable
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Введите размеры таблицы:");

            Console.Write("n = ");
            int n = GetInputNuber();

            Console.Write("m = ");
```

```
            int m = GetInputNuber();

            for (int i = 1; i <= n; i++)
            {
                for (int j = 1; j <= m; j++)
                    Console.Write($"{i*j,6}");

                Console.WriteLine();
            }
        }

        static int GetInputNuber()
        {
            string? console_input = Console.ReadLine();

            return int.TryParse(console_input,
                out int number) ? number : 1;
        }
    }
}
```

2.8.4. Рекомендации учителю

- Обучение циклическим конструкциям начинайте с обсуждения практических примеров и ситуаций, где требуется повтор операций.
- Обозначьте, какие операторы циклов реализованы в C# (без описания синтаксиса), устно проработайте примеры, в каких ситуациях более рационально использовать тот или иной тип цикла.
- Изучение синтаксиса циклов начните с инструкций `while` и `do-while`. Далее переходите к разбору цикла `for`. Покажите на нескольких примерах, что возможен переход от одного цикла к другому без нарушения результата работы программы.
- Разберите алгоритмы и задачи, требующие вложения циклов. Необходимо затронуть как вложение одинаков типов циклов, так и смешанных.
- Проработайте алгоритмы, в которых предусмотрено прерывание цикла. Обратите внимание на то, в каких ситуациях

операция прерывания оптимизирует выполнение цикла, а в каких в ней нет необходимости (а цикл рационально заменить другим типом).

- Цикл для работы с коллекцией `foreach` рационально вынести на изучение темы, связанной с массивами.

Вопросы для самопроверки

1. Опишите синтаксис и возможности цикла `for`.
2. Перечислите возможные вариации цикла for на примерах.
3. В чем преимущества локального описания переменных в циклах?
4. Каким образом можно прервать выполнение цикла или его очередного витка?
5. Какими преимуществами и недостатками обладает цикл `foreach` и в каких ситуациях его рационально использовать?

Практикум

Задание 1

1. Создайте проект *SumOfProgression*.
2. С клавиатуры запрашивается натуральное число $\boldsymbol{n} \geq 10$. Найдите частичную сумму:
$$S(n) = 1 + \frac{1}{4} + \frac{1}{9} + \cdots + \frac{1}{n^2}.$$
3. Сравните значение с пределом суммы ряда:
$$\lim_{n\to\infty} S(n) = \frac{\pi^2}{6}.$$
4. Экспериментально подберите n, при котором относительная погрешность частичной суммы не превышает 1%.

```
PS D:\Documents\Programming\NET\2024\Program> dotnet run
Введите n: ?
Частичная сумма = 1,63075
Сумма ряда = 1,64493
Относительная погрешность = 0,87%
```

Рис. 2.95. Результат работы программы из задания 1

Задание 2

1. Создайте проект *TeylorCos*.
2. С клавиатуры запрашивается натуральное число $\boldsymbol{n} \geq 10$ и вещественное $\boldsymbol{x}$.
3. Вычислите приближенное значение $\cos x$ в заданной точке x по разложению функции в ряд Тейлора:
$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots + (-1)^n \frac{x^{2n}}{(2n)!} + \cdots.$$
4. Сравните результат с работой встроенного метода `Cos()` из класса `Math`.

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите натуральное число n >= 1: 15
Введите x: 6,28
Наш алгоритм = 0,99999
Встроенный метод = 0,99999
```

Рис. 2.96. Результат работы программы из задания 2

Задание 3

1. Создайте проект *Perfect_Number*.
2. Натуральное число назовем «совершенным», если сумма его всевозможных делителей (кроме самого числа) равна этому числу. Например, $28 = 1 + 2 + 4 + 7 + 14$ является совершенным.
3. Из диапазона от 1 до $\boldsymbol{n}$ вывести все совершенные числа.

Задание 4

1. Создайте проект *PrimeNumber*.
2. Для заданного числа $\boldsymbol{n} \geq 2$ определить, является ли оно простым.
3. Используйте следующий код:

Глава 2\PrimeNumber\Program.cs

```
using System;

namespace PerfectNumber
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            Console.Write("Введение n >= 2: ");
            string? text_input = Console.ReadLine();

            if (!int.TryParse(text_input, out int n) || n < 2)
            {
                Console.WriteLine("Значение исправлено на 2");
                n = 2;
            }

            // Устанавливаем верхнюю границу перебора
            // возможных делителей
            bool is_prime = true;
            int limit = (int) Math.Truncate(Math.Sqrt(n));

            // Далее поиск и вывод

        }
    }
}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введение n > 2: 113
Число простое
Поиск завершен.
```

Рис. 2.97. Результат работы программы из задания 4

Задание 5

1. Создайте проект *ForeachCycle*.
2. Для заданного массива с помощью цикла foreach найти и вывести числа, дробная часть которых меньше 0.5 по абсолютной величине. Используйте ограничение

$$|x - [x]| < 0.5,$$

где $[x]$ - целое от x.

```
double[] arr = { 45.7, 5.04, 9.51, -4.6, 7, 12.46 };
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
5,04    0,040000000000000036
7       0
12,46   0,46000000000000085
```

Рис. 2.98. Результат работы программы из задания 5

2.9. Массивы

2.9.1. Понятие массива

Массивы как наборы элементов одного типа

Определение

Массив *– структура данных в C#, представляющая собой совокупность переменных одного типа, элементы которой индексируются по порядку.*

Массивы являются удобной структурой для организации простых коллекций, к элементам которых можно обращаться по индексу. Массивы могут иметь разную размерность:

- *одномерные массивы* представляют собой последовательность проиндексированных элементов, аналогичных спискам, очередям, стекам и т.п. структурам;
- *двумерные массивы* можно рассматривать в качестве таблиц или матриц значений;
- *трехмерные массивы* подходят для задач моделирования 3D-объектов;
- *многомерные массивы* могут быть востребованы в статистических задачах и измерительном оборудовании, которое фиксирует данные по нескольким критериям.

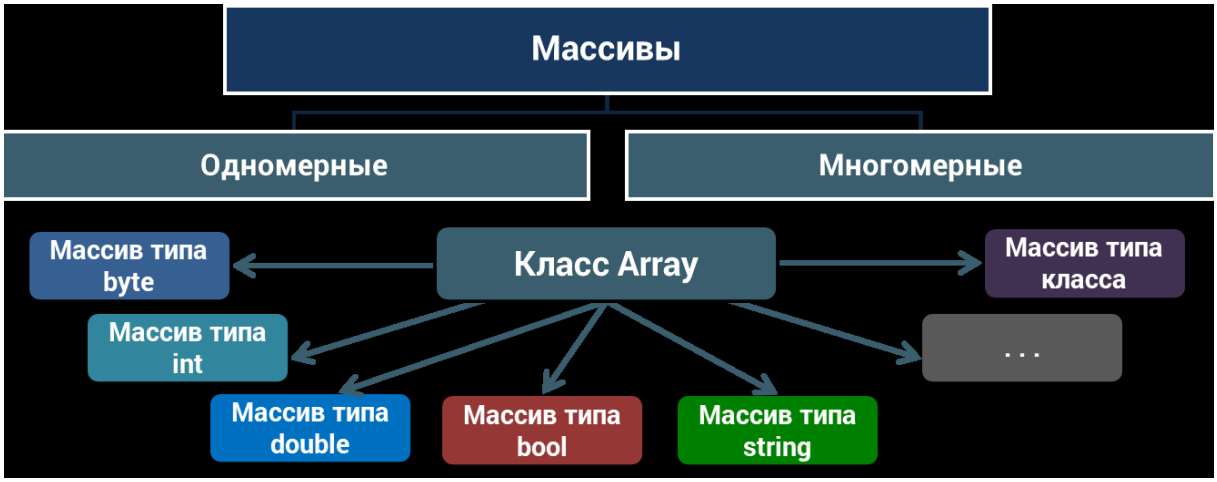


Рис. 2.99. Массивы в .NET

Массивы в .NET являются объектами, которые производны от базового класса **`Array`** из пространства имен `System`.

Описание массива

Определение

Одномерный массив *представляет собой список проиндексированных переменных. Общий синтаксис описания:*

```
тип[] имя_массива = new тип[размер];
```

где размер определяет число элементов массива.

Объявление массива отличается от описания обычной переменной тем, что:

- в конце типа без пробела ставятся пустые квадратные скобки;
- в правой части после оператора присваивания используется оператор `new`.

```
int[] array = new int[100];
```

Пустые скобки слева указывают, что переменная ссылается на массив целых чисел. Правая часть с помощью оператора **`new`** выделяет оперативную память под указанной число элементов массива (оператор `new` используется при создании объектов класса и вызывает **конструктор** класса). Пока он не вызван, работа с элементами массива невозможна.

Переменная `mass` является **ссылкой** на массив.

Описание ссылки может быть отделено от резервации памяти под массив:

```
int[] array;
array = new int[100];
```

После выполнения команды `new` все элементы инициализируются нулями. Предполагается, что далее массив будет заполнен конкретными значениями.

Индексация элементов массива

Это важно знать!

В C# массивы индексируются с нуля!

Для обращения к элементу массива в квадратных скобках указывается его индекс:

```
mass[50] = 2025;  // 51-й элемент, т.к. индексация с нуля
Console.WriteLine($"M[51] = {mass[50]}");
```

Обращение к элементам неинициализированного массива невозможно:

```
// Создана только ссылка на массив типа int
int[] array;

// Ошибка! Массив не инициализирован
Console.WriteLine(array[4]);
```

У каждого массива доступно свойство **Length**, которое хранит **длину массива** (количество элементов) и автоматически ее обновляет:

```
Console.WriteLine("Длина массива: " + mass.Length);
```

В версии C# 8.0 был добавлен оператор ^, который позволяет указывать индексы с конца массива (или коллекции в целом).

Так, в следующем примере для обращения к последнему элементу можно использовать один из двух вариантов:

```
int last_element_1 = mass[mass.Length - 1];
int last_element_2 = mass[^1];
```

Инициализация с помощью new

Необходимо учитывать, что оператор new изначально устанавливает элементам массива «нулевые» значения. При этом для значимых и ссылочных типов данных они разнятся.

Для значимых типов данных числовые массивы инициализируются нулями, а для логического типа – ложью.

```
// все элементы <- 0
int[] indexes = new int[40];

// все элементы <- 0
double[] array = new double[10];

// все элементы <- false
bool[] flags = new bool[8];
```

Элементы массивов ссылочного типа (в частности – классы) инициализируются ссылкой `null`:

```
// все элементы <- null
Student[] journal = new Student[30];
```

Непосредственная инициализация

Заданные автоматически после объявления массива «нулевые» значения элементов перезаписываются конкретными значениями, например – введенными с клавиатуры.

Однако массивы можно объявлять и сразу инициализировать. Для этого в правой части вместо вызова оператора `new` в фигурных скобках перечисляются элементы массива:

```
double[] table = { 2.3, 4.0, 20.13, 5, -2.1 };

string[] verse =
{
    "Люблю грозу в начале мая,",
    "Когда весенний, первый гром,",
    "Как бы резвяся и играя,",
    "Грохочет в небе голубом."
};
```

Достоинством этого подхода является то, что нет необходимости явно указывать длину массива.

Однако инициализировать массив можно и с явным указанием оператора `new`. Ниже все четыре описания приводят к аналогичному результату, однако рекомендуется использовать именно последний:

```
int[] mass = new int[5] { 1, 2, 3, 4, 5 };
int[] mass = new int[] { 1, 2, 3, 4, 5 };
int[] mass = new[] { 1, 2, 3, 4, 5 };
int[] mass = { 1, 2, 3, 4, 5 };
```

А начиная с версии C# 12.0 объявлять и инициализировать массивы можно с помощью **выражения коллекций**, где вместо фигурных скобок используются квадратные:

```
int[] mass = [ 1, 2, 3, 4, 5 ];
```

Кроме того, можно создавать и пустой массив:

```
int[] mass = {};
int[] mass = [];
```

Переменная как ссылка

Массивы относятся к ссылочному типу данных, поэтому переменная типа массив содержит ссылку (адрес) на данные массива в оперативной памяти.

В следующем примере массив `array2` создается как копия `array1`. Однако изменение элементов в первом автоматически меняет элементы второго и наоборот.

```
double[] array1 = { 56.3, 20.0, 33.2 };
double[] array2 = array1;

array1[0] = 2025;

Console.WriteLine(array1[0]);
Console.WriteLine(array2[0]);

array2[^1] = -2025;

Console.WriteLine(array1[^1]);
Console.WriteLine(array2[^1]);
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
2025
2025
-2025
-2025
```

Рис. 2.100. Особенности копирования ссылок

Это связано с тем, что на самом деле скопирован не массив, а ссылка на массив: теперь переменные `array1` и `array2` ссылаются на один и тот же массив в памяти.

Подобное копирование называет **поверхностным**, в противовес **глубокому**, когда ссылочные объекты создаются как невзаимосвязанные сущности.

Проверка на null и выход за границы индексирования

Работая с массивами, важно следить за следующими деталями:

- массив не должен быть `null`;
- индекс не должен выходить за допустимый диапазон.

Если существует риск, что в ходе операций массив может ссылаться на значение `null`, то важно установить проверку, например:

```
int[]? arr = null;

if (arr == null)
{
    // Массив неинициализирован
    // Можно задать некоторый размер по умолчанию.
    arr = new int[1024];
}
```

Так, если по каким-либо причинам массив будет ссылаться на `null`, то для него зарезервируется указанный объем ячеек, заполненных нулями. Код можно сократить, используя оператор нулевого слияния:

```
arr ??= new int[1024];
```

Что касается индекса при обращении к элементу массива, то его необходимо контролировать в пределах от `0` до `arr.Length - 1` включительно. Если указан индекс вне этого диапазона, то во время работы программы возникнет исключение `IndexOutOfRangeException`.

Пример обработки:

```
if (0 <= index && index <= arr.Length - 1)
{
    // Обращение к существующему элементу
    Console.WriteLine(arr[index]);
}
else
{
    // Обращение к несуществующему элементу
    Console.WriteLine("Индекс вне диапазона!");
}
```

2.9.2. Одномерные массивы

Обход элементов массива в цикле

Обычно массивы обрабатываются в циклах. В C# для этой цели можно использовать все четыре вида циклических операторов, но более удобными являются циклы `for` и `foreach`.

1. Обход массива циклом for

Цикл `for` используется, когда в теле цикла возникает необходимость явных операций с индексом элемента.

```
double[] array = { 3.56, 4.23, 4.98, 4.62, 5.00 };

for (int i = 0; i < array.Length; i++)
{
    Console.Write($"{array[i]:f2}  ");
}
```

Здесь используется знак строго меньше, т.к. индекс последнего элемента `array.Length - 1`. Также можно было использовать и ограничение `i <= array.Length - 1`.

2. Обход массива циклом foreach

Цикл `foreach` дает более короткую форму обращение, перебирая элементы массива как элементы из множества (коллекции).

```
double[] array = { 3.56, 4.23, 4.98, 4.62, 5.00 };

foreach (var number in array)
{
    Console.Write($"{number:f2}  ");
}
```

Здесь уже не требуется обращение к индексу элемента. Кроме того – избегаем явного контроля длины массива.

```
PS D:\Documents\Programming\NET\Program> dotnet run
3,56  4,23  4,98  4,62  5,00
```

Рис. 2.101. Вывод элементов массива (работает одинаково для приведенных выше фрагментов кода)

Это полезно знать!

Цикл ***for*** *используется, когда в теле цикла требуется работа с индексом элемента массива. Также в теле цикла можно обращаться к разным элементам массива, указывая их индексы.*
Цикл ***foreach*** *дает более короткий синтаксис перебора элементов массива. Однако он не позволяет менять элементы массива, а только их получать (читать).*

Стилизация цикла for под foreach

Из цикла `for` можно реализовать конструкцию, аналогичную циклу `foreach`. Для этого в начале каждой итерации текущий элемент записывается в локальную переменную:

```
double[] array = { 3.56, 4.23, 4.98, 4.62, 5.00 };

for (int i = 0; i < array.Length; i++)
{
    double number = array[i];

    // Далее работаем с number как с текущим
    // элементом массива
    // При этом также сохраняем возможность работать
    // с индексом элемента
}
```

Достоинство этого подхода:

- для считывания элемента используется более понятное название без индекса;
- можно обращаться к индексу элемента, если необходимо.

Пример обработки одномерного массива

Требуется описать массив вещественных чисел. Число элементов и сами элементы запрашиваются с клавиатуры.

Далее подсчитать количество четных положительных и нечетных отрицательных элементов. Вывести индекс первого равного нулю. Также распечатать сам массив.

Решение

Глава 2\SearchInArray_1\Program.cs

```
using System;

namespace SearchInArray
{
    class Program
    {
        static void Main(string[] args)
        {
            int n;

            Console.Write("Введите размер массива: ");
            while (!int.TryParse(Console.ReadLine(), out n) ||
                n <= 0)
            {
                Console.Write("Некорректное значение.
                    Введите повторно: ");
            }

            double[] array = new double[n];

            for (int i = 0; i < n; i++)
            {
                Console.Write($"A({i}) = ");
                array[i] = double.TryParse(
                    Console.ReadLine(), out double number)
                    ? number : 0;
            }

            Console.Clear();

            int even_positive = 0;
            int odd_negative = 0;

            for (int i = 0; i < n; i++)
            {
                if (array[i] > 0 && array[i] % 2 == 0)
                    even_positive++;

                if (array[i] < 0 && array[i] % 2 != 0)
                    odd_negative++;
            }
```

```
            int first_zero_index = -1;

            for (int i = 0; i < n; i++)
            {
                if (array[i] == 0)
                {
                    first_zero_index = i;
                    break;
                }
            }

            for (int i = 0; i < n; i++)
            {
                Console.Write(array[i] + "  ");
            }

            Console.WriteLine();

            Console.WriteLine("Количество четных положительных
                чисел: " + even_positive);
            Console.WriteLine("Количество нечетных
                положительных чисел: " + odd_negative);

            if (first_zero_index != -1)
                Console.WriteLine("Индекс первого нуля: " +
                    first_zero_index);
            else
                Console.WriteLine("Нулей в массиве нет");
        }
    }
}
```

Комментарии к реализации

В начале создается массив `array` указанного пользователем размера. Далее в цикле последовательно запрашиваем ввод элементов, записываем их в массив. Если введенное значение недопустимо, то будет скорректировано на нуль. По завершению ввода очистим окно консоли от лишней информации.

Для подсчета четных положительных и нечетных отрицательных используем простой алгоритм циклического перебора и проверки всех чисел в массиве.

В переменные `even_positive` и `odd_negative` будем накапливать количество соответствующих чисел (изначально предполагаем, что таковых нет).

В теле цикла отдельно проверяем текущий элемент на соответствие первой или второй группе чисел. Если число удовлетворяет требованиям, наращиваем счетчик.

Далее ищем индекс первого нуля. Изначально допускаем, что нулей в массиве нет, поэтому установим условно значение -1.

Аналогично задействуем цикл перебора всех элементов. В теле цикла проверяем, является ли текущий элемент нулем. Если да, то в переменную `first_zero_index` записываем индекс нуля и прерываем выполнение цикла, поскольку требуется лишь первое вхождение.

В случае отсутствия нулей в массиве переменная сохранит значение -1.

Важно заметить, что поиск индекса нерационально включать в предыдущий цикл, поскольку здесь возможен досрочный выход из цикла.

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите размер массива: 8
A(0) = -5
A(1) = 8
A(2) = -12
A(3) = 0
A(4) = 6
A(5) = -1
A(6) = 0
A(7) = 7
```

```
-5  8  -12  0  6  -1  0  7
Количество четных положительных чисел: 2
Количество нечетных положительных чисел: 2
Индекс первого нуля: 3
```

Рис. 2.102. Результат вычислений

Оптимизация

Некоторые из приведенных в решении циклов `for` можно заменить на более короткие `foreach`.

- Цикл ввода не меняем, поскольку в теле цикла в подсказке требуется выводить индекс элемента.

- Второй цикл можно заменить: здесь при проверке элементов их индекс неважен.
- Третий цикл требует явной работы с индексом, его заменить нельзя.
- Цикл вывода также не требует явного обращения к индексам элементов, заменяем на `foreach`.

Глава 2\SearchInArray_2\Program.cs

```
using System;

namespace SearchInArray
{
    class Program
    {
        static void Main(string[] args)
        {
            int n;

            Console.Write("Введите размер массива: ");
            while (!int.TryParse(Console.ReadLine(), out n) ||
                n <= 0)
            {
                Console.Write("Некорректное значение.
                    Введите повторно: ");
            }

            double[] array = new double[n];

            for (int i = 0; i < n; i++)
            {
                Console.Write($"A({i}) = ");
                array[i] = double.TryParse(
                    Console.ReadLine(), out double number)
                    ? number : 0;
            }

            Console.Clear();

            int even_positive = 0;
            int odd_negative = 0;
```

```
            foreach (var number in array)
            {
                if (number > 0 && number % 2 == 0)
                    even_positive++;

                if (number < 0 && number % 2 != 0)
                    odd_negative++;
            }

            int first_zero_index = -1;

            for (int i = 0; i < n; i++)
            {
                if (array[i] == 0)
                {
                    first_zero_index = i;
                    break;
                }
            }

            foreach (var number in array)
            {
                Console.Write(number + "  ");
            }

            Console.WriteLine();

            Console.WriteLine("Количество четных положительных
                чисел: " + even_positive);
            Console.WriteLine("Количество нечетных
                положительных чисел: " + odd_negative);

            if (first_zero_index != -1)
                Console.WriteLine("Индекс первого нуля: " +
                    first_zero_index);
            else
                Console.WriteLine("Нулей в массиве нет");
        }
    }
}
```

2.9.3. Многомерные массивы

Описание многомерного массива

Определение

Многомерный массив *представляет собой таблицу проиндексированных переменных размерности N.*
Общий синтаксис описания:

```
тип[,...,] имя_массива = new тип[размер_1,...,размер_N];
```

где размеры определяют число элементов массива по каждой размерности.

Многомерными считают массивы, имеющие более одной размерности. Чаще всего используются двумерные массивы, который можно отождествить с таблицей или матрицей (в случае числовых значений).

Например, ниже описан трехмерный массив, резервирующий память под $100 * 50 * 3 = 15000$ элементов:

```
double[,,] coords_web = new double[100, 50, 3];
```

Здесь в левой части в квадратных скобках запитые отделяют размерности, указывая, что тип связан с трехмерным массивом. Правая часть вызывает конструктор, задающий число элементов в каждой размерности. Общее количество элементов определяется произведением длин каждой размерности.

Двумерный массив

Определение

Двумерный массив представляет собой таблицу проиндексированных переменных:

```
тип[,] имя_массива = new тип[размер_1,размер_2];
```

где размеры определяют число элементов массива по каждой размерности.

Пример описания двумерного массива вещественных чисел:

```
double[,] table = new double[10, 20];
```

Формально в двумерном массиве (как таблице или матрице) можно выделять строки и колонки. При этом порядок циклической обработки размерностей не важен и зависит лишь от того, какая задача поставлена.

Объявление и инициализация двумерных массивов

В двумерном массиве при обращении к его элементу указывается два индекса, в соотвествие с каждой размерностью.

```
double[,] table = new double[10, 20];
table[8, 15] = 13.42;
```

Инициализация двумерного массива, как и в случае одномерного, может осуществляться как последовательная запись значений в элементы, так непосредственно в момент описания:

```
double[,] table =
{
    { 23.4, 22.5, 22.9 },
    { 28.0, 27.8, 29.7 }
};
```

При такой инициализации важно следить, чтобы число элементов в «колонках» было одинаковым для каждой «строки», т.е. массив должен быть «прямоугольным».

Работа с размерами и размерностями массива

В процессе обработки двумерных массивов удобно обращаться к ряду свойств и методов, которые автоматически отслеживают ранг (количество размерностей), длину каждой размерности и общее число элементов:

```
double[,] table = new double[10, 20];

// size <- 200 (10 * 20, число всех элементов)
int size = table.Length;

// n <- 10 (число элементов 1-й размерности)
int n = table.GetLength(0);
```

```
// m <- 20 (число элементов 2-й размерности)
int m = table.GetLength(1);

// rank <- 2 (число измерений массива)
int rank = table.Rank;
```

Указанные методы и свойства позволяют получить размеры массива в любой момент времени.

Циклический обход двумерного массива

1. Обход массива циклом for

Цикл `for` наиболее универсальный для обработки двумерных массивов, поскольку позволяет обращаться к элементам по индексам.

```
for (int i = 0; i < table.GetLength(0); i++)
{
    for (int j = 0; j < table.GetLength(1); j++)
    {
        // Операции с элементами table[i,j]
    }
}
```

Для обхода двумерного массива требуется два цикла. Внешний цикл перебирает элементы по первому измерению, а внутренний – по второму.

Число элементов каждого измерения получаем с помощью метода `GetLength()`, где параметр указывает индекс размерности.

2. Обход массива циклом foreach

Цикл `foreach` позволяет укоротить код обхода и ограничиться всего лишь одним циклом. Все элементы двумерного массива перебираются как элементы множества.

```
foreach (var elem in table)
{
    // Операции с текущим элементом elem
}
```

Однако такой обход можно использовать лишь в том случае, когда порядок обхода и индексы элемента не влияют на алгоритм обработки. И, разумеется, когда не требуется менять элементы массива.

Количество строк и столбцов

В случае, когда алгоритм обработки двумерного массива предполагает неоднократно использование циклических обходов, рационально записать количество строк и столбцов в переменные:

```
int rows = table.GetLength(0);
int cols = table.GetLength(1);

for (int i = 0; i < rows; i++)
{
    for (int j = 0; j < cols; j++)
    {
        // Операции с элементами table[i,j]
    }
}

// Далее использование row и cols в других циклах
```

Пример обработки двумерного массива

Требуется описать целочисленную матрицу и заполнить ее случайными числами от -1000 до 1000. Найти максимальное и минимальное значение, а также сумму положительных элементов.

Решение

Глава 2\MatrixProcessing_1\Program.cs

```
using System;

namespace MatrixProcessing
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Введите размеры матрицы:");

            Console.Write("n = ");
            int n = GetIntInput(default_number: 1);

            Console.Write("m = ");
            int m = GetIntInput(default_number: 1);

            int[,] matrix = new int[n, m];
```

```
        Random rnd = new Random();

        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++)
                matrix[i, j] = rnd.Next(-1000, 1000);

        int max = matrix[0, 0];
        int min = matrix[0, 0];
        int posit_sum = 0;

        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < m; j++)
            {
                if (matrix[i, j] > max)
                    max = matrix[i, j];

                if (matrix[i, j] < min)
                    min = matrix[i, j];

                if (matrix[i, j] > 0)
                    posit_sum += matrix[i, j];
            }
        }

        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < m; j++)
                Console.Write("{0,5}", matrix[i, j]);

            Console.WriteLine();
        }

        Console.WriteLine($"Максимальное значение =
            {max}");
        Console.WriteLine($"Минимальное значение =
            {min}");
        Console.WriteLine($"Сумма положительных =
            {posit_sum}");
    }

    static int GetIntInput(int default_number = 1)
    {
        return int.TryParse(Console.ReadLine(),
```

```
                    out int number) && (number > default_number)
                    ? number : default_number;
            }
        }
}
```

Комментарии к реализации

В начале осуществляется ввод значения матрицы. Для удобства оформлена функция `GetIntInput()`. Здесь используется значение параметра по умолчанию: если длина размерности не будет указана, то она берется по умолчанию единицей. Также в случае некорректного ввода значение тоже будет исправлено на 1. Метод возвращает полученную размерность в основную часть программы.

Далее объявляется двумерный массив. Создаем экземпляр `rnd` генератора случайных чисел. В двух циклах генерируем случайные целые числа в диапазоне от -1000 до 1000 и записываем их в массив.

Изначально предполагаем, что элемент $A_{0,0}$ матрицы наибольший и наименьший по значению. В циклах перебираем элементы, если необходимо, перезаписываем минимальное и максимальное значение. Также суммируем положительные числа в `posit_sum`.

После того, как пройдены все элементы, в новых циклах выводим элементы построчно, резервируем по 5 знаков на колонку. Наконец, выводится количество искомых элементов.

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите размеры матрицы:
n = 5
m = 8
 -181 -410  -94  261  192 -665  758    1
 -134 -229    2 -435  102  946  681  744
  975  733  132  959  -49 -935  603 -375
 -799  713 -438  -13 -600 -246 -712  -37
  104  669 -912  758  290  641 -244 -588
Максимальное значение = 975
Минимальное значение = -935
Сумма положительных = 10264
```

Рис. 2.103. Результат выполнения программы

Оптимизация

Циклы поиска максимального, минимального и суммы положительных чисел массива можно заменить на `foreach`. Здесь это возможно, поскольку алгоритм перебора не требует работы с индексами и порядок перебора элементов также непринципиален.

Глава 2\MatrixProcessing_2\Program.cs

```
// См. MatrixProcessing_1
// Заменяем циклы for на foreach

foreach (var number in matrix)
{
    if (number > max)
        max = number;

    if (number < min)
        min = number;

    if (number > 0)
        posit_sum += number;
}
```

2.9.4. Ступенчатые массивы

Понятие и особенности двумерных ступенчатых массивов

Определение

*Двумерный **ступенчатый массив** представляет собой таблицу проиндексированных переменных, где вторая размерность может варьировать длину для каждой «строки».*
Общий синтаксис описания:

```
тип[][] имя_массива = new тип[размер][];
```

где размер определяет число элементов первой размерности.

Главной особенностью ступенчатого массива является то, что его вторая размерность может иметь разную длину, т.е. для каждой строки таблицы количество колонок может варьироваться.

На самом деле ступенчатый массив является одномерным массивом, в котором каждый элемент – массив. Поэтому часто говорят, что ступенчатый массив – это массив массивов.

Объявление ступенчатого массива задается следующим образом:

```
int[][] jagged = new int[3][];
```

В отличие от «прямоугольного» двумерного массива здесь скобки отделены. А в правой части вторые скобки пустые, т.е. число элементов каждой «строки» может быть разным.

Инициализация ступенчатого массива

Последовательная инициализация

Поскольку ступенчатый массив является массивом, элементы которого тоже массивы, то инициализация каждой строки предполагает создание нового массива:

```
int[][] jagged = new int[3][];
jagged[0] = new int[2];
jagged[1] = new int[5];
jagged[2] = new int[3];

jagged[1][3] = 2025;
```

Каждый элемент первой размерности является одномерным массивом и создается отдельно (рис. 2.104).

0	0			
0	0	0	2025	0
0	0	0		

Рис. 2.104. Структура ступенчатого массива из примера

Инициализация при объявлении

Ступенчатый массив также можно задать непосредственно при определении:

```
int[][] jagged =
{
    new int[] { 3, -5, 0, 10 },
    new int[] { 40, -4, 1 },
    new int[] { 2025, 1, 0, -34, 56, 22 }
};
```

В версии C# 12.0 появилась более удобная форма описания:

```
var jagged = new[]
{
    new[] { 3, -5, 0, 10 },
    new[] { 40, -4, 1 },
    new[] { 2025, 1, 0, -34, 56, 22 }
};
```

Также можно использовать выражение коллекции:

```
int[][] jagged =
[
    [3, -5, 0, 10],
    [40, -4, 1],
    [2025, 1, 0, -34, 56, 22]
];
```

Циклический обход двумерного ступенчатого массива

1. Обход массива циклом for

Нужно заметить, что свойство `Length` относительно ступенчатого массива возвращает число вложенных массивов («строк»).

Для получения длины каждого отдельного массива (числа «колонок») обращение ведется по индексу элемента:

```
for (int i = 0; i < jagged.Length; i++)
{
    for (int j = 0; j < jagged[i].Length; j++)
    {

    }
}
```

В данном фрагменте:

- `jagged.Length` – длина первой размерности (количество строк);
- `jagged[i].Length` – длина второй размерности (количество колонок или ячеек), которая может быть разной.

2. Обход массива циклом foreach

Гораздо более прозрачной обработка ступенчатого массива будет с использованием цикла `foreach`. Здесь тоже необходимо вложение циклов:

```
foreach (var row in jagged)
{
    foreach (var elem in row)
    {

    }
}
```

Однако внешний цикл перебирает элементы как строки таблицы, записывая их поочередно в переменную `row`. А внутренний перебирает элементы `elem` из каждой строки `row` как из одномерного массива.

Код получился не только короче, но и существенно проще для чтения и понимания.

Массив строк как ступенчатый массив

1. Массив из массива символов

Нетрудно заметить, что одномерный массив строк является двумерным ступенчатым массивом символов.

Например:

```
string[] verse =
{
    "Люблю грозу в начале мая,",
    "Когда весенний, первый гром,",
    "Как бы резвяся и играя,",
    "Грохочет в небе голубом."
};

// row = "Люблю грозу в начале мая,"
string row = verse[0];

char simbol1 = row[4];        // simbol1 = 'ю'
char simbol2 = verse[0][4];   // simbol2 = 'ю'
```

Здесь массив `verse` можно является обычным одномерным массивам из текстовых строк. Однако каждая строка – индексируемая коллекция символов, поэтому получаем ступенчатый массив.

2. Массив из массива строк

Если же создать ступенчатый массив, где каждый элемент является массивом строк, то речи идет уже о ступенчатом массиве символов с тремя размерностями.

Например, создадим массив покупок `purchases`, разделенных на группы товаров:

```
string[][] purchases =
[
    ["бананы", "яблоки", "груши"],
    ["хлеб"],
    ["средство для мытья посуды", "зубная паста"],
];

foreach (var shopping_group in purchases)
{
    foreach (var product in shopping_group)
    {
        Console.Write(product + " | ");
    }

    Console.WriteLine();
}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
бананы | яблоки | груши |
хлеб |
средство для мытья посуды | зубная паста |
```

Рис. 2.105. Вывод ступенчатого массива строк

Здесь для обращения к отдельным видам покупок, покупкам и символам можно обращаться как напрямую через тройную индексацию, так и создавая промежуточные переменные:

```
string[] shopping_group = purchases[2];
string product = shopping_group[0];

string product1 = purchases[2][0];
string product2 = shopping_group[0];

char sim1 = purchases[2][0][0];
char sim2 = shopping_group[0][0];
char sim3 = product[0];
```

```
Console.WriteLine($"Третья группа покупок:
    {shopping_group}");
Console.WriteLine($"Первый продукт третьей группы
    покупок: {product}");

Console.WriteLine($"Первый продукт третьей группы
    покупок: {product1}");
Console.WriteLine($"Первый продукт третьей группы
    покупок: {product2}");

Console.WriteLine($"Первый символ искомого продукта:
    {sim1}");
Console.WriteLine($"Первый символ искомого продукта:
    {sim2}");
Console.WriteLine($"Первый символ искомого продукта:
    {sim3}");
```

```
Третья группа покупок: System.String[]
Первый продукт третьей группы покупок: средство для мытья посуды
Первый продукт третьей группы покупок: средство для мытья посуды
Первый продукт третьей группы покупок: средство для мытья посуды
Первый символ искомого продукта: с
Первый символ искомого продукта: с
Первый символ искомого продукта: с
```

Рис. 2.106. Вариации обращения к элементам массива

Пример обработки двумерного массива

Необходимо обработать ступенчатый массив, организовав ввод его элементов с клавиатуры и дальнейший вывод в надлежащей форме. Количество элементов в каждой строке запрашивается в процессе ввода элементов.

Решение

Глава 2\JagedArray_1\Program.cs

```
using System;

namespace JaggedArray
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            Console.Write("Число строк: ");
            int n = ConsoleInput.GetIntInput();

            double[][] jagged = new double[n][];

            for (int i = 0; i < n; i++)
            {
                Console.Write($"Число элементов в {i}й
                    строке: ");
                int m = ConsoleInput.GetIntInput();
                jagged[i] = new double[m];

                for (int j = 0; j < m; j++)
                {
                    Console.Write($"A({i};{j}) = ");
                    jagged[i][j] =
                        ConsoleInput.GetDoubleInput();
                }
            }

            Console.Clear();

            for (int i = 0; i < jagged.Length; i++)
            {
                for (int j = 0; j < jagged[i].Length; j++)
                    Console.Write($"{jagged[i][j],6}");

                Console.WriteLine();
            }
        }
    }

    static class ConsoleInput
    {
        public static int GetIntInput(int default_number = 1)
        {
            return int.TryParse(Console.ReadLine(),
                out int number) && (number > default_number)
                ? number : default_number;
        }

        public static double GetDoubleInput(
            double default_number = 0)
        {
```

```
            return double.TryParse(Console.ReadLine(),
            out double number) && (number > default_number)
            ? number : default_number;
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Program> dotnet run
Число строк: 4
Число элементов в 0й строке: 3
A(0;0) = 4
A(0;1) = 8
A(0;2) = -3
Число элементов в 1й строке: 7
A(1;0) = 0
A(1;1) = 5
A(1;2) = 3,5
A(1;3) = 2
A(1;4) = 0,02
A(1;5) = 6
A(1;6) = 7
Число элементов в 2й строке: 1
A(2;0) = 9
Число элементов в 3й строке: 4
A(3;0) = -3
A(3;1) = 2
A(3;2) = 10
A(3;3) = 7
```

```
    4    8   -3
    0    5  3,5    2 0,02    6    7
    9
   -3    2   10    7
```

Рис. 2.107. Ввод и вывод ступенчатого массива

Комментарии к реализации

В начале запрашивается количество строк. Для каждой строки в цикле запрашиваем число элементов (колонок) и по этому размеру создается массив.

Ввод размерностей и самих чисел обрабатывают описанные нами функции `GetIntInput()` и `GetDoubleInput()`, которые при необходимости испрвляют значения при некорректном вводе.

Вывод элементов также организуем в два цикла. Число строк получаем из свойства `Length`. Для внутреннего цикла указывается индекс строки как элемента-массива.

Оптимизация

Вывод существенно упрощается, если циклы `for` заменить на `foreach`.

Глава 2\JagedArray_2\Program.cs

```
// См. код JagedArray_1

foreach (var row in jagged)
{
    foreach (var elem in row)
        Console.Write($"{elem,6}");

    Console.WriteLine();
}
```

2.9.5. Методы и свойства массивов

Часто используемые свойства и методы

Поскольку все массивы производны от класса `Array`, то они наблюдают целым набором встроенных свойств и методов, которые помогают преобразовывать массивы и извлекать из них данные.

Свойствами `Length` и `Rank` мы рассмотрели ранее.

Рассмотрим некоторые методы.

Метод	Описание
`Clear()`	Очищает массив, устанавливая для всех его элементов значение по умолчанию.
`Copy()`	Копирует часть одного массива в другой массив.
`Exists()`	Проверяет, содержит ли массив определенный элемент.
`Find()`	Находит элемент, удовлетворяющий определенному условию.
`FindAll()`	Находит все элементы, которые удовлетворяют определенному условию.
`IndexOf()`	Возвращает индекс элемента.
`Resize()`	Изменяет размер одномерного массива.
`Reverse()`	Располагает элементы массива в обратном порядке.
`Sort()`	Сортирует элементы одномерного массива.

Рис. 2.108. Ряд методов массива

Перечисленные методы являются статическими и вызываются напрямую из класса `Array`.

Простые примеры

Для сортировки заданного массива используется метод `Sort()`. По умолчанию он сортирует массив по возрастанию. Если с помощью метода `Reverse()` обратить расположение элементов в обратном порядке, то это будет равносильно сортировке по убыванию:

```
int[] mass = { 34, 7, -4, 0, 2, 0, 1, 5 };

Array.Sort(mass);
Array.Reverse(mass);
```

Для поиска индекса элемента, равного указанному, используется метод `IndexOf()`. Например, найдем индекс первого нуля:

```
int indexOfZero = Array.IndexOf(mass, 0);
```

Для создания полных или частичных копий используется метод `Copy()`. В качестве примера скопируем первые четыре элемента в новый массив:

```
int[] massCopy = new int[4];
Array.Copy(mass, massCopy, 4);
```

Метод `Exists()` позволяет проверить наличие хотя-бы одного элемента по заданному условию. Например, проверим наличие хотя-бы одного положительного четного элемента:

```
bool itExists = Array.Exists(
    mass, x => (x > 0) && (x % 2 == 0)
);
```

Второй параметр является функцией, записанной в более коротком синтаксисе **лямбда-выражения** (**функциональное выражение**). Переменная `x` является параметром и в нее последовательно подставляется каждый элемент массива. Название может быть и любым другим. Если хотя-бы один элемент будет найден, функция возвратит `true`.

Приведенный пример также показывает гибкость языка C#. Он сочетает в себе как классический императивный подход, так и элементы функционального программирования.

Глубокое копирование массивов

Важно заметить, что копирование массивов выносится отдельным методом в силу особенности работы с классами (которыми и представлены массивы).

Так, в следующем примере в переменную `copy` на самом деле запишется не новый массив как копия `test`, а будет создана ссылка на тот же массив.

```
double[] test = { 23.5, 56.3, -67.9, 5.0, 12.3 };
double[] copy = test;
```

Иными словами, переменные `test` и `copy` будут ссылаться на один и тот же массив. Меняя элементы в `test` автоматически меняются элементы в copy и наоборот.

Для копирования переменных класса используют **глубокое копирование** и оно зависит от структуры объекта.

В следующем примере для получения полной и независимой копии создается новый массив по размеру искомого и с помощью метода `Copy()` осуществляется копирование элементов:

Глава 2\DeepCopy\Program.cs

```
using System;

namespace DeepCopy
{
    class Program
    {
        static void Main(string[] args)
        {
            double[] test = { 23.5, 56.3, -67.9, 5.0, 12.3 };

            int size = test.Length;
            double[] copy = new double[size];
            Array.Copy(test, copy, size);

            Console.WriteLine("Массив и его копия:");
            Console.WriteLine(ArrayToString(test));
            Console.WriteLine(ArrayToString(copy));

            test[0] = 2025;

            Console.WriteLine("Массив и его копия после
```

```
                изменения одного из элементов:");
            Console.WriteLine(ArrayToString(test));
            Console.WriteLine(ArrayToString(copy));
        }

        // Функция возвращает запись массива в одну строку
        static string? ArrayToString(double[] array)
        {
            string? result = " | ";

            foreach (var elem in array)
                result += elem + " | ";

            return result;
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Массив и его копия:
 | 23,5 | 56,3 | -67,9 | 5 | 12,3 |
 | 23,5 | 56,3 | -67,9 | 5 | 12,3 |
Массив и его копия после изменения одного из элементов:
 | 2025 | 56,3 | -67,9 | 5 | 12,3 |
 | 23,5 | 56,3 | -67,9 | 5 | 12,3 |
```

Рис. 2.109. Результаты копирования

Видно, что после изменении одного из элементов искомого массива второй массив не меняется, потому что переменные ссылки ссылаются на разные объекты в памяти.

2.9.6. Рекомендации учителю

- Изучение массивов следует начать с анализа примеров, в каких ситуациях необходима работа с массивами как с индексированными последовательностями элементов одного типа.
- Важно проработать процедуру описания, заполнения и обхода одномерных массивов. Обратите внимание на то, что переменная массива является ссылкой и как ее правильно использовать. В операциях необходимо затронуть разные типы циклов.

- Далее переходите к изучению алгоритмов поиска, преобразований и сортировки в одномерном массиве. Важно подбирать задачи, требующие комбинирования операций в теле цикла: ветвления, вложенные циклы.
- Для двумерных массивов следует отработать решение задач, предполагающих операции с табличными данными (матрицами).
- Покажите, в каких ситуациях ступенчатые массивы более удобны, чем «прямоугольные». Также оперируйте циклом `foreach` там, где он оптимизирует и упрощает код.
- Продемонстрируйте ряд свойств и методов, которые упрощают обработку элементов массива и массивов в целом.

Вопросы для самопроверки

1. Что представляют собой массивы и какие данные в них рационально хранить?
2. Опишите варианты инициализации одномерных массивов.
3. Чем отличаются алгоритмы обхода массива с помощью цикла `for` и `foreach`? Приведите примеры, когда рационально использовать первый и второй операторы цикла соответственно.
4. Как объявляются и инициализируются двумерные массивы?
5. Какие свойства и методы помогают в преобразовании массивов и извлечении из них данных?
6. В чем особенности структуры ступенчатых массивов?
7. Проведите сравнительный анализ возможностей обычных и ступенчатых массивов.

Практикум

1. Прямоугольные массивы

Задание 1

1. Создайте проект *ExpectedValue*.
2. Заданы два массива (одного размера):
    - ***X*** – хранит значения случайной величины;

- ***P*** – хранит значения вероятностей появления величины ***X*** (сумма всех вероятностей равна 1).

3. Составьте программу, вычисляющую математическое ожидание случайной величины ***X***. Вывести данные массивов (следуя комментариям в коде), рис. 2.110.
4. Математическое ожидание вычисляется по формуле:

$$M(X) = x_1p_1 + x_2p_2 + \ldots + x_np_n.$$

5. Разберите и используйте для основы следующий код:

Глава 2\ExpectedValue\Program.cs

```
using System;

namespace ExpectedValue
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Введите число вариант события: ");
            int n;
            while (!int.TryParse(Console.ReadLine(), out n) ||
                n <= 0)
            {
                Console.Write("Некорректное значение.
                    Введите повторно: ");
            }

            double[] X = new double[n];
            double[] P = new double[n];

            for (int i = 0; i < n; i++)
            {
                Console.Write($"X({i}) = ");
                X[i] = GetDoubleInput();

                Console.Write($"P({i}) = ");
                P[i] = GetDoubleInput();
            }

            Console.Clear();

            Console.Write("X:\t");
```

```
            // Вывести циклом строку с X[i]:

            Console.Write("\nP:\t");

            // Вывести циклом строку с P[i]:


            // Вычислить циклом математическое ожидание
            double M = 0;

            Console.WriteLine("\n\nM(X) = " + M);
        }

        static double GetDoubleInput()
        {
            return double.TryParse(Console.ReadLine(),
                out double num) ? num : 0;
        }
    }
}
```

```
Введите число вариант события: 4
X(0) = 0
P(0) = 0,1
X(1) = 1
P(1) = 0,1
X(2) = 3
P(2) = 0,75
X(3) = 10
P(3) = 0,05
```

```
X:      0       1       3       10
P:      0,1     0,1     0,75    0,05

M(X) = 2,85
```

Рис. 2.110. Результат работы программы из задания 1

Задание 2

1. Создайте проект *Quotes*.
2. Для заданного массива строк с текстом цитат вывести те, которые имеют длину не более 50 символов и оканчиваются точкой. Для проверки используйте цикл `foreach`.
3. Замечание: к символам в строках можно обращаться по индексу, как в массивах. Используйте заготовку ниже.

Глава 2\Quotes\Program.cs

```
using System;

namespace Quotes
{
    class Program
    {
        static void Main(string[] args)
        {
            // Массив цитат
            string[] quotes =
            {
                "Не волнуйтесь, если что-то не работает.
                    Если бы всё работало, вас бы уволили.",
                "Некоторые проблемы лучше не решать,
                    а избегать.",
                "Работает? Не трогай!",
                "Если сразу не получилось хорошо, назовите это
                    версией 1.0.",
                "Молодые специалисты не умеют работать,
                    а опытные специалисты умеют не работать.",
                "В теории - теория и практика неразделимы.
                    На практике - это не так"
            };

            // С помощью цикла foreach отобрать искомые
            // цитаты и вывести на экран:


            Console.WriteLine("Поиск завершен");
        }
    }
}
```

Задание 3

1. Создайте проект *FindZeros*.
2. С клавиатуры задается произвольная матрица.
3. Для каждой строки подсчитать и вывести на экран количество нулей и их позицию (индексы).
4. Разберите предложенный ниже код заготовки и используйте его в качестве основы.
5. Также вывести общее количество нулей в матрице (см. рис. 2.111).

Глава 2\FindZeros\Program.cs

```
using System;

namespace FindZeros
{
    class Program
    {
        static void Main(string[] args)
        {
            // Вводим размеры матрицы, создаем массив
            // и вводим числа
            Console.WriteLine("Введите размеры матрицы");

            Console.Write("Число строк N: ");
            int n = GetIntInput();

            Console.Write("Число колонок M: ");
            int m = GetIntInput();

            double[,] matrix = new double[n,m];

            for (int i = 0; i < n; i++)
            {
                for (int j = 0; j < m; j++)
                {
                    Console.Write($"A[{i};{j}] = ");
                    matrix[i,j] = GetDoubleInput();
                }
            }

            // Далее с помощью двух циклов for ищем нули.
            // Внешний цикл перебирает строки,
            // внутренний – колонки.
            // Общее число нулей накапливать в переменную
            // zeros_counter.
            // А для каждой строки считать число нулей
            // отдельно, накапливая их в переменную
            // row_zeros_counter


        }

        static int GetIntInput(int default_number = 1)
        {
```

```
            return int.TryParse(
                Console.ReadLine(),
                out int number) && (number > default_number)
                ? number : default_number;
        }

        static double GetDoubleInput(
            double default_number = 0
        )
        {
            return double.TryParse(
                Console.ReadLine(),
                out double number) &&
                (number > default_number)
                ? number : default_number;
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Program> dotnet run
Введите размеры матрицы
Число строк N: 3
Число колонок M: 6
A[0;0] = 2
A[0;1] = 0
A[0;2] = -5
A[0;3] = 2,45
A[0;4] = 1,3
A[0;5] = 6
A[1;0] = 0
A[1;1] = -5,3
A[1;2] = 0
A[1;3] = 2,05
A[1;4] = 0
A[1;5] = 0
A[2;0] = 2
A[2;1] = -4,5
A[2;2] = -0,01
A[2;3] = 2
A[2;4] = 5
A[2;5] = 7
```

```
(0;1)
В 0-й строке 1 нулей
(1;0)
(1;2)
(1;4)
(1;5)
В 1-й строке 4 нулей
В 2-й строке нет нулей
Всего 5
```

Рис. 2.111. Результат работы программы из задания 3

Задание 4

1. Создайте проект *MaximumInRows*.
2. С клавиатуры задается произвольная матрица.
3. Сформировать одномерный массив, состоящий из наибольших значений в каждой строке матрицы. Оформите ввод и вывод, как изображено на рис. 2.112.
4. Используйте заготовку.

Глава 2\MaximumInRows\Program.cs

```
using System;

namespace MaximumInRows
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Введите размеры матрицы");

            Console.Write("Число строк N: ");
            int n = ConsoleInput.GetIntInput();

            Console.Write("Число колонок M: ");
            int m = ConsoleInput.GetIntInput();

            double[,] matrix = new double[n,m];

            for (int i = 0; i < n; i++)
            {
                for (int j = 0; j < m; j++)
                {
                    Console.Write($"A[{i};{j}] = ");
                    matrix[i,j] =
                        ConsoleInput.GetDoubleInput();
                }
            }

            // Резервируем массив, в который далее
            // записываются максимальные для каждой строки
            double[] rows_maximum = new double[n];

            // Далее поиск максимальных и запись
            // в rows_maximum
```

```
            // Вывод искомой матрицы

            // Вывод максимальных значений в каждой строке
        }
    }

    static class ConsoleInput
    {
        public static int GetIntInput(int default_number = 1)
        {
            return int.TryParse(Console.ReadLine(),
                out int number) &&
                (number > default_number)
                ? number : default_number;
        }

        public static double GetDoubleInput(
            double default_number = 0
        )
        {
            return double.TryParse(Console.ReadLine(),
                out double number) &&
                (number > default_number)
                ? number : default_number;
        }
    }
}
```

```
Введите размеры матрицы
Число строк N: 3
Число колонок M: 5
A[0;0] = -5
A[0;1] = 10
A[0;2] = 3,56
A[0;3] = 2
A[0;4] = 0
A[1;0] = 1
A[1;1] = 0
A[1;2] = -2
A[1;3] = 3,6
A[1;4] = 5
A[2;0] = 9
A[2;1] = -3
A[2;2] = -10
A[2;3] = 2,5
A[2;4] = 2,7
      -5      10    3,56       2       0
       1       0      -2     3,6       5
       9      -3     -10     2,5     2,7
Строка 0. Максимум равен 10
Строка 1. Максимум равен 5
Строка 2. Максимум равен 9
```

Рис. 2.112. Результат работы программы из задания 4

2. Ступенчатые массивы и методы массивов

Задание 1

1. Создайте проект *Temperature*.
2. В ходе эксперимента исследователь фиксировал показатели температуры нагрева железной пластины и записывал их в журнал (каждая строка соответствует одному дню наблюдений).
3. Вывести таблицу температур на экран с точностью до десятых (см. рис. 2.113).
4. Для каждого дня посчитайте среднее значение температуры.

Глава 2\Temperature\Program.cs

```
using System;

namespace Temperature
{
    class Program
    {
        static void Main(string[] args)
        {
            // Классический обновленный синтаксис
            // инициализации:
            // double[][] temperature =
            // {
            //     new[] { 34.2, 35.6, 35.5, 34.8 },
            //     new[] { 34.3, 34.1, 35.0 },
            //     new[] { 37.9, 37.8, 38.1, 38.6, 38.5, 38.6,
                           37.8 }
            // };

            // Через новый синтаксис выражения коллекции
            double[][] temperature =
            [
                [34.2, 35.6, 35.5, 34.8],
                [34.3, 34.1, 35.0],
                [37.9, 37.8, 38.1, 38.6, 38.5, 38.6, 37.8]
            ];

            // Вывести таблицу (используйте вложенные
            // циклы foreach)

            Console.WriteLine();
```

```
            Console.WriteLine("Средние показатели:");

            // Вычислить средний показатель для каждой
            // строки (используйте цикл foreach)


        }
    }
}
```

```
    34,2    35,6    35,5    34,8
    34,3    34,1    35,0
    37,9    37,8    38,1    38,6    38,5    38,6    37,8

Средние показатели:
35,0
34,5
38,2
```

Рис. 2.113. Результат работы программы из задания 1

Задание 2

1. Создайте проект *ArrayMethods*.
2. Для заданного массива `test`:
   - Создайте новый массив `copy` и скопируйте в него искомый `test`, далее работайте с копией.
   - Отсортируйте массив по возрастанию.
   - Выведите оба массива.
   - Проверьте, есть ли в массиве хотя бы один нуль (с помощь метода `Exists()`).
   - Проверьте, есть ли в массиве хотя бы одно число из интервала [0,15] (с помощь метода `Exists()`).
   - Создайте новый массив из положительных элементов copy (с помощь метода `FindAll()`).

Глава 2\ArrayMethods\Program.cs

```
using System;

namespace ArrayMethods
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            double[] test = { 23.5, 56.3, -67.9, 22.3, 10.0,
                -5.0, 12.3 };

            int n = test.Length;
            double[] copy = new double[n];
            Array.Copy(test, copy, n);
            Array.Sort(copy);

            Console.WriteLine("Основной массив:");
            PrintArray(test);
            Console.WriteLine("Отсортированный массив:");
            PrintArray(copy);

            if (Array.Exists(copy, (x) => x == 0))
                Console.WriteLine("Есть нуль(и)");
            else
                Console.WriteLine("Нулей нет");

            // Другие проверки и поиск...

        }

        // Метод для вывода массива
        static void PrintArray(double[] arr)
        {
            foreach (var num in arr)
            {
                Console.Write($"{num}  ");
            }

            Console.WriteLine();
        }
    }
}
```

```
Основной массив:
23,5  56,3  -67,9  22,3  10  -5  12,3
Отсортированный массив:
-67,9  -5  10  12,3  22,3  23,5  56,3
Нулей нет
Есть число(а) из интервала [0,15]
Положительные элементы:
10  12,3  22,3  23,5  56,3
```

Рис. 2.114. Результат работы программы из задания 2

2.10. Методы как функции внутри класса

2.10.1. Объявление метода класса

Метод как подпрограмма

Определение

Метод *– подпрограмма (процедура или функция), которая является частью класса или экземпляра класса.*

В языке C# методы отвечают за определённые операции и преобразования данных, что позволяет выстраивать логику приложения.

До появления объектно-ориентированного программирования в языках программирования использовалось понятие функций и процедур – модулей, которые позволяли избавлять от многократного дублирования некоторой логики обработки данных.

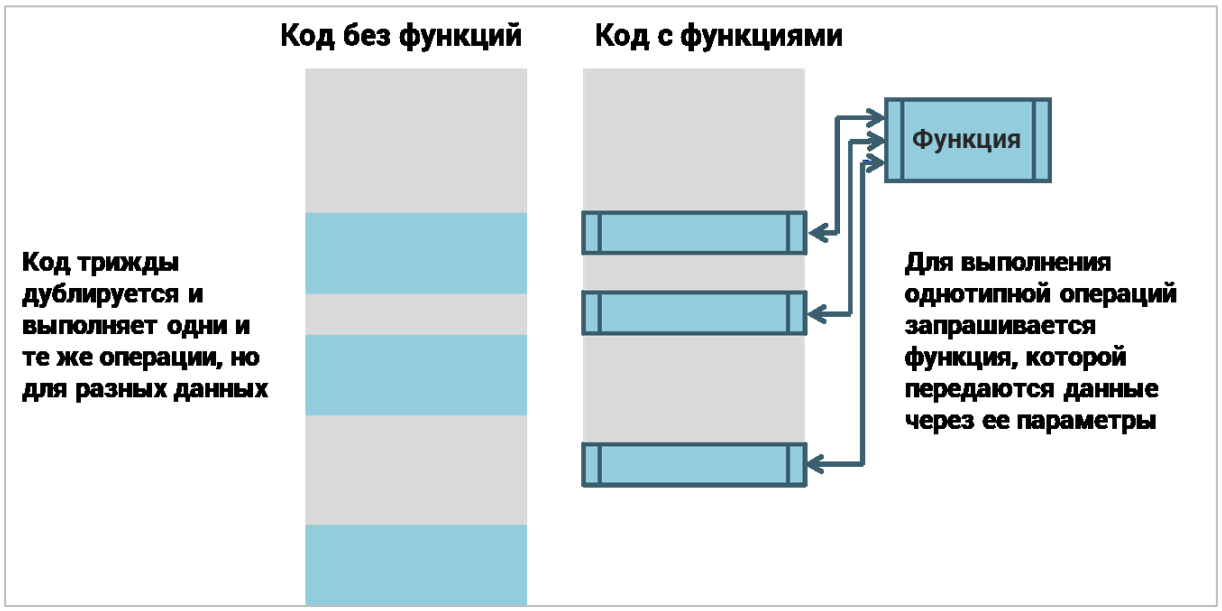


Рис. 2.115. Использование функций позволяет упростить работу с кодом

Использование подпрограмм позволяет решать следующие проблемы:

- возможность повторно использовать логику обработки данных;

- упрощает чтение кода и улучшает общую структуру программы;
- упрощает отладку и тестирование проекта;
- разделяет программу на модули, каждый из которых отвечает за свою область задач;
- делает более прозрачной поддержку и масштабирование проекта.

Это полезно знать!

В архитектуре кода языка C# методы являются частью класса, т.е. они отвечают за обработку данных и логики класса или объектов этого класса.

Синтаксис метода

Определение

Синтаксис метода:

```
[доступ] [static] возвращаемый_тип имя(
    список_параметров
)
{
    // тело функции
}
```

Рассмотрим далее подробнее компоненты метода, основываясь на простом примере вычисления факториала числа:

```
private static long Factorial(int n)
{
    long factor = 1;

    for (int i = 1; i <= n; i++)
        factor *= n;

    return factor;
}
```

Описание метода определяет логику обработки данных. Чтобы метод заработал, его необходимо вызвать. В данном примере метод является функцией, возвращающей вычисленное значение факториала числа, поэтому его можно присвоить переменной:

```
long f = Factorial(6);
```

Модификаторы доступа

Модификатор доступа устанавливает уровень видимости метода по отношению к другим классам.

Есть несколько модификаторов, которые определяют разные правила видимости методов в классах. Чаще всего используются следующие уровни:

- **`public`** – открытый, т.е. метод можно использовать как внутри класса, так и при обращении из других классов;
- **`private`** – закрытый, т.е. метод можно использовать только внутри класса (по умолчанию доступ к методам считается закрытым, `private` указывать необязательно).

```
private static long Factorial(int n)
{
    long factor = 1;

    for (int i = 1; i <= n; i++)
        factor *= n;

    return factor;
}
```

Рис. 2.116. Модификатор доступа: метод закрыт в классе

Обычно закрытыми делают методы, которые обеспечивают обработку данных внутри класса и их нежелательно использовать внешне.

Отношение к классу или экземпляру

Тип отношения показывает, является ли метод статическим или методом экземпляра:

- **Статический метод** привязан к классу и вызывается относительно него, существует в единственном экземпляре.

- **Метод экземпляра** (объекта класса) привязывается к объектам, которые создаются на базе класса, т.е. каждый объект получает копию метода.

Подробно значение этой команды будет раскрыто в разделе, посвященному классам.

```
static long Factorial(int n)
{
    long factor = 1;

    for (int i = 1; i <= n; i++)
        factor *= n;

    return factor;
}
```

Рис. 2.117. Статичный метод привязан к классу

Тип возвращаемого значения

Тип метода определяется значением, которое он возвращает. За возврат значения отвечает оператор **`return`**. Справа от оператора указывается выражение, которое возвращается. Тип метода должен совпадать с типом возвращаемого значения. Тип `long` взят в силу быстрого роста факториала числа.

В методе может быть несколько операторов `return`. Однако необходимо, чтобы любая возможная ветвь метода завершалась оператором `return`, т.е. возвратом значения. Иначе при компиляции генерируется ошибка.

```
static long Factorial(int n)
{
    long factor = 1;

    for (int i = 1; i <= n; i++)
        factor *= n;

    return factor;
}
```

Рис. 2.118. Метод возвращает значение типа `long`

Название метода

Название метода – это его именной идентификатор. При обращении к методу указывается его имя и в круглых скобках параметры, если имеются.

В C# названия методов, как и классов, и пространств имен, принято начинать с заглавной буквы.

```
static long Factorial(int n)
{
    long factor = 1;

    for (int i = 1; i <= n; i++)
        factor *= n;

    return factor;
}


long f = Factorial(6);
```

Рис. 2.119. Название метода в описании и вызове

Список параметров

Список параметров – это одна или более переменных, через которые методу передаются дополнительные данные. Каждый **формальный параметр** указывается вместе с типом, он должен соответствовать типу фактических параметров (аргументов). Если у метода несколько параметров, то они перечисляются через запятую.

Методы могут быть и без параметров: в этом случае при вызове метода скобки остаются пустыми.

Формальные параметры являются копиями **аргументов** (фактические параметры), и их названия могут совпадать.

```
static long Factorial(int n)
```

Рис. 2.120. Методу передается одни параметр – число, факториал которого он должен возвратить

Тело метода

Тело метода реализует его работу. В этом блоке кода допускается использование переменных, которые указаны в качестве параметров в заголовке метода. А переменные, описанные в теле метода, являются локальными, т.е. доступными только в области метода.

```
static long Factorial(int n)
{
    long factor = 1;

    for (int i = 1; i <= n; i++)
        factor *= n;

    return factor;
}
```

Рис. 2.121. Тело метода реализует код обработки данных

Вызов метода

Для вызова метода указывается его имя и фактические аргументы. Аргументами могут выступать литералы или переменные, значение которых будет скопировано в формальные параметры метода.

```
long f = Factorial(6);
```

Т.к. метод статичный, то его можно вызвать напрямую из класса `Program`.

```
long f = Program.Factorial(6);
```

Однако это излишне, поскольку метод уже вызывается внутри класса `Program`, где и описан.

Приведем код программы целиком:

Глава 2\FactorialFunction\Program.cs

```
using System;

namespace FactorialFunction
{
    class Program
    {
        static void Main(string[] args)
        {
            int n = 0;
```

```
            Console.Write(
                "Введите целое неотрицательное n: "
            );
            while (!int.TryParse(Console.ReadLine(), out n) ||
                n < 0)
            {
                Console.Write("Недопустимое значение!
                    Введите целое неотрицательное n: ");
            }

            Console.WriteLine($"{n}! = {Factorial(n)}");
        }

        static long Factorial(int n)
        {
            long result = 1;

            for (int i = 1; i <= n; i++)
                result *= i;

            return result;
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Введите целое неотрицательное n: 8
8! = 40320
```

Рис. 2.122. Результат работы программы по вычислению факториала числа

Здесь важно обратить внимание, что метод `Factorial()` описан на одном уровне с методом `Main()` внутри класса. При этом его можно было описать и выше: компилятору языка C# порядок описания методов непринципиален.

Добавление других методов

Дополним возможности проекта *FactorialFunction*.

На основе метода `Factorial()` добавьте еще два метода для вычисления комбинаторных формул для сочетаний и размещений.

- Метод `Combo()`, возвращающий количество комбинаций выборки m элементов из n элементов ($m \leq n$).

$$C_n^m = \frac{n!}{m!\,(n-m)!}$$

- Метод `Accomodation()`, возвращающий число размещений m элементов из n элементов ($m \le n$).

$$A_n^m = \frac{n!}{(n-m)!}$$

Решение

Глава 2\Combinatorics\Program.cs

```
using System;

namespace Combinatorics
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("n = ");
            int n = GetIntegerInput();

            Console.Write("m = ");
            int m = GetIntegerInput();

            long combinations = Combo(n, m);
            long accommodations = Accomodation(n, m);

            Console.WriteLine($"{n}! = {Factorial(n)}");
            Console.WriteLine($"{m}! = {Factorial(m)}");
            Console.WriteLine($"Сочетания C({n};{m}) =
                {combinations}");
            Console.WriteLine($"Размещения A({n};{m}) =
                {accommodations}");
        }

        // Функция возвращает факториал числа n
        static long Factorial(int n)
        {
            long result = 1;

            for (int i = 1; i <= n; i++)
                result *= i;

            return result;
```

```
        }

        // Функция возвращает число сочетаний без повтора
        // m элементов из набора n элементов
        static long Combo(int n, int m)
        {
            return Factorial(n) /
                (Factorial(m) * Factorial(n - m));
        }

        // Функция возвращает число размещений без повтора
        // m элементов из набора n элементов
        static long Accomodation(int n, int m)
        {
            return Factorial(n) / Factorial(n - m);
        }

        // Функция возвращает введенное целое число
        static int GetIntegerInput()
        {
            int number = 0;

            while (!int.TryParse(
                Console.ReadLine(), out number) || number < 0
            )
            {
                Console.Write("Недопустимое значение!
                    Введите целое неотрицательное число: ");
            }

            return number;
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
n = 10
m = 7
10! = 3628800
7! = 5040
Сочетания C(10;7) = 120
Размещения A(10;7) = 604800
```

Рис. 2.123. Результат вычисления по комбинаторным формулам

Комментарии к реализации

Формулы для расчета числа сочетаний и размещений используют вычисление факториала, причем неоднократно. Поэтому рационально использовать уже описанный метод `Factorial()` в определении методов `Combo()` и `Accomodation()`.

Оба метода принимают по два целочисленных параметра. Задача методов – возвратить вычисленные значение числа комбинаций. Эти числа всегда получаются целыми (можно доказать путем сокращения в формулах), поэтому методы имеют возвращаемый тип `long`. С другой стороны, ошибки округления исключаются в силу использования целочисленных переменных в операциях, и возвращаемое значение командой `return` будет целым числом (согласно типу метода).

Также для безопасной обработки ввода данных в классе описан метод `GetIntegerInput()`, который возвратит только допустимое для работы значение.

2.10.2. Тип метода

1. Метод с возвратом значения

Метод, который возвращает значение, является **функцией**.

Особенности метода с возвратом значения:

- возвращает некоторое значение с помощью оператора `return`;
- тип метода совпадает с типом возвращаемого значения;
- оператор `return` обязателен и указывает на значение, подлежащее возврату;
- может быть присвоен переменной или использоваться в других допустимых операциях как значение.

В рассмотренных ранее примерах мы создавали методы с возвратом значения, т.е. функции:

```
static long Combo(int n, int m)
{
    return Factorial(n) /
        (Factorial(m) * Factorial(n - m));
}
. . . . . .
long combinations = Combo(n, m);
```

2. Метод без возврата значения

Метод, который не возвращает значение, является аналогом **процедуры**.

Особенности метода без возврата значения:

- не возвращает значение;
- всегда имеет тип **`void`** («пустой»);
- оператор `return` необязателен, но может использоваться как команда выхода из метода.

Пример описания и вызова метода без возврата, т.е. процедуры:

```
static void Print(string text)
{
    Console.WriteLine(text);
}
. . . . . .
Print("Привет!");
```

Это полезно знать!

В C# не делают различия между понятиями «функция» и «процедура»: процедура тоже является функцией, которая не возвращает значения (вернее – возвращает пустое значение).
*Ну а поскольку функция всегда привязана к какому либо классу, то обычно используется название «**метод**».*

2.10.3. Статический метод и метод экземпляра

1. Статический метод

Метод, который привязывается к классу, называют **статическим**.

Особенности статических методов:

- описываются с модификатором **`static`**;
- являются частью самого класса и существуют в одном экземпляре;
- вызывается относительно имени класса;
- не могут быть привязаны к объекту, который создается на базе класса.

Можно сказать, что если сам класс является объектом, то статический метод является частью этого объекта класса.

Например, все методы класса `Math` для вычисления математических функций являются статическими:

```
double t = Math.Sqrt(20.4);
```

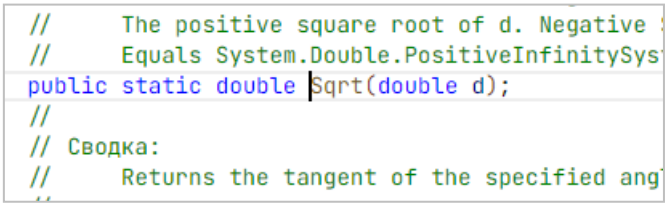


Рис. 2.124. Метод `Sqrt()` статический и вызывается напрямую из класса `Math`

2. Метод экземпляра класса

Метод, который привязывается к объекту, созданному на основе класса, называют **методом экземпляра** или просто **экземплярным**. Понятие экземпляра можно отождествить с понятием **объекта**.

Особенности методов экземпляра класса:

- является частью экземпляра класса, т.е. объекта, который создается на базе класса;
- каждый создаваемы экземпляр класса получает свою копию метода;
- вызываются относительно экземпляра класса.

Например, методы класса `Random` для генерирования случайных чисел являются экземплярными:

```
Random rnd = new Random();
int rnd_num = rnd.Next(100);
```

Здесь объект `rnd` создается как экземпляр класса `Random`.

Метод `Next()` «привязан» к объекту `rnd` и вызывается как часть этого объекта. Таких независимых объектов (как отдельные переменные) можно создать любое число и они будут взаимно независимы.

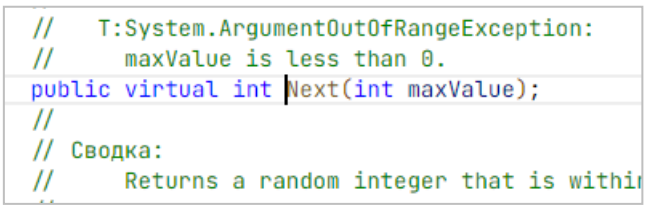


Рис. 2.125. Методы класса `Random` привязываются к экземплярам класса

Пример решения задач

Требуется написать метод `IsAmicable()`, проверяющий, являются ли два заданных числа дружественными. Дружественными называют пары целых чисел, для которых первое равно сумме делителей второго, а второе равно сумме делителей первого.

Например, дружественными являются числа 220 и 284.

$$n = 220$$
$$(284\colon\ 1 + 2 + 4 + 71 + 142 = 220)$$
$$m = 284$$
$$(220\colon\ 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284)$$

Решение

Для проверки двух чисел потребуется дважды вычислять сумму делителей первого и второго числа. Поэтому рационально описать еще одну функцию `SumOfDivisors()`, которая будет вычислять сумму двух заданных чисел.

Глава 2\FrendlyNumbers\Program.cs

```
using System;

namespace FrendlyNumbers
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("n = ");
            int n = GetIntegerNumber();
            Console.Write("m = ");
            int m = GetIntegerNumber();

            Console.WriteLine($"Сумма делителей {n} =
                {SumOfDivisors(n)}");
            Console.WriteLine($"Сумма делителей {m} =
                {SumOfDivisors(m)}");

            if (IsAmicable(n, m))
                Console.WriteLine("Дружественные");
            else
                Console.WriteLine("Не дружественные");
        }
```

```
        static bool IsAmicable(int n, int m)
        {
            return (n != m) &&
                   (n == SumOfDivisors(m)) &&
                   (m == SumOfDivisors(n));
        }

        static int SumOfDivisors(int n)
        {
            int sum = 1;
            int divisor = 2;

            while (divisor * divisor <= n)
            {
                if (n % divisor == 0)
                    sum += divisor + n / divisor;

                divisor++;
            }

            return sum;
        }

        static int GetIntegerNumber()
        {
            return int.TryParse(Console.ReadLine(),
                out int number) ? number : 1;
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
n = 220
m = 284
Сумма делителей 220 = 284
Сумма делителей 284 = 220
```

Рис. 2.126. Проверка чисел на дружественность

Комментарии к реализации

Метод `IsAmicable()` возвращает логическое значение `true`, если заданные числа оказываются дружественными, иначе `false`. Его удобно использовать в условной проверке как признак.

Согласно определению дружественных числе одновременно требуется, чтобы сумма делителей первого была равна второму числу и наоборот. Кроме того, исключаем случай проверки одинаковых чисел.

Для подсчета суммы делителей в методе `SumOfDivisors()` изначально включаем в сумму делителей 1. Перебор и проверку возможных делителей начинаем с 2.

Если `divisor` является делителем, то добавляем его в сумму вместе с парным делителем. В этом случае перебирать возможные делители достаточно, пока `divisor` $\leq \sqrt{n}$. Чтобы не вычислять корень, возведем обе части неравенства в квадрат.

Обратите внимание: в методе `IsAmicable()`не требуется использовать конструкцию `if-else` или тернарный оператор. Выражение может быть либо истинным, либо ложным, поэтому его можно сразу передать оператору `return`.

2.10.4. Переменное число аргументов

Массив как параметр метода

Часто требуется создавать методы, которые должны работать с любым числом связанных параметров. Например, требуется метод, возвращающий сумму заданных чисел. В зависимости от ситуации может потребоваться искать сумму от 2-х, 3-х или более чисел.

Одно из решений – в качестве параметра передавать методу массив чисел.

Глава 2\ArrayParameters\Program.cs

```
using System;

namespace ArrayParameters
{
    class Program
    {
        static void Main(string[] args)
        {
            double[] array = [5.6, 9.03, 6];

            double sum1 = GetSum(array);
            double sum2 = GetSum([5.6, 9.03, 6]);
```

```
            Console.WriteLine($"Сумма элементов массива:
                {sum1}");
            Console.WriteLine($"Сумма элементов массива:
                {sum2}");
        }

        // Метод, возвращающий сумму элементов заданного
        // массива numbers
        static double GetSum(double[] numbers)
        {
            double sum = 0;

            foreach (var x in numbers)
                sum += x;

            return sum;
        }
    }
}
```

Представленный способ передачи нескольких значений через массив рабочий, но не очень удобен, поскольку потребуется предварительно описывать массив, или создавать его «на лету».

Модификатор params

C# поддерживает модификатор **params**, который описывает специальный параметр метода, способный принимать переменное число аргументов. Тип этого параметра должен быть одномерным массивом.

В модифицированном ниже коде в параметр `numbers` передается любое количество аргументов.

Глава 2\ParamsModifier\Program.cs

```
using System;

namespace ParamsModifier
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(GetSum());
```

```
            Console.WriteLine(GetSum(6));
            Console.WriteLine(GetSum(3, 8));
            Console.WriteLine(GetSum(5, 0, 1, -2, 7, 56));
        }

        // Метод, возвращающий сумму элементов заданного
        // массива numbers
        static double GetSum(params double[] numbers)
        {
            double sum = 0;

            foreach (var x in numbers)
                sum += x;

            return sum;
        }
    }
}
```

В отличие от первоначальной реализации, при вызове аргументы передаются как отдельные параметры. Допустимо даже их отсутствие. Вне зависимости от числа переданных аргументов они формируются в единый массив.

2.10.5. Операции с параметрами

Необязательные параметры

Методы C# позволяют использовать **необязательные параметры** или **параметры по умолчанию**. Это означает, что при вызове метода некоторые аргументы могут быть не указаны и в этом случае метод сам задает эти значения.

Любому необязательному параметру необходимо задать значение по умолчанию с помощью оператора присваивания.

В следующем примере метод `IntelliDiv()` в зависимости от необязательного параметра `rounded` осуществляет деление с округлением или без.

Глава 2\DefaultParameters\Program.cs

```
using System;

namespace DefaultParameters
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Без округления: " +
                IntelliDiv(23, 6));
            Console.WriteLine("С округлением: " +
                IntelliDiv(23, 6, true));
        }

        static double IntelliDiv(
            double a, double b, bool rounded = false
        )
        {
            return rounded ? Math.Round(a / b) : a / b;
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Без округления: 3,8333333333333335
С округлением: 4
```

Рис. 2.127. Использование параметров по умолчанию

По умолчанию третий необязательный параметр принимает значение `false`, т.е. если его не указать при вызове явно, то возвращается частное без округления. Но если явно указать значение `true`, то выполняется деление с округлением.

Именованные параметры

При вызове метода можно явно указывать название параметров. Для этого пишут название параметра, как оно обозначено в описании метода и через двоеточие указывают значение.

У такого подхода есть два достоинства:

1. Если явно указать имена всем параметрам, то их можно перечислить в любом порядке.
2. Явно указывать имена параметрам удобно для специфических методов, которые имеют много параметров и уточнение их имени делает вызов более информативным.

Так приведенном выше проекте *DefaultParameters* роль третьего параметра в методе неочевидна и потребуется изучать код метода. Если передать его с указанием имени, то уже интуитивно понятно, что дополнительно осуществляется округление результата.

Глава 2\NamedParameters\Program.cs

```
using System;

namespace DefaultParameters
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Без округления: " +
                IntelliDiv(a: 23, b: 6));
            Console.WriteLine("Без округления: " +
                IntelliDiv(b: 6, a: 23));

            Console.WriteLine("С округлением: " +
                IntelliDiv(23, 6, rounded: true));
            Console.WriteLine("С округлением: " +
                IntelliDiv(a: 23, b: 6, true));
            Console.WriteLine("С округлением: " +
                IntelliDiv(a: 23, b: 6, rounded: true));
            Console.WriteLine("С округлением: " +
                IntelliDiv(b: 6, a: 23, rounded: true));
            Console.WriteLine("С округлением: " +
                IntelliDiv(rounded: true, b: 6, a: 23));
        }

        static double IntelliDiv(
            double a, double b, bool rounded = false
        )
        {
            return rounded ? Math.Round(a / b) : a / b;
        }
    }
}
```

В приведенной модификации параметры можно передавать в разном порядке и чтение кода заметно упрощается.

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Без округления: 3,8333333333333335
Без округления: 3,8333333333333335
С округлением: 4
С округлением: 4
С округлением: 4
С округлением: 4
С округлением: 4
```

Рис. 2.128. Использование именованных параметров

2.10.6. Перегрузка метода

Перегрузка как элемент полиморфизма

В ООП поведение метода часто требует вариации в зависимости от числа параметров, их типа и др.

Например, методы `Write()` и `WriteLine()` из класса `Console` могут принимать в качестве аргументов объекты разных типов. И в каждом случае C# корректно определяет алгоритм их обработки и вывода.

Так, если открыть описании класса `Console` в библиотеке .NET методу `WriteLine()` задано 19 различных определений, которые отличаются количеством параметров и описанием. На самом деле каждому заголовку метода соответствует своя программная реализация.

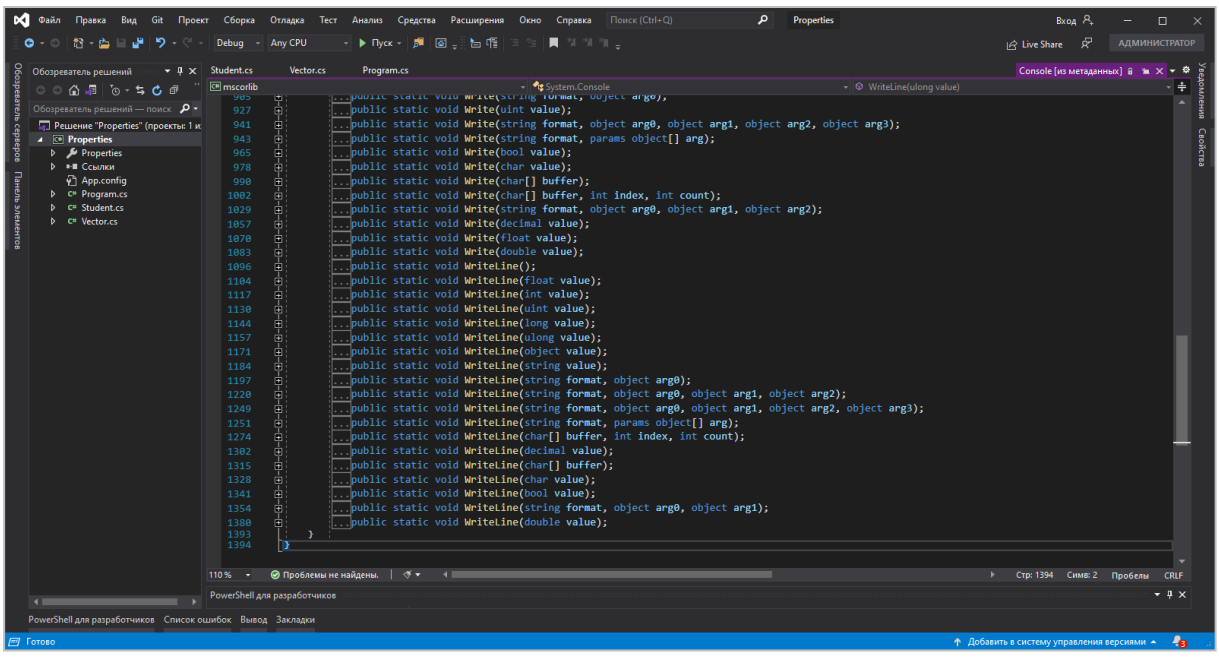


Рис. 2.129. Варианты описания заголовков методов `Write()` и `WriteLine()`

Если не брать во внимание название параметров, а только их типы, порядок следования, число и ключевые модификаторы в определении метода, то это называют **сигнатурой метода**.

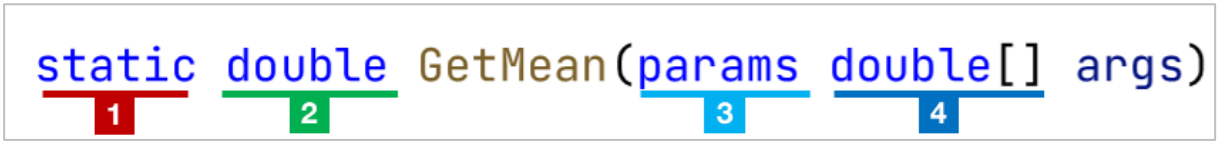


Рис. 2.130. Компоненты сигнатуры метода

Возможность методов принимать различную реализацию в зависимости от сигнатуры называется **перегрузкой методов**. Перегрузка относится к одному из трех принципов ООП – **полиморфизму**, т.е. возможности одного объекта принимать разные формы и реализацию.

При вызове метода компилятор анализирует его сигнатуру и выбирает подходящую перегрузку.

Практический пример перегрузки методов

Пусть требуется создать перегруженный метод, который может вычислять длину вектора в зависимости от размерности пространства, т.е. числа координат.

Глава 2\MethodOverload\Program.cs

```
using System;

namespace MethodOverload
{
    class Program
    {
        // Далее
        static void Main(string[] args)
        {
            Console.WriteLine($"V1: {GetVectorLength(2)}");
            Console.WriteLine($"V2: {GetVectorLength(4, 7)}");
            Console.WriteLine($"V3: {GetVectorLength(9, 0,
                3.5)}");
        }

        static double GetVectorLength(double x)
        {
            return Math.Abs(x);
        }
```

```
        static double GetVectorLength(double x, double y)
        {
            return Math.Sqrt(x * x + y * y);
        }

        static double GetVectorLength(double x, double y,
            douuble z)
        {
           return Math.Sqrt(x * x + y * y + z * z);
        }
    }
}
```

Метод `GetVectorLength()` имеет три перегрузки, т.е. три отдельных реализации в зависимости от числа координат.

В этом примере сигнатуры метода отличаются только количеством параметров.

Компилятор при вызове метода определяет подходящую сигнатуру и соответствующую ей реализацию.

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
V1: 2
V2: 8,06225774829855
V3: 9,656603957913983
```

Рис. 2.131. В зависимости от количества параметров как числа координат длина вектора вычисляется по соответствующей размерности пространства формуле

Важность перегрузки методов

Если бы C# не поддерживал перегрузку методов, то потребовалось бы писать три метода с разным названием, каждый на определенное число параметров. Такой подход как раз и использовался в языках программирования до реализации подобного механизма полиморфизма.

Например, для языка C отдельно предусматривались функции `abs(x)` и `fabs(x)`, вычисляющие модуль значения для целого и вещественного числа `x` соответственно. Подобное разнообразие имен для родственных функций может вносить трудности при сопровождении кода.

2.10.7. Оператор тела выражения

Оператор =>

Начиная с версии C# 6.0 в язык добавлена возможность определения **тела выражения** с использованием оператора `=>`. Он позволяет упростить определение коротких методов, которые задаются одной инструкцией.

Следующее описание метода:

```
static double GetVectorLength(double x)
{
    return Math.Abs(x);
}
```

может быть свернуто в более короткую форму:

```
static double GetVectorLength(double x) => Math.Abs(x);
```

В этом случае не нужны фигурные скобки тела метода и оператор `return`, оператор тела выражения их заменяет.

Такие функции часто неофициально называют **стрелочными**.

Пример использования оператора тела выражения

Оператор тела выражения может существенно упростить описание функций в проекте *MethodOverload*.

Глава 2\RowOperator\Program.cs

```
using System;

namespace MethodOverload
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine($"V1: {GetVectorLength(2)}");
            Console.WriteLine($"V2: {GetVectorLength(4, 7)}");
            Console.WriteLine($"V3: {GetVectorLength(9, 0,
                3.5)}");
        }

        static double GetVectorLength(double x) =>
            Math.Abs(x);
```

```
        static double GetVectorLength(double x, double y) =>
            Math.Sqrt(x * x + y * y);

        static double GetVectorLength(double x, double y,
            double z) =>
            Math.Sqrt(x * x + y * y + z * z);
    }
}
```

2.10.8. Рекурсивный вызов методов

Рекурсивные функции

Рекурсивная функция – функция, которая в своем определении обращается к самой себе.

Рекурсивные функции тесно связаны с математической моделью **рекуррентных соотношений**, где для вычисления (поиска) n-го члена последовательности необходимо знать значение предыдущего(их).

Простым примером рекуррентного соотношения является формула для вычисления арифметической или геометрической прогрессии:

$$a_n = a_{n-1} + h \qquad\qquad b_n = b_{n-1} \cdot h$$

Для рекуррентных соотношений важно определить начальные условия, иначе этот процесс потенциально бесконечен. Например, для обозначенных прогрессий стартовым значением является некоторое a_0 и b_0.

Из рекуррентного соотношения нетрудно определить и рекурсивную функцию (здесь n понимается уже не в смысле индекса, а аргумента функции):

$$f(n) \coloneqq f(n-1) + h,\ \ f(0) = a_0,$$
$$f(n) \coloneqq f(n-1) \cdot h,\ \ f(0) = b_0.$$

При использовании циклического алгоритма обработки рекуррентное соотношение обычно удобнее разворачивать в «естественном порядке», вычисляя от начального значения(ий) к конечному(ым).

Для арифметической прогрессии вычисления в цикле выглядят следующим образом:

$$a_0$$
$$a_1 = a_0 + h$$
$$a_2 = a_1 + h$$
$$\dots$$
$$a_n = a_{n-1} + h$$

Для геометрической прогрессии:

$$b_0$$
$$b_1 = b_0 \cdot h$$
$$b_2 = b_1 \cdot h$$
$$\dots$$
$$b_n = b_{n-1} \cdot h$$

Для рекурсивного вызова принцип работы обратный: в начале рекурсия накапливает конечные значения до тех пор, пока вложенные вызовы функции не достигают начального значения. Далее погружение в рекурсию останавливается и осуществляется вычисление по сохраненным значениям.

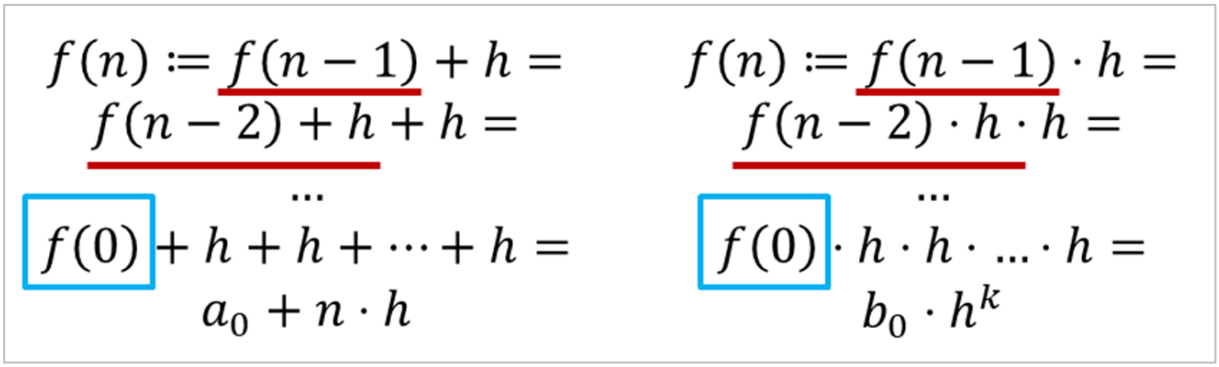


Рис. 2.132. Принцип погружения в рекурсию

Рекурсия иногда дает более простой и прозрачный алгоритм реализации задачи, которая может быть решена итерационно (с помощью циклов).

Пример с факториалом

Пусть требуется вычислить факториал натурального числа n с использованием цикла и рекурсии:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n.$$

Каждый из алгоритмов вычисления реализовать отдельными функциями.

Решение

Введем функцию $F(n) = n!$

Вычисление факториала через цикл тривиально: достаточно организовать цикл домножения переменной (изначально равна 1) на числа от 1 до n. Действительно, согласно определению факториала:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \ (n-1) \cdot n = [1 \cdot 2 \cdot 3 \cdot \ldots \ (n-1)] \cdot n = (n-1)! \cdot n,$$

откуда следует рекуррентное соотношение:

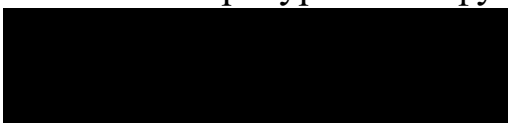
$$F_n = F_{n-1} \cdot n, \ F_0 = 1.$$

Переходя в терминологию рекурсивных функций, задаем целочисленную функцию $F(n)$:

$$F(n) = F(n-1) \cdot n.$$

С учетом начального условия считают $F(0) = 1$.

Тогда формально описание рекурсивной функции имеет вид:



Опишем две функции:

- `IterFactorial()` – возвращает факториал числа, используя вычисления в цикле;
- `RecFactorial()` – возвращает факториал числа, используя рекурсивный вызов.

Глава 2\RecursionFactorial\Program.cs

```
using System;

using System;

namespace RecursionFactorial
{
    class Program
    {
        static void Main(string[] args)
        {
            int n = 8;
            Console.WriteLine($"{n}! = {IterFactorial(n)}");
            Console.WriteLine($"{n}! = {RecFactorial(n)}");
        }

        static long IterFactorial(int n)
        {
```

```
            long factor = 1;

            for (int i = 1; i <= n; i++)
                factor *= i;

            return factor;
        }

        static long RecFactorial(int n)
        {
            if (n == 0)
                return 1;

            return RecFactorial(n - 1) * n;
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
8! = 40320
8! = 40320
```

Рис. 2.133. Итерационный и рекурсивный способы вычисления факториала

Комментарии к реализации

Итерационный алгоритм вычисления ранее был разобран в проекте *FactorialFunction*. Рекурсивное вычисление предполагает следование заданной функции:

- если значение параметра равно нулю, достигнуто дно рекурсии и осуществляется возврат 1;
- иначе идет погружение в рекурсию: функция вызывает себя для вычисления значения от предыдущего аргумента, а текущий множитель n записывается в стек.

Когда функция достигает «дна», возвращается значение 1, которое последовательно перемножается на сохраненные ранее множители в виде переданных параметров («всплытие» рекурсии).

Реализацию рекурсивной функции можно сократить, если воспользоваться тернарным оператором:

```
static long RecFactorial(int n) =>
    n == 0 ? 1 : RecFactorial(n - 1) * n;
```

2.10.9. Пример объединения методов в классы

Постановка задачи

Требуется реализовать проект *MatrixProcessing*, способный осуществлять преобразования над матрицами.

Требования к архитектуре кода:

- Описать класс `MatrixOperations`, реализующий методы:
  - `InputMatrix(n,m)` возвращает матрицу вещественных чисел, элементы которой вводят с клавиатуры (с подсказками индексов);
  - `PrintMatrix(matrix)` выводит матрицу в форме таблицы;
  - `GetMatrixSum(matrix)` возвращает сумму элементов матрицы.
- Описать класс `ConsoleOperations`, реализующий методы:
  - `GetIntegerInput()` – возвращает введенное целое положительное число, а в случае некорректного ввода – 0;
  - `GetDoubleInput()` – возвращает введенное вещественное число, а в случае некорректного ввода – 0.
- В основной части программы продемонстрировать создание матрицы и использование описанных методов ее обработки.

Реализация

Для реализации проекта потребуется три файла:

- `Program.cs` – основной файл с кодом программы.
- `MatrixOperations.cs` – файл с классом, описывающим операции с матрицами.
- `ConsoleOperations.cs` – файл с классом, описывающим некоторые методы ввода данных.

Чтобы упростить проект, все классы опишем в едином пространстве имен `MatrixProcessing`.

Начнем с реализации класса `ConsoleOperations`, поскольку его методы необходимы для ввода данных в матрицу. Оба метода сделаем статическими, поскольку они будут осуществлять операции с объектом консоли и нет необходимости создавать для этой цели разные эк-

земпляры класса. Сам класс в этом случае тоже обозначим статическим (чтобы не было смысла создавать на его основе экземпляры).

Глава 2\MatrixProcessing\ConsoleOperations.cs

```
namespace MatrixProcessing
{
    public static class ConsoleOperations
    {
        public static int GetIntegerInput()
        {
            return int.TryParse(Console.ReadLine(),
                out int number) && number > 0 ? number : 0;
        }

        public static double GetDoubleInput()
        {
            return double.TryParse(Console.ReadLine(),
                out double number) ? number : 0;
        }
    }
}
```

Следующим шагом реализуем класс для работы с матрицами. Постараемся учесть в реализации нежелательные случаи (некорректно заданные длины размерностей, пустые матрицы, некорректный ввод коэффициентов):

Глава 2\MatrixProcessing\MatrixOperations.cs

```
namespace MatrixProcessing
{
    public static class MatrixOperations
    {
        public static double[,] InputMatrix(int n, int m)
        {
            if (n <= 0 || m <= 0)
                throw new ArgumentException(
                    "Размерности матрицы должны быть >= 1"
                );

            Console.WriteLine($"Введите элементы матрицы
                {n}x{m}");
            double[,] matrix = new double[n,m];

            for (int i = 0; i < n; i++)
```

```
            {
                for (int j = 0; j < m; j++)
                {
                    Console.Write($"M[{i};{j}] = ");
                    matrix[i,j] =
                        ConsoleOperations.GetDoubleInput();
                }
            }

            return matrix;
        }

        public static void PrintMatrix(double[,] matrix)
        {
            if (matrix == null)
                return;

            Console.WriteLine();

            for (int i = 0; i < matrix.GetLength(0); i++)
            {
                for (int j = 0; j < matrix.GetLength(1); j++)
                    Console.Write($"{matrix[i,j],10:F3}");

                Console.WriteLine();
            }

            Console.WriteLine();
        }

        public static double GetMatrixSum(double[,] matrix)
        {
            if (matrix == null)
                return 0;

            double sum = 0;

            foreach (var elem in matrix)
                sum += elem;

            return sum;
        }
    }
}
```

Наконец, в основной части программы создадим матрицу, заполним ее значения с клавиатуры, выведем на экран матрицу в форме таблицы и вычислим сумму элементов матрицы.

Глава 2\MatrixProcessing\Program.cs

```
using System;

namespace MatrixProcessing
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Введите размеры матрицы:");
            Console.Write("n = ");
            int n = ConsoleOperations.GetIntegerInput();
            Console.Write("m = ");
            int m = ConsoleOperations.GetIntegerInput();

            var matrix = MatrixOperations.InputMatrix(n,m);

            MatrixOperations.PrintMatrix(matrix);
            Console.WriteLine($"Сумма элементов матрицы равна
                {MatrixOperations.GetMatrixSum(matrix)}");
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Program> dotnet run
Введите размеры матрицы:
n = 3
m = 5
Введите элементы матрицы 3x5
M[0;0] = -567
M[0;1] = 2,67
M[0;2] = 1000
M[0;3] = 6
M[0;4] = 0
M[1;0] = 1
M[1;1] = 0
M[1;2] = 20,04
M[1;3] = -23
M[1;4] = 67
M[2;0] = 900
M[2;1] = 234,1
M[2;2] = 0,03
M[2;3] = 12
M[2;4] = 67

  -567,000     2,670  1000,000     6,000     0,000
     1,000     0,000    20,040   -23,000    67,000
   900,000   234,100     0,030    12,000    67,000

Сумма элементов матрицы равна 1719,84
```

Рис. 2.134. Ввод матрицы и вывод результатов вычислений

2.10.10. Рекомендации учителю

Методы как подпрограммы внутри класса

- Изучение методов важно начать с конкретных примеров ситуаций, когда требуется повторное использование кода при разных искомых параметрах.
- Далее формулируется понятие подпрограммы как модуля, решающего локальную задачу обработки или преобразования данных.
- Необходимо четко разделить методы на функции с возвратом и функции без возврата значения, привести простые, но ярко выраженные примеры обоих типов методов. Укажите на роль модификатора `static` (на этом этапе курса пока не стоит подробно акцентировать внимание на нем).
- Объяснение, почему важно задавать наделенное смыслом название метода.
- Дополнительно можете обсудить с учащимися, как передаются параметры по значению и по ссылке (`ref`, `out`).
- Проработайте типовые алгоритмы на описание функций (сумма элементов, среднее арифметическое, факториал, передача в параметрах массивов и т.д.).
- Приведите примеры задач, где рационально вызывать одну функцию внутри другой.

Рекурсивные функции

- Изучение рекурсии следует начать с простого примера, на котором доступно иллюстрируется разница между циклическим вычислением и вычислением с помощью рекурсии (например – факториал числа).
- Продемонстрируйте, как работает стек (заранее подготовьте иллюстративный или анимационный демонстрационный материал).
- В простых задачах покажите, что они могут быть решены как итерационно, так и рекурсивно.
- В качестве примеров покажите, что некоторые задачи ЕГЭ могут решаться с привлечением рекурсии.

Вопросы для самопроверки

1. Для каких целей в программный код вводят подпрограммы?
2. Опишите синтаксис метода на конкретном примере.
3. Для чего используются модификаторы доступа?
4. Какие типы значений может возвращать метод? В каком случае метод можно назвать функцией или процедурой?
5. В чем отличие статического метода от метода экземпляра?
6. Чем удобен модификатор `params`?
7. Почему рационально задавать параметрам значения по умолчанию?
8. В каком случае указывать имена аргументов может оказаться полезным?
9. Что представляет собой перегрузка метода и как она связана с его сигнатурой?
10. Опишите преимущества использования оператора тела выражения.
11. Что представляют собой рекурсивные функции?
12. Каким образом реализовать рекурсивный вызов функции.

Практикум

1. Общие задачи по методам

Задание 1

1. Создайте проект *Combinatorics*.
2. Скопируйте код и повторите рассмотренного проекта в текущем занятии.
3. Доработайте его таким образом, чтобы:
   a. метод `Factorial()` в случает отрицательного аргумента считал факториал от противоположного (т.е. положительного) по знаку числа;
   b. методы `Combo()` и `Accomodation()` следили, чтобы $n \geq m$; в противном случае считать $n = m$.

Задание 2*

1. Создайте проект *CombinatoricsTables*.
2. Предварительно повторите и внимательно изучите реализованный в конце занятия проект *MatrixProcessing*.
3. Перенесите описание всех трех ранее созданных методов в static-класс *Combinatorica*, опишите его в отдельном файле. Каждому методу сделайте открытый доступ извне с помощью модификатора `public`.
4. Используя созданные методы, в основной части программы вывести на экран:
   a. факториалы чисел от 0 до n;
   b. «треугольник сочетаний» (треугольник Паскаля, $0 \leq m \leq n$);
   c. «треугольник размещений» ($1 \leq m \leq n$).
5. Внимательно изучите его структуру кода проекта и самостоятельно реализуйте.

```
Введите n: 6

Таблица факториалов
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720

Треугольник Паскаля
1
1       1
1       2       1
1       3       3       1
1       4       6       4       1
1       5       10      10      5       1
1       6       15      20      15      6       1

Треугольник размещений
1
2       2
3       6       6
4       12      24      24
5       20      60      120     120
6       30      120     360     720     720
```

Рис. 2.135. Результат работы программы из задания 2

Задание 3

1. Создайте проект *OddEven*.
2. Написать методы логического типа `IsEven(n)` и `IsOdd(n)`, проверяющие, является ли заданное число четным и нечетным соответственно.
3. Сократите их синтаксис, используя возможности языка C#.

```
PS D:\Documents\Programming\NET\Program> dotnet run
Введите n: 45845
45845 - нечетное
PS D:\Documents\Programming\NET\Program> dotnet run
Введите n: 58234
58234 - четное
```

Рис. 2.136. Результат работы программы из задания 3

Задание 4

1. Создайте проект *PrimeNumbers*.
2. Напишите логическую функцию `IsPrime(n)`, проверяющую заданное натуральное число $\boldsymbol{n} \geq 2$ на простоту.
3. Используя эту функцию, вывести на экран все простые числа в пределах от 2 до 1000.
4. Используйте следующий код:

Глава 2\PrimeNumbers\Program.cs

```
using System;

namespace PrimeNumbers
{
    class Program
    {
        static void Main(string[] args)
        {
            const int m = 1000;

            for (int i = 2; i <= m; i++)
            {
                // Проверяем с помощью функции
                if (IsPrimeNumber(i))
                    Console.WriteLine(i);
            }
        }
```

```
        static bool IsPrimeNumber(int n)
        {
            // Предположим, что простое
            bool result = true;

            // Здесь код проверки числа n на простоту

            return result;
        }
    }
}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
2
3
5
7
11
13
17
```

Рис. 2.137. Результат работы программы из задания 4

Задание 5

1. Создайте проект *MeanValue*.
2. Определите метод `GetMean()`, вычисляющий среднее значение среди заданных чисел. Метод должен принимать любое количество параметров (чисел) и корректно работать вне зависимости от их числа (используйте модификатор `params`).
3. Предусмотреть случай, когда параметры не передаются. В этом случае должен возвращаться нуль.

```
Console.WriteLine(GetMean());
Console.WriteLine(GetMean(5));
Console.WriteLine(GetMean(3.6, 2, -1));
Console.WriteLine(GetMean(4, 8, 11, 23, -2, 3.4));
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
0
5
1,5333333333333332
7,8999999999999995
```

Рис. 2.138. Результат работы программы из задания 5

Задание 6

1. Создайте проект *MatrixProcessing*.
2. Скопируйте рассмотренный в занятии пример, отладьте и протестируйте его.

Задание 7

1. Создайте проект *NuberTypeOverload*.
2. Требуется описать три перегрузки функции, которая возвращает название типа числа в зависимости от переданного параметра. Параметр может быть задан целым числом, вещественным числом и строкой. Для третьего случая ,если строку нельзя сконвертировать в число, возвратить запись «NULL».
3. Протестируйте код на следующих вызовах метода:

```
Console.WriteLine(NumberType(5));
Console.WriteLine(NumberType(0));
Console.WriteLine(NumberType(-4) + "\n");

Console.WriteLine(NumberType(5.4));
Console.WriteLine(NumberType(0));
Console.WriteLine(NumberType(-2.4) + "\n");

Console.WriteLine(NumberType("23"));
Console.WriteLine(NumberType("0"));
Console.WriteLine(NumberType("-45,5"));
Console.WriteLine(NumberType("-45.5"));
Console.WriteLine(NumberType("34gf"));
Console.WriteLine(NumberType(""));
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
положительное
нулевое
отрицательное

положительное
нулевое
отрицательное

положительное
нулевое
отрицательное
NULL
NULL
NULL
```

Рис. 2.139. Результат работы программы из задания 7

2. Задачи по рекурсии

Задание 1

1. Создайте проект *RecursionFactorial*.
2. Скопируйте рассмотренный в занятии одноименный пример, отладьте и протестируйте его.

Задание 2

1. Создайте проект *GCD*.
2. Напишите рекурсивную функцию для вычисления НОД двух заданных чисел, используя следующую разностную формулу:

$$GCD(n,m) = \begin{cases} n, & \text{если } n = m, \\ GCD(n-m,m), & \text{если } n > m, \\ GCD(n,m-n), & \text{если } m > n. \end{cases}$$

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
НОД(20400,53400) = 600
```

Рис. 2.140. Результат работы программы из задания 2

Задание 3

1. Создайте проект *Fibonacci*.
2. Написать рекурсивную функцию вычисления ***n***-го числа Фибоначчи. Рядом Фибоначчи называют последовательность чисел, где каждое последующее равно сумме двух предыдущих. Первые два числа считают равными по 1.

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Номер числа = 1
Fib(1) = 1
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Номер числа = 2
Fib(2) = 1
PS D:\Documents\Programming\NET\Manual2025> dotnet run
Номер числа = 10
Fib(10) = 55
```

Рис. 2.141. Результат работы программы из задания 3

Задание 4*

1. Создайте проект *RecursiveProgressionSum*.
2. Написать рекурсивную функцию для вычисления частичной суммы членов последовательности (при $\boldsymbol{n} \geq 1$):

$$S(n) = 1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \cdots + \frac{1}{n^2}.$$

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
n = 10
S(10) = 1,5497677311665408
```

Рис. 2.142. Результат работы программы из задания 4

Задание 5*

1. Создайте проект *RecursiveSum*.
2. Вычислить сумму ряда через рекурсию при заданном $\boldsymbol{x}$ и $\boldsymbol{n}$:

$$S(x,n) = 1 + \frac{1}{x} + \frac{1}{x^2} + \ldots + \frac{1}{x^n}$$

3. Используйте формулу:

$$\boldsymbol{S(x,n) = \begin{cases} 1, \ n = 0, \\ S(x,n-1) + \dfrac{1}{x^n} \end{cases}}$$

4. Для оптимизации расчетов учтите зависимость между соседними членами ряда:

$$\frac{a_n}{a_{n-1}} = \frac{1}{x}, \quad a_n = \frac{1}{x} a_{n-1}, \ a_{n-1} = a_n \cdot x.$$

$$S(x, a_n, n) = S(x, a_{n-1}, n-1) + a_n, \ S(x, a_0, 0) = 1,$$

5. Опишите две функции: основная `RecSum(x,n)` вызывает (и по существу запускает) вспомогательную `_RecSum(x,an,n)`, которая и вычисляется рекурсивно:

```
static double _RecSum(double x, double an, int n) {}
static double RecSum(double x, int n) {}
```

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
x = 2,45
n = 10
S(2,45;10) = 1,689566666731561
```

Рис. 2.143. Результат работы программы из задания 5

Глава 3. КЛАССЫ И ИХ КОМПОНЕНТЫ

3.1. Классы как составные типы данных

3.1.1. Понятие класса

Необходимость использования классов

В простых проектах, которые были продемонстрированы в текущем курсе ранее, в большинстве случаев было достаточным работать с простыми встроенными типами данных: числовыми, логическими, символьным, строковым. Однако реальные практических задачи предполагают более сложное взаимодействие объектов, поэтому данные требуется структурировать и объединять с логикой обработки.

Например, если мы пишем программу обработки данных сотрудников на предприятии, то для каждого работника потребуется описывать множество атрибутов: ФИО, возраст, должность, стаж и т.п. Кроме того, необходимо осуществлять различные операции: вычислять заработную плату, полагающиеся премии и надбавки, описывать характеристику работника по показателям производительности труда, а также управлять учёными записями о каждом работнике. Т.е. данные и логика, как правило, взаимосвязаны.

Именно поэтому реализация современных программ требует моделировать объекты окружающего мира, их поведение и взаимодействие. **Объекты** – одно из важнейших понятий **объектно-ориентированного программирования** (ООП).

Объекты в программировании характеризуются **состоянием** и **поведением**:

- **Состояние** – это данные, которые объект хранит в определенный момент времени. Эти данные характеризуют свойства объекта и его компонент.
- **Поведение** – это действия, которые объект может выполнять для управления своим состоянием. Результатом пове-

дения может являться последовательность операций, преобразований данных объекта или извлечение новых данных.

В C# объекты реализованы на базе **классов**. Каждый класс нужно рассматривать как отдельный **тип данных**, наряду со всеми остальными типами.

Класс как шаблон структуры

Определение

Класс *– это шаблон, по которому определяется форма объекта. В нем указываются данные и код, который будет оперировать этими данными.*

Класс необходимо представлять в форме шаблона или схемы устройства объекта. Класс является **логической абстракцией**, которая определяет **структуру** объекта.

Например, определим класс `Student` для описания студентов. Каждого студента можно охарактеризовать рядом атрибутов, например: ФИО, возраст, курс, группа, уникальный идентификатор, характеристика студента (разумеется, можно задать и множество других характеристик, в зависимости от поставленной задачи).

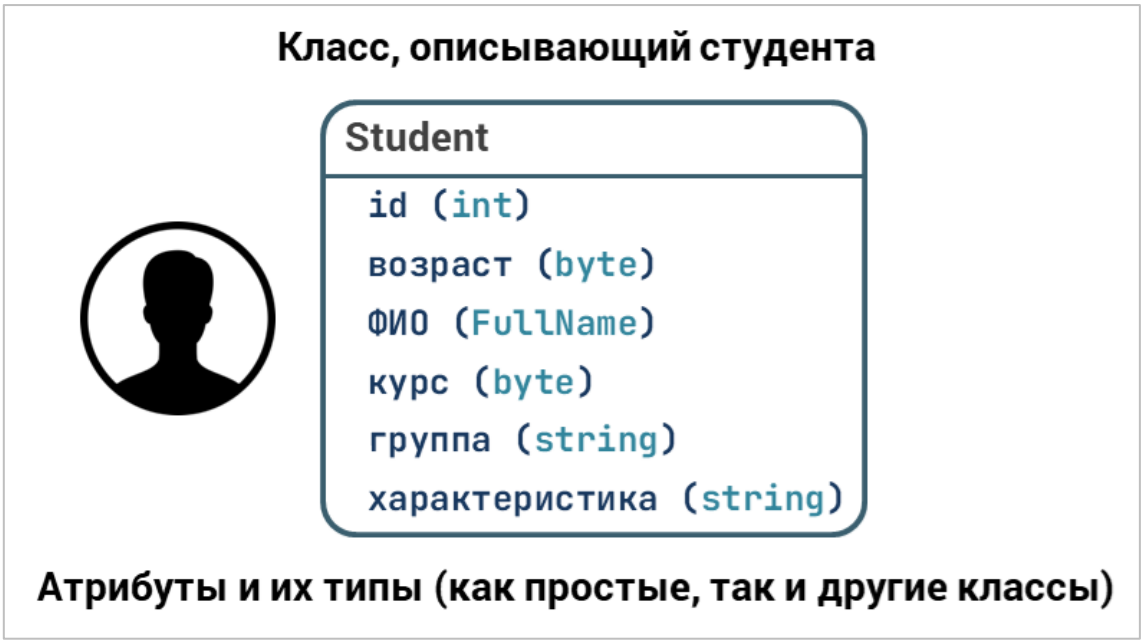


Рис. 3.1. Класс как шаблон описания объектов

Каждая из характеристик имеет определенный тип данных: это может быть как простой базовый тип, так и некоторый другой класс. Т.е. класс можно считать **составным типом данных**.

Именно поэтому классы C# и являются составными типами данных: они объединяют разные виды данных и функциональных компонент в единый тип.

Объект как экземпляр класса

Определение

Объект *представляет собой* ***экземпляр класса****, а сам класс – описание структуры объекта.*

Объект класса получает структуру согласно описанию класса.

Можно создавать множество объектов одного класса: они будут иметь одинаковую структуру, но разные значения состояний. При этом все создаваемые на базе класса объекты изначально взаимно независимы.

Рис. 3.2. Экземпляр `denis` касса `Student`

На рис. 3.2 приведен пример создания экземпляра `denis` класса `Student`. Видно также, что один из атрибутов (ФИО) описан в качестве составной сущности (`FullName`, рис. 3.2), также являющейся классом.

Понятие объекта и экземпляра класса часто отождествляют, что вполне допустимо. В дальнейшем мы также будем использовать оба термина. Однако предполагаем, что понятие объекта более широкое и прежде всего – это связанные в сущность данные, которые записываются в оперативную память. В то время как экземпляр класса – это конкретно созданный элемент программного кода в виде переменной, через которую мы будем ссылаться на этот объект.

Программирование с классами

Парадигма ООП не вносит дополнительные возможности как таковые в программирование. Все, что можно реализовать в ООП, достигается и средствами «классического» процедурного программирования. ООП лишь определяет более совершенный уровень абстрагирования кода.

Работа с объектами и классами более естественна для человека и позволяет очевиднее организовывать взаимодействие компонент в программе, а также масштабировать архитектуру проекта под новые задачи.

Использовать архитектуру кода с классами удобнее по ряду причин:

- данные и методы для их обработки будут связаны с объектом, т.е. инкапсулированы в него (один из фундаментальных принципов ООП);
- на базе классов как типов данных можно создавать другие классы путем наследования и расширения их возможностей;
- на основе класса можно описывать любое число экземпляров; они будут иметь одинаковую архитектуру, но разные значения данных;
- повышение абстракции кода и более прозрачная модульная структура проектов.

3.1.2. Структурные компоненты класса

Данные и логика

Компоненты класса формально можно разделить на компоненты-данные и компоненты-функции.

1. **Компоненты-данные** – это переменные, которые, которые хранят данные о состоянии объекта.
2. **Компоненты функций** – это методы, которые обеспечивают функциональную обработку и манипулирование данными.

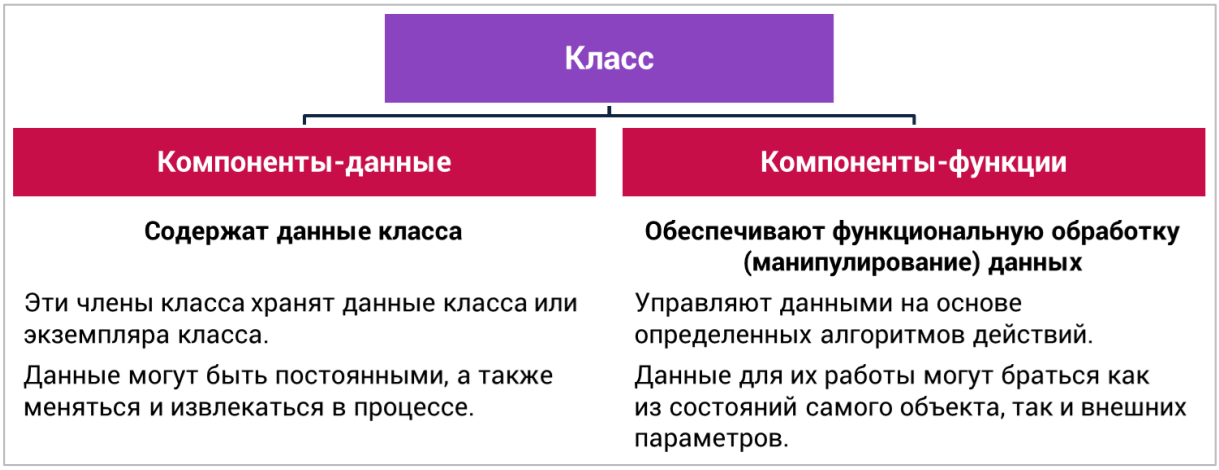


Рис. 3.3. Структурные компоненты класса

В классе C# доступны несколько разновидностей компонентов-данных и компонентов-функций. Класс может содержать одновременно всевозможные типы компонентов, а может лишь некоторые из них (класс даже может быть пустым).

Компоненты-данные

Контейнерами для данных класса являются **поля** – переменные, объявленные внутри класса. Поля хранят информацию о свойствах и состояниях объекта класса.

Поля могут ограничиваться уровнями доступа, которые позволяют с ними работать только в определенной области видимости.

Компоненты-функции

Для обработки данных в классе описываются **методы**. Они описывают логику взаимодействия с данными. Обычно методы управля-

ют содержимым полей, могут его менять или считывать. Также методы могут обращаться к другим методам класса.

Как и в случае полей, методы тоже имеют уровни доступа, что позволяет писать защищенный от нежелательных вмешательств код.

3.1.3. Описание класса

Общий синтаксис

Определение

Класс описывается следующим образом:

```
[доступ] [static] class ИмяКласса
{
    // элементы класса
}
```

Ключевая команда **`class`** определяет описание класса с заданным названием. Название принято писать с заглавной буквы.

Следующее описание задает шаблон класса `Student` с уровнем доступности этого класса в рамках текущего проекта:

```
class Student
{
    // Здесь описываются компоненты класса
}
```

Модификаторы доступа

Доступ задается модификатором доступа, который определяет область видимости класса относительно проекта. Среди часто используемых выделяют следующие:

- **`public`**: доступ к классу открыт в текущей и любых других сборках (проектах). К такому классу можно обращаться из других пространств имен, классов и проектов в рамках единого решения.
- **`internal`**: доступ к классу открыт только в текущей сборке. Класс будет доступен для других пространств имен и классов, но только в рамках текущей сборки. Подразумевается

по умолчанию, т.е. этот модификатор можно не указывать явно.

Кроме того, доступны следующие модификаторы:

- **`private`** – класс доступен только внутри другого класса, если он в него вложен (т.е. максимально закрыт);
- **`protected`** – класс будет доступен внутри класса, где описан, а также в наследующих его классах;
- **`protected internal`** – область видимости класса распространяется на сборку и классы наследники;
- **`private protected`** – класс будет доступен внутри сборки и для наследников этого класса (добавлен в C# 7.2).

Это полезно знать!

Перечисленные модификаторы также используются и для описания полей и методов класса.

Статичность класса

При изучении методов как функций (в п. 2.10.3) мы обращали внимание на специальный модификатор `static`, который привязывал метод непосредственно к классу, а не к экземплярам этого класса.

Классы тоже могут быть статическими. Далее будет показано, в каких случаях рационально описывать класс с модификатором `static`.

3.1.4. Размещение описания класса

Работа с классами предполагает модульный принцип: каждый класс описывается в отдельном файле, даже если он небольшой.

1. В одном файле с основным классом

В одном пространстве имен

Самый простой способ – описать класс внутри основного пространства имен:

Глава 3\ClassesStructure_1\Program.cs

```
using System;

namespace ClassesStructure
{
    class Program
    {
        static void Main(string[] args)
        {
            Student denis = new Student();
        }
    }

    class Student
    {

    }
}
```

Здесь новый класс описывается наравне с другими классами этого пространства имен. Описание класса `Student` можно разместить выше или ниже основного `Program` (для компилятора порядок непринципиален), но обязательно на одном уровне.

В методе `Main()` создаём один экземпляр этого класса, по аналогии с описанием переменной:

```
Student denis = new Student();
```

Однако таким образом объединять несколько классов в одном файле не следует.

В одном пространстве имен по отдельности

Одно и тоже пространство имен допускается «делить» на отдельные части:

Глава 3\ClassesStructure_2\Program.cs

```
using System;

namespace ClassesStructure
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            Student denis = new Student();
        }
    }
}

namespace ClassesStructure
{
    class Student
    {

    }
}
```

В этом примере осуществлено раздельное описание классов в одном и том же пространстве имен `ClassesStructure` и в одном файле. Технически это абсолютно аналогично предыдущему варианту.

В разных пространствах имен

Описание класса может содержаться и в другом пространстве имен:

Глава 3\ClassesStructure_3\Program.cs

```
using System;
using NewSpace;

namespace Classes_Structure
{
    class Program
    {
        static void Main(string[] args)
        {
            Student denis = new Student();
        }
    }
}

namespace NewSpace
{
    class Student
    {

    }
}
```

В этом случае не забудьте в начале подключить к нему короткий доступ директивой `using`. Иначе перед каждым вызовом класса извне потребуется уточнять его пространство имен:

```
NewSpace.Student denis = new NewSpace.Student();
```

или так:

```
var denis = new NewSpace.Student();
```

2. В отдельном файле

В рамках одного пространства имен

В следующем примере описание класса `Student` выносится в отдельный файл. Имя файла может быть любым, но желательно назвать аналогично классу, чтобы избежать путаницы.

Глава 3\ClassesStructure_4\Program.cs

```
using System;

namespace ClassesStructure
{
    class Program
    {
        static void Main(string[] args)
        {
            Student denis = new Student();
        }
    }
}
```

Глава 3\ClassesStructure_4\Student.cs

```
namespace ClassesStructure
{
    class Student
    {

    }
}
```

В основном классе обращение к классу `Student` возможно, поскольку он описан в том же пространстве имен `ClassesStructure`, что и `Program`. Несмотря на то, что код разбит по разным файлам, классы `Program` и `Student` находятся в единой логической области.

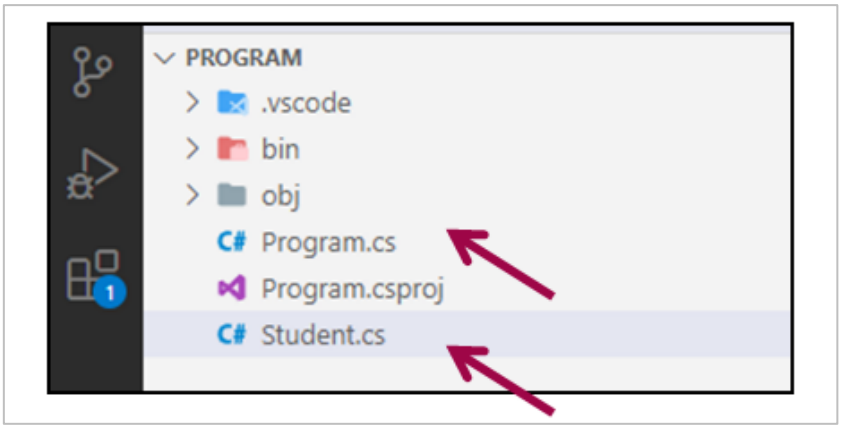


Рис. 3.4. Разбиение кода классов на отдельные файлы

В разных пространствах имен

В общем же случае новые классы создаются в отдельных пространствах имен. Для короткого обращения извне используйте `using` в основной части программы:

Глава 3\ClassesStructure_5\Program.cs

```
using System;
using NewSpace;

namespace Classes_Structure
{
    class Program
    {
        static void Main(string[] args)
        {
            Student denis = new Student();
        }
    }
}
```

Глава 3\ClassesStructure_5\Student.cs

```
namespace NewSpace
{
    class Student
    {

    }
}
```

Это полезно знать!

В отличие от описания обычной переменной, переменная класса описывается в следующем виде:

```
Student denis = new Student();
```

Название класса ***Student*** *играет роль типа данных. Правая часть отвечает за создание нового объекта в памяти.*

В современном C#-программировании левую часть обычно сокращают, заменяя название класса на неявную типизацию (компилятор выводит тип переменной по правой части):

```
var denis = new Student();
```

3.1.5. Модификаторы доступа к классу

Пример установки области видимости

Модификаторы доступа к классу позволяют ограничить его область доступности в по отношению к другим классам, сборке и правилам наследования.

Рассмотрим пример, в котором определим классы с разными модификаторами доступа.

Глава 3\Modifier\Program.cs

```
using System;
using NewSpace;

namespace Classes_Structure
{
    class Program
    {
        static void Main(string[] args)
        {
            Opened obj1 = new Opened();
            Assembly1 obj2 = new Assembly1();
            Assembly2 obj3 = new Assembly2();
        }
    }
}
```

Глава 3\Modifier\Modifier.cs

```
namespace NewSpace
{
    public class Opened
    {

    }

    internal class Assembly1
    {

    }

    class Assembly2
    {

    }
}
```

Классы `Opened`, `Assembly1` и `Assembly2` доступны внутри текущей сборки (проекта). Класс `Opened` из пространства `NewSpace` можно использовать и в других проектах, т.к. он описан открытым.

А вот обращение к `Assembly1` и `Assembly2` из других проектов невозможно, т.к. они доступны только внутри этого проекта. По умолчанию классы считаются «запечатанными», модификатор `internal` можно не указывать.

Разбиение на файлы

В рассмотренном примере *Modifier* все три класса описаны в одном файле *Modifier.cs*. Более приемлемой считается практика описания каждого класса в отдельном файле, при этом пространство имен можно сохранить единым.

Так, в условии предыдущей задачи предпочтительнее организовать следующую структуру проекта:

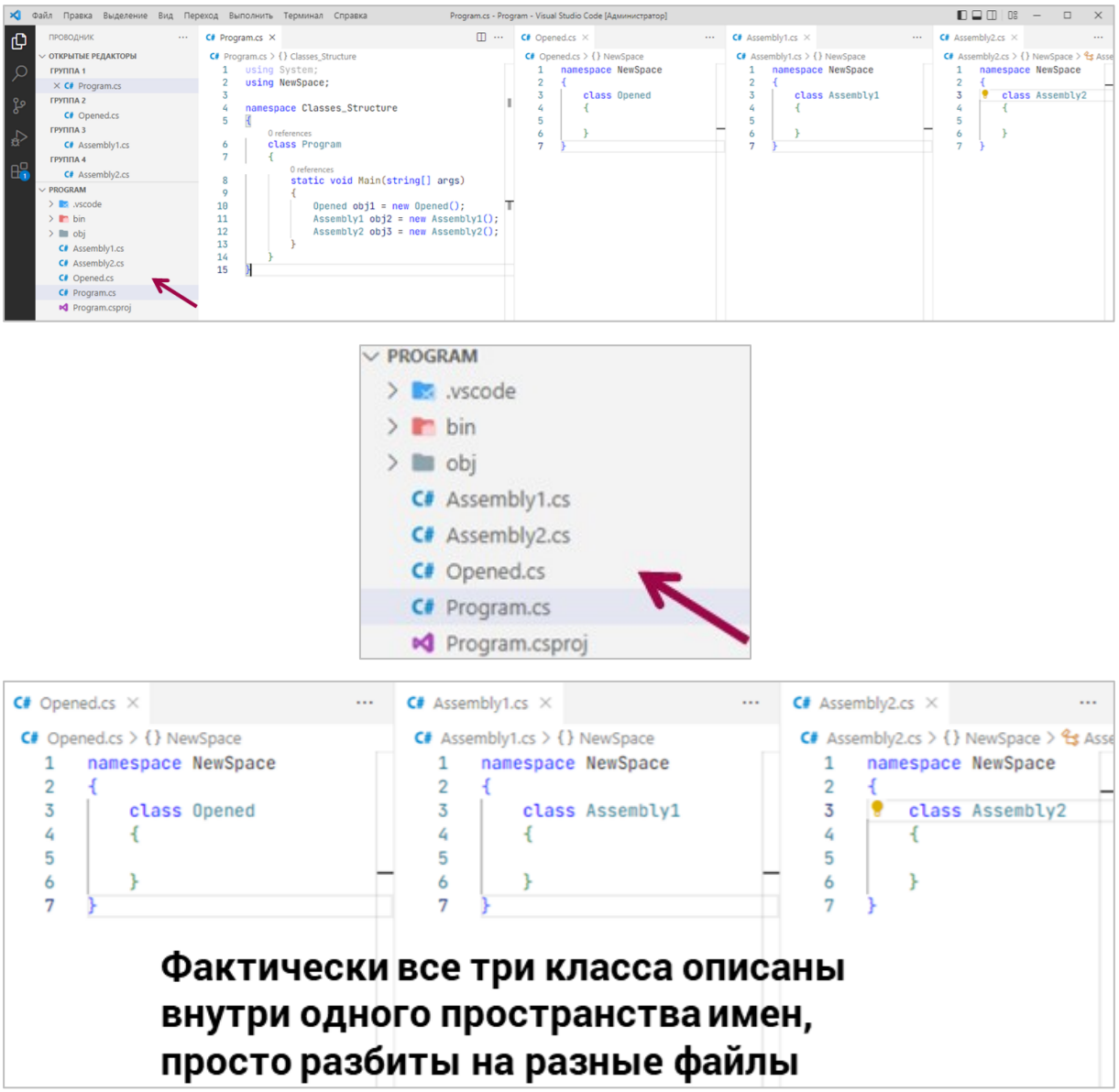


Рис. 3.5. Деление проекта на отдельные файлы

Вопросы для самопроверки

1. Почему классы называют шаблонами описания переменных и типами данных?
2. Что такое объект и экземпляр класса? Приведите примеры описания классов и возможных атрибутов.
3. Что представляют собой компоненты-данные и компоненты-функции класса, каковы их роль?
4. Для чего нужны модификаторы доступа к классу?
5. Почему каждый класс следует описывать в отдельном файле?

3.2. Поля и методы класса

3.2.1. Поля как компоненты-данные

Описание полей в классе

Ране было отмечено, что класс является лишь шаблоном для создания объектов. Объекты представляют собой экземпляры класса с конкретными данными и функциями.

Для размещения данных в классах и экземплярах класса используются **поля**. Поля хранят значения, которые описывают состояние объекта (его характеристики). Для каждого объекта они должны принять определенные значения.

На самом деле каждое поле является глобальной переменной внутри класса.

Каждому полю указывается модификатор доступа, тип и название.

Среди модификаторов доступа отметим два основных:

- **`public`** – открытые поле, т.е. доступно как внутри класса, так и за его пределами;
- **`private`** – закрытое поле, т.е. доступно только внутри класса.

По умолчанию все поля класса изначально считаются закрытыми и `private` можно не указывать.

Так, для описания студента определим в качестве полей его идентификатор (некоторый уникальный номер), фамилию, имя и возраст. На данном этапе курса все поля делаем открытыми:

```
class Student
{
    public long Id;
    public string? FirstName;
    public string? LastName;
    public int Age;
}
```

Каждое поле описано типом, который определяет данные для записи в него. Здесь и далее названия полей заданы с заглавных букв, чтобы их отличать от обычных переменных (хотя далее будет показано более приемлемое правило именования полей).

Создание экземпляров класса

Создадим экземпляр класса `Student`. Каждому полю этого объекта зададим значение. Здесь используется точечная нотация, показывающая привязку поля к объекту `denis`. Порядок инициализации полей не важен.

```
class Program
{
    static void Main(string[] args)
    {
        Student denis = new Student();
        denis.Id = 12345;
        denis.FirstName = "Денис";
        denis.LastName = "Якубович";
        denis.Age = 22;

        Console.WriteLine($"ID: {denis.Id}\nИмя:
            {denis.FirstName}\nФамилия:
            {denis.LastName}\nВозраст: {denis.Age}\n");
    }
}
```

Каждое поле по существу – переменная, поэтому далее его можно использовать в различных операциях, например – выводе, присваивании нового значения, вычислениях и т.д.

Открытый и закрытый доступ к полям

В классе все поля были заданы с модификатором `public`, поэтому их можно вызывать вне класса `Student`.

Если же, например, полю `Id` указать модификатор `private`,

```
class Student
{
    private long Id;
    // Остальные поля
}
```

то здесь обращение к нему вне класса `Student` уже невозможно:

```
Student denis = new Student();
denis.Id = 12345;
```

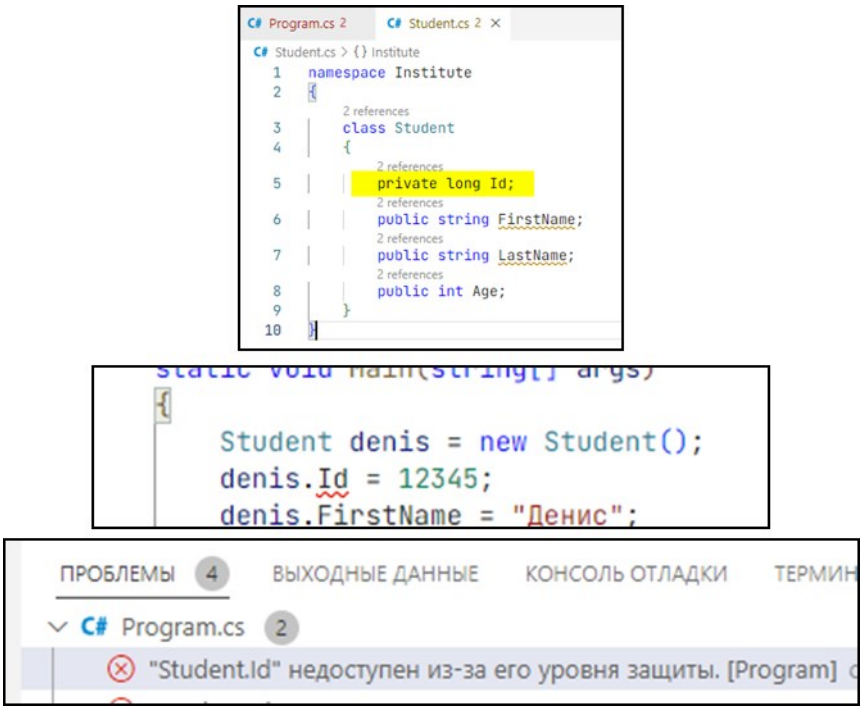


Рис. 3.6. Доступ к private-полю доступен только внутри класса

Пример описания полей и создания экземпляров класса

Требуется задать класс, описывающий студента. В качестве атрибутов определить уникальный идентификатор (числовой), фамилию, имя и возраст (целым числом).

Создать несколько экземпляров класса и задать полям конкретные значения. Вывести на экран информацию по каждому.

Описание класса вынести в отдельный файл. Пространство имен можно использовать единым.

Решение

Глава 3\Institute_1\Student.cs

```
namespace Institute
{
    class Student
    {
        public long Id;
        public string? FirstName;
        public string? LastName;
        public int Age;
    }
}
```

Глава 3\Institute_1\Program.cs

```
using System;

namespace Institute
{
    class Program
    {
        static void Main(string[] args)
        {
            Student denis = new Student();
            denis.Id = 12345;
            denis.FirstName = "Денис";
            denis.LastName = "Якубович";
            denis.Age = 22;

            Console.WriteLine(
                $"ID: {denis.Id}\n
                Имя: {denis.FirstName}\n
                Фамилия: {denis.LastName}\n
                Возраст: {denis.Age}\n"
            );

            Student andrew = new Student();
            andrew.Id = 12000;
            andrew.FirstName = "Андрей";
            andrew.LastName = "Иванов";
            andrew.Age = 21;

            Console.WriteLine(
                $"ID: {andrew.Id}\n
                Имя: {andrew.FirstName}\n
                Фамилия: {andrew.LastName}\n
                Возраст: {andrew.Age}"
            );
        }
    }
}
```

Комментарии к реализации

Здесь в отдельном файле *Student.cs* (одноименный с названием класса) описываем класс с четырьмя открытыми полями, а в основном *Program.cs* создаем два экземпляра класса. Оба класса помещены в единое пространство имен `Institute`.

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
ID: 12345
Имя: Денис
Фамилия: Якубович
Возраст: 22

ID: 12000
Имя: Андрей
Фамилия: Иванов
Возраст: 21
```

Рис. 3.7. Результаты вывода в окно консоли

Замечания по форматированию кода

При описании листингов кода далее нам потребуется работать с длинным строками. Поскольку ширина пространства страницы текущего пособия не позволяет размещать такие записи в одну строку, то мы будем их разбивать на несколько строк, как, например здесь:

```
Console.WriteLine(
    $"ID: {denis.Id}\n
    Имя: {denis.FirstName}\n
    Фамилия: {denis.LastName}\n
    Возраст: {denis.Age}\n"
);
```

Однако, если скопировать код из пособия в редактор, то возникнет ошибка:

```
Console.WriteLine(
    $"ID: {denis.Id}\n
    Имя: {denis.FirstName}\n
    Фамилия: {denis.LastName}\n
    Возраст: {denis.Age}\n"
);
```

Рис. 3.8. Ошибка! Таким образом переносить строки в C# нельзя

Чтобы переносить текст в строке корректно, необходимо использовать оператор конкатенации +, и каждую часть строки заключать в кавычки:

```
Console.WriteLine(
    $"ID: {denis.Id}\n" +
    $"Имя: {denis.FirstName}\n" +
    $"Фамилия: {denis.LastName}\n" +
    $"Возраст: {denis.Age}\n"
);
```

```
Console.WriteLine(
    $"ID: {denis.Id}\n" +
    $"Имя: {denis.FirstName}\n" +
    $"Фамилия: {denis.LastName}\n" +
    $"Возраст: {denis.Age}\n"
);
```

Рис. 3.9. Такое разбиение строки в листинге уже корректно

Важное замечание!

Далее при необходимости разбиения длинной строки мы будем в основном оформлять код первым способом. Однако не забывайте, что при копировании кода из этого пособия обязательно нужно выстроить строку в одну линию, иначе редактор будет указывать на ошибку. Либо разбивайте ее на части с помощью оператора + и отдельных кавычек (см. рис. 3.9).

Рис. 3.10. Таким образом код оформлен в редакторе (см. 15 и 23-ю строки)

Пример с добавлением поля перечислимого типа

Пусть требуется расширить класс Student еще одним полем Degree, которое определяет уровень подготовки студента: бакалавриат, специалитет либо магистратура.

Решение

Поскольку набор возможных значений этого поля будет строго ограничен тремя значениями, то рационально описать его типом перечисления (enum).

Глава 3\Institute_2\EducationalDegree.cs

```
namespace Institute
{
    enum EducationalDegree
    {
        Bachelor,
        Specialist,
        Master
    }
}
```

Глава 3\Institute_2\Student.cs

```
namespace Institute
{
    class Student
    {
        public long Id;
        public string? FirstName;
        public string? LastName;
        public int Age;
        public EducationalDegree Degree;
    }
}
```

Глава 3\Institute_2\Program.cs

```
using System;

namespace Institute
{
    class Program
    {
        static void Main(string[] args)
        {
            Student denis = new Student();
            denis.Id = 12345;
            denis.FirstName = "Денис";
            denis.LastName = "Якубович";
```

```
            denis.Age = 22;
            denis.Degree = EducationalDegree.Specialist;

            Console.WriteLine(
                $"ID: {denis.Id}\n
                Имя: {denis.FirstName}\n
                Фамилия: {denis.LastName}\n
                Возраст: {denis.Age}\n
                Уровень образования: {denis.Degree}\n
            );

            Student andrew = new Student();
            andrew.Id = 12000;
            andrew.FirstName = "Андрей";
            andrew.LastName = "Иванов";
            andrew.Age = 21;
            andrew.Degree = EducationalDegree.Bachelor;

            Console.WriteLine(
                $"ID: {andrew.Id}\n
                Имя: {andrew.FirstName}\n
                Фамилия: {andrew.LastName}\n
                Возраст: {andrew.Age}\n
                Уровень образования: {andrew.Degree}"
            );
        }
    }
}
```

Комментарии к реализации

Для определения уровня подготовки выпускника опишем отдельный тип в виде перечисления `EducationalDegree`. Это позволит замкнуть возможные значения выбора всего тремя вариантами.

Описание также выносим в отдельный файл.

Поле `Degree` имеет тип `EducationDegree`, т.е. задается типом этого перечисления. Как и отмечалось ранее, поля могут иметь любой тип, в т.ч. перечисления или класса.

При инициализации полей двух созданных экземпляров `denis` и `andrew` задаем полям значения.

В текущей реализации уровень образования выводится на английском языке (т.е. по формальному названию значения перечисления), однако в дальнейшем этот код будет доработан.

```
PS D:\Documents\Programming\NET\Manual2025> dotnet run
ID: 12345
Имя: Денис
Фамилия: Якубович
Возраст: 22
Уровень образования: Specialist

ID: 12000
Имя: Андрей
Фамилия: Иванов
Возраст: 21
Уровень образования: Bachelor
```

Рис. 3.11. Результат работы программы после добавления еще одного поля в класс `Student`

3.2.2. Методы как компоненты-функции

Необходимость работы с методами

При создании экземпляров класса `Student` для каждого объекта требуется явно задавать значения полей. Кроме того, команда для вывода задается длинной строкой:

```
Student denis = new Student();
denis.Id = 12345;
denis.FirstName = "Денис";
denis.LastName = "Якубович";
denis.Age = 22;
denis.Degree = EducationalDegree.Specialist;

Console.WriteLine($"ID: {denis.Id}\nИмя: {denis.FirstName}\nФамилия: {denis.Last
```

Рис. 3.12. Инициализация полей объекта и вывод информации необходимо дублировать для каждого нового экземпляра класса

Если потребуется создавать несколько объектов, то для каждого из них операции повторяются и требуется дублировать код для каждого нового объекта.

Необходимо выстроить стратегию, которая позволит упростить обозначенные задачи.

Описание методов в классе

Метод для инициализации полей

Для управления данными внутри класса описывают **методы** (при необходимости повторите предыдущий раздел).

Каждый метод имеет тип и название, при необходимости методу могут передаваться параметры.

Модификаторы доступа к методу аналогичны полям. В текущем курсе нам достаточно двух:

- **`public`** открывает доступ к методу вне класса;
- **`private`** позволяет использовать метод только внутри класса; по умолчанию методы закрыты, `private` необязательно указывать явно.

Опишем в классе Student метод `SetInfo()`, который позволит упростить процедуру заполнения полей объекта. Метод не будет возвращать что-либо, поэтому имеет тип `void`. Необходимые для полей данные будем передавать через параметры метода. В теле метода поочередно копируем в поля значения из параметров.

```
class Student
{
    // ...
    // Выше поля

    // Метод для записи значений в поля
    public void SetInfo(
        long id,
        string? first_name,
        string? last_name,
        int age,
        EducationalDegree degree
    )
    {
        Id = Math.Abs(id);
        FirstName = first_name;
        LastName = last_name;
        Age = (16 <= age) && (age <= 120) ? age : 18;
        Degree = degree;
    }
}
```

Для вызова метода используем все тот же оператор точку:

```
Student denis = new Student();
denis.SetInfo(12345, "Денис", "Якубович", 134,
    EducationalDegree.Specialist);
```

Порядок работы следующий:

1. В основном методе после создания экземпляра класса `denis` осуществляется вызов метода `SetInfo()`, которому передаются значения параметров для заполнения соответствующих полей этого объекта.
2. Метод `SetInfo()` согласно описанию класса записывает переданные в параметрах значения в поля.
3. При этом перед записью поля `Id` значение будет скорректировано: если будет передано отрицательное значение, то исправится на противоположное положительное.
4. Для поля `Age` возраст тоже будет при необходимости скорректирован: если параметр будет вне пределов от 16 до 120, то метод исправит его на некоторое значение по умолчанию (возьмем равным 18).

Здесь код проверки значения, которое будет записываться в поле `Age`, используется тернарный оператор (вместо инструкции `if-else`). Он позволяет сократить выражение проверки в одну строку.

Метод для формирования сводной информации

Также необходимо упростить и вывод информации. Делегируем эту задачу второму методу – `GetInfo()`: пусть он формирует и возвращает информацию о студенте в виде текстовой строки, т.е. возвращает значение типа `string`. Этому методу не нужны параметры: данные он будет брать из полей класса.

Каждое поле доступно внутри обоих методов как глобальная переменная. Методы могут как получать значения полей, так их и менять.

```
public string? GetInfo() =>
    $"ID: {Id}\n
    Имя: {FirstName}\n
    Фамилия: {LastName}\n
    Возраст: {Age}\n
    Уровень образования: {Degree}";
```

Вызова метода в методе `Main()`:

```
Console.WriteLine(denis.GetInfo());
```

Использование методов сокращает код и делает его более безопасным.

Теперь после описания объекта не требуется явно задавать каждое поле, поскольку метод `SetInfo()` передает все необходимые данные и «скрытно» записывает их в соответствующие поля. А метод `GetInfo()` возвращает необходимую информацию в виде строки.

Приведем полный код программы:

Глава 3\Institute_3_1\EducationalDegree.cs

```
namespace Institute
{
    enum EducationalDegree
    {
        Bachelor,
        Specialist,
        Master
    }
}
```

Глава 3\Institute_3_1\Student.cs

```
namespace Institute
{
    class Student
    {
        public long Id;
        public string? FirstName = "";
        public string? LastName = "";
        public int Age;
        public EducationalDegree Degree;

        public void SetInfo(
            long id,
            string? first_name,
            string? last_name,
            int age,
            EducationalDegree degree
        )
        {
            Id = Math.Abs(id);
            FirstName = first_name;
            LastName = last_name;
            Age = (16 <= age) && (age <= 120) ? age : 18;
            Degree = degree;
```

```
        }

        public string? GetInfo() =>
            $"ID: {Id}\n
            Имя: {FirstName}\n
            Фамилия: {LastName}\n
            Возраст: {Age}\n
            Уровень образования: {Degree}";
    }
}
```

Глава 3\Institute_3_1\Program.cs

```
using System;

namespace Institute
{
    class Program
    {
        static void Main(string[] args)
        {
            Student denis = new Student();
            denis.SetInfo(12345, "Денис", "Якубович", 134,
                EducationalDegree.Specialist);
            Console.WriteLine(denis.GetInfo() + "\n");

            Student andrew = new Student();
            andrew.SetInfo(12000, "Андрей", "Иванов", 21,
                EducationalDegree.Bachelor);
            Console.WriteLine(andrew.GetInfo());
        }
    }
}
```

Если сравнить этот проект с предыдущим *Institute_2*, то создание экземпляров класса и вывод информации о каждом на экран существенно упростились. Кроме того, методу `SetInfo()` обязательно требуется передать все пять значений параметров. Если же инициализировать поля по отдельности, то можно забыть установить какому-либо из них значение. Т.о. метод также в некотором степени организует более защищенную инициализацию объекта класса.

Результаты вывода будут аналогичными.

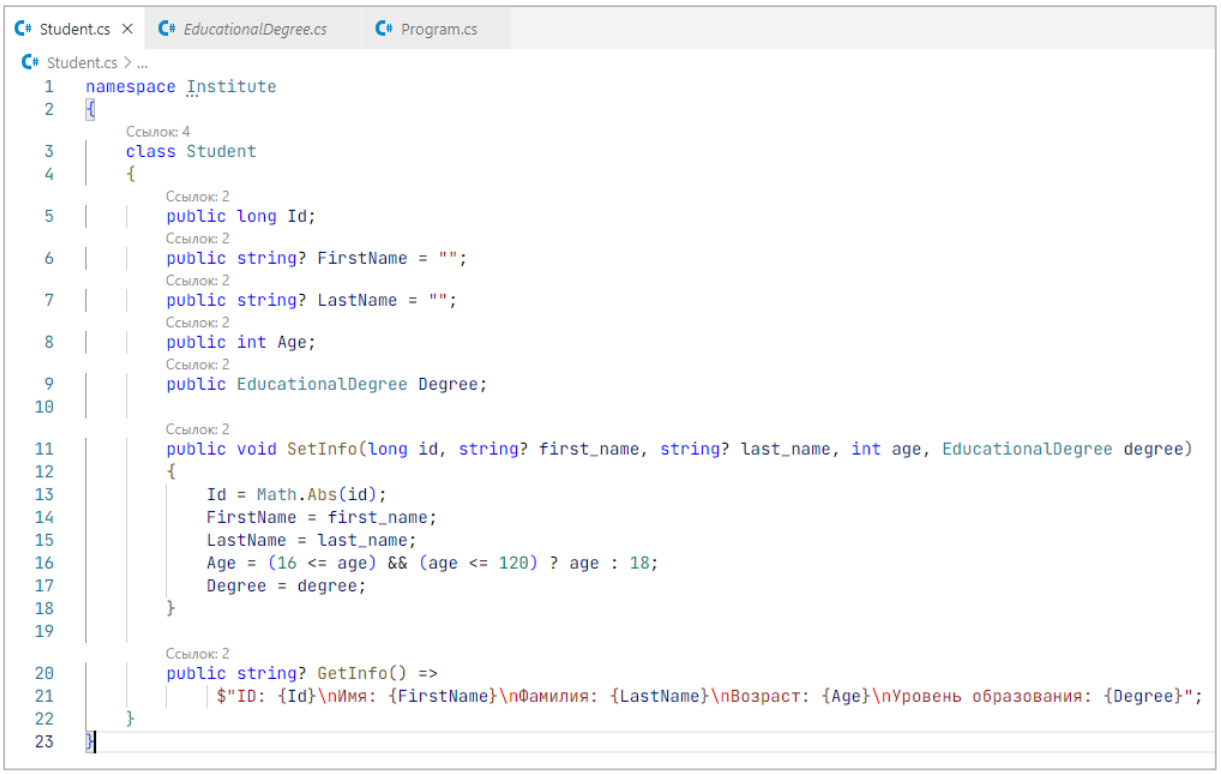


Рис. 3.13. Поля и методы класса описываются на одном уровне

Расширение класса дополнительными методами

Уже из предыдущего примера видно, что методы позволяют скрывать логику обработки данных внутри класса.

Расширим класс Student еще двумя полезными методами:

- `IsAdult()` – проверяет, является ли студент совершеннолетним (логический).
- `GetEducationalType()` – возвращает название уровня подготовки на русском языке (строковый).

Проверка студента на совершеннолетие

Метод `IsAdult()` будет возвращать значение `true` или `false` в зависимости от возраста студента:

```
class Student
{
    // Код выше ...

    public bool IsAdult() =>
        Age >= 18;

    // Код ниже ...
}
```

Здесь также используются возможности нового синтаксиса C# для описания коротких функций, хотя можно оставить и классическую полную форму с фигурными скобками и оператором `return`:

```
public bool IsAdult()
{
    return Age >= 18;
}
```

В основной части программы теперь можно использовать этот метод как проверку признака, например:

```
if (denis.IsAdult())
    Console.WriteLine(
        $"{denis.FirstName} совершеннолетний"
    );
```

Здесь метод избавляет нас от необходимости обращаться к полю возраста и сравнивать его со значением 18, как если бы мы работали с полем:

```
if (denis.Age >= 18)
    Console.WriteLine(
        $"{denis.FirstName} совершеннолетний"
    );
```

Название метода уже скрывает подразумевает проверку, при этом прячет это код внутри класса.

Возврат уровня подготовки студента

Метод `GetEducationalType()` в зависимости от абстрактной константы из поля `Degree` (перечислимого типа) вернет соответствующую строковую запись на русском языке.

Здесь для лаконичности оператор переключения `switch` заменен на более новый аналог шаблона сопоставления `switch`:

```
class Student
{
    // См. выше ...

    public string GetEducationalType()
    {
        string educational_type = Degree switch
        {
            EducationalDegree.Bachelor => "бакалавриат",
            EducationalDegree.Specialist => "специалитет",
```

```
                EducationalDegree.Master => "магистратура",
                _ => "не определено"
            };

            return educational_type;
        }
            // Далее ...
    }
```

В зависимости от значения в поле `Degree` в локальную переменную `educational_type` запишется название уровня подготовки студента, эта строка и будет возвращена методом.

Теперь можно внести изменения в метод `GetInfo()`. Методы внутри класса могут взаимодействовать. Чтобы скорректировать название уровня образования на русский язык, вызываем ранее описанный метод `GetEducationalType()`: он возвратит необходимую запись, которая подставится в указанное место строки, формируемой методом `GetInfo()`.

```
public string? GetInfo() =>
    $"ID: {Id}\n
    Имя: {FirstName}\n
    Фамилия: {LastName}\n
    Возраст: {Age}\n
    Уровень образования: {GetEducationalType()}"
```

Код метода `GetEducationalType()` можно упростить, используя все тот же оператор тела выражения `=>`. Шаблон сопоставления `swicth` является одной вычисляемой конструкцией, возвращающей значение, поэтому отпадает необходимость использовать промежуточную переменную `educational_type`:

```
public string GetEducationalType() =>
    Degree switch
    {
        EducationalDegree.Bachelor => "бакалавриат",
        EducationalDegree.Specialist => "специалитет",
        EducationalDegree.Master => "магистратура",
        _ => "не определено"
    };
```

Приведем полный код программы:

Глава 3\Institute_3_2\EducationalDegree.cs

```
namespace Institute
{
    enum EducationalDegree
    {
        Bachelor,
        Specialist,
        Master
    }
}
```

Глава 3\Institute_3_2\Student.cs

```
namespace Institute
{
    class Student
    {
        public long Id;
        public string? FirstName = "";
        public string? LastName = "";
        public int Age;
        public EducationalDegree Degree;

        public void SetInfo(
            long id,
            string? first_name,
            string? last_name,
            int age,
            EducationalDegree degree
        )
        {
            Id = Math.Abs(id);
            FirstName = first_name;
            LastName = last_name;
            Age = (16 <= age) && (age <= 120) ? age : 18;
            Degree = degree;
        }

        public string? GetInfo() =>
            $"ID: {Id}\n
            Имя: {FirstName}\n
            Фамилия: {LastName}\n
            Возраст: {Age}\n
```

```
            Уровень образования: {GetEducationalType()}";

        public bool IsAdult() =>
            Age >= 18;

        public string GetEducationalType() =>
            Degree switch
            {
                EducationalDegree.Bachelor => "бакалавриат",
                EducationalDegree.Specialist => "специалитет",
                EducationalDegree.Master => "магистратура",
                _ => "не определено"
            };
    }
}
```

Глава 3\Institute_3_2\Program.cs

```
using System;

namespace Institute
{
    class Program
    {
        static void Main(string[] args)
        {
            Student denis = new Student();
            denis.SetInfo(12345, "Денис", "Якубович", 134,
                EducationalDegree.Specialist);
            Console.WriteLine(denis.GetInfo());

            Console.WriteLine("\nДругая информация:");

            if (denis.IsAdult())
                Console.WriteLine(
                    $"{denis.FirstName} совершеннолетний"
                );

            Console.WriteLine(
                $"Уровень подготовки:
                 {denis.GetEducationalType()}"
            );
        }
    }
}
```

```
PS D:\Documents\Programming\NET\2024\Institute> dotnet run
ID: 12345
Имя: Денис
Фамилия: Якубович
Возраст: 18
Уровень образования: специалитет

Другая информация:
Денис совершеннолетний
Уровень подготовки: специалитет
```

Рис. 3.14. Использование новых методов в классе `Student`

Student.cs | EducationalDegree.cs | Program.cs

Student.cs > Student

```
1   namespace Institute
2   {
        Ссылок: 2
3       class Student
4       {
            Ссылок: 2
5           public long Id;
            Ссылок: 3
6           public string? FirstName = "";
            Ссылок: 2
7           public string? LastName = "";
            Ссылок: 3
8           public int Age;
            Ссылок: 2
9           public EducationalDegree Degree;
10
            Ссылок: 1
11          public void SetInfo(long id, string? first_name, string? last_name, int age,
            EducationalDegree degree)
12          {
13              Id = Math.Abs(id);
14              FirstName = first_name;
15              LastName = last_name;
16              Age = (16 <= age) && (age <= 120) ? age : 18;
17              Degree = degree;
18          }
19
            Ссылок: 1
20          public string? GetInfo() =>
21              $"ID: {Id}\nИмя: {FirstName}\nФамилия: {LastName}\nВозраст: {Age}\nУровень образования:
                {GetEducationalType()}";
22
            Ссылок: 1
23          public bool IsAdult() =>
24              Age >= 18;
25
            Ссылок: 2
26          public string GetEducationalType() =>
27              Degree switch
28              {
29                  EducationalDegree.Bachelor => "бакалавриат",
30                  EducationalDegree.Specialist => "специалитет",
31                  EducationalDegree.Master => "магистратура",
32                  _ => "не определено"
33              };
34      }
35  }
```

Рис. 3.15. Описание класса `Student` на текущем этапе

Это полезно знать!

Методы в классе решают целый ряд важных задач.
С одной стороны, метод скрывает логику обработки данных. Для его использования достаточно знать, что делает метод, какие параметры ему необходимы и что он возвращает.
С другой стороны, методы можно вызывать многократно, что избавляет от дублирования кода.
Наконец, метод предполагает действие(я) в привязке к объекту, т.е. рассматривается как его неотъемлемая функциональная часть.

Метод ToString()

Все классы, в т.ч. создаваемые пользователем, производны от класса **`Object`**. Говорят, что `Object` является **базовым классом** для всех остальных. Производные классы автоматически наследуют ряд методов из `Object`.

Среди таких методов отметим метод **`ToString()`**. Этот метод возвращает некоторую строковую запись об объекте. Изначально в качестве записи служит только название класса, но разработчик в своем классе может переопределить действие этого метода так, как посчитает нужным.

Для простых типов данных `ToString()` преобразует значение в строковую запись. В общем случае для объектов классов он возвратит только название пространства имен и класса, которым описан объект (как для объекта `denis`):

```
Console.WriteLine(45.ToString());
Console.WriteLine(true.ToString());
Console.WriteLine("Привет!".ToString());
Console.WriteLine(denis.ToString());
```

```
45
True
Привет!
Institute.Student
```

Рис. 3.16. Явный вызов метода `ToString()`

Однако при использовании методов `Write()` и `WriteLine()` вызывать у объекта в параметре метод `ToString()` излишне, т.к. одна из его перегрузок может принимать объект любого типа и скрыто вызывать `ToString()`:

```
Console.WriteLine(45);
Console.WriteLine(true);
Console.WriteLine("Привет!");
Console.WriteLine(denis);
```

Нетрудно заметить, что подобное поведения метода `ToString()` рационально распространить и в нашем классе `Student`:

```
class Student
{
    // Код выше

    public override string ToString() =>
        $"ID: {Id}\n
        Имя: {FirstName}\n
        Фамилия: {LastName}\n
        Возраст: {Age}\n
        Уровень образования: {GetEducationalType()}";

    // Код ниже
}
```

Модификатор **`override`** показывает, что базовая реализация метода `ToString()` может быть переопределена в нашем классе. Тело метода аналогично коду из `GetInfo()`, его достаточно скопировать.

Метод `GetInfo()` больше не нужен, его задачу будет решать переопределенный `ToString()`.

Теперь при выводе объекта в окно консоли осуществлялся неявный вызов метода ToString() (т.е. его вызывать необязательно):

```
Student denis = new Student();
denis.SetInfo(12345, "Денис", "Якубович", 134,
    EducationalDegree.Specialist);
Console.WriteLine(denis);
```

```
PS D:\Documents\Programming\NET\2024\Institute> dotnet run
ID: 12345
Имя: Денис
Фамилия: Якубович
Возраст: 18
Уровень образования: специалитет
```

Рис. 3.17. Использование метода `ToString()`

Полный код программы

Приведем полный код программы:

Глава 3\Institute_3_3\EducationalDegree.cs

```
namespace Institute
{
    enum EducationalDegree
    {
        Bachelor,
        Specialist,
        Master
    }
}
```

Глава 3\Institute_3_3\Student.cs

```
namespace Institute
{
    class Student
    {
        public long Id;
        public string? FirstName = "";
        public string? LastName = "";
        public int Age;
        public EducationalDegree Degree;

        public void SetInfo(
            long id,
            string? first_name,
            string? last_name,
            int age,
            EducationalDegree degree
        )
        {
            Id = Math.Abs(id);
            FirstName = first_name;
```

```
            LastName = last_name;
            Age = (16 <= age) && (age <= 120) ? age : 18;
            Degree = degree;
        }

        public override string ToString() =>
            $"ID: {Id}\n
            Имя: {FirstName}\n
            Фамилия: {LastName}\n
            Возраст: {Age}\n
            Уровень образования: {GetEducationalType()}";

        public bool IsAdult() =>
            Age >= 18;

        public string GetEducationalType() =>
            Degree switch
            {
                EducationalDegree.Bachelor => "бакалавриат",
                EducationalDegree.Specialist => "специалитет",
                EducationalDegree.Master => "магистратура",
                _ => "не определено"
            };
    }
}
```

Глава 3\Institute_3_3\Program.cs

```
using System;

namespace Institute
{
    class Program
    {
        static void Main(string[] args)
        {
            Student denis = new Student();
            denis.SetInfo(12345, "Денис", "Якубович", 134,
                EducationalDegree.Specialist);
            Console.WriteLine(denis);

            Console.WriteLine("\nДругая информация:");

            if (denis.IsAdult())
```

```
                Console.WriteLine($"{denis.FirstName}
                    совершеннолетний");

            Console.WriteLine($"Уровень подготовки:
                {denis.GetEducationalType()}");
        }
    }
}
```

3.2.3. Решение практических задач

Расширение класса методами инициализации

Расширим описание класса `Student` тремя методами, которые заполняют поля:

- `InitBachelor()` – инициализирует студента бакалавриата;
- `InitSpecialist()` – инициализирует студента специалитета;
- `InitMaster()` – инициализирует студента магистратуры.

Опишем метод `InitBachelor()` (остальные два описываются по аналогии). Метод `InitBachelor()` будет частным случаем метода `SetInfo()`: он делает то же самое, но поле `Degree` задается уже предопределенным значением. Поэтому внутри достаточно вызвать метод `SetInfo()` и передать ему 4 параметра из метода `InitBachelor()`, а 5-й параметр задать константой:

```
namespace Institute
{
    class Student
    {
        // Код выше ...

        public void InitBachelor(
            long id, string? first_name,
            string? last_name, int age
        ) =>
            SetInfo(id, first_name, last_name, age,
                EducationalDegree.Bachelor);

        // Код ниже ...
    }
}
```

Аналогично описываются методы `InitSpecialist()` и `InitMaster()`: они вызывают метод `SetInfo()`, первые 4 параметра копируются, а 5-й параметр задается определенной константой:

```
// Код выше ...

Ссылок: 1
public void InitBachelor(long id, string? first_name, string? last_name, int age) =>
    SetInfo(id, first_name, last_name, age, EducationalDegree.Bachelor);

Ссылок: 1
public void InitSpecialist(long id, string? first_name, string? last_name, int age) =>
    SetInfo(id, first_name, last_name, age, EducationalDegree.Specialist);

Ссылок: 1
public void InitMaster(long id, string? first_name, string? last_name, int age) =>
    SetInfo(id, first_name, last_name, age, EducationalDegree.Master);

// Код ниже ...
```

Рис. 3.18. Все три метода инициализации используют метод `SetInfo()` как вспомогательный

В методе `Main()` заполняем поля объектов после их создания с использованием частных случаев метода `SetInfo()`:

```
Student denis = new Student();
denis.InitSpecialist(12345, "Денис", "Якубович", 22);
Console.WriteLine(denis);

Student andrew = new Student();
andrew.InitBachelor(13002, "Андрей", "Иванов", 21);
Console.WriteLine(andrew);

Student olga = new Student();
olga.InitMaster(14450, "Ольга", "Куликова", 23);
Console.WriteLine(olga);
```

Здесь вызов методов сокращается, т.к. уменьшается число требуемых параметров.

Прием, показанный в примере, часто находит использование на практике. Зачастую требуется создавать объекты определенного подкласса. Чтобы не прописывать все параметры, описывают частные случаи методов.

Например, следующие команды приводят к одинаковому результату:

```
denis.SetInfo(12345, "Денис", "Якубович", 22, EducationalDegree.Specialist);
denis.InitSpecialist(12345, "Денис", "Якубович", 22);
```

Рис. 3.19. Общий метод инициализации и частный случай

Однако второй вызов метода не только короче, его проще осмысливать: название вносит больше конкретики в создаваемый объект, а также требуется меньшее число параметров.

Полный код примера:

Глава 3\Institute_4\EducationalDegree.cs

```
namespace Institute
{
    enum EducationalDegree
    {
        Bachelor,
        Specialist,
        Master
    }
}
```

Глава 3\Institute_4\Student.cs

```
namespace Institute
{
    class Student
    {
        public long Id;
        public string? FirstName = "";
        public string? LastName = "";
        public int Age;
        public EducationalDegree Degree;

        public void SetInfo(
            long id, string? first_name,
            string? last_name, int age,
            EducationalDegree degree
        )
        {
            Id = id;
            FirstName = first_name;
            LastName = last_name;
            Age =  (16 <= age) && (age <= 120) ? age : 18;
            Degree = degree;
        }
```

```
        public void InitBachelor(
            long id, string? first_name, string? last_name,
            int age
        ) =>
            SetInfo(id, first_name, last_name, age,
                EducationalDegree.Bachelor);

        public void InitSpecialist(
            long id, string? first_name, string? last_name,
            int age
        ) =>
            SetInfo(id, first_name, last_name, age,
                EducationalDegree.Specialist);

        public void InitMaster(
            long id, string? first_name, string? last_name,
            int age
        ) =>
            SetInfo(id, first_name, last_name, age,
                EducationalDegree.Master);

        public override string? ToString() =>
            $" ID: {Id}\n
            Имя: {FirstName}\n
            Фамилия: {LastName}\n
            Возраст: {Age}\n
            Уровень образования: {GetEducationalType()}";

        public bool IsAdult() =>
            Age >= 18;

        public string GetEducationalType() =>
            Degree switch
            {
                EducationalDegree.Bachelor => "бакалавриат",
                EducationalDegree.Specialist => "специалитет",
                EducationalDegree.Master => "магистратура",
                _ => "не определено"
            };
    }
}
```

Глава 3\Institute_4\Program.cs

```
using System;

namespace Institute
{
    class Program
    {
        static void Main(string[] args)
        {
            Student denis = new Student();
            denis.InitSpecialist(
                12345, "Денис", "Якубович", 22
            );
            Console.WriteLine(denis);

            Student andrew = new Student();
            andrew.InitBachelor(
                13002, "Андрей", "Иванов", 21
            );
            Console.WriteLine(andrew);

            Student olga = new Student();
            olga.InitMaster(
                14450, "Ольга", "Куликова", 23
            );
            Console.WriteLine(olga);
        }
    }
}
```

```
PS D:\Documents\Programming\NET\2024\Institute> dotnet run
ID: 12345
Имя: Денис
Фамилия: Якубович
Возраст: 22
Уровень образования: специалитет

ID: 13002
Имя: Андрей
Фамилия: Иванов
Возраст: 21
Уровень образования: бакалавриат

ID: 14450
Имя: Ольга
Фамилия: Куликова
Возраст: 23
Уровень образования: магистратура
```

Рис. 3.20. Результат работы программы

Закрытый доступ к «обслуживающим» методам класса

Вспомогательные методы

Пусть требуется сделать закрытым доступ к методу `SetInfo()`. Добавим метод `InitStudent()`, который будет заполнять все поля (метод открытый).

```
namespace Institute
{
    class Student
    {
        // Код выше ...

        private void _SetInfo(
            long id, string? first_name,
            string? last_name, int age,
            EducationalDegree degree
        )
        {
            Id = Math.Abs(id);
            FirstName = first_name;
            LastName = last_name;
            Age = (16 <= age) && (age <= 120) ? age : 18;
            Degree = degree;
        }

        // Код ниже ...
    }
}
```

С помощью модификатора `private` закрываем доступ к методу `SetInfo()`. Также переименуем его, указав в начале нижнее подчеркивание (так часто принято называть закрытые методы класса).

Теперь этот метод уже нельзя использовать вне класса `Student`. Однако метод `_SetInfo()` доступен внутри класса.

Используем его, чтобы задать поля для каждого из четырех возможных методов-инициализаторов.

```
namespace Institute
{
    class Student
    {
        // Код выше ...
```

```
        public void InitBachelor(
            long id, string? first_name,
            string? last_name, int age
        ) =>
            _SetInfo(id, first_name, last_name, age,
                EducationalDegree.Bachelor);

        public void InitSpecialist(
            long id, string? first_name,
            string? last_name, int age
        ) =>
            _SetInfo(id, first_name, last_name, age,
                EducationalDegree.Specialist);

        public void InitMaster(
            long id, string? first_name,
            string? last_name, int age
        ) =>
            _SetInfo(id, first_name, last_name, age,
                EducationalDegree.Master);

        public void InitStudent(
            long id, string? first_name,
            string? last_name, int age,
            EducationalDegree degree
        ) =>
            _SetInfo(id, first_name, last_name, age,
                degree);

        // Код ниже ...
    }
}
```

Метод `InitStudent()` будет наиболее общим случаем и фактически делать то же, что и `_SetInfo()` – заполнять поля по 5-ти произвольным параметрам (эти параметры просто передаются как копии значений из `InitStudent()`).

В силу модификатора `private` метод `_SetInfo()` больше недоступен извне своего класса. Его роль теперь играет метод `InitStudent()`, а также частные методы-инициализаторы `InitBachelor()`, `InitSpecialist()` и `InitMaster()`.

Значимость запечатанных методов

Закрывая метод `_SetInfo()` мы делаем его методом «внутренней кухни» класса `Student`. Он берет на себя задачу обслуживания открытых методов `InitStudent()`, `InitBachelor()`, `InitSpecialist()` и `InitMaster()`.

Этим мы обеспечиваем еще один принцип ООП – **инкапсуляция**, т.е. сокрытие данных, доступ к которым должен быть закрыт в определенной области. Иными словами, закрытые методы решают локальные задачи внутри класса, а открытые методы необходимы для внешних операций с объектом, хотя и внутри класса их тоже можно использовать.

На самом деле этот подход может быть усовершенствован, если использовать конструктор класса (см. следующий пункт).

Полная реализация задачи

Глава 3\Institute_5\EducationalDegree.cs

```
namespace Institute
{
    enum EducationalDegree
    {
        Bachelor,
        Specialist,
        Master
    }
}
```

Глава 3\Institute_5\Student.cs

```
namespace Institute
{
    class Student
    {
        public long Id;
        public string? FirstName = "";
        public string? LastName = "";
        public int Age;
        public EducationalDegree Degree;

        private void _SetInfo(
            long id, string? first_name, string? last_name,
            int age, EducationalDegree degree
        )
```

```
        {
            Id = Math.Abs(id);
            FirstName = first_name;
            LastName = last_name;
            Age = (16 <= age) && (age <= 120) ? age : 18;
            Degree = degree;
        }

        public void InitBachelor(
            long id, string? first_name, string? last_name,
            int age
        ) =>
            _SetInfo(id, first_name, last_name, age,
                EducationalDegree.Bachelor);

        public void InitSpecialist(
            long id, string? first_name, string? last_name,
            int age
        ) =>
            _SetInfo(id, first_name, last_name, age,
                EducationalDegree.Specialist);

        public void InitMaster(
            long id, string? first_name, string? last_name,
            int age
        ) =>
            _SetInfo(id, first_name, last_name, age,
                EducationalDegree.Master);

        public void InitStudent(
            long id, string? first_name, string? last_name,
            int age, EducationalDegree degree
        ) =>
            _SetInfo(id, first_name, last_name, age, degree);

        public override string ToString() =>
            $"ID: {Id}\n
            Имя: {FirstName}\n
            Фамилия: {LastName}\n
            Возраст: {Age}\n
            Уровень образования: {GetEducationalType()}";

        public bool IsAdult() =>
            Age >= 18;
```

```
        public string GetEducationalType() =>
            Degree switch
            {
                EducationalDegree.Bachelor => "бакалавриат",
                EducationalDegree.Specialist => "специалитет",
                EducationalDegree.Master => "магистратура",
                _ => "не определено"
            };
    }
}
```

Глава 3\Institute_5\Program.cs

```
using System;

namespace Institute
{
    class Program
    {
        static void Main(string[] args)
        {
            Student margo = new Student();
            margo.InitStudent(
                20034, "Маргарита", "Березкина",
                20, EducationalDegree.Bachelor
            );
            Console.WriteLine(margo);
        }
    }
}
```

```
PS D:\Documents\Programming\NET\2024\Institute> dotnet run
ID: 20034
Имя: Маргарита
Фамилия: Березкина
Возраст: 20
Уровень образования: бакалавриат
```

Рис. 3.21. Результат работы программы

Вопросы для самопроверки

1. Какую роль играют поля класса и как их описывать?
2. Можно ли полям задать произвольный тип данных?
3. Каким образом создаются экземпляры класса и заполняются их поля?
4. Почему поля и методы в классе тесно взаимосвязаны?
5. Каким образом описывается и вызываются метод?
6. Чем полезно переопределение метода `ToString()`?
7. Приведите примеры описания классов с несколькими полями и методами, которые их обрабатывают.

Практикум

1. Закрепление пройденного

Задание 1

1. Создайте каталог *Institute*.
2. Далее внутри подготовьте новые каталоги *Insutute_1*, *Insutute_2* и т.д., согласно рассмотренным в этом параграфе примерам.

Задание 2

1. Последовательно повторите и воспроизведите рассмотренные примеры, каждый в отдельном каталоге (см. по названию в примерах).
2. Ваша задача:
    - изучать взаимодействие между методами и полями класса;
    - проследить за логикой вызова методов;
    - разобрать фрагменты кода, которые упрощены;
    - понять значимость предложенных преобразований.

2. Работа с полями и методами класса

Задание 1

1. Создайте каталог *WebSystem*.
2. Далее в нем реализуйте в отдельных каталогах описанные ниже задания.
3. Замечание: общая концепция работы с заданиями похожа на рассматриваемый здесь и далее пример с классом *Student*. Предполагается, что каждый последующий шаг предполагает некоторую модификацию кода.

Задание 2

1. Создайте каталог *Account_1*.
2. Опишите класс `Account`, который характеризует данные учетной записи сотрудника некоторой компании. В классе задайте следующие поля:
   a. `Id` – уникальный номер сотрудника (целое число);
   b. `Login` – логин пользователя (строка);
   c. `FullName` – полные ФИО сотрудника (строка);
   d. `Age` – возраст сотрудника (целое число).
3. Создайте 2-3 экземпляра этого класса и выведите информацию о них на экран.
4. На данном этапе методы для заполнения полей и вывода информации не описывать.
5. Оформить результат следующим образом:

```
PS D:\Documents\Programming\NET\2024\WebSystem> dotnet run
ID: 10345
Логин: YDA2023
ФИО: Якубович Денис Андреевич
Возраст: 24

ID: 20108
Логин: miha_krapiva
ФИО: Крапивницкий Михаил Эдуардович
Возраст: 31
```

Рис. 3.22. Результат работы программы из задания 2

Задание 3

1. Создайте каталог *Account_2*.
2. Используйте в качестве основы код из *Account_1*.
3. Добавьте в класс `Account` поле `AccessRights`, хранящее права доступа пользователя (может быть администратором, менеджером, штатным сотрудником либо гостем).
4. Поле должно иметь тип перечисления `AccessRightsType` с четырьмя возможными значениями: `Admin`, `Manager`, `Employee`, `Guest`.
5. Описание перечисления вынести в отдельный файл.
6. Дополните работу с этим полем при создании объектов класса `Account`.

```
PS D:\Documents\Programming\NET\2024\WebSystem> dotnet run
ID: 10345
Логин: YDA2023
ФИО: Якубович Денис Андреевич
Возраст: 24
Права доступа: Admin

ID: 20108
Логин: miha_krapiva
ФИО: Крапивницкий Михаил Эдуардович
Возраст: 31
Права доступа: Employee
```

Рис. 3.23. Результат работы программы из задания 3

Задание 4

1. Создайте каталог *Account_3*.
2. Используйте в качестве основы код из *Account_2*.
3. Внесите в описание класса `Account` ряд методов и протестируйте их на 2-3 экземплярах класса:
   - `SetData()` – записывает произвольные данные в поля (передаются через параметры метода);
   - `ToString()` – переопределить для возврата строки, как изображено на рисунке ниже;
   - `IsAdult()` – проверяет, является ли студент совершеннолетним (логический);

- `GetAccessRightsType()` – возвращает название уровня подготовки на русском языке (т.е. в виде текстовой строки).

4. Воспользуйтесь ими для задания и вывода дополнительной информации.

```
var denis = new Account();
denis.SetData(10345, "YDA2023", "Якубович Денис Андреевич",
    24, AccessRightsType.Admin);
Console.WriteLine(denis);

if (denis.IsAdult())
    Console.WriteLine($"{denis.FullName} совершеннолетний");

Console.WriteLine();

var misha = new Account();
misha.SetData(20108, "miha_krapiva", "Крапивницкий Михаил Эдуардович",
    31, AccessRightsType.Employee);
Console.WriteLine(misha);

if (misha.IsAdult())
    Console.WriteLine($"{misha.FullName} совершеннолетний");
```

Рис. 3.24. Фрагмент кода для создания экземпляров класса `Account` и использование его методов

```
PS D:\Documents\Programming\NET\2024\WebSystem> dotnet run
ID: 10345
Логин: YDA2023
ФИО: Якубович Денис Андреевич
Возраст: 24
Права доступа: администратор
Якубович Денис Андреевич совершеннолетний

ID: 20108
Логин: miha_krapiva
ФИО: Крапивницкий Михаил Эдуардович
Возраст: 31
Права доступа: сотрудник
Крапивницкий Михаил Эдуардович совершеннолетний
```

Рис. 3.25. Результат работы программы из задания 4

Задание 5

1. Создайте каталог *Account_4*.
2. Используйте в качестве основы код из *Account_3*.
3. Внесите коррективы в метод `SetData()`, чтобы он автоматически корректировал значения, записываемые в поля:
   - в случае попытки задать возраст меньше 18 или больше 100 в поле будет записано значение 18;
   - идентификатор должен быть неотрицательным числом, в противном случае исправить его на положительное.

```
denis.SetData(-10345, "YDA2023", "Якубович Денис Андреевич", 24, AccessRightsType.Admin);

misha.SetData(20108, "miha_krapiva", "Крапивницкий Михаил Эдуардович", 16, AccessRightsType.Employee);
```

Рис. 3.26. Вызов метода `SetData()` с «плохими» значениями параметров идентификатора и возраста

```
ID: 10345
Логин: YDA2023
ФИО: Якубович Денис Андреевич
Возраст: 24
Права доступа: администратор

ID: 20108
Логин: miha_krapiva
ФИО: Крапивницкий Михаил Эдуардович
Возраст: 18
Права доступа: сотрудник
```

Рис. 3.27. Результат работы программы из задания 5

Задание 6

1. Создайте каталог *Account_5*.
2. Используйте в качестве основы код из *Account_4*.
3. Добавьте в описание класса четыре метода:
   - `IsAdmin()` – возвращает истину, если сотрудник имеет права администратора;
   - `IsManager()` – возвращает истину, если сотрудник имеет права администратора;
   - `IsEmployee()` – возвращает истину, если сотрудник имеет права штатного сотрудника;

- `IsGuest()` – возвращает истину, если пользователь имеет права доступа как гость.

4. Протестируйте эти методы для одного созданного объекта.

```
var denis = new Account();
denis.SetData(10345, "YDA2023", "Якубович Денис Андреевич",
    24, AccessRightsType.Admin);
Console.WriteLine(denis + "\n");

if (denis.IsAdmin())
    Console.WriteLine("Пользователь имеет права администратора");

if (denis.IsManager())
    Console.WriteLine("Пользователь имеет права менеджера");

if (denis.IsEmployee())
    Console.WriteLine("Пользователь имеет права штатного сотрудника");

if (denis.IsGuest())
    Console.WriteLine("Пользователь имеет права гостя");
```

Рис. 3.28. Фрагмент кода для создания экземпляров класса `Account` и использование его методов

```
PS D:\Documents\Programming\NET\2024\WebSystem> dotnet run
ID: 10345
Логин: YDA2023
ФИО: Якубович Денис Андреевич
Возраст: 24
Права доступа: администратор

Пользователь имеет права администратора
```

Рис. 3.29. Результат работы программы из задания 6

Задание 7

1. Создайте каталог *Account_6*.
2. Используйте в качестве основы код из *Account_5*.
3. Добавьте в описание класса три метода:
   - `GetFirstName()` – возвращает имя сотрудника;
   - `GetLastName()` – возвращает фамилию сотрудника;
   - `GetPatronymic()` – возвращает отчество сотрудника.
4. Для извлечения фрагментов строки используйте метод `Split()`.

5. Протестируйте эти методы для одного созданного объекта.

```
var denis = new Account();
denis.SetData(10345, "YDA2023", "Якубович Денис Андреевич",
    24, AccessRightsType.Admin);
Console.WriteLine(denis + "\n");

Console.WriteLine($"Фамилия: {denis.GetLastName()}");
Console.WriteLine($"Имя: {denis.GetFirstName()}");
Console.WriteLine($"Отчество: {denis.GetPatronymic()}");
```

Рис. 3.30. Фрагмент кода для создания экземпляра класса `Account` и использование его методов обработки ФИО

```
PS D:\Documents\Programming\NET\2024\WebSystem> dotnet run
ID: 10345
Логин: YDA2023
ФИО: Якубович Денис Андреевич
Возраст: 24
Права доступа: администратор

Фамилия: Якубович
Имя: Денис
Отчество: Андреевич
```

Рис. 3.31. Результат работы программы из задания 7

Задание 8

1. Создайте каталог *Account_7*.
2. Используйте в качестве основы код из *Account_6*.
3. Расширьте описание класса Account методами, которые заполняют поля:
   - `InitAccount()` – осуществляет обработку, аналогичную методу `SetData()`;
   - `InitAdmin()` – инициализирует администратора;
   - `InitManager()` – инициализирует менеджера;
   - `InitEmployee()` – инициализирует штатного сотрудника;
   - `InitGuest()` – инициализирует аккаунт с правами гостя.
4. Оптимизируйте код, используя ранее созданный метод `SetData()`. Сделайте его закрытым методом в классе и переименуйте в `_SetData()`.

5. Внешний доступ должен осуществляться через обозначенные 5 методов.

```
var denis = new Account();
denis.InitAdmin(10345, "YDA2023", "Якубович Денис Андреевич", 24);

var olga = new Account();
olga.InitManager(20246, "olalaaa", "Лесных Ольга Аркадиевна", 27);

var misha = new Account();
misha.InitEmployee(30023, "miha_krapiva", "Крапивницкий Михаил Эдуардович", 31);

var nastya = new Account();
nastya.InitGuest(20108, "nastusha.flower", "Цветкова Анастасия Филимоновна", 28);

var vladislav = new Account();
vladislav.InitAccount(30532, "vlad-tankist19", "Ермоленко Владислав Николаевич", 37,
    AccessRightsType.Employee);


Console.WriteLine($"{denis}\n");
Console.WriteLine($"{olga}\n");
Console.WriteLine($"{misha}\n");
Console.WriteLine($"{nastya}\n");
Console.WriteLine($"{vladislav}");
```

Рис. 3.32. Фрагмент кода для создания экземпляров класса `Account` и их инициализации

```
PS D:\Documents\Programming\NET\2024\WebSystem> dotnet run
ID: 10345
Логин: YDA2023
ФИО: Якубович Денис Андреевич
Возраст: 24
Права доступа: администратор

ID: 20246
Логин: olalaaa
ФИО: Лесных Ольга Аркадиевна
Возраст: 27
Права доступа: менеджер

ID: 30023
Логин: miha_krapiva
ФИО: Крапивницкий Михаил Эдуардович
Возраст: 31
Права доступа: менеджер

ID: 20108
Логин: nastusha.flower
ФИО: Цветкова Анастасия Филимоновна
Возраст: 28
Права доступа: менеджер

ID: 30532
Логин: vlad-tankist19
ФИО: Ермоленко Владислав Николаевич
Возраст: 37
Права доступа: сотрудник
```

Рис. 3.33. Результат работы программы из задания 8

3.3. Конструктор класса

3.3.1. Назначение конструктора класса

Методы инициализации

На данном этапе в класс `Student` добавлен ряд методов, которые отвечают за заполнение полей объекта значениями, т.е. за его инициализацию: `InitBachelor()`, `InitSpecialist()`, `InitMaster()`, `InitStudent()`. Каждый из этих методов можно вызывать многократно, что позволяет перезаписывать атрибуты одного и того же объекта любое число раз.

Однако в реальных задачах такая возможность далеко не всегда необходима. Часто объекты должны представлять собой уникальные сущности и определяться постоянными характеристиками, которые задаются один раз и, как правило – в момент создания объекта. А методы объекта используют поля для извлечения данных или преобразования для других целей.

Понятие конструктора класса

Определение

Конструктор класса *– это компонент класса, вызываемый при создании экземпляра этого класса.*

```
Student denis = new Student();
```

Правая часть выражения – это конструктор объекта.

Обычно конструктор используется для задания первоначальных значений объектам, или же для выполнения любых других установочных процедур, которые требуются для создания объекта.

Описание конструктора очень похоже на описание метода класса, однако:

- у конструктора название такое же, как и у класса;
- конструктор не имеет возвращаемого типа.

3.3.2. Конструктор по умолчанию

У любого класса имеется **конструктор по умолчанию**, независимо от того, определите вы его в классе самостоятельно или нет. Этот конструктор не имеет параметров и в момент создания экземпляра класса устанавливает его полям некоторые значения по умолчанию, например, для числовых типов нуль, для логического `false`, для ссылочных – значение `null`.

```
Student denis = new Student();
Console.WriteLine(denis);
```

```
ID: 0
Имя:
Фамилия:
Возраст: 0
Уровень образования: бакалавриат
```

Рис. 3.34. Конструктор по умолчанию

Однако можно переопределить конструктор по умолчанию. Например – задать значения некоторым (или всем) полям.

В этом случае изначально поля каждого создаваемого экземпляра класса получат копии этих значений.

```
class Student
{
    // Поля класса...

    public Student()
    {
        FirstName = "none";
        LastName = "none";
    }

    // Методы класса...
}
```

```
ID: 0
Имя: none
Фамилия: none
Возраст: 0
Уровень образования: бакалавриат
```

Рис. 3.35. Переопределение конструктора по умолчанию

3.3.3. Параметризированный конструктор

Создание конструктора с параметрами

На практике больший интерес вызывает **параметризированный конструктор**, который позволяет заполнять поля конкретными значениями.

Как и в случае метода, данные передаются через параметры. Реализация этого конструктора полностью аналогична методу `_SetInfo()`, поскольку перед конструктором стоит аналогичная задача.

```
class Student
{
    // Поля класса...

    public Student(
        long id, string? first_name, string? last_name,
        int age, EducationalDegree degree
    )
    {
        Id = id;
        FirstName = first_name;
        LastName = last_name;
        Age = (16 <= age) && (age <= 120) ? age : 18;
        Degree = degree;
    }

    // Методы класса...
 }
```

Поскольку C# поддерживает **перегрузку конструкторов**, то в классе можно описывать разные варианты конструктора.

Для создания объекта `andrew` вызываем конструктор по умолчанию, а для `denis` – пареметризированный конструктор.

```
Student andrew = new Student();
Console.WriteLine(andrew);

Console.WriteLine();

Student denis = new Student(12345, "Денис", "Якубович",
    22, EducationalDegree.Specialist);
Console.WriteLine(denis);
```

```
ID: 0
Имя: none
Фамилия: none
Возраст: 0
Уровень образования: бакалавриат

ID: 12345
Имя: Денис
Фамилия: Якубович
Возраст: 22
Уровень образования: специалитет
```

Рис. 3.36. Параметризированный конструктор

Полный код программы (для краткости уберем методы инициализации из класса `Student`):

Глава 3\Institute_6\EducationalDegree.cs

```
namespace Institute
{
    enum EducationalDegree
    {
        Bachelor,
        Specialist,
        Master
    }
}
```

Глава 3\Institute_6\Student.cs

```
namespace Institute
{
    class Student
    {
        public long Id;
        public string? FirstName = "";
        public string? LastName = "";
        public int Age;
        public EducationalDegree Degree;

        public Student()
        {
            FirstName = "none";
            LastName = "none";
        }
```

```
        public Student(
            long id, string? first_name, string? last_name,
            int age, EducationalDegree degree
        )
        {
            Id = id;
            FirstName = first_name;
            LastName = last_name;
            Age =  (16 <= age) && (age <= 120) ? age : 18;
            Degree = degree;
        }

        public override string ToString() =>
            $"ID: {Id}\n
            Имя: {FirstName}\n
            Фамилия: {LastName}\n
            Возраст: {Age}\n
            Уровень образования: {GetEducationalType()}";

        public bool IsAdult() =>
            Age >= 18;

        public string GetEducationalType() =>
            Degree switch
            {
                EducationalDegree.Bachelor => "бакалавриат",
                EducationalDegree.Specialist => "специалитет",
                EducationalDegree.Master => "магистратура",
                _ => "не определено"
            };
    }
}
```

Глава 3\Institute_6\Program.cs

```
using System;

namespace Institute
{
    class Program
    {
        static void Main(string[] args)
        {
            Student andrew = new Student();
```

```
            Console.WriteLine(andrew);

            Console.WriteLine();

            Student denis = new Student(12345, "Денис",
                "Якубович", 22, EducationalDegree.Specialist);
            Console.WriteLine(denis);
        }
    }
}
```

Это важно знать!

Когда создан конструктор с параметрами, то необходимо также явно описать и конструктор по умолчанию. Если ему не требуется задавать какой либо реализации, достаточно оставить пустыми скобки его тела.

Пример взаимодействия конструктора с методами

Упростим реализацию конструкторов класса `Student`, используя закрытый обслуживающий метод `_SetInfo()`.

Кроме того, добавим конструктор, который создает студента по заданной фамилии и имени, а остальные поля оставим по умолчанию.

Глава 3\Institute_7\EducationalDegree.cs

```
namespace Institute
{
    enum EducationalDegree
    {
        Bachelor,
        Specialist,
        Master
    }
}
```

Глава 3\Institute_7\Student.cs

```
namespace Institute
{
    class Student
    {
```

```
    public long Id;
    public string? FirstName = "";
    public string? LastName = "";
    public int Age;
    public EducationalDegree Degree;

    public Student() =>
        _SetInfo(0, "none", "none", 18,
            EducationalDegree.Bachelor);

    public Student(
        string? first_name,
        string? last_name
    ) =>
        _SetInfo(0, first_name, last_name, 18,
            EducationalDegree.Bachelor);

    public Student(
       long id, string? first_name,
       string? last_name, int age,
       EducationalDegree degree
    ) =>
        _SetInfo(id, first_name, last_name, age, degree);

    private void _SetInfo(
        long id, string? first_name, string? last_name,
        int age, EducationalDegree degree
    )
    {
        Id = id;
        FirstName = first_name;
        LastName = last_name;
        Age = (16 <= age) && (age <= 120) ? age : 18;
        Degree = degree;
    }

    public void InitBachelor(
        long id, string? first_name, string? last_name,
        int age
    ) =>
        _SetInfo(id, first_name, last_name, age,
            EducationalDegree.Bachelor);
```

```
        public void InitSpecialist(
            long id, string? first_name, string? last_name,
            int age
        ) =>
            _SetInfo(id, first_name, last_name, age,
                EducationalDegree.Specialist);

        public void InitMaster(
            long id, string? first_name, string? last_name,
            int age
        ) =>
            _SetInfo(id, first_name, last_name, age,
                EducationalDegree.Master);

        public void InitStudent(
            long id, string? first_name, string? last_name,
            int age, EducationalDegree degree
        ) =>
            _SetInfo(id, first_name, last_name, age, degree);

        public override string ToString() =>
            $"ID: {Id}\n
            Имя: {FirstName}\n
            Фамилия: {LastName}\n
            Возраст: {Age}\n
            Уровень образования: {GetEducationalType()}";

        public bool IsAdult() =>
            Age >= 18;

        public string GetEducationalType() =>
            Degree switch
            {
                EducationalDegree.Bachelor => "бакалавриат",
                EducationalDegree.Specialist => "специалитет",
                EducationalDegree.Master => "магистратура",
                _ => "не определено"
            };
    }
}
```

Глава 3\Institute_7\Program.cs

```
using System;

namespace Institute
{
    class Program
    {
        static void Main(string[] args)
        {
            Student andrew = new Student();
            Console.WriteLine(andrew + "\n");

            Student olga = new Student("Ольга", "Березкина");
            Console.WriteLine(olga + "\n");

            Student denis = new Student(12345, "Денис",
                "Якубович", 22, EducationalDegree.Specialist);
            Console.WriteLine(denis);
        }
    }
}
```

Ранее созданный обслуживающий и закрытый внутри класса метод `_SetInfo()` заполняет поля во всех трех вариациях конструктора. Необходимые значения передаются внешне через параметры, а фиксированные устанавливаются напрямую.

В основной части программы создаем объекты конструктором по умолчанию, конструктором с двумя параметрами и пятью (наиболее полный).

3.3.4. Динамическое выделение памяти

На основании указанного материала можно точно описать, каким образом работает команда по созданию экземпляра класса с помощью конструктора класса:

```
Student denis = new Student();
```

1. В начале создается переменная `denis` ссылочного типа `Student`. Сама переменная не является объектом, но может ссылаться на объект этого типа (в ОЗУ).

2. Оператор `new` выделяет необходимый объем памяти для объекта.
3. Операция `Student()` – это конструктор (может быть по умолчанию или с параметрами).
4. Ссылка `denis` связывается с объектом в оперативной памяти и в дальнейшем мы работаем через нее как удобный интерфейс переменной.

3.3.5. Инициализаторы объекта

При создании экземпляра класса необязательно использовать вызов конструктора в описанной форме. C# поддерживает альтернативный синтаксис, называемый **инициализатором объекта**.

Для этого используют фигурные скобки, а значения в поля задаются с указанием их имени.

```
Student denis = new Student(12345, "Денис", "Якубович",
    22, EducationalDegree.Specialist);
```

можно создать объект следующим образом:

```
Student denis = new Student
{
    Id = 12345,
    FirstName = "Денис",
    LastName = "Якубович",
    Age = 22,
    Degree = EducationalDegree.Specialist
};
```

Несмотря на то, что такая инициализация кажется более трудоемкой, значения передаваемых параметров очевидны. В таком случае поля можно заполнять в любом порядке.

Инициализаторы объекта очень полезны, когда требуется создать объект в процессе некоторой операции, или объект, который собирается из компонент разных классов.

Также инициализаторы объекта упрощают анализ структуры объекта, особенно если она «древовидная» (т.е. поля описываются классами, имеющими вложенную структуру).

Вопросы для самопроверки

1. Почему возможность многократно менять свойства объекта не всегда рационально или безопасно?
2. Что представляет собой конструктор класса и чем он отличается от метода?
3. Какие особенности работы с конструктором по умолчанию следуюет учитывать?
4. Почему часто приходиться создавать разные перегрузки конструкторов класса?
5. Приведите примеры, когда конструктор класса может использовать описанные в классе методы.

Практикум

Задание 1

1. Создайте проект *Institute_6* в каталоге *Institute*.
2. Последовательно повторите и воспроизведите рассмотренный в параграфе пример.
3. Проследите, что произойдет, если убрать конструктор по умолчанию или оставить пустым его теле (рис. 3.37).

Задание 2

1. Создайте проект *Institute_7* в каталоге *Institute*.
2. Последовательно повторите и воспроизведите рассмотренный в параграфе пример (рис. 3.38).
3. Постарайтесь понять, каким образом осуществляется взаимодействие конструкторов класса с методом `_SetInfo()`. Можно ли обойтись без него?

Задание 3

1. Создайте проект *Account_8* в каталоге *WebSystem*.
2. Опишите в классе `Account` конструктор по умолчанию и параметризированный конструктор. Рекомендуется использовать закрытый метод `_SetData()` для обслуживания конструкторов (рис. 3.39, рис. 3.40).
3. Используйте идеи, которые были заложены в проекте *Institute_7.*

```
PS D:\Documents\Programming\NET\2024\Institute> dotnet run
ID: 0
Имя: none
Фамилия: none
Возраст: 0
Уровень образования: бакалавриат

ID: 12345
Имя: Денис
Фамилия: Якубович
Возраст: 22
Уровень образования: специалитет
```

Рис. 3.37. Результат работы программы из задания 1

```
PS D:\Documents\Programming\NET\2024\Institute> dotnet run
ID: 0
Имя: none
Фамилия: none
Возраст: 18
Уровень образования: бакалавриат

ID: 0
Имя: Ольга
Фамилия: Березкина
Возраст: 18
Уровень образования: бакалавриат

ID: 12345
Имя: Денис
Фамилия: Якубович
Возраст: 22
Уровень образования: специалитет
```

Рис. 3.38. Результат работы программы из задания 2

```
var denis = new Account();
var olga = new Account(
    20246, "olalaaa", "Лесных Ольга Аркадиевна",
    27, AccessRightsType.Employee
);
```

Рис. 3.39. Пример создания экземпляров класса конструктором по умолчанию и параметризированным конструктором

```
PS D:\Documents\Programming\NET\2024\WebSystem> dotnet run
ID: 0
Логин: none
ФИО: none
Возраст: 18
Права доступа: гость

ID: 20246
Логин: olalaaa
ФИО: Лесных Ольга Аркадиевна
Возраст: 27
Права доступа: сотрудник
```

Рис. 3.40. Результат работы программы из задания 3

3.4. Классы как ссылочные типы данных. Ссылка this

3.4.1. Переменная ссылочного типа

Переменная как ссылка на объект

Классы C# относятся к **ссылочным типам** данных. Это означает, что переменная класса хранит не значение объекта, а является **ссылкой** на него.

Так, если присвоить одной ссылочной переменной другую того же типа (т.е. попытаться скопировать), то обе переменные станут ссылаться на один объект:

```
Student denis = new Student(12345, "Денис", "Якубович",
    22, EducationalDegree.Specialist);

Student oleg = denis;

Console.WriteLine($"Возраст Дениса: {denis.Age}");
Console.WriteLine($"Возраст Олега: {oleg.Age}\n");

denis.Age = 25;

Console.WriteLine($"Возраст Дениса: {denis.Age}");
Console.WriteLine($"Возраст Олега: {oleg.Age}");
```

В примере в переменную `andrew` копируется ссылка на объект, на который ссылается переменная `denis`. Теперь обе переменные ссылаются на один и тот же объект! Это означает, что изменение значения любого поля объекта `denis` автоматически меняет соответствующее поле объекта `andrew`. Аналогично и наоборот.

```
Возраст Дениса: 22
Возраст Олега: 22

Возраст Дениса: 25
Возраст Олега: 25
```

Рис. 3.41. Особенность копирования ссылки на объект

Создание нового объекта

Чтобы разорвать связь ссылок, экземпляру `oleg` необходимо вызвать конструктор класса, т.е. привязать эту ссылку к новому, созданному конструктором объекту:

```
Student denis = new Student(12345, "Денис", "Якубович",
    22, EducationalDegree.Specialist);
Student oleg = denis;

oleg = new Student(20034, "Олег", "Нечаев",
    19, EducationalDegree.Bachelor);

denis.Age = 25;

Console.WriteLine($"Возраст Дениса: {denis.Age}");
Console.WriteLine($"Возраст Олега: {oleg.Age}");
```

```
Возраст Дениса: 25
Возраст Олега: 19
```

Рис. 3.42. Привязка ссылки к новому объекту

3.4.2. Ссылка this

Ссылка на экземпляр класса

В языке C# имеется команда **`this`**, которая является **ссылкой на текущий объект** внутри класса. Ее можно указывать явно для полей, методов и конструктора, а также в том случае, когда необходимо получить ссылку на весь этот объект. Для каждого нового объекта класса `this` всегда ссылается на него.

В следующем фрагменте кода с помощью `this` подчеркивается, что в методе `_SetInfo()` слева ведется обращение к полям объекта:

```
class Student
{
    // ...
    private void _SetInfo(
        long id, string? first_name, string? last_name,
        int age, EducationalDegree degree
    )
    {
        this.Id = Math.Abs(id);
```

```
            this.FirstName = first_name;
            this.LastName = last_name;
            this.Age = (16 <= age) && (age <= 120)
                       ? age
                       : 18;
            this.Degree = degree;
        }

        public override string ToString() =>
            $"ID: {Id}\n
            Имя: {FirstName}\n
            Фамилия: {LastName}\n
            Возраст: {Age}\n
            Уровень образования:
                {this.GetEducationalType()}";

        // ...
    }
```

В текущем контексте `this` лишь подчеркивает, что ведется работа с полями или методами объекта. Однако внутри класса указывать `this` в явной форме излишне, поскольку поля и методы здесь видны глобально.

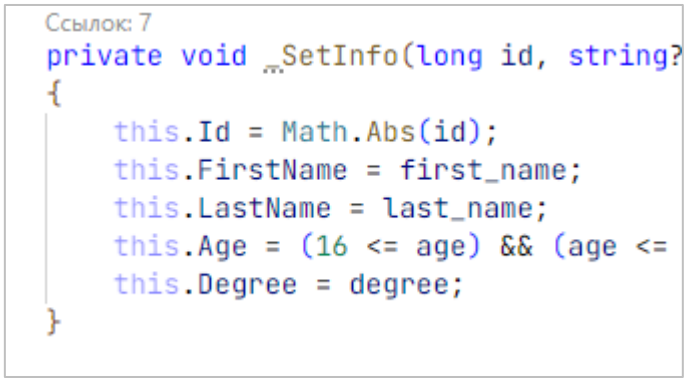


Рис. 3.43. Среда разработки подсвечивает `this` как необязательный компонент

Явное использование this

Одно из возможных применений `this` состоит в том, чтобы уточнить, что переменная является полем или методом класса, а не локальной переменной внутри метода или конструктора. Такая ситуация возникнет, когда разработчик решит задать полям и параметрам в методах и конструкторе одинаковые названия.

Предположим, что поля в классе названы так же, как и параметры методов. Тогда, если убрать `this`, возникает неопределенность, какую переменную считать полем, а какую – параметром метода.

В этом случае явное указание `this` устраняет неоднозначность: слева работаем с полем объекта, справа – с параметром метода (однако в нашей реализации такой проблемы совпадающих имен нет, поэтому `this` и не указываем):

```
private void _SetInfo(
    long id, string? first_name, string? last_name,
    int age, EducationalDegree degree
)
{
    this.id = Math.Abs(id);
    this.first_name = first_name;
    this.last_name = last_name;
    this.age = (16 <= age) && (age <= 120) ? age : 18;
    this.degree = degree;
}
```

3.4.3. Метод для возврата копии объекта

Попытка копирования с помощью this

Пусть в классе `Student` требуется реализовать метод `Copy()`, который возвращает независимую копию текущего объекта.

Поскольку метод должен возвратить копию объекта, то тип возвращаемого значения должен быть `Student`, как и сам объект, копия которого требуется.

Изначально можно пойти некорректным путем и попробовать возвратить объект `this`.

```
class Student
{
    // Другие члены класса...

    public Student Copy() => this;
}
```

Однако копирование с помощью метода `Copy()` будет не таким, как требовалось:

```
Student denis = new Student(12345, "Денис", "Якубович",
    22, EducationalDegree.Specialist);

Student denis_clone = denis.Copy();

Console.WriteLine($"Возраст оригинала: {denis.Age}");
Console.WriteLine($"Возраст клона: {denis_clone.Age}\n");

denis.Age = 25;

Console.WriteLine($"Возраст оригинала: {denis.Age}");
Console.WriteLine($"Возраст клона: {denis_clone.Age}");
```

```
Возраст оригинала: 22
Возраст клона: 22

Возраст оригинала: 25
Возраст клона: 25
```

Рис. 3.44. Переменные все равно ссылаются на один и тот же объект

В таком случае `this` возвратит только ссылку на объект, относительно которого вызывается метод `Copy()`. Поэтому и `denis`, и `denis_clone` будут ссылаться на один объект.

На самом деле вызов метода

```
Student denis_clone = denis.Copy();
```

приводит к такому же результату, что и обычное присваивание ссылок:

```
Student denis_clone = denis;
```

Для создания независимой копии потребуется создавать новый объект с помощью конструктора.

Копирование посредством создания нового объекта

В следующем примере копирование осуществляется через вызов конструктора, которому передаются значения полей текущего объекта. Поскольку параметры метода передаются в качестве копий, а конструктор выделяет память под новый объект, метод `Copy()` возвращает уже независимую копию объекта.

Глава 3\Institute_8\EducationalDegree.cs

```
namespace Institute
{
    enum EducationalDegree
    {
        Bachelor,
        Specialist,
        Master
    }
}
```

Глава 3\Institute_8\Student.cs

```
namespace Institute
{
    class Student
    {
        public long Id;
        public string? FirstName = "";
        public string? LastName = "";
        public int Age;
        public EducationalDegree Degree;

        public Student() =>
            _SetInfo(0, "none", "none", 18,
                EducationalDegree.Bachelor);

        public Student(
            string? first_name, string? last_name
        ) =>
            _SetInfo(0, first_name, last_name, 18,
                EducationalDegree.Bachelor);

        public Student(
            long id, string? first_name, string? last_name,
            int age, EducationalDegree degree
        ) =>
            _SetInfo(id, first_name, last_name, age, degree);

        private void _SetInfo(
            long id, string? first_name, string? last_name,
            int age, EducationalDegree degree
        )
        {
```

```
        Id = Math.Abs(id);
        FirstName = first_name;
        LastName = last_name;
        Age = (16 <= age) && (age <= 120) ? age : 18;
        Degree = degree;
    }

    public void InitBachelor(
        long id, string? first_name, string? last_name,
        int age
    ) =>
        _SetInfo(id, first_name, last_name, age,
            EducationalDegree.Bachelor);

    public void InitSpecialist(
        long id, string? first_name, string? last_name,
        int age
    ) =>
        _SetInfo(id, first_name, last_name, age,
            EducationalDegree.Specialist);

    public void InitMaster(
        long id, string? first_name, string? last_name,
        int age
    ) =>
        _SetInfo(id, first_name, last_name, age,
            EducationalDegree.Master);

    public void InitStudent(
        long id, string? first_name, string? last_name,
        int age, EducationalDegree degree
    ) =>
        _SetInfo(id, first_name, last_name, age, degree);

    public override string ToString() =>
        $"ID: {Id}\n
        Имя: {FirstName}\n
        Фамилия: {LastName}\n
        Возраст: {Age}\n
        Уровень образования: {GetEducationalType()}";

    public bool IsAdult() =>
        Age >= 18;
```

```
        public string GetEducationalType() =>
            Degree switch
            {
                EducationalDegree.Bachelor => "бакалавриат",
                EducationalDegree.Specialist => "специалитет",
                EducationalDegree.Master => "магистратура",
                _ => "не определено"
            };

        public Student Copy() =>
            new Student(Id, FirstName, LastName, Age, Degree);
    }
}
```

Глава 3\Institute_8\Program.cs

```
using System;

namespace Institute
{
    class Program
    {
        static void Main(string[] args)
        {
            Student denis = new Student(12345, "Денис",
                "Якубович", 22, EducationalDegree.Specialist);

            Student denis_clone = denis.Copy();

            Console.WriteLine($"Возраст оригинала:
                {denis.Age}");
            Console.WriteLine($"Возраст клона:
                {denis_clone.Age}\n");

            denis.Age = 25;

            Console.WriteLine($"Возраст оригинала:
                {denis.Age}");
            Console.WriteLine($"Возраст клона:
                {denis_clone.Age}");
        }
    }
}
```

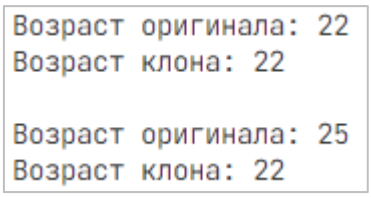


Рис. 3.45. Теперь копия действительно независима

Вообще говоря, создание полных копий объектов ссылочных типов называется **глубоким копированием**. И в более общих случаях, когда приходится работать с объектами классов сложной структуры, оно требует отдельной проработки.

3.4.4. Цепочки методов

У ссылки `this` имеется важное практическое применение: она позволяет выстраивать вызов методов в **цепочки**.

Пусть требуется метод, который будет менять значение `Id` студента на указанный в параметре:

```
class Student
{
    // Другие члены класса...

    public void ChangeId(long id)
    {
        Id = Math.Abs(id);
    }
}
```

В таком случае операции требуют пошагового выполнения:

```
denis.ChangeId(20012);
Console.WriteLine($"Новое ID: {denis.Id}");
```

В примере сначала меняем `Id`. Далее используем его в операциях.

Расширим возможности метода, используя ссылку `this`.

Пусть теперь метод не только меняет поле, но и дополнительно возвращает ссылку на объект через `this`.

Метод будет возвращать ссылку на объект типа `Student` (в данном случае ссылаемся на тот же объект):

```
class Student
{
    // Другие члены класса...

    public Student ChangeId(long id)
    {
        Id = Math.Abs(id);

        return this;
    }
}
```

Теперь можно организовать цепочку вызова методов и скомбинировать несколько операций в одну:

```
Console.WriteLine(
    $"Новое ID: {denis.ChangeId(20012).Id}"
);
```

Сначала метод `ChangeID()` меняет `Id` студента и возвращает ссылку на тот же объект `denis`. А это позволяет продолжить обращение к тому же объекту: например, обратиться к другому методу или полю.

При этом мы также оставляем возможность работать поэтапно:

```
denis.ChangeId(20012);
Console.WriteLine($"Новое ID: {denis.Id}");
```

Несмотря на то, что `ChangeId()` не только меняет значение поля `Id`, но и возвращает ссылку на `denis`, его необязательно присваивать в переменную (т.е. можно использовать не как функцию, а как процедуру).

Это полезно знать!

Описывая подобным образом методы с «побочным эффектом» их вызов можно выстраивать в цепочки в любом порядке. Это не является обязательным, но часто позволяет расширить гибкость работы с методами класса.

3.4.5. Цепочки конструкторов

При реализации нескольких конструкторов в классе часто рационально выделить один главный. Как правило, все остальные конструкторы имеют аналогичную реализацию, но с другими значениями.

Так, в классе `Student` наиболее полным является третий конструктор, которому при создании объекта передаются все 5 параметров:

```
public Student() =>
    _SetInfo(0, "none", "none", 18,
        EducationalDegree.Bachelor);

public Student(string? first_name, string? last_name) =>
    _SetInfo(0, first_name, last_name, 18,
        EducationalDegree.Bachelor);

public Student(
    long id, string? first_name, string? last_name,
    int age, EducationalDegree degree
) =>
    _SetInfo(id, first_name, last_name, age, degree);
```

Однако здесь реализация уже упрощена, поскольку используется вспомогательный закрытый метод `_SetInfo()`, управляющий записью в поля.

Ссылка `this` допускает еще одно применение – проектирование **цепочки конструкторов**.

Шаблон работает следующим образом. В начале реализуется наиболее полный конструктор (главный). Далее в частичных конструкторах через двоеточие вызывается ссылка **`this`** с передаваемыми главному конструктору параметрами. С точки зрения синтаксиса C# двоеточие означает **наследование**, т.е. каждый конструктор наследует главный с указанными значениями параметров.

```
public Student() :
    this(0, "none", "none", 18,
        EducationalDegree.Bachelor) {}

public Student(string? first_name, string? last_name) :
    this(0, first_name, last_name, 18,
        EducationalDegree.Bachelor) {}
```

```
public Student(
    long id, string? first_name, string? last_name,
    int age, EducationalDegree degree
) =>
    _SetInfo(id, first_name, last_name, age, degree);
```

В примере главный конструктор (последний) – содержит полную реализацию логики инициализации полей (в данном случае через прослойку вспомогательного метода `_SetInfo()`). Частичные конструкторы наследуют главный. Т.е. они сначала выполняют его при заданных параметрах, а далее могут выполнить и другой код, указанный в теле (в нашем случае это не требуется, поэтому скобки тела конструктора 1 и 2 остаются пустыми).

Вопросы для самопроверки

1. Опишите особенности переменных ссылочного типа.
2. Почему при копирование переменных типа класса требует отдельных методов?
3. Приведите примеры, когда явное обращение к ссылке `this` необходимо с технической точки зрения.
4. Для каких целей методы выстраиваются в цепочки?
5. Что представляет собой цепочка конструкторов и как ее организовать?

Практикум

Задание 1

1. Создайте проект *Institute_8 в каталоге Institute*.
2. Последовательно повторите и воспроизведите рассмотренный в параграфе пример.
3. Проследите за логикой зависимости в конструкторах.

Задание 2

1. Создайте проект *Account_9 в каталоге WebSystem*.
2. В классе `Account` реализовать метод `Copy()`, который возвращает независимую копию текущего объекта.
3. Проверьте его работу (рис. 3.46)

```
Права доступа у оригинала: администратор
Права доступа у копии: администратор

Меняем права одному из пользователей
Теперь у оригинала права: менеджер
Теперь у копии права: администратор
```

Рис. 3.46. Результат работы программы из задания 2

Задание 3

1. Создайте проект *Account_10* в каталоге *WebSystem*.
2. Преобразуйте методы инициализации так, чтобы появилась возможность организовывать цепочки вызова методов. Например, следующим образом (рис. 3.47).
3. Учтите зависимости (это позволит упростить код).

```
Console.WriteLine(
    new Account(10345, "YDA2023", "Якубович Денис Андреевич",24, AccessRightsType.Admin).Login
);
```

Рис. 3.47. Цепочка вызова для конструктора класса

```
Логин пользователя 'Денис':
YDA2023
```

Рис. 3.48. Результат работы программы из задания 3

Задание 4

1. Создайте проект *Account_11* в каталоге *WebSystem*.
2. За основу взять реализацию задачи *Account_9*.
3. Скорректируйте конструктор по умолчанию, используя подход с ссылкой `this` и шаблоном вызова цепочки конструкторов. Добавьте конструктор, который способен создавать аккаунт только по наличию `Id` и логина.

```
PS D:\Documents\Programming\NET\2024\WebSystem> dotnet run
ID: 10345
Логин: YDA2023
ФИО: Якубович Денис Андреевич
Возраст: 24
Права доступа: администратор

ID: 20623
Логин: newacc
ФИО: none
Возраст: 18
Права доступа: гость
```

Рис. 3.49. Результат работы программы из задания 4

3.5. Статические компоненты класса

3.5.1. Необходимость общедоступных компонент

Поля и методы класса

При создании экземпляров класса предполагается, что каждый объект будет наделен собственными значениями компонент. Однако часто в описании класса необходимы общедоступные компоненты, которыми можно манипулировать в независимости от состояния объектов.

Для этих целей компоненты делают **статическими**, т.е. принадлежащими самому классу, а не его экземплярам. Статический компонент помечается модификатором **`static`**.

Предположим, требуется задать класс `Family`, описывающий членов семьи и операции с семейным бюджетом. Ограничимся полями, задающими ФИО и возраст члена семьи. Добавим два конструктора: по умолчанию и с параметрами. Переопределим для получения общей информации о члене семьи.

Глава 3\StaticModifier_1\Family.cs

```
namespace StaticModifier
{
    class Family
    {
        public string? Full_name = "";
        public int Age;

        public Family() { }

        public Family(string? full_name, int age)
        {
            Full_name = full_name;
            Age = age;
        }

        public override string ToString() =>
            $"{Full_name}\t{Age}";
    }
}
```

Глава 3\StaticModifier_1\Program.cs

```
using System;

namespace StaticModifier
{
    class Program
    {
        static void Main(string[] args)
        {
            Family father = new Family(
                "Карпов Олег Николаевич", 41
            );
            Family mother = new Family(
                "Карпова Лариса Андреевна", 39
            );
            Family son = new Family(
                "Карпов Валентин Олегович", 16
            );
            Family daughter = new Family(
                "Карпова Анна Олеговна", 17
            );

            Console.WriteLine($"Отец: {father}");
            Console.WriteLine($"Мать: {mother}");
            Console.WriteLine($"Сын: {son}");
            Console.WriteLine($"Дочь: {daughter}");
        }
    }
}
```

```
Отец: Карпов Олег Николаевич    41
Мать: Карпова Лариса Андреевна  39
Сын: Карпов Валентин Олегович   16
Дочь: Карпова Анна Олеговна     17
```

Рис. 3.50. Простой пример описания класса и создания нескольких экземпляров

Пусть в классе `Family` требуется поле, хранящее бюджет семьи (назовем далее `Budget`). Однако привязывать такое поле к каждому объекту нерационально: бюджет относится ко всей семье, а не к ее конкретному представителю. Это поле должно приобрести некоторую общедоступность и быть независимым от объектов класса `Family`.

Поле следует связать с самим классом, используя модификатор `static`.

Модификатор static

Модификатор **static** обозначает, что компонент принадлежит всему классу, а не конкретному экземпляру этого класса. Иными словами, если сам класс считать объектом, то static-компонента класса (поле, метод или конструктор) принадлежит этому объекту.

С другой стороны, поскольку static-компонента принадлежит самому классу, то она существует в единственном экземпляре. При этом не имеет значения, созданы ли экземпляры этого класса.

3.5.2. Статические поля

Поле класса делают статическим, если:

- оно должно принадлежать всему классу, а не его конкретному экземпляру;
- поле должно хранить общую для всего класса информацию.

В обозначенном выше примере *Family_1* бюджет рационально хранить именно в статическом поле.

Глава 3\StaticModifier_2\Family.cs

```
namespace StaticModifier
{
    class Family
    {
        public string? Full_name = "";
        public int Age;
        public static double Budget;

        public Family() { }

        public Family(string? full_name, int age)
        {
            Full_name = full_name;
            Age = age;
        }

        public override string ToString() =>
            $"{Full_name}\t{Age}";
    }
}
```

Глава 3\StaticModifier_2\Program.cs

```
using System;

namespace StaticModifier
{
    class Program
    {
        static void Main(string[] args)
        {
            Family father = new Family(
                "Карпов Олег Николаевич", 41
            );
            Family mother = new Family(
                "Карпова Лариса Андреевна", 39
            );
            Family son = new Family(
                "Карпов Валентин Олегович", 16
            );
            Family daughter = new Family(
                "Карпова Анна Олеговна", 17
            );

            Family.Budget = 472000;

            Console.WriteLine($"Отец: {father}");
            Console.WriteLine($"Мать: {mother}");
            Console.WriteLine($"Сын: {son}");
            Console.WriteLine($"Дочь: {daughter}");
            Console.WriteLine($"Бюджет: {Family.Budget}");
        }
    }
}
```

```
Отец: Карпов Олег Николаевич    41
Мать: Карпова Лариса Андреевна  39
Сын: Карпов Валентин Олегович   16
Дочь: Карпова Анна Олеговна     17
Бюджет: 472000
```

Рис. 3.51. Работа со статическим полем

В приведенном примере поле `Budget` связываем с классом с помощью модификатора `static`. Обращение к статическому полю ведется от имени класса: оно принадлежит ему и существует в единственном экземпляре.

Важное замечание!

Существует только одна копия статического поля ***Budget****, независимо от того, сколько будет создано экземпляров класса до / после. Статическое поле инициализируются перед первым обращением к классу, даже если еще не создано ни одного экземпляра класса.*

3.5.3. Статические методы

Статические методы в обработке статических полей

Наряду со статическими полями возникает естественная необходимость и в статических методах.

Метод рационально описывать статическим, если:

- метод должен принадлежать классу, а не создаваемым на его основе экземплярам;
- метод выполняет задачу общего (глобального) плана, не требующую данных экземпляров (но дополнительные данные могут быть переданы в параметрах методу).

Разумеется, часто статические методы нужны, чтобы работать со статическими полями.

Добавим в класс `Family` два статических метода:

- `GetBudget()` – возвращает величину текущего бюджета (т.е. значение поля `Budget`);
- `AddBudget()` – наращивает семейный бюджет на указанную сумму.

Глава 3\StaticModifier_3\Family.cs

```
namespace StaticModifier
{
    class Family
    {
        public string? Full_name = "";
        public int Age;
        public static double Budget;

        public Family() { }
```

```
        public Family(string? full_name, int age)
        {
            Full_name = full_name;
            Age = age;
        }

        public override string ToString() =>
            $"{Full_name}\t{Age}";

        public static double GetBudget() =>
            Budget;

        public static void AddBudget(double sum) =>
            Budget += sum;
    }
}
```

Глава 3\StaticModifier_3\Program.cs

```
using System;

namespace StaticModifier
{
    class Program
    {
        static void Main(string[] args)
        {
            Family.Budget = 472000;
            Console.WriteLine(
                $"Бюджет в начале: {Family.GetBudget()}"
            );
            Family.AddBudget(50000);
            Family.AddBudget(43000);
            Console.WriteLine(
                $"Бюджет после вложений: {Family.GetBudget()}"
            );
        }
    }
}
```

Оба `static`-метода работают со статическим полем:

- `GetBudget()` возвращает значение поля `Budget`;
- `AddBudget()` наращивает `Budget` на указанную сумму.

В основной части программы устанавливаем стартовый бюджет и проводим ряд пополнений.

```
Бюджет в начале: 472000
Бюджет после вложений: 565000
```

Рис. 3.52. Расширения класса `static`-методами

Отметим, что в приведенном примере роль метода `GetBudget()` здесь не столь велика. Для получения значения бюджета можно просто напрямую обратиться к полю `Family.Budget`: метод `GetBudget()` его же и возвращает. По аналогии и метод AddBudget(): вместо его вызова можно оперировать напрямую с полем `Family.Budget`. (Методы добавлен исключительно для демонстрации)

Методы экземпляра в обработке статических полей и методов

Доступ к статическим полям и методам имеют не только статические методы, но и методы экземпляра. Экземпляры класса совместно могут пользоваться переменной типа `static`: получать значение или его менять.

Например, каждый член семьи может использовать часть накоплений для покупки чего-либо.

Внесем в описание класса метод `GetMoney()`, который будет уменьшать накопления на заданную сумму.

Глава 3\StaticModifier_4\Family.cs

```
namespace StaticModifier
{
    class Family
    {
        public string? Full_name = "";
        public int Age;
        public static double Budget;

        public Family() { }

        public Family(string? full_name, int age)
        {
            Full_name = full_name;
            Age = age;
        }
```

```
        public override string ToString() =>
            $"{Full_name}\t{Age}";

        public static double GetBudget() =>
            Budget;

        public static void AddBudget(double sum) =>
            Budget += sum;

        public void GetMoney(double sum)
        {
            if (Age >= 18)
            {
                if (Budget >= sum)
                    Budget -= sum;
                else
                    Budget = 0;
            }
            else
            {
                if ((Budget >= 150000) && (sum <= 50000))
                    Budget -= sum;
                else
                    Console.WriteLine("Бюджет не резиновый!");
            }
        }
    }
}
```

Глава 3\StaticModifier_4\Program.cs

```
using System;

namespace StaticModifier
{
    class Program
    {
        static void Main(string[] args)
        {
            Family father = new Family(
                "Карпов Олег Николаевич", 41
            );
            Family mother = new Family(
                "Карпова Лариса Андреевна", 39
```

```
            );
            Family son = new Family(
                "Карпов Валентин Олегович", 16
            );
            Family daughter = new Family(
                "Карпова Анна Олеговна", 17
            );

            Family.Budget = 472000;

            Console.WriteLine(
                $"Бюджет в начале: {Family.GetBudget()}"
            );

            father.GetMoney(24500);
            mother.GetMoney(4900);
            daughter.GetMoney(8000);
            son.GetMoney(138000);

            Console.WriteLine(
                $"Бюджет после снятий: {Family.GetBudget()}"
            );
        }
    }
}
```

```
Бюджет в начале: 472000
Бюджет не резиновый!
Бюджет после снятий: 434600
```

Рис. 3.53. Метод экземпляра исполбзует статическое поле

Метод `GetMoney()` является экземплярным, т.е. привязывается к объектам класса.

Согласно заданной в методе логике обработки совершеннолетний член семьи может взять любую сумму. Если запрашиваемая сумма больше бюджета, то забираются все деньги, т.е. бюджет обнулятся. Несовершеннолетний член семьи может взять не более 50000 р, но при условии, что в бюджете есть не менее 150000 р.

В основной части программы метод `GetMoney()` вызывается относительно объектов класса. Здесь каждый экземпляр класса участвует в изменении статического поля `Budget`.

Это важно знать!

Из приведенного примера видно, что методы экземпляра могут управлять статическими полями как общими.
Однако наоборот неверно: статические методы не могут управлять полями объектов. Это следует из того, что статический метод привязан к классу, а не к какому-либо объекту.

3.5.4. Статические и экземплярные компоненты

Общие правила

При взаимодействии со статическими полям и методами важно следить за рядом правил.

1. Статический метод может оперировать только статическими полями и методами класса.
2. Методы экземпляра могут оперировать как статическими полями, так и вызывать статические поля и методы.
3. Статическому методу недоступна ссылка `this` (т.к. он не связан с объектами).

Выбор между методом экземпляра и static-методом

Методы экземпляра

Если метод характеризует или извлекает состояние объекта, его рационально описывать как метод экземпляра. Каждый экземпляр класса получает свою копию этого метода.

Например, в классе `Family` методы `ToString()` и `GetMoney()` по смыслу привязаны к объектам, результат работы этих методов зависит от самого объекта или его характеристик.

Статические методы

Если метод определяет общее поведение класса или его компоненты, а также влияет на все объекты класса, его следует описать как статический.

Например, в классе `Family` методы `GetBudget()` и `AddBudget()` не зависят от объектов класса и обрабатывают общедоступное статическое поле `Budget`.

3.5.5. Статический конструктор

Статический конструктор для глобальных операций

Статические конструкторы обычно используются для инициализации статических данных.

Особенности статического конструктора:

1. Статическому конструктору не указывают модификатор доступа и он не имеет параметров;
2. В статическом конструкторе ссылка `this` недоступна. Можно обращаться только к статическим членам класса.
3. Статический конструктор нельзя вызвать в программе вручную. Он выполняется автоматически при первом создании экземпляра класса или при любом первом обращении к его статическим членам.

Добавим статический конструктор, который будет задавать стартовое значение бюджета семьи (иначе берется нулем). Поле `Budget` тоже является статическим, поэтому работа конструктора допустима:

```
class Family
{
    // ...
    static Family() => Budget = 50000;
    // ...
}
```

Статический конструктор сработает автоматически при первом обращении к классу (оба обращения равнозначны):

```
static class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(
            $"Начальный бюджет: {Family.Budget}"
        );
        Console.WriteLine(
            $"Начальный бюджет: {Family.GetBudget()}"
        );
    }
}
```

```
Начальный бюджет: 50000
Начальный бюджет: 50000
```

Рис. 3.54. Статический конструктор задает начальный бюджет семьи

В простых случаях (как в нашем примере) создавать статический конструктор излишне: достаточно инициализировать статическое поле прямо в классе.

Этот прием еще называют **статической инициализацией**.

```
class Family
{
    // ...
    public static double Budget = 50000;
    // ...
}
```

Пример использования статического конструктора

В качестве примера работы со статическим конструктором реализуем в классе `Family` автоматизированный счетчик созданных экземпляров класса. Оформим его в виде метода `GetCounter()`, который возвращает число созданных объектов класса `Family`. Счетчик должен автоматически обновляться при создании нового члена семьи.

```
namespace StaticModifier
{
    class Family
    {
        // Поля ...
        private static int _counter = 0;

        public Family() => _counter++;

        public Family(string full_name, int age)
        {
            Full_name = full_name;
            Age = age;
            _counter++;
        }

        public static int GetCounter() => _counter;
        // Другие методы...
    }
}
```

В примере добавлено статическое поле `_counter`, которое будет хранить число созданных объектов. Важно, что поле описано закрытым, чтобы внешне оно было недоступно для изменения (иначе счетчик может быть сброшен или установлен некорректным значением).

Счетчик должен наращиваться при каждом создании экземпляра класса, т.е. как в конструкторе по умолчанию, так и параметризированном конструкторе.

Чтобы иметь возможность получать текущее значение закрытого счетчика, делегируем эту задачу на открытый статический метод `GetCounter()`.

Наращивание счетчика не лишним будет оформить в виде закрытого метода `_AddCounter()`, тогда смысл этой операции в конструкторах будет более очевидной:

```
namespace StaticModifier
{
    class Family
    {
        // ...
        private static int _counter = 0;

        private static void _AddCounter() => _counter++;

        public Family() => _AddCounter();

        public Family(string full_name, int age)
        {
            Full_name = full_name;
            Age = age;
            _AddCounter();
        }

        // ...
    }
}
```

Полный код программы:

Глава 3\StaticModifier_5\Family.cs

```
namespace StaticModifier
{
    class Family
    {
```

```
        public string? Full_name = "";
        public int Age;
        public static double Budget = 50000;
        private static int _counter = 0;

        private static void AddCounter() =>
            _counter++;

        public Family() =>
            AddCounter();

        public Family(string? full_name, int age)
        {
            Full_name = full_name;
            Age = age;
            AddCounter();
        }

        public static int GetCounter() =>
            _counter;

        public override string ToString() =>
            $"{Full_name}\t{Age}";

        public static double GetBudget() =>
            Budget;

        public static void AddBudget(double sum) =>
            Budget += sum;

        public void GetMoney(double sum)
        {
            if (Age >= 18)
            {
                if (Budget >= sum)
                    Budget -= sum;
                else
                    Budget = 0;
            }
            else
            {
                if ((Budget >= 150000) && (sum <= 50000))
                    Budget -= sum;
                else
```

```
                    Console.WriteLine("Бюджет не резиновый!");
                }
            }
        }
    }
```

Глава 3\StaticModifier_5\Program.cs

```
using System;

namespace StaticModifier
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine($"Изначальное число членов
                семьи: {Family.GetCounter()}");

            Family father = new Family(
                "Карпов Олег Николаевич", 41
            );
            Family mother = new Family(
                "Карпова Лариса Андреевна", 39
            );
            Family son = new Family(
                "Карпов Валентин Олегович", 16
            );
            Family daughter = new Family(
                "Карпова Анна Олеговна", 17
            );

            Console.WriteLine($"В семье человек:
                {Family.GetCounter()}");
        }
    }
}
```

```
Изначальное число членов семьи: 0
В семье человек: 4
```

Рис. 3.55. Использование счетчика созданных экземпляров класса

3.5.6. Статический класс

Статические классы и их особенности

Если все поля и методы класса объявлены с модификатором `static`, то создание экземпляров этого класса теряет смысл: у созданных объектов не будет собственных полей и методов. Им лишь доступны стандартные методы `Equals()`, `GetHashCode()`, `GetType()` и `ToString()`, которые наследуются всеми классами от базового класса `Object`.

В этом случае класс также следует описать с модификатором `static`. Тогда создавать экземпляры **статического класса** будет запрещено. А обращение к полям, методам и другим компонентам будет осуществляться по имени класса.

Примером статических классов являются классы `Math` и `Console` из пространства имен `System`. Ранее в задачах мы не создавали экземпляров этих классов, а вызывали их методы и свойства, напрямую обращаясь к классу.

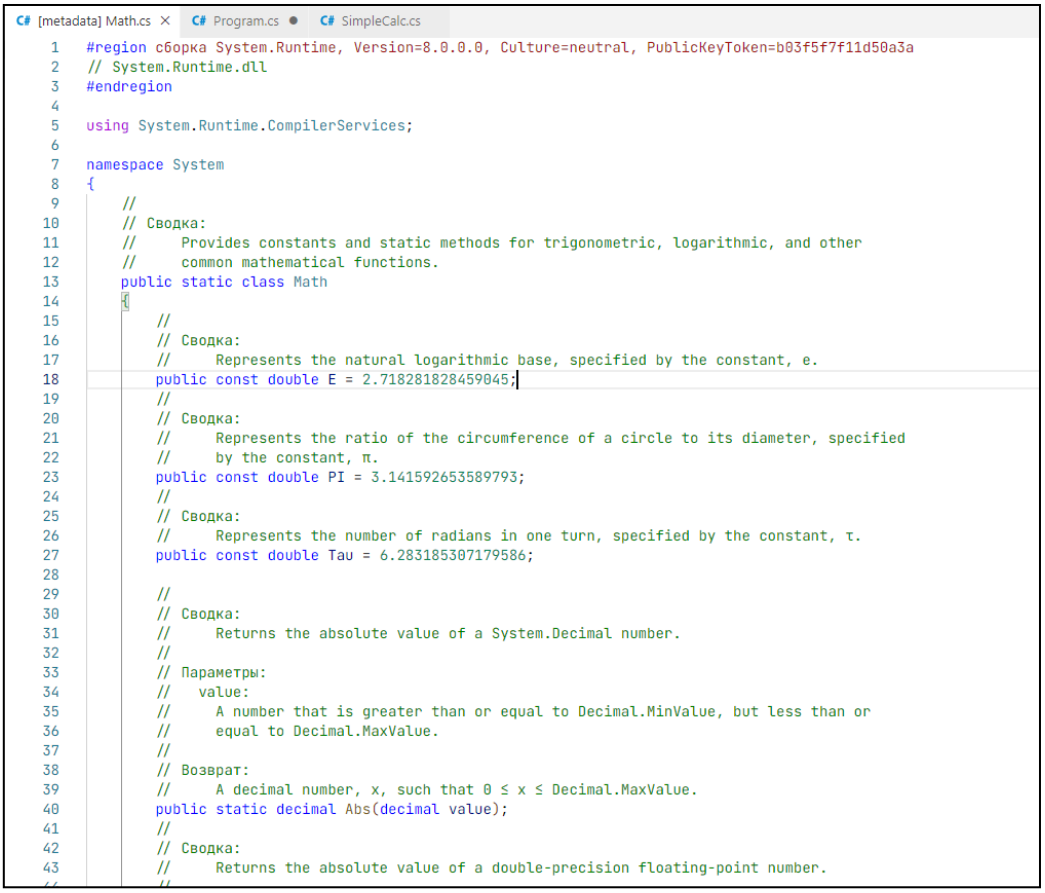


Рис. 3.56. Класс Math описан с модификатором `static`

Пример статического класса

Определим класс с методами, осуществляющими простейшие арифметические операции. Поскольку операции не привязаны к конкретным числам, а осуществляются над ними, то все методы рационально сделать статическими. Каждому методу передаются два числа, а метод возвращает результат операции.

Глава 3\SimpleCalc_1\SimpleCalc.cs

```
namespace StaticModifier
{
    static class SimpleCalc
    {
        public static double GetSum(double a, double b) =>
            a + b;

        public static double GetSub(double a, double b) =>
            a - b;

        public static double GetMul(double a, double b) =>
            a * b;

        public static double GetDiv(double a, double b) =>
            a / b;
    }
}
```

Глава 3\SimpleCalc_1\Program.cs

```
using System;

namespace StaticModifier
{
    static class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Сумма: " +
                SimpleCalc.GetSum(10, 4.5));
            Console.WriteLine("Разность: " +
                SimpleCalc.GetSub(10, 4.5));
            Console.WriteLine("Произведение: " +
                SimpleCalc.GetMul(10, 4.5));
            Console.WriteLine("Частное: " +
```

```
                SimpleCalc.GetDiv(10, 4.5));
        }
    }
}
```

```
Сумма: 14,5
Разность: 5,5
Произведение: 45
Частное: 2,2222222222222223
```

Рис. 3.57. Статический класс с набором статических методов

Поля константы

Если в описании статического класса присутствует поле в качестве константы, то указывать ключевое слово `static` не требуется. Константу можно считать частным случаем статического поля, однако оно неизменно и обязательно имеет заранее заданное значение.

Добавим в класс `SimpleCalc` несколько констант.

Глава 3\SimpleCalc_2\SimpleCalc.cs

```
namespace StaticModifier
{
    static class SimpleCalc
    {
        public const double PI = 3.1415926535897931;
        public const double E = 2.7182818284590451;

        public static double GetSum(double a, double b) =>
            a + b;

        public static double GetSub(double a, double b) =>
            a - b;

        public static double GetMul(double a, double b) =>
            a * b;

        public static double GetDiv(double a, double b) =>
            a / b;
    }
}
```

Глава 3\SimpleCalc_2\Program.cs

```
using System;

namespace StaticModifier
{
    static class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Сумма: " +
                SimpleCalc.GetSum(10, 4.5));
            Console.WriteLine("Разность: " +
                SimpleCalc.GetSub(10, 4.5));
            Console.WriteLine("Произведение: " +
                SimpleCalc.GetMul(10, 4.5));
            Console.WriteLine("Частное: " +
                SimpleCalc.GetDiv(10, 4.5));

            Console.WriteLine("Пи = " + SimpleCalc.PI);
            Console.WriteLine("E = " + SimpleCalc.E);
        }
    }
}
```

```
Сумма: 14,5
Разность: 5,5
Произведение: 45
Частное: 2,2222222222222223
Пи = 3,141592653589793
E = 2,718281828459045
```

Рис. 3.58. Статический класс с набором const-полей

Вопросы для самопроверки

1. Почему возникает необходимость описывать поля и методы, принадлежащие объекту класса?
2. В чем разница между статическими компонентом и компонентом экземпляра класса?
3. Опишите правила использования `static`-методов.
4. Когда следует использовать статические конструкторы?
5. Какими особенностями обладают статические классы? Приведите примеры.

Практикум

Задание 1

1. Создайте каталог *StaticModifier*.
2. Последовательно повторите и воспроизведите рассмотренный в параграфе примеры проектов от *StaticModifier_1-5*.

Задание 2

1. Создайте каталог *Institute_9*. Используйте наработки проекта *Institute_8*.
2. Требуется вместо методов `InitStudent()`, `InitBachelor()`, `InitSpecialist()` и `InitMaster()` реализовать статические методы, которые возвращают новый объект студента. Назовите их `CreateStudent()`, `CreateBachelor()`, `CreateSpecialist()` и `CreateMaster()`.
3. Для этого создайте обслуживающий приватный статический метод `_CreateStudent()`, который использует конструктор для возврата нового объекта. Это возможно, поскольку статический метод может создавать и возвращать новые объекты класса, обращаясь к конструктору класса. Такие методы называют **фабричными**:

```
private static Student _CreateStudent(
    long id, string? first_name, string? last_name,
    int age, EducationalDegree degree
) =>
    new Student(
        id, first_name, last_name, age, degree
    );
```

4. Далее используйте метод `_CreateStudent()`, чтобы реализовать работу выше обозначенных четырех статических методов. Например, для `CreateBachelor()` реализация имеет вид:

```
public static Student CreateBachelor(
    long id, string? first_name, string? last_name,
    int age
) =>
    _CreateStudent(
        id, first_name, last_name, age,
        EducationalDegree.Bachelor
);
```

5. Реализация перегрузок конструкторов меняться не должна: они по-прежнему должны работать с нестатическим методом `_SetInfo()`.
6. Еще раз внимательно проследите за цепочками зависимостей между методами и конструкторами.
7. В методе `Main()` с помощью обозначенных методов создайте массив из нескольких студентов. В цикле осуществите обход элементов с выдачей информации о каждом (рис. 3.59).
8. Оформите вывод, как изображено на рис. 3.60.

```
Student denis = new Student(12345, "Денис", "Якубович", 22, EducationalDegree.Specialist);
Student andrew = Student.CreateMaster(23001, "Андрей", "Мясищев", 24);
Student olga = Student.CreateBachelor(30253, "Ольга", "Рыбникова", 19);
Student sergey = Student.CreateSpecialist(10253, "Сергей", "Полянский", 19);

Student[] journal = [denis, andrew, olga, sergey];

foreach (var student in journal)
{
    // Здесь операции с элементом student

}
```

Рис. 3.59. Создание массива объектов

```
ID: 12345
Имя: Денис
Фамилия: Якубович
Возраст: 22
Уровень образования: специалитет
Студент совершеннолетний

ID: 23001
Имя: Андрей
Фамилия: Мясищев
Возраст: 24
Уровень образования: магистратура
Студент совершеннолетний

ID: 30253
Имя: Ольга
Фамилия: Рыбникова
Возраст: 19
Уровень образования: бакалавриат
Студент совершеннолетний

ID: 10253
Имя: Сергей
Фамилия: Полянский
Возраст: 19
Уровень образования: специалитет
Студент совершеннолетний
```

Рис. 3.60. Результат работы программы из задания 2

3.6. Разбор примеров проектов

3.6.1. Проект «Геометрия треугольника»

Постановка задачи

Требуется создайте класс `Triangle` («Треугольник»).

1. Определить в классе три вещественных поля `a`, `b`, `c` – длина соответствующей стороны.
2. Определить конструктор по умолчанию, создающий объект с нулевыми сторонами и параметризированный конструктор, заполняющий поля по указанным значениям.
3. Переопределить метод `ToString()` для вывода данных о треугольнике в следующей форме:

   ```
   Стороны: a = 4, b = 5.3, c = 3
   ```

4. В отдельном файле описать перечисление `TriangleType`, задающее возможный тип треугольника (равносторонний, равнобедренный, обыкновенный, не треугольник).
5. В классе определить метод `GetTriangleType()`, который возвращает значение типа треугольника (в зависимости от его сторон, см. п. 4).
6. Также добавить метод `GetTriangleTypeRus()`, который возвращает название треугольника на русском языке (т.е. строку).
7. Определить логический метод `IsTriangle()`, возвращающий истину, если объект является треугольником, иначе – ложь. Используйте этот метод далее в остальных методах, где требуется.
8. Определить метод `GetPerimeter()`, возвращающий периметр треугольника. Если фигура не является треугольником, то возвратить нуль.
9. Определить метод `GetSquare()`, возвращающий площадь треугольника. Если фигура не является треугольником, то возвратить нуль.
10. Протестировать методы на одном экземпляре класса.

Комментарии к реализации

Начнем с описания полей и конструкторов класса `Triangle`.

Любой треугольник можно определить по длине трех сторон (при условии его существования, см. далее).

Каждая сторона будет определяться одноименным открытым полем класса. Определим два конструктора:

- конструктор по умолчанию задает объект с нулевыми значениями всех сторон (т.е. это не треугольник);
- конструктор с параметрами задает фигуру по трем указанным сторонам, а отрицательные значения исправит на положительные.

Для возврата сводной информации о сторонах треугольника переопределим метод `ToString()`:

```
public class Triangle
{
    public double a;
    public double b;
    public double c;

    public Triangle() { }

    public Triangle(double a, double b, double c)
    {
        this.a = Math.Abs(a);
        this.b = Math.Abs(b);
        this.c = Math.Abs(c);
    }

    public override string ToString() =>
        $"Стороны: a={a}   b={b}   c={c}";
}
```

Треугольники могут быть разных типов. Выделим правильный, равнобедренный, обыкновенный треугольник, а также вариант, что фигура не является треугольником. Рационально замкнуть эти варианты перечислением (опишем в отдельном файле):

```
namespace TriangleGeometry
{
    public enum TriangleType
    {
       Equilateral,    // Равносторонний
       Isosceles,      // Равнобедренный
       Ordinary,       // Обыкновенный
       Nontrangle      // Не треугольник
    }
}
```

Теперь опишем метод, который вычисляет и возвращает тип треугольника:

- у равностороннего (правильного) треугольника все стороны равны;
- у равнобедренного треугольника равны какие-либо две стороны;
- у обыкновенного треугольника достаточно проверить, что сумма двух любых сторон больше третей (теорема существования треугольника);
- иначе фигура не является треугольником.

```
public class Triangle
{
    // Поля и методы описаны выше ...

    public TriangleType GetTriangleType()
    {
        if ((a >= b + c) || (b >= a + c) || (c >= a + b))
            return TriangleType.Nontrangle;
        if ((a == b) && (b == c))
            return TriangleType.Equilateral;
        if ((a == b) || (a == c) || (b == c))
            return TriangleType.Isosceles;
        return TriangleType.Ordinary;
    }
}
```

Метод `GetTriangleType()` возвратит тип треугольника как значение абстрактного перечисления. Чтобы его писать на русском языке, создадим метод `GetTriangleTypeRus()`, который возвратит название типа уже в виде текстовой строки:

```
public class Triangle
{
    // Поля и методы описаны выше ...

    public string GetTriangleTypeRus() =>
        GetTriangleType() switch
        {
            TriangleType.Equilateral => "правильный",
            TriangleType.Isosceles => "равнобедренный",
            TriangleType.Ordinary => "обыкновенный",
            _ => "не треугольник"
        };

}
```

Здесь мы используем уже созданный метод `GetTriangleType()` и сокращаем код благодаря шаблону сопоставления `switch`.

Поскольку в дальнейших операциях предварительно нужно будет убедиться, что фигура является треугольником, рационально делегировать эту задачу на метод логического типа `IsTriangle()`. Фигура является треугольником, если ее тип любой, кроме `Nontrangle`:

```
public class Triangle
{
    // Поля и методы описаны выше ...

    public bool IsTriangle() =>
        GetTriangleType() != TriangleType.Nontrangle;
}
```

Для вычисления периметра предварительно необходимо проверить, что фигура является треугольником, иначе периметр нужно считать равным нулю:

```
public class Triangle
{
    // Поля и методы описаны выше ...
    public double GetPerimeter()
    {
        if (!IsTriangle())
            return 0;
        return a + b + c;
    }
}
```

Площадь вычислим по формуле Герона (относительно сторон и полупериметра). При этом проверять, является ли фигура треугольником, необязательно: для не треугольника полупериметр будет равным нулю, а следовательно – и площадь:

```
public class Triangle
{
    // Поля и методы описаны выше ...

    public double GetSquare()
    {
        double p = GetPerimeter() / 2;

        return Math.Sqrt(
            p * (p - a) * (p - b) * (p - c)
        );
    }
}
```

Далее приведем пример создания треугольника и извлечения его свойств.

Программный код

Глава 3\TriangleGeometry\TriangleType.cs

```
namespace TriangleGeometry
{
    public enum TriangleType
    {
       Equilateral,   // Равносторонний
       Isosceles,     // Равнобедренный
       Ordinary,      // Обыкновенный
       Nontrangle     // Не треугольник
    }
}
```

Глава 3\TriangleGeometry\Triangle.cs

```
namespace TriangleGeometry
{
    public class Triangle
    {
        public double a;
        public double b;
        public double c;
```

```
public Triangle() { }

public Triangle(double a, double b, double c)
{
    this.a = Math.Abs(a);
    this.b = Math.Abs(b);
    this.c = Math.Abs(c);
}

public override string ToString() =>
    $"Стороны: a={a}   b={b}   c={c}";

public TriangleType GetTriangleType()
{
    if ((a >= b + c) || (b >= a + c) || (c >= a + b))
        return TriangleType.Nontrangle;
    if ((a == b) && (b == c))
        return TriangleType.Equilateral;
    if ((a == b) || (a == c) || (b == c))
        return TriangleType.Isosceles;
    return TriangleType.Ordinary;
}

public string GetTriangleTypeRus() =>
    GetTriangleType() switch
    {
        TriangleType.Equilateral => "правильный",
        TriangleType.Isosceles => "равнобедренный",
        TriangleType.Ordinary => "обыкновенный",
        _ => "не треугольник"
    };

public bool IsTriangle() =>
    GetTriangleType() != TriangleType.Nontrangle;

public double GetPerimeter()
{
    if (!IsTriangle())
        return 0;
    return a + b + c;
}
```

```
        public double GetSquare()
        {
            double p = GetPerimeter() / 2;
            return Math.Sqrt(p * (p - a) * (p - b) * (p - c));
        }
    }
}
```

Глава 3\TriangleGeometry\Program.cs

```
using System;

namespace TriangleGeometry
{
    class Program
    {
        static void Main(string[] args)
        {
            var triangle = new Triangle(7.5, 5, 7.5);

            Console.WriteLine(triangle);

            if (triangle.IsTriangle())
                Console.WriteLine(
                    "Фигура является треугольником"
                );
            else
                Console.WriteLine(
                    "Фигура не является треугольником"
                );

            Console.WriteLine($"Тип треугольника:
                {triangle.GetTriangleTypeRus()}");
            Console.WriteLine($"Периметр:
                {triangle.GetPerimeter():F5}");
            Console.WriteLine($"Площадь:
                {triangle.GetSquare():F5}");
        }
    }
}
```

```
Стороны: a=7,5   b=5   c=7,5
Фигура является треугольником
Тип треугольника: равнобедренный
Периметр: 20,00000
Площадь: 17,67767
```

Рис. 3.61. Результат выполнения программы

3.6.2. Проект «Линейные функции»

Постановка задачи

Требуется реализовать класс `Linear`, определяющий работу с линейными функциями вида $f(x) = ax + b$. В качестве полей класса взять коэффициенты $a,\ b$.

В описание класса внести ряд методов:

- Метод `ToString()` возвращает строковую запись функции.
- Метод `GetValue(x)` возвращает значение функции в заданной точке x (если параметр не передается, считать $x = 0$).
- Метод `IsConstant()`, проверяющий, является ли функция постоянной.
- Метод `Opposite()`, возвращающий противоположную по знаку функцию.
- Метод `Sum(f,g)`, возвращающий сумму двух функций.
- Метод `Sub(f,g)`, возвращающий разность двух функций.
- Метод `Scale(k,f)`, возвращающий произведение функции на число. Также предусмотрите возможность вызова метода с измененным порядком следования параметров: `Scale(f,k)`.

Комментарии к реализации

Начнем с описания полей класса и конструкторов. Линейная функция определяется двумя числовыми коэффициентами, на которые нет каких-либо ограничений.

```
public class Linear
{
    public double a;
    public double b;

    public Linear() { }
```

```
    public Linear(double a, double b)
    {
        this.a = a;
        this.b = b;
    }
}
```

Для вывода функции в виде $ax + b$ необходимо учесть возможные значения коэффициентов, чтобы их оформить в надлежащей форме:

```
public class Linear
{
    // Поля и методы выше ...

    public override string ToString()
    {
        if (a == 0 && b == 0)
            return "0";
        else if (a == 0)
            return $"{b}";
        else if (b == 0)
            return $"{a}x";
        else if (b < 0)
            return $"{a}x - {-b}";
        else
            return $"{a}x + {b}";
    }
}
```

Для вычисления значения линейной функции в указанной точке достаточно подставить значение в формулу. Благодаря перегрузке функций учтем также случай, когда аргумент x не передается (считаем равным нулю):

```
public class Linear
{
    // Поля и методы выше ...

    public double GetValue() => b;
    public double GetValue(double x) => a * x + b;
}
```

Приведенную реализацию двух методов также можно свернуть в одну, используя установку значения параметра по умолчанию:

```
public class Linear
{
    // Поля и методы выше ...

    public double GetValue(double x = 0) => a * x + b;
}
```

Для проверки, является ли функция постоянной, анализируем коэффициент при x:

```
public class Linear
{
    // Поля и методы выше ...

    public bool IsConstant() => a == 0;
}
```

Противоположная по знаку функция, сумма, разность и умножение линейной функции на число тоже являются линейными функциями. Поскольку эти операции будут осуществляться над функциями и числами, возвращать их результаты, то рационально описать методы статическими:

```
public class Linear
{
    // Поля и методы выше ...

    public static Linear Opposite(Linear f) =>
        new Linear(-f.a, -f.b);

    public static Linear Sum(Linear f, Linear g) =>
        new Linear(f.a + g.a, f.b + g.b);

    public static Linear Sub(Linear f, Linear g) =>
        new Linear(f.a - g.a, f.b - g.b);

    public static Linear Scale(double k, Linear f) =>
        new Linear(k * f.a, k * f.b);
}
```

Здесь вычисления удается свернуть в одну операцию. Каждый метод обращается к конструктору новой линейной функции, а в качестве параметров берутся коэффициенты функций.

Программный код

Глава 3\FunctionSpace_1\Linear.cs

```
namespace FunctionSpace
{
    public class Linear
    {
        public double a;
        public double b;

        public Linear() { }

        public Linear(double a, double b)
        {
            this.a = a;
            this.b = b;
        }

        public override string ToString()
        {
            if (a == 0 && b == 0)
                return "0";
            else if (a == 0)
                return $"{b}";
            else if (b == 0)
                return $"{a}x";
            else if (b < 0)
                return $"{a}x - {-b}";
            else
                return $"{a}x + {b}";
        }

        public double GetValue() => b;
        public double GetValue(double x) => a * x + b;

        // public double GetValue(double x = 0) => a * x + b;

        public bool IsConstant() => a == 0;

        public static Linear Opposite(Linear f) =>
            new Linear(-f.a, -f.b);

        public static Linear Sum(Linear f, Linear g) =>
            new Linear(f.a + g.a, f.b + g.b);
```

```
        public static Linear Sub(Linear f, Linear g) =>
            new Linear(f.a - g.a, f.b - g.b);

        public static Linear Scale(double k, Linear f) =>
            new Linear(k * f.a, k * f.b);
    }
}
```

Глава 3\FunctionSpace_1\Program.cs

```
using System;

namespace FunctionSpace
{
    class Program
    {
        static void Main(string[] args)
        {
            Linear f = new Linear(1.5, 6);
            Linear g = new Linear(0, -4);

            Console.WriteLine($"f(x) = {f}");
            Console.WriteLine($"g(x) = {g}\n");

            double x = -2;
            Console.WriteLine($"f({x}) = {f.GetValue(x)}");
            Console.WriteLine($"g({x}) = {g.GetValue(x)}\n");

            if (f.IsConstant())
                Console.WriteLine("f постоянная");

            if (g.IsConstant())
                Console.WriteLine("g постоянная");

            Console.WriteLine();
            Console.WriteLine($"-f = {Linear.Opposite(f)}");
            Console.WriteLine($"-g = {Linear.Opposite(g)}");

            Linear sum = Linear.Sum(f, g);
            Linear sub = Linear.Sub(f, g);

            Linear scale = Linear.Scale(
                2, Linear.Sum(f, Linear.Scale(3.5, g))
            );
```

```
            Console.WriteLine();
            Console.WriteLine($"f(x) + g(x) = {sum}");
            Console.WriteLine($"f(x) - g(x) = {sub}");
            Console.WriteLine($"2 * (f(x) + 3.5 * g(x)) =
                {scale}");
        }
    }
}
```

```
f(x) = 1,5x + 6
g(x) = -4

f(-2) = 3
g(-2) = -4

g постоянная

-f = -1,5x - 6
-g = 4

f(x) + g(x) = 1,5x + 2
f(x) - g(x) = 1,5x + 10
2 * (f(x) + 3.5 * g(x)) = 3x - 16
```

Рис. 3.62. Результат выполнения программы

Перегрузка операторов

Проблема

При организации вычислений с линейными функциями требуется вызывать методы. Когда количество аргументов в формулах растет, требуется вкладывать вызов функций, что неудобно. Например, для вычисления выражения $2(f(x) + 3.5g(x))$ необходимо тройное вложение:

```
Linear scale = Linear.Scale(
    2, Linear.Sum(f, Linear.Scale(3.5, g))
);
```

Такой синтаксис неудобен и мало естественный для человека. Гораздо удобнее иметь возможность писать выражения в естественной форме:

```
Linear scale = 2 * (f + 3.5*g);
```

Решение проблемы

И такая возможность существует благодаря поддержке языком C# **перегрузки операторов**.

Перегрузка операторов – возможность переопределить стандартное поведение оператора в зависимости от используемого класса.

В случае одного операнда оператор называют **унарным** (например оператор - как оператор противоположного по знаку), а для двух операндов – **бинарным** (например, +, -, *, / как арифметические).

Переопределение оператора – это подвид статического метода. Разница лишь в том, что вместо названия метода указывается оператор и ключевое слово **`operator`**. Параметры метода – это один или два операнда, между которыми ставится оператор.

Так, чтобы перегрузить унарный оператор знака, опишем следующий код:

```
public static Linear operator -(Linear f) =>
    new Linear(-f.a, -f.b);
```

Очевидно, что разница по сравнению с методом `Opposite()` минимальна:

```
public static Linear Opposite(Linear f) =>
    new Linear(-f.a, -f.b);
```

Теперь получить противоположную по знаку функцию можно двумя способами:

```
Linear neg_f = Linear.Opposite(f);
Linear neg_g = -g;
```

Для перегрузки бинарных операторов подход аналогичен, добавляется только еще один операнд. Например, перегрузка оператора + и соответствующий ранее описанный метод сложения:

```
public static Linear Sum(Linear f, Linear g) =>
    new Linear(f.a + g.a, f.b + g.b);

public static Linear operator +(Linear f, Linear g) =>
    new Linear(f.a + g.a, f.b + g.b);
```

Теперь для сложения функций можно использовать короткий подход:

```
Linear sum = f + g;            // Linear.Sum(f,g);
```

Оптимизация

Нетрудно также заметить, что описанные методы и перегрузки соответствующих им операторов имеют одинаковую реализацию тела метода. Поэтому рационально для каждой операции описать вспомогательные закрытые в классе методы, которые будут обслуживать вычисления.

Например, для взятия противоположного по знаку значения создадим метод `_Opposite()`:

```
private static Linear _Opposite(Linear f) =>
    new Linear(-f.a, -f.b);

public static Linear Opposite(Linear f) => _Opposite(f);
public static Linear operator -(Linear f) => _Opposite(f);
```

По аналогии создаются методы для других операций, например, для сложения:

```
private static Linear _GetSum(Linear f, Linear g) =>
    new Linear(f.a + g.a, f.b + g.b);

public static Linear Sum(Linear f, Linear g) =>
    _GetSum(f,g);
public static Linear operator +(Linear f, Linear g) =>
    _GetSum(f,g);
```

Полное решение:

Глава 3\FunctionSpace_2\Linear.cs

```
namespace FunctionSpace
{
    public class Linear
    {
        public double a;
        public double b;

        public Linear() { }

        public Linear(double a, double b)
        {
            this.a = a;
            this.b = b;
        }
```

```
        public override string ToString()
        {
            if (a == 0 && b == 0)
                return "0";
            else if (a == 0)
                return $"{b}";
            else if (b == 0)
                return $"{a}x";
            else if (b < 0)
                return $"{a}x - {-b}";
            else
                return $"{a}x + {b}";
        }

        public double GetValue() => b;
        public double GetValue(double x) => a * x + b;

        public bool IsConstant() => a == 0;

        private static Linear _Opposite(Linear f) =>
            new Linear(-f.a, -f.b);
        public static Linear Opposite(Linear f) =>
            _Opposite(f);
        public static Linear operator -(Linear f) =>
            _Opposite(f);

        private static Linear _GetSum(Linear f, Linear g) =>
            new Linear(f.a + g.a, f.b + g.b);
        public static Linear Sum(Linear f, Linear g) =>
            _GetSum(f,g);
        public static Linear operator +(Linear f, Linear g) =>
            _GetSum(f,g);

        private static Linear _GetSub(Linear f, Linear g) =>
            new Linear(f.a - g.a, f.b - g.b);
        public static Linear Sub(Linear f, Linear g) =>
            _GetSub(f,g);
        public static Linear operator -(Linear f, Linear g) =>
            _GetSub(f,g);

        private static Linear _GetScale(double k, Linear f) =>
            new Linear(k * f.a, k * f.b);
        public static Linear Scale(double k, Linear f) =>
```

```
            _GetScale(k,f);
        public static Linear Scale(Linear f, double k) =>
            _GetScale(k,f);
        public static Linear operator *(double k, Linear f) =>
            _GetScale(k,f);
        public static Linear operator *(Linear f, double k) =>
            _GetScale(k,f);
    }
}
```

Глава 3\FunctionSpace_2\Program.cs

```
using System;

namespace FunctionSpace
{
    class Program
    {
        static void Main(string[] args)
        {
            Linear f = new Linear(1.5, 6);
            Linear g = new Linear(0, -4);

            Console.WriteLine($"f(x) = {f}");
            Console.WriteLine($"g(x) = {g}\n");

            double x = -2;
            Console.WriteLine($"f({x}) = {f.GetValue(x)}");
            Console.WriteLine($"g({x}) = {g.GetValue(x)}\n");

            if (f.IsConstant())
                Console.WriteLine("f постоянная");

            if (g.IsConstant())
                Console.WriteLine("g постоянная");

            Console.WriteLine();
            Console.WriteLine($"-f = {-f}");
            Console.WriteLine($"-g = {-g}");

            Linear sum = f + g;
            Linear sub = f - g;
            Linear scale = 2 * (f + 3.5 * g);

            Console.WriteLine();
```

```
            Console.WriteLine($"f(x) + g(x) = {sum}");
            Console.WriteLine($"f(x) - g(x) = {sub}");
            Console.WriteLine($"2 * (f(x) + 3.5 * g(x)) =
                {scale}");
        }
    }
}
```

В реализации метода `Scale()` было создано по две перегрузки метода и оператора *. Это необходимо, что иметь возможность домножать на функцию на число как слева, так и справа. Хотя с математической точки зрения разницы нет, поэтому все четыре перегрузки обращаются к одному и тому же метода `_Scale()`.

```
f(x) = 1,5x + 6
g(x) = -4

f(-2) = 3
g(-2) = -4

g постоянная

-f = -1,5x - 6
-g = 4

f(x) + g(x) = 1,5x + 2
f(x) - g(x) = 1,5x + 10
2 * (f(x) + 3.5 * g(x)) = 3x - 16
```

Рис. 3.63. Результат работы программы

Вопросы для самопроверки

1. Обоснуйте причины, по которым в рассмотренных примерах одни методы описаны как методы экземпляра, а другие – как статические.
2. Какие возможности еще можно предусмотреть в каждом из проектов?
3. Что представляет собой перегрузка операторов? Приведите примеры задач, где перегрузка операторов тоже может быть крайне востребована.

Практикум

Задание 1

1. Создайте каталог *TriangleGeometry*.
2. Внутри этого каталога подготовьте проект *TriangleGeometry_1*.
3. Последовательно повторите и воспроизведите рассмотренный в параграфе пример с геометрией треугольника.
4. Проследите за логикой зависимости в методах.
5. Добавьте в проект логический метод `IsRight()`, проверяющий, является ли треугольник прямоугольным. Если фигура не является треугольником, то возвратить ложь. Рекомендуется дополнительно описать закрытый в классе метод `_IsPithagorTriangle(double a, double b, double c)`, который возвращает истину, если `a`, `b` и `c` образуют пифагорову тройку чисел. Далее используйте этот метод, чтобы проверить всевозможные пары внутри `IsRight()`. Также не забудьте убедиться, что фигура вообще является треугольником.
6. Протестируйте работу с этим методом.

Задание 2

1. Создайте проект *TriangleGeometry_2*.
2. Опишите массив треугольников и протестируйте и выведите всю информацию о каждом в цикле.

```
Triangle[] test_array =
{
    new Triangle(),
    new Triangle(2, 0.1, 5),
    new Triangle(23, 20, 31),
    new Triangle(7.5, 5, 7.5),
    new Triangle(6,10, 8),
    new Triangle(4,4,4)
};

foreach (var triangle in test_array)
{
    // Операции с треугольником


}
```

Рис. 3.64. Тестирование методов объекта

```
Стороны: a=0   b=0   c=0
Фигура не является треугольником
Тип треугольника: не треугольник
Периметр: 0
Площадь: 0

Стороны: a=2   b=0,1   c=5
Фигура не является треугольником
Тип треугольника: не треугольник
Периметр: 0
Площадь: -0

Стороны: a=23   b=20   c=31
Фигура является треугольником
Тип треугольника: обыкновенный
Периметр: 74
Площадь: 229,8608274587038

Стороны: a=7,5   b=5   c=7,5
Фигура является треугольником
Тип треугольника: равнобедренный
Периметр: 20
Площадь: 17,67766952966369

Стороны: a=6   b=10   c=8
Фигура является треугольником
Тип треугольника: обыкновенный
Периметр: 24
Площадь: 24
Прямоугольный

Стороны: a=4   b=4   c=4
Фигура является треугольником
Тип треугольника: правильный
Периметр: 12
Площадь: 6,928203230275509
```

Рис. 3.65. Результат работы программы из задания 2

Задание 3

1. Создайте каталог *FunctionSpace*.
2. Внутри этого каталога подготовьте проект *FunctionSpace_1*.
3. Последовательно повторите и воспроизведите рассмотренный в параграфе пример с классом линейных функций.
4. Проследите за тем, как организованы операции с методами.

Задание 4

1. Создайте проект *FunctionSpace_2*.
2. Последовательно повторите и воспроизведите рассмотренный в параграфе пример с классом линейных функций.
3. Проследите за логикой зависимости в методах, организующих операции. Также внимательно разберите, как работают перегрузки функций и операторов, в чем их преимущество.
4. Расширьте описание класса методами:
   - `IsIncreasing()` возвращает истину, если функция возрастающая.
   - `IsDecreasing()` возвращает истину, если функция убывающая.
   - `GetRoot()` возвращает корень уравнения $f(x) = 0$ в форме текстовой строки (или запись, что корня нет).
5. Оформите результат в следующем виде:

```
f(x) = 1,5x + 6
g(x) = -4

f(-2) = 3
g(-2) = -4

f возростающая
g постоянная

-f = -1,5x - 6
-g = 4

f(x) + g(x) = 1,5x + 2
f(x) - g(x) = 1,5x + 10
2 * (f(x) + 3.5 * g(x)) = 3x - 16

Решение уравнения 1,5x + 6 = 0:
x = -4
Решение уравнения -4 = 0:
решений нет
```

Рис. 3.66. Результат работы программы из задания 4

3.7. Методы доступа

3.7.1. Проблема открытых полей класса

Описание проблемы

Ранее в примерах и задачах текущего раздела использовался открытый доступ к полям класса, т.е. поля были доступны как внутри класса, так и за его пределами.

```
Student denis = new Student(12345, "Денис", "Якубович", 220, EducationalDegree.Specialist);
Console.WriteLine($"Возраст студента: {denis.Age}");

denis.Age = 220;
Console.WriteLine($"Возраст студента: {denis.Age}");
```

Рис. 3.67. Доступ к полям класса `Student` открыт

Однако открытые поля класса могут быть источником проблем при работе с объектами. Внешний доступ к внутренним данным зачастую способен нарушить корректную логику взаимодействия данных внутри объекта класса.

Исходный проект

Для дальнейшей работы воспользуемся проектом *AccessMethods* (см. код далее). За основу возьмем упрощенную реализацию класса `Student`: оставим в нем конструктор по умолчанию, конструктор с параметрами, методы `ToString()` и `IsAdult()`.

Глава 3\AccessMethods_1\EducationalDegree.cs

```
namespace AccessMethods
{
    enum EducationalDegree
    {
        Bachelor,
        Specialist,
        Master
    }
}
```

Глава 3\AccessMethods_1\Student.cs

```
namespace AccessMethods
{
    class Student
    {
        public long Id;
        public string? FirstName = "";
        public string? LastName = "";
        public int Age;
        public EducationalDegree Degree;

        public Student() =>
            Age = 18;

        public Student(
            long id, string? first_name, string? last_name,
            int age, EducationalDegree degree
        )
        {
            Id = Math.Abs(id);
            FirstName = first_name;
            LastName = last_name;
            Age = (16 <= age) && (age <= 120) ? age : 18;
            Degree = degree;
        }

        public override string ToString() =>
            $"ID: {Id}\n
            Имя: {FirstName}\n
            Фамилия: {LastName}\n
            Возраст: {Age}\n
            Уровень образования: {Degree}";

        public bool IsAdult() =>
            Age >= 18;
    }
}
```

Глава 3\AccessMethods_1\Program.cs

```
using System;

namespace AccessMethods
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            Student denis = new Student(
                12345, "Денис", "Якубович", 220,
                EducationalDegree.Specialist
            );
            Console.WriteLine($"Возраст студента:
                {denis.Age}");

            denis.Age = 220;
            Console.WriteLine($"Возраст студента:
               {denis.Age}");
        }
    }
}
```

```
Возраст студента: 18
Возраст студента: 220
```

Рис. 3.68. Код исходной задачи

Проблема в реализации проекта

В примере поле Age хранит возраст студента. Предполагается, что при создании объектов класса этому полю будет задано значение в разумных пределах (как и ранее, от 16 до 120).

Частично проблема уже решена: в конструкторе осуществляется предварительная проверка переданного в параметре значения и при необходимости оно будет исправлено на значение по умолчанию – 18.

```
public Student(long id, string? first_name, string? last_n
{
    Id = Math.Abs(id);
    FirstName = first_name;
    LastName = last_name;
    Age =  (16 <= age) && (age <= 120) ? age : 18;
    Degree = degree;
}
```

Рис. 3.69. Возраст при необходимости корректируется конструктором

Поэтому при создании объекта в поле Age записывается корректное значение.

Однако поле `Age` открыто, поэтому некорректное значение 220 может быть присвоено напрямую:

```
denis.Age = 220;
Console.WriteLine($"Возраст студента: {denis.Age}");
```

```
Возраст студента: 220
```

Рис. 3.70. Некорректное значение возраста можно установить через обращение к полю

Открытые поля как источник проблем

Работа с открытыми полями имеет следующие недостатки:

1. Внешний доступ опасен, т.к. может привести к нежелательному изменению значения поля, что нарушит логику обработки данных внутри класса, вплоть до ошибки выполнения кода.
2. Полю можно забыть присвоить необходимое значение.
3. Значение поля класса ограничивается только типом данных, которым оно описано. Но на практике каждая задача накладывает свои ограничения.

Решение проблемы состоит в том, чтобы закрыть внешний доступ к полям.

3.7.2. Геттеры и сеттеры

Закрываем доступ к полю

Закроем доступ к полю `Age`; также переименуем его везде, используя символ нижнего подчеркивания (как было отмечено ранее, так принято обозначать `private`-члены класса).

```
private int _Age;
```

Тогда внешне обратиться к полю уже не получится:

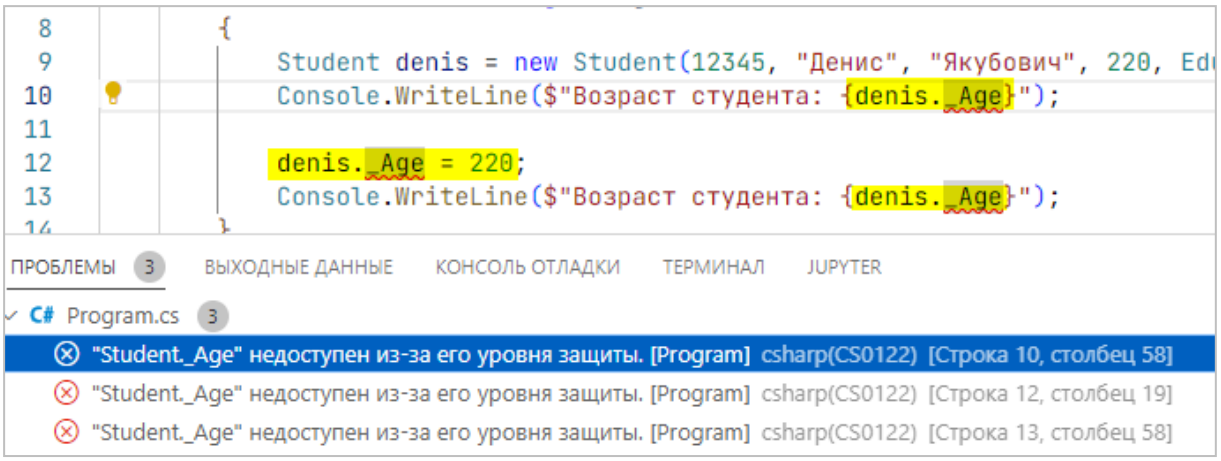


Рис. 3.71. Поле сделано закрытым, поэтому недоступно внешне

Для внешнего управления закрытыми полями создаются открытые обслуживающие **методы доступа**, получившие название сеттеры и геттеры.

Методы доступа к закрытым полям

Метод для записи значения

Определение

Сеттер *(«метод-установщик») – это метод, который записывает в поле требуемое значение. Метод всегда имеет тип* ***void****, т.к. он ничего не возвращает, а только записывает значение в поле.*

Значение, которое необходимо записать, передается через параметр метода. В теле метода, если требуется, осуществляется его предварительная проверка, поэтому закрытое поле получит корректное значение возраста:

```
public void SetAge(int age) =>
    _Age = (16 <= age) && (age <= 120) ? age : 18;
```

В данному случае код метода упрощен благодаря оператору тела выражения методов, однако в общем случае логика обработки может быть более сложной и требовать полной формы описания метода.

Метод для чтения значения

Определение

Геттер *(«метод-получатель») – это метод, который предназначен для чтения поля, т.е. возврата его значения. Тип метода совпадает с типом возвращаемого значения поля. Этот метод только возвращает значение из закрытого поля.*

Геттеру не требуется передавать параметр, его задача только возвратить значение из закрытого в классе поля:

```
public int GetAge() => _Age;
```

Описание закрытого поля и обслуживающих его методов доступа удобно группировать:

```
Ссылок: 5
private int _Age;
Ссылок: 2
public void SetAge(int age) => _Age = (16 <= age) && (age <= 120) ? age : 18;
Ссылок: 2
public int GetAge() => _Age;
```

Рис. 3.72. Закрытое поле `_Age` и открытые методы доступа к нему для записи и чтения значения

Защита закрытого поля

Теперь в основной части программы закрытое поле `_Age` уже недоступно. Однако оно здесь и не нужно: с ним можно работать опосредовано, используя методы `SetAge()` и `GetAge()`.

- Чтобы записать значение в поле, используем метод `SetAge()` и передаем ему в качестве параметра значение для записи.
- Чтобы получить значение возраста, используем метод `GetAge()`.

```
denis.SetAge(220);
Console.WriteLine($"Возраст студента: {denis.GetAge()}");
```

```
Возраст студента: 18
Возраст студента: 18
```

Рис. 3.73. Теперь возраст полностью контролируется

Избавляемся от дублирования логики обработки

Поскольку в конструкторе возраст также проходит аналогичную проверку, то вместо дублирования ее кода можно вызвать уже описанный сеттер `SetAge()`:

```
public Student(
    long id, string? first_name, string? last_name,
    int age, EducationalDegree degree
)
{
    Id = Math.Abs(id);
    FirstName = first_name;
    LastName = last_name;
    SetAge(age);
    Degree = degree;
}
```

Таким образом мы получили полный контроль над закрытым полем `_Age`: как в момент создания объекта конструктором, так и в процессе его внешнего изменения с помощью `SetAge()`.

Доработка других методов

Что касается других методов класса, которые используют закрытое поле `_Age`, то они всегда работают с корректным значением возраста: до их вызова значение будет записано методом `SetAge()` либо в конструкторе (т.е. при создании объекта), либо в процессе изменения значения с помощью этого метода.

А это означает, что вместо обращения к полю `_Age` можно вызывать и геттер `GetAge()`:

```
public bool IsAdult() =>
    _Age >= 18;

public bool IsAdult() =>
    GetAge() >= 18;
```

Обе реализации приводят к аналогичному результату, т.к. `GetAge()` возвращает значение поля `_Age`. Если использовать Get-методы, в этом случае можно будет навесить дополнительную логику обработки внутри этого метода (если это необходимо). А при обращении к полю не будет дополнительной прослойки в виде метода, что немного экономит время вычисления.

Это полезно знать!

Методы сеттер и геттер принято называть с приставкой Set и Get соответственно.
Как правило, закрытому полю создаются два обслуживающих метода – сеттер и геттер. Этот прием сокрытия данных распространен и в других языках в языках ООП, например C++, Java.
*С помощью такого простого механизма обработки данных обеспечивается один из важнейших принципов ООП – **инкапсуляция** данных (т.е. их сокрытие).*

Преобразованный код с закрытым полем:

Глава 3\AccessMethods_2\EducationalDegree.cs

```
namespace AccessMethods
{
    enum EducationalDegree
    {
        Bachelor,
        Specialist,
        Master
    }
}
```

Глава 3\AccessMethods_2\Student.cs

```
namespace AccessMethods
{
    class Student
    {
        public long Id;
        public string? FirstName = "";
        public string? LastName = "";

        private int _Age;
        public void SetAge(int age) =>
            _Age = (16 <= age) && (age <= 120) ? age : 18;
        public int GetAge() => _Age;

        public EducationalDegree Degree;
```

```
        public Student() =>
            _Age = 18;

        public Student(
            long id, string? first_name, string? last_name,
            int age, EducationalDegree degree
        )
        {
            Id = Math.Abs(id);
            FirstName = first_name;
            LastName = last_name;
            SetAge(age);
            Degree = degree;
        }

        public override string ToString() =>
            $"ID: {Id}\n
            Имя: {FirstName}\n
            Фамилия: {LastName}\n
            Возраст: {GetAge()}\n
            Уровень образования: {Degree}";

        public bool IsAdult() =>
            GetAge() >= 18;
    }
}
```

Глава 3\AccessMethods_2\Program.cs

```
using System;

namespace AccessMethods
{
    class Program
    {
        static void Main(string[] args)
        {
            Student denis = new Student(
                12345, "Денис", "Якубович", 220,
                EducationalDegree.Specialist
            );
            Console.WriteLine($"Возраст студента:
                {denis.GetAge()}");
```

```
            denis.SetAge(220);
            Console.WriteLine($"Возраст студента:
                {denis.GetAge()}");
        }
    }
}
```

```
Возраст студента: 18
Возраст студента: 18
```

Рис. 3.74. Возраст полностью контролируется методами

В приведенном примере обратите внимание, что работа с закрытым полем `_Age` осуществляется лишь в методах доступа `SetAge()` и `GetAge()`. Конструктор и остальные методы работают через прослойку этих методов.

Закрываем внешний доступ ко всем полям

Очевидно, что подобную стратегию нужно применить ко всем полям класса `Student`. Управление этими полями должно быть обеспечено посредством парных методов-сеттеров и геттеров.

При этом:

- возраст студента должен быть в интервале (16, 120) , иначе скорректирован на 18;
- ID может быть только неотрицательным числом, иначе брать его модуль;
- для фамилии, имени и уровня образования ограничений не предусматривается.

Глава 3\AccessMethods_3\EducationalDegree.cs

```
namespace AccessMethods
{
    enum EducationalDegree
    {
        Bachelor,
        Specialist,
        Master
    }
}
```

Глава 3\AccessMethods_3\Student.cs

```
namespace AccessMethods
{
    class Student
    {
        private long _Id;
        public void SetId(long id) => _Id = Math.Abs(id);
        public long GetId() => _Id;

        private string? _FirstName = "";
        public void SetFirstName(string? first_name) =>
            _FirstName = first_name;
        public string? GetFirstName() => _FirstName;

        private string? _LastName = "";
        public void SetLastName(string? last_name) =>
            _LastName = last_name;
        public string? GetLastName() => _LastName;

        private int _Age;
        public void SetAge(int age) =>
            _Age = (16 <= age) && (age <= 120) ? age : 18;
        public int GetAge() => _Age;

        private EducationalDegree _Degree;
        public void SetDegree(EducationalDegree degree) =>
            _Degree = degree;
        public EducationalDegree GetDegree() => _Degree;

        public Student() =>
            SetAge(18);

        public Student(
            long id, string? first_name, string? last_name,
            int age, EducationalDegree degree
        )
        {
            SetId(id);
            SetFirstName(first_name);
            SetLastName(last_name);
            SetAge(age);
            SetDegree(degree);
        }
```

```
        public override string ToString() =>
            $"ID: {GetId()}\n
            Имя: {GetFirstName()}\n
            Фамилия: {GetLastName()}\n
            Возраст: {GetAge()}\n
            Уровень образования: {GetDegree()}";

        public bool IsAdult() =>
            GetAge() >= 18;
    }
}
```

Глава 3\AccessMethods_3\Program.cs

```
using System;

namespace AccessMethods
{
    class Program
    {
        static void Main(string[] args)
        {
            Student denis = new Student(
                12345, "Денис", "Якубович", 24,
                EducationalDegree.Specialist
            );
            Console.WriteLine($"Id: {denis.GetId()}");
            Console.WriteLine($"Фамилия:
                {denis.GetLastName()}");
            Console.WriteLine($"Имя: {denis.GetFirstName()}");
            Console.WriteLine($"Возраст: {denis.GetAge()}");
            Console.WriteLine($"Уровень подготовки:
                {denis.GetDegree()}\n");

            Console.WriteLine("Меняем уровень подготовки:");
            denis.SetDegree(EducationalDegree.Master);
            Console.WriteLine($"Уровень подготовки:
                {denis.GetDegree()}");
        }
    }
}
```

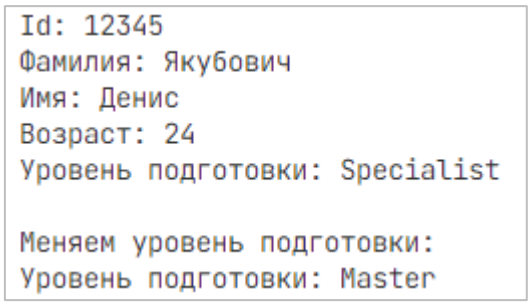


Рис. 3.75. Все поля класса закрыты, доступ к ним организован через интерфейс методов доступа (сеттер и геттер)

Каждому закрытому в классе полю назначаем по два обслуживающих метода для записи и чтения значения.

В конструкторе используются описанные методы сеттеры, которые проверят и запишут корректные значения в поля.

Теперь внешнее управление данными объектов контролируется только посредством методов. По существу при использовании класса `Student` теперь не требуется задумываться о наличие полей: достаточно обращаться к методам.

3.7.3. Поле только для чтения

Если требуется, чтобы значение поля можно было задать только один раз и в момент создания объекта, в этом случае поле делается доступным **только для чтения**.

Для этого метод-сеттер убирается из реализации класса, а необходимые установочные операции с закрытым полем осуществляются только в конструкторе класса. При этом метод-геттер остается: с помощью него в любой момент можно получить необходимое значение.

```
private int _Age;
public int GetAge() => _Age;
```

Теперь установить значение возраста можно только в момент создания объекта. Внешне изменить его невозможно:

```
public Student(
    long id, string first_name, string last_name,
    int age,
    EducationalDegree degree
)
{
    Id = Math.Abs(id);
```

```
        FirstName = first_name;
        LastName = last_name;
        _Age = (16 <= age) && (age <= 120) ? age : 18;
        Degree = degree;
    }
```

3.7.4. Проверка логики в геттерах

Однако проверять корректность значения, которое записывается в поле, необязательно в методе-сеттере. Ее можно привязывать и к методам-геттерам.

А именно, можно допустить, что метод-сеттер записывает значение в поле без предварительной проверки, а метод-геттер проверяет его и при необходимости корректирует, прежде чем возвратит значение.

Так, можно применить следующий подход:

```
private int _Age;
public void SetAge(int age) => _Age = age;
public int GetAge() =>
    (16 <= _Age) && (_Age <= 120) ? _Age : 18;
```

Результат получится аналогичным.

Однако при такой «замене ролей» нужно учитывать важное отличие:

- Если логика проверки привязана к Set-методу, то в поле записывается при необходимости исправленное значение, а оригинальное в этом случае теряется. Get-метод всегда возвращает уже обработанное значение.
- Если логика проверки привязана к Get-методу, то Set-методом в поле записывается оригинальное значение. А при чтении Get-методом возвращается исправленное значение. Т.е. оригинальное значение уже не теряется.

Как правило, используется именно первый подход, поскольку на практике редко требуется хранить в закрытых полях некорректные значения и исправлять их в момент считывания. Тем не менее такая возможность тоже поддерживается.

3.7.5. Решение задачи

Постановка задачи

Создадим класс `Rectangle`, описывающий прямоугольники и их свойства. Класс описывается двумя закрытыми полями (ширина и высота), а дополнительная информация извлекается с помощью методов.

В структуру класса должны войти:

- конструктор по умолчанию (создает прямоугольник размером 0 x 0);
- конструктор с двумя параметрами (создает прямоугольник);
- `ToString()` возвращает размеры прямоугольника;
- `GetPerimeter()` возвращает периметр прямоугольника;
- `GetSquare()` возвращает площадь прямоугольника;
- `IsSquare()` проверяет, является ли прямоугольник квадратом.

Решение

Поля, характеризующие высоту и ширину, делаем закрытыми. Сеттеры не описываем, чтобы не было возможности менять эти параметры в процессе. Установить высоту и ширину можно будет только при создании объекта, т.е. в конструкторе.

При попытке создать прямоугольник с отрицательными или нулевыми сторонами сбросим их к значению нуль. Необходимую информацию будем извлекать с помощью методов.

В основной части программы результат в начале будем накапливать в текстовую переменную и далее выводить целиком одной командой.

Глава 3\Figures\Rectangle.cs

```
namespace Figures
{
    class Rectangle
    {
        private double _height;
        public double GetHeight() => _height;

        private double _width;
        public double GetWidth() => _width;
```

```
        public Rectangle() { }

        public Rectangle(double height, double width)
        {
            _height = (height > 0) ? height : 0;
            _width = (width > 0) ? width : 0;
        }

        public override string ToString() =>
            $"[{_height} x {_width}]";

        public double GetPerimeter() =>
            2 * (_height + _width);

        public double GetSquare() =>
            _height * _width;

        public double GetDiagonal() =>
            Math.Sqrt(_height * _height + _width * _width);

        public bool IsSquare() =>
            _height == _width;
    }
}
```

Глава 3\Figures\Program.cs

```
using System;

namespace Figures
{
    class Program
    {
        static void Main(string[] args)
        {
            Rectangle rec = new Rectangle(3.5, 6);

            string? text_result =
                $"Прямоугольник: {rec}.\n" +
                $"Ширина: {rec.GetWidth():#.###} ед.\n" +
                $"Высота: {rec.GetHeight():#.###} ед.\n" +
                $"Периметр: {rec.GetPerimeter():#.###} ед.\n" +
                $"Площадь: {rec.GetSquare():#.###} ед².\n" +
                $"Диагональ: {rec.GetDiagonal():#.###} ед.\n";
```

```
            if (rec.IsSquare())
                text_result += "Также является квадратом";

            Console.WriteLine(text_result);
        }
    }
}
```

```
Прямоугольник: [3,5 x 6].
Ширина: 6 ед.
Высота: 3,5 ед.
Периметр: 19 ед.
Площадь: 21 ед².
Диагональ: 6,946 ед.
```

Рис. 3.76. Оформление результатов вычислений

Здесь необходимо обратить внимание на то, что в методах для чтения значений высоты и ширины не используются методы `GetHeight()` и `GetWidth()`, а идет обращение к закрытым полям. Это сделано в силу того, что:

- внутри класса мы уверены, что их значения корректны;
- вызов методов-геттеров является прослойкой, которая требует дополнительного времени на обработку.

3.7.6. Общий итог

Методы сеттер и геттер хороши тем, что организуют взаимодействие по принципу черного ящика. При внешнем использовании объектов не нужно знать, какие поля есть в классе и как они взаимодействуют. Управление состоянием и поведением объекта осуществляется через интерфейс методов доступа.

На самом деле один метод может управлять сразу несколькими полями.

Общая схема работы с геттером и сеттером на примере одного поля приведена на рис. 3.77.

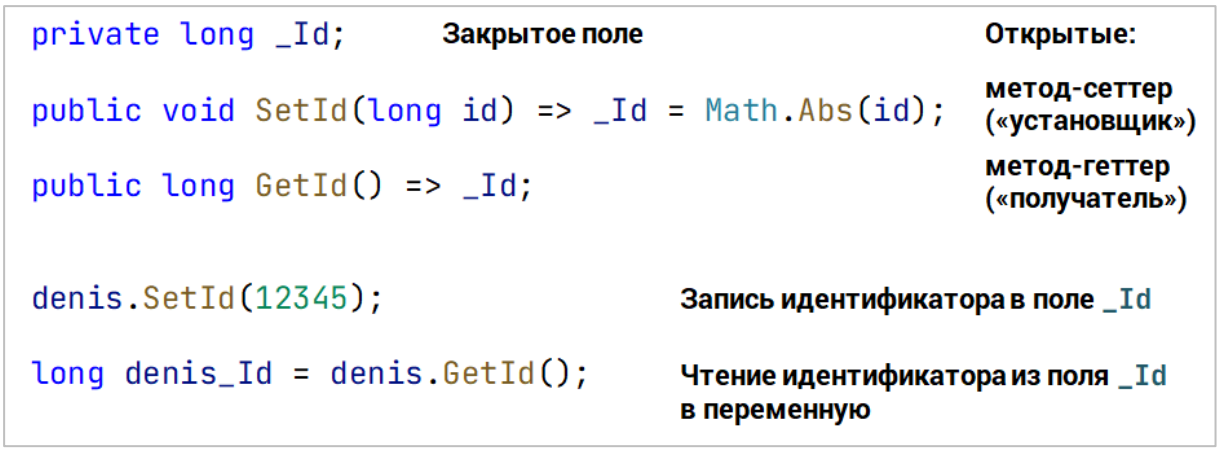


Рис. 3.77. Методы сеттер и геттер для управления доступа к внутренним данным

Вопросы для самопроверки

1. Какие потенциальные проблемы могут возникнуть в случае открытых полей класса?
2. Что представляют собой методы доступа и какие задачи они решают?
3. Опишите синтаксис и принцип работы с методом-сеттером.
4. Опишите синтаксис и принцип работы с методом-геттером.
5. Какие варианты обработки логики геттеров и сеттеров могут использоваться для организации контроля данных?

Практикум

Задание 1

1. Создайте проект *AccessMethods_3*.
2. Последовательно повторите и воспроизведите рассмотренный в параграфе пример с закрытыми полями и методами доступа.
3. В качестве эксперимента попробуйте добавить методы, которые ранее были описаны в рамках проектов *Institute*.
4. Можно ли вместо вызова Get-методов внутри методов класса работать с полями напрямую так, чтобы не нарушить логику и безопасность кода, т.е. принцип инкапсуляции?

Задание 2

1. Создайте проект *Figures*.
2. Последовательно воспроизведите рассмотренный в параграфе пример с закрытыми полями и методами доступа.
3. Дополните проект следующими компонентами:
   - конструктор класса с одним параметром, создающим прямоугольник;
   - `GetAngleBetweenHeightDiag()` – возвращает угол в градусах между диагональю и высотой;
   - `GetAngleBetweenWidthDiag()` – возвращает угол в градусах между диагональю и шириной.

```
Прямоугольник: [7 x 3,5].
Ширина: 3,5 ед.
Высота: 7 ед.
Периметр: 21 ед.
Площадь: 24,5 ед².
Диагональ: 7,826 ед.
Угол между диагональю и высотой: 26,565°.
Угол между диагональю и шириной: 63,435°.
```

Рис. 3.78. Результат работы программы из задания 2

Задание 3

1. Создайте проект *Account_13*.
2. За основу возьмите проект *Account_12* (или другой).
3. Далее:
   - Сделайте поле `Age` закрытым и переименуйте его в `_Age`. Организуйте доступ на запись и чтение поля посредством методов (т.е. методы по типу сеттер и геттер). При этом возраст должен быть задан в пределах от 16 до 100, иначе установить 18.
   - По аналогии закройте доступ ко всем остальным полям и реализуйте механизм доступа посредством методов.
4. Продемонстрируйте работу с методами сеттерами и геттерами в основной части программы.

3.8. Свойства класса

3.8.1. Недостатки сеттеров и геттеров

Механизм работы методов доступа

Методы доступа к закрытым полям (геттеры и сеттеры) решают задачу сокрытия данных класса. Все преобразования данных в полях изолированы от внешнего использования, а необходимые внешние операции осуществляются с помощью открытых методов. Это один из простых и универсальных шаблонов инкапсуляции данных.

Вернемся к последнему примеру с классом `Student` (см. *AccessMethods_3*). В нем поля сделаны закрытыми в классе, каждое из них обслуживает метод-сеттер (записывает значение в поле) и метод геттер (читает значение из поля). В некоторых сеттерах предусмотрена дополнительная проверка предлагаемого значения, что позволяет безопасно управлять данными. Также методы-сеттеры используются и в конструкторе, попутно упрощая его код.

Для удобства каждое поле сгруппировано с его сеттером и геттером (что, вообще говоря, необязательно). Например, для поля возраста:

```
private int _Age;
public void SetAge(int age) =>
    _Age = (16 <= age) && (age <= 120) ? age : 18;
public int GetAge() => _Age;
```

Недостатки сеттеров и геттеров

Однако у предложенного шаблона есть и ряд недостатков.

1. *Некоторая избыточность*. Каждому закрытому полю создаются обслуживающие методы. Приходится следовать определенному соглашению по именованию таких методов и приписывать приставки Set и Get.
2. *Несвойственное поведение*. Оба метода оперируют полем, записывая и извлекая данные из него. Они отвечают за установку и получение количественно качественной характеристики экземпляра класса. Иными словами, мы делегируем методам роль «умных» полей. Но у методов класса иная роль – в идеале они должны осуществлять извлечение

или преобразование данных, которое необходимо в разных операциях.

Например, возраст студента – это числовая характеристика объекта. Если нам требуется проверить, что студент старше 20 лет, то определим следующее условие:

```
if (denis.GetAge() > 20)
    Console.WriteLine(
        $"{denis.GetFirstName()} старше 20"
    );
```

Вызывая методы `GetAge()` и `GetFirstName()` мы рассматриваем их в форме функций объекта `denis` – «получить возраст» и «получить имя». Такой код понятен и отлично работает, но в некотором смысле противоречит роли методов. Логичнее считать, что поля – это данные, а методы – механизм управления данными.

Иными словами, более приемлемым будет синтаксис, где обращение к возрасту, имени и т.д. будет производится как к характеристике (свойству) объекта, а не как к его функции.

Таким образом основная цель – получить синтаксис, в котором к характеристике объекта можно обращаться как ранее к полю:

```
if (denis.Age > 20)
    Console.WriteLine($"{denis.FirstName} старше 20");
```

С другой стороны, желательно сохранить возможность контролировать записываемое значение и делать это неявно:

```
// Будет записано 18, т.к. 220 не входит
// в ограничение (16, 120)
denis.Age = 220;
```

Вместо	Желаем
`denis.SetAge(20);`	`denis.Age = 20;`
Вместо	**Желаем**
`int denis_age = denis.GetAge();`	`int denis_age = denis.Age;`

Рис. 3.79. Синтаксис, к которому стремимся

Эту задачу в языке C# решают специальные члены класса под названием *свойства*.

3.8.2. Синтаксис свойства

Понятие свойства

Определение

Свойство *– член класса, содержащий методы доступа к закрытым полям.*

```
доступ тип имя
{
    set { . . . }
    get { . . . }
}
```

Свойства сочетают в себе краткость полей и функциональность методов. Они реализованы на основе использования методов-сеттеров и геттеров, но в более короткой и удобной форме. Для этого в них реализованы специальные методы-аксессоры get и set:

- блок **`set`** – метод-сеттер;
- блок **`get`** – метод-геттер.

Преимущества описания свойств:

- свойства управляют доступом к закрытым полям класса;
- позволяют реализовать дополнительную автоматическую логику обработки данных при записи данных;
- в C# свойства доступ к данным классам обычно осуществляется именно через свойства, а не прямое обращение к полям.

Описание свойства на примере

Описание свойства в C# включает следующие компоненты:

1. модификатор доступа;
2. модификатор `static` (если оно статичное);
3. название свойства;
4. `get` и `set`-методы, которые реализуют методы доступа.

Опишем свойство Age, которое будет управлять закрытым полем _Age:

```
private int _Age;
public int Age
{
    set
    {
        _Age = (16 <= value) && (value <= 120)
               ? value
               : 18;
    }
    get
    {
        return _Age;
    }
}
```

В начале задается закрытое поле _Age, для которого опишем свойство с названием Age.

Свойство Age уже открыто. Тип совпадает с типом поля _Age. Оно нестатично, как и поле. Далее фигурные скобки размечают тело свойства.

В теле свойства описаны методы сеттер и геттер, но в более короткой форме.

Метод **set** – аналог SetAge(). Ранее методу SetAge() передавался параметр – значение, которое нужно записать в поле _Age. В свойствах методу set он передается в специальное ключевое слово **value**.

Метод **get** – аналог GetAge(). Он только возвращает значение из закрытого поля _Age.

Теперь для обращения к данным объекта нам доступен следующий синтаксис:

```
denis.Age = 220;
Console.WriteLine($"Возраст: {denis.Age}");
```

В случае присваивания (записи) сработает set-метод свойства Age, значение будет неявно скопировано в переменную value, проверено и далее в поле _Age запишется при необходимости исправленное значение.

При выводе на экран (чтении) сработает get-метод свойства Age и возраст будет считан из закрытого поля _Age.

Сокращенный вариант

Если тело методов `set` и `get` содержит одну инструкцию, то можно использовать оператор тела выражения:

```
private int _Age;
public int Age
{
    set => _Age = (16 <= value) && (value <= 120)
                  ? value : 18;
    get => _Age;
}
```

Синтаксис свойств удобен тем, что связанные с полем `set` и `get` методы объединяются в едином блоке.

Использование свойства в конструкторе

Поскольку свойство `Age` реализовано, то его можно использовать далее во всем классе.

В первую очередь – в конструкторе:

```
public Student(
    long id, string? first_name, string? last_name,
    int age,
    EducationalDegree degree
)
{
    SetId(id);
    SetFirstName(first_name);
    SetLastName(last_name);
    Age = age;
    SetDegree(degree);
}
```

Свойству `Age` присваиваем значение из параметра. В этом случае срабатывает set-метод свойства `Age`, осуществляется проверка значения `age` из параметра конструктора и запись в поле `_Age`.

Свойства в других методах класса

В методах класса, например `IsAdult()`, вместо обращения к закрытому полю теперь можно обращаться к созданному свойству:

```
public bool IsAdult() =>
    _Age >= 18;
```

```
public bool IsAdult() =>
    Age >= 18;
```

Так во второй форме реализации при обращении к свойству сработает метод get, который и возвратит значение возраста из закрытого поля для дальнейшей проверки условия.

Разумеется, методы (внутри класса) также могут продолжать управлять состоянием поля _Age напрямую. Однако в этом случае важно следить, чтобы не создавалось логических противоречий со свойством Age.

Здесь, как и в случае с методами сеттерами и геттерами, нужно учитывать, что:

- при обращении к закрытому полю значение извлекается напрямую, однако это не так удобно, если требуется добавить некоторую логику обработки на возвращаемое значение;
- при обращении к свойству вначале выполняется get-метод, что несколько замедляет выполнение, однако позволяет замкнуть внутри get-метода дополнительную логику обработки возвращаемого из поля значения.

Описание свойств для всех полей класса

По аналогии для всех полей класса Student из проекта *AccessMethods_3* реализуем свойства. Ранее описанные методы Set- и Get- уберем: их функционал полностью заменят свойства.

Каждому закрытому полю назначаем открытое свойство.

Ранее описанные Set- и Get-методы заменяем set и get методами свойств.

Глава 3\Properties_1\EducationalDegree.cs

```
namespace Properties
{
    enum EducationalDegree
    {
        Bachelor,
        Specialist,
        Master
    }
}
```

Глава 3\Properties_1\Student.cs

```
namespace Properties
{
    class Student
    {
        private long _Id;
        public long Id
        {
            set => _Id = Math.Abs(value);
            get => _Id;
        }

        private string? _FirstName = "";
        public string? FirstName
        {
            set => _FirstName = value;
            get => _FirstName;
        }

        private string? _LastName = "";
        public string? LastName
        {
            set => _LastName = value;
            get => _LastName;
        }

        private int _Age;
        public int Age
        {
            set => _Age = (16 <= value) && (value <= 120)
                        ? value : 18;
            get => _Age;
        }

        private EducationalDegree _Degree;
        public EducationalDegree Degree
        {
            set => _Degree = value;
            get => _Degree;
        }
```

```
        public Student() =>
            Age = 18;

        public Student(
            long id, string? first_name, string? last_name,
            int age, EducationalDegree degree
        )
        {
            Id = id;
            FirstName = first_name;
            LastName = last_name;
            Age = age;
            Degree = degree;
        }

        public override string ToString() =>
            $"ID: {Id}\n
            Имя: {FirstName}\n
            Фамилия: {LastName}\n
            Возраст: {Age}\n
            Уровень образования: {Degree}";

        public bool IsAdult() =>
            Age >= 18;
    }
}
```

Глава 3\Properties_1\Program.cs

```
using System;

namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Student denis = new Student(
                12345, "Денис", "Якубович", 24,
                EducationalDegree.Specialist
            );

            Console.WriteLine($"Первоначальный Id:
                {denis.Id}");
```

```
            denis.Id = -55555;

            Console.WriteLine($"Новый Id: {denis.Id}");
        }
    }
}
```

```
Первоначальный Id: 12345
Новый Id: 55555
```

Рис. 3.80. Результат работы программы с описанными в классе свойствами

В конструкторах и методах вместо полей теперь обращаемся к свойствам, которые их обслуживают. В методах эта замена была необязательной, но она позволяет сохранить единообразие кода.

Как и в случае сеттеров и геттеров, удалось закрыть поля класса от внешнего вмешательства, однако теперь работать с данными можно как со свойствами объекта.

Общий итог

Общая схема реализации свойства для записи и чтения значения выглядит следующим образом:

```
private long _Id;       Закрытое поле
public long Id          Открытое свойство
{                                          Открытые:
    set => _Id = Math.Abs(value);          метод-сеттер («установщик»)
    get => _Id;                            метод-геттер («получатель»)
}


denis.Id = 12345;                  Запись идентификатора в поле _Id

long denis_id = denis.Id;          Чтение идентификатора из поля _Id
                                   в переменную
```

Рис. 3.81. Свойства объединяют в себе методы сеттер и геттер для управления доступа к внутренним данным (полям)

Это важно знать!

В практике .NET разработчиков действует следующее правило: поля класса всегда нужно описывать закрытыми, а доступ к ним реализовывать через синтаксис свойств.

3.8.3. Автоматические свойства

Далеко не все свойства в классе требуют дополнительных проверок или корректировок значений.

Так в примере *Properties_1* свойства `FirstName`, `LastName` и `Degree` не накладывают ограничений на значение: методы `set` записывают значение, `get` – его возвращают.

Для таких случаев можно использовать автоматические свойства. Здесь методам доступа `get` и set не требуется задавать какой-либо реализации.

Например, для свойства FirstName полная форма реализации:

```
private string _FirstName = "";
public string FirstName
{
    set => _FirstName = value;
    get => _FirstName;
}
```

может быть свернута к автоматическому свойству:

```
public string FirstName { get; set; } = "";
```

Можно также заметить, что для автоматического свойства не указано закрытое поле. На самом деле оно создается компилятором неявно автоматически.

Кроме того, автоматическим свойствам можно задавать значения по умолчанию.

3.8.4. Вычисляемые свойства

Свойства с аксессором set

Если от свойства требуется только возврат некоторого вычисленного значения, которое зависит от полей или свойств, то свойство

является вычислимым. В этом случае метод `set` в свойстве не реализуют, а реализуют только метод `get`.

Так, в классе `Student` напрашивается замена метода `IsAdult()` на свойство: проверка на совершеннолетие – это проверка соответствия объекта определенному признаку. Оно не будет вносить изменения в поля, поэтому используем только метод доступа `get`:

```
public bool IsAdult
{
    get { return Age >= 18; }
}
```

Более того, если тело свойства состоит из одной команды, то свойство только для чтения можно свернуть в короткую форму:

```
public bool IsAdult => Age >= 18;
```

Доработка проекта

Внесем изменения в проект *Properties_1*:

- Заменим свойства на автоматические, где возможно.
- Заменим метод `IsAdult()` на одноименное свойство.
- Добавим свойство `FullName`, возвращающее строку в виде «Фамилия Имя».
- Скопируем метод `EducationalType()` из проекта `Institute` и переделаем его в вычисляемое свойство. Воспользуемся им в методе `ToString()` (чтобы уровень подготовки писался на русском языке).

Глава 3\Properties_2\EducationalDegree.cs

```
namespace Properties
{
    enum EducationalDegree
    {
        Bachelor,
        Specialist,
        Master
    }
}
```

Глава 3\Properties_2\Student.cs

```
namespace Properties
{
    class Student
    {
        private long _Id;
        public long Id
        {
            set => _Id = Math.Abs(value);
            get => _Id;
        }

        public string? FirstName { get; set; } = "";

        public string? LastName { get; set; } = "";

        private int _Age;
        public int Age
        {
            set => _Age = (16 <= value) && (value <= 120)
                          ? value : 18;
            get => _Age;
        }

        public EducationalDegree Degree { get; set; }

        public Student() =>
            Age = 18;

        public Student(
            long id, string? first_name, string? last_name,
            int age, EducationalDegree degree
        )
        {
            Id = id;
            FirstName = first_name;
            LastName = last_name;
            Age = age;
            Degree = degree;
        }

        public override string ToString() =>
            $"ID: {Id}\n
            Имя: {FirstName}\n
```

```
            Фамилия: {LastName}\n
            Возраст: {Age}\n
            Уровень образования: {EducationalType}";

        public string? FullName =>
            (FirstName != null) || (LastName != null)
            ? $"{LastName} {FirstName}"
            : null;

        public bool IsAdult =>
            Age >= 18;

        public string EducationalType =>
            Degree switch
            {
                EducationalDegree.Bachelor => "бакалавриат",
                EducationalDegree.Specialist => "специалитет",
                EducationalDegree.Master => "магистратура",
                _ => "не определено"
            };
    }
}
```

Глава 3\Properties_2\Program.cs

```
using System;

namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Student denis = new Student(
                12345, "Денис", "Якубович", 21,
                EducationalDegree.Specialist
            );

            Console.WriteLine("Общая информация:");
            Console.WriteLine(denis);
            Console.WriteLine(
                "Фамилия и имя: {denis.FullName}"
            );
```

```
            if (denis.IsAdult)
                Console.WriteLine(
                    $"Студент совершеннолетний"
                );
            else
                Console.WriteLine(
                    $"Студент несовершеннолетний"
                );

            // Console.WriteLine(denis.IsAdult
            // ? " Студент совершеннолетний"
            // : " Студент несовершеннолетний");
        }
    }
}
```

```
Общая информация:
ID: 12345
Имя: Денис
Фамилия: Якубович
Возраст: 21
Уровень образования: специалитет
Фамилия и имя: Якубович Денис
Студент совершеннолетний
```

Рис. 3.82. Работа с автоматическими и вычисляемыми свойствами

3.8.5. Свойства только для чтения

Модификатор **readonly** в описании поля означает, что его значение можно задать только во время создания объекта или в конструкторе класса. После этого оно не может быть изменено, а может быть только читаться, например – свойством.

```
public class Person
{
    private readonly string _Name;
    public string Name => _Name;

    public Person(string name)
    {
        _Name = name;
    }
}
```

В приведенном примере поле _Name описано как поле только для чтения. Его значение может быть задано только в конструкторе объекта, а далее только читаться через интерфейс свойства Name.

```
Person person = new Person("Денис");

person.Name = "Андрей";                 // Ошибка!
Console.WriteLine(person.Name);         // OK
```

Этот механизм также может быть реализован и для автоматического свойства:

```
public class Person
{
    public string Name { get; }

    public Person(string name)
    {
        Name = name;
    }
}
```

3.8.6. Название полей и свойств

Разработчики C# используют следующие соглашения именования полей и свойств.

1. Закрытые поля класса принято называть с нижнего подчеркивания и далее маленькой буквы. (Однако в наших примерах мы не следовали этому правилу и называли поля с заглавной буквы)
2. Название свойств задаются с заглавной буквы.
3. В конструкторе класса и методах передаваемые параметры пишутся с маленькой буквы, как локальные переменные.
4. Название поля, свойства и параметра рационально делать одинаковым, чтобы не было путаницы. Для названий из нескольких слов можно использовать Camel- или Snake-нотацию.

Рис. 3.83. Правила именования полей, методов и свойств

3.8.7. Пример с описанием векторов

Формулировка задачи

Необходимо создайте проект *VectorSpace* и описать класс `Vector`, задающий векторы на плоскости.

Далее реализовать в классе компоненты (методы и свойства) для получения следующих данных:

- текстовое представление вектора в форме (x,y);
- вычисление длины вектора;
- проверку, является ли вектор нулевым или базисным;
- сумму двух векторов;
- разность двух векторов;
- умножение вектора на число;
- скалярное произведение двух векторов.

1. Реализация основы класса Vector

Основные свойства и методы класса

Основными свойствами вектора будут его координаты. Делаем их автоматическими, т.к. координаты могут быть любые и ограничиваться только рамками типа `double`. Они должны быть доступны как на запись, так и на чтение.

Конструктор по умолчанию будет создавать нуль-вектор. Конструктор с параметрами создает произвольный вектор.

```
public class Vector
{
    public double X { get; set; }
    public double Y { get; set; }

    public Vector() { }

    public Vector(double x, double y)
    {
        X = x;
        Y = y;
    }
}
```

Метод `ToString()` переопределим для возврата записи вектора в формате `(X; Y)`:

```
public class Vector
{
    // Поля и конструкторы выше...

    public override string ToString() =>
        $"({X}; {Y})";
}
```

Свойства `Length`, `IsZero`, `IsUnit` рассчитывают новые характеристики объекта относительно известных данных (координат), поэтому опишем как свойства

```
public class Vector
{
    // Поля, конструкторы и методы выше...

    public double Length =>
        Math.Sqrt(X*X + Y*Y);

    public bool IsZero =>
        X == 0 && Y == 0;

    public bool IsUnit =>
        (X == 1 && Y == 0) || (X == 0 && Y == 1);
}
```

Арифметика с векторами выносится в статические методы, т.к. операции не являются свойствами или функциями объектов, это – преобразования над объектами. Векторы и числа, над которыми осуществляются операции, передаем в параметрах.

Для суммы, разности векторов и умножения числа на вектор тип метода описываем классом `Vector`, т.к. методы тоже возвращают вектор. А результатом скалярного произведения векторов является число

```
public class Vector
{
    // Поля, конструкторы и методы выше...

    public static Vector Sum(Vector a, Vector b) =>
        new Vector(a.X + b.X, a.Y + b.Y);

    public static Vector Sub(Vector a, Vector b) =>
        new Vector(a.X - b.X, a.Y - b.Y);

    public static Vector Mul(Vector a, double k) =>
        new Vector(k * a.X, k * a.Y);

    public static double DotProduct(
        Vector a, Vector b
    ) =>
        a.X * b.X + a.Y * b.Y;
}
```

Полный код программы

Глава 3\VectorSpace_1\Vector.cs

```
namespace VectorSpace
{
    public class Vector
    {
        public double X { get; set; }
        public double Y { get; set; }

        public Vector() { }

        public Vector(double x, double y)
        {
            X = x;
            Y = y;
        }
```

```
        public override string ToString() =>
            $"({X}; {Y})";

        public double Length =>
            Math.Sqrt(X * X + Y * Y);

        public bool IsZero =>
            X == 0 && Y == 0;

        public bool IsUnit =>
            (X == 1 && Y == 0) || (X == 0 && Y == 1);

        public static Vector Sum(Vector a, Vector b) =>
            new Vector(a.X + b.X, a.Y + b.Y);

        public static Vector Sub(Vector a, Vector b) =>
            new Vector(a.X - b.X, a.Y - b.Y);

        public static Vector Mul(Vector a, double k) =>
            new Vector(k * a.X, k * a.Y);

        public static double DotProduct(Vector a, Vector b) =>
            a.X * b.X + a.Y * b.Y;
    }
}
```

Глава 3\VectorSpace_1\Program.cs

```
using System;

namespace VectorSpace
{
    class Program
    {
        static void Main(string[] args)
        {
            var vec0 = new Vector();
            var vec1 = new Vector(-3, 6);
            var vec2 = new Vector(0, 1);

            Vector[] vectors = { vec0, vec1, vec2 };

            foreach (var vector in vectors)
            {
                Console.WriteLine(
```

```
                    $"Свойства вектора {vector}"
                );
                Console.WriteLine(
                    $"Длина: {vector.Length:F5}"
                );

                if (vector.IsZero)
                    Console.WriteLine(
                        "Является нуль-вектором"
                    );

                if (vector.IsUnit)
                    Console.WriteLine("Является ортом");

                Console.WriteLine();
            }

            Vector sum = Vector.Sum(vec1, vec2);
            Vector sub = Vector.Sub(vec1, vec2);
            Vector mul = Vector.Mul(vec1, 3.5);
            double scalar_prod =
                Vector.DotProduct(vec1, vec2);

            Console.WriteLine($"Сумма = {sum}");
            Console.WriteLine($"Разность = {sub}");
            Console.WriteLine($"Умножение на число = {mul}");
            Console.WriteLine(
                $"Скалярное произведение = {scalar_prod}"
            );
        }
    }
}
```

```
Свойства вектора (0; 0)
Длина: 0,00000
Является нуль-вектором

Свойства вектора (-3; 6)
Длина: 6,70820

Свойства вектора (0; 1)
Длина: 1,00000
Является ортом

Сумма = (-3; 7)
Разность = (-3; 5)
Умножение на число = (-10,5; 21)
Скалярное произведение = 6
```

Рис. 3.84. Результат вычислений по нескольким векторам

2. Расширение функционала класса

Частные случаи векторов

Расширим класс дополнительными компонентами, чтобы повысить читаемость кода.

Выделим частные случаи векторов:

- нуль вектор;
- единичный вектор по оси X;
- единичный вектор по оси Y.

В текущей реализации создавать такие векторы необходимо с помощь конструктора класса:

```
var vec0 = new Vector();        // или  new Vector(0, 0);
var vec1 = new Vector(1, 0);
var vec2 = new Vector(0, 1);
```

Однако в таком присваивании недостаточно смысловой нагрузки, которая подчеркивает, что это какие-то особые векторы.

Зададим этим векторам имена. Один из способов – реализовать их в форме свойств только для чтения. Однако эти свойства сделаем статическими, поскольку они характеризуют не состояние какого-либо вектора, а представляют собой векторы как подвид объектов самого класса:

```
public static Vector Zero => new Vector();
public static Vector UnitX => new Vector(1,0);
public static Vector UnitY => new Vector(0,1);
```

Каждое обращение к свойству создает новый вектор и возвращает на него ссылку.

Теперь создавать нуль-вектор и единичные векторы можно следующим образом:

```
Vector vec0 = Vector.Zero;
Vector vec1 = Vector.UnitX;
Vector vec2 = Vector.UnitY;
```

Свойсва для проверки частных случаев

Поскольку введены частные случаи векторов, не помешает добавить логические свойства, которые позволяют проверить, является ли заданный вектор нуль-вектором, ортом по оси X, ортом по оси Y или просто ортом. Здесь все четыре свойства являются экземплярны-

ми, т.к. они проверяют состояние конкретного вектора (относительно которого идет проверка):

```
public bool IsZero => (X == 0) && (Y == 0);
public bool IsUnitX => (X == 1) && (Y == 0);
public bool IsUnitY => (X == 0) && (Y == 1);
public bool IsUnit => IsUnitX || IsUnitY;
```

Здесь свойство `IsUnit` упрощено, поскольку теперь можно обратиться к реализованным выше свойствам `IsUnitX` и `IsUnitY`.

Полный код программы

Глава 3\VectorSpace_2\Vector.cs

```
namespace VectorSpace
{
    public class Vector
    {
        public double X { get; set; }
        public double Y { get; set; }

        public Vector() { }

        public Vector(double x, double y)
        {
            X = x;
            Y = y;
        }

        public override string ToString() =>
            $"({X}; {Y})";

        public static Vector Zero => new Vector();
        public static Vector UnitX => new Vector(1,0);
        public static Vector UnitY => new Vector(0,1);

        public bool IsZero => (X == 0) && (Y == 0);
        public bool IsUnitX => (X == 1) && (Y == 0);
        public bool IsUnitY => (X == 0) && (Y == 1);
        public bool IsUnit => IsUnitX || IsUnitY;

        public double Length =>
            Math.Sqrt(X*X + Y*Y);

        public static Vector Sum(Vector a, Vector b) =>
            new Vector(a.X + b.X, a.Y + b.Y);
```

```
        public static Vector Sub(Vector a, Vector b) =>
            new Vector(a.X - b.X, a.Y - b.Y);

        public static Vector Mul(Vector a, double k) =>
            new Vector(k * a.X, k * a.Y);

        public static double DotProduct(Vector a, Vector b) =>
            a.X * b.X + a.Y * b.Y;
    }
}
```

Глава 3\VectorSpace_2\Program.cs

```
using System;

namespace VectorSpace
{
    class Program
    {
        static void Main(string[] args)
        {
            var vec0 = Vector.Zero;
            var vec1 = Vector.UnitX;
            var vec2 = Vector.UnitY;
            var vec3 = new Vector(1, 1);

            if (vec0.IsZero)
                Console.WriteLine(
                    $"{vec0} - это нуль-вектор"
                );

            if (vec1.IsUnitX)
                Console.WriteLine(
                    $"{vec1} - это орт по оси X"
                );

            if (vec2.IsUnitY)
                Console.WriteLine(
                    $"{vec2} - это орт по оси Y"
                );

            if (vec3.IsUnit)
                Console.WriteLine(
                    $"{vec3} - это орт по одной из осей"
```

```
                );
            else
                Console.WriteLine(
                    $"{vec3} - не является ортом"
                );
        }
    }
}
```

```
(0; 0) - это нуль-вектор
(1; 0) - это орт по оси X
(0; 1) - это орт по оси Y
(1; 1) - не является ортом
```

Рис. 3.85. Результат проверок частных случаев векторов

3. Сравнение векторов

Особенности сравнения на равенство

Для операций с векторами важной является возможность их сравнения. Однако нельзя упускать тот факт, что классы являются ссылочными типами данных. Так, следующее условие проверки ложно, поскольку ссылки на объекты `vec` и `Vector.UnitX` различны:

```
Vector vec = Vector.UnitX;

if (vec == Vector.UnitX)
    Console.WriteLine($"{vec} является ортом по оси X");
```

Связано это с тем, что каждое обращение к свойству `UnitX` создает новый вектор, т.е. объекты `vec` и `Vector.UnitX` являются разными с точки зрения хранения в памяти, хотя с математической точки зрения это два одинаковых базисных вектора.

Метод Equals()

Чтобы сравнивать векторы не по ссылкам, а по значениям координат, необходимо переопределить метод **`Equals()`**, который наследуется любым классом из базового класса `Object`.

Метод `Equals()` в качестве параметра передается объект общего типа `object`, а в теле метода осуществляется попытка его приведения к типу искомого объекта:

```
public override bool Equals(object? obj)
{
    if (obj == null || GetType() != obj.GetType())
    {
        return false;
    }
    else
    {
        Vector obj_vec = (Vector) obj;
        return (X == obj_vec.X) && (Y == obj_vec.Y);
    }
}
```

Предложенный алгоримт сравнения работает следующим образом:

- Если объект `obj` пустой или его тип не совпадает с типом текущего объекта, то сравнение прерывается с отрицательным результатом.
- Иначе `obj` – это объект класса `Vector`. Теперь можно сравнить их по координатам с текущим вектором.

Вызов метода Equals(), как метода объекта, осуществляется следующим образом:

```
var vec = new Vector(3.4, 0);

if (vec.Equals(Vector.UnitX))
    Console.WriteLine($"{vec} является ортом по оси X");
```

Метод GetHashCode()

Метод `Equals()` должен реализовываться в паре с другим наследуемым методом – **`GetHashCode()`**.

Мы не будем вдаваться в детали этого метода, упрощенно лишь отметим, что он добавлен для ускорения сравнения двух объектов. Для этого он использует **хэш-коды** (уникальные числовые идентификаторы), которые присваиваются объектам класса по определенному алгоритму.

Иными словами, если требуется узнать, одинаковы ли два объекта, то сначала сравниваются их хэш-коды. Если они различаются, то объекты различны. Если же совпадают, то тогда начинается выполняется более трудоемкое сравнение через `Equals()`.

Таким образом `GetHashCode()`:

- должен совпадать для одинаковых объектов;
- по возможности отличаться для разных объектов;
- достаточно быстро вычисляться (быстрее, чем `Equals()`).

Оформим метод следующим образом (можно взять и другой алгоритм вычисления):

```
public override int GetHashCode() =>
    X.GetHashCode() ^ Y.GetHashCode();
```

Здесь берем хэш-коды координат и связываем их побитовым оператором ЛИБО `^`. Каждая комбинация координат вектора `X` и `Y` будет давать разный итоговый хэш-код, т.е. разные векторы.

Полный код программы

Глава 3\VectorSpace_3\Vector.cs

```
namespace VectorSpace
{
    public class Vector
    {
        public double X { get; set; }
        public double Y { get; set; }

        public Vector() { }

        public Vector(double x, double y)
        {
            X = x;
            Y = y;
        }

        public override string ToString() =>
            $"({X}; {Y})";

        public static Vector Zero => new Vector();
        public static Vector UnitX => new Vector(1,0);
        public static Vector UnitY => new Vector(0,1);

        public bool IsZero => (X == 0) && (Y == 0);
        public bool IsUnitX => (X == 1) && (Y == 0);
        public bool IsUnitY => (X == 0) && (Y == 1);
        public bool IsUnit => IsUnitX || IsUnitY;
```

```
        public double Length =>
            Math.Sqrt(X*X + Y*Y);

        public static Vector Sum(Vector a, Vector b) =>
            new Vector(a.X + b.X, a.Y + b.Y);

        public static Vector Sub(Vector a, Vector b) =>
            new Vector(a.X - b.X, a.Y - b.Y);

        public static Vector Mul(Vector a, double k) =>
            new Vector(k * a.X, k * a.Y);

        public static double DotProduct(Vector a, Vector b) =>
            a.X * b.X + a.Y * b.Y;

        public override bool Equals(object? obj)
        {
            if (obj == null || GetType() != obj.GetType())
            {
                return false;
            }
            else
            {
                Vector vec = (Vector) obj;
                return (X == vec.X) && (Y == vec.Y);
            }
        }

        public override int GetHashCode() =>
            X.GetHashCode() ^ Y.GetHashCode();
    }
}
```

Глава 3\VectorSpace_3\Program.cs

```
using System;

namespace VectorSpace
{
    class Program
    {
        static void Main(string[] args)
        {
            Vector vec1 = Vector.UnitX;
```

```
            if (vec1.Equals(Vector.UnitX))
                Console.WriteLine(
                    $"{vec1} является ортом по оси X"
                );
        }
    }
}
```

```
(1; 0) является ортом по оси X
```

Рис. 3.86. Сравнение двух векторов

4. Перегрузка операторов

Наконец, более удобным будет сравнение, в котором используются операторы == и !=.

Ранее было отмечено, что язык C# поддерживает перегрузку операторов, т.е. возможность переопределять действия операторов +, -, *, /, ++, -- и т.д. для собственных классов.

Например, переопределим операторы сравнения == и !=:

```
public static bool operator ==(Vector a, Vector b) =>
    (a.X == b.X) && (a.Y == a.Y);

public static bool operator !=(Vector a, Vector b) =>
    (a.X != b.X) || (a.Y != a.Y);
```

Теперь сравнивать объекты класса `Vector` на равенство / неравенство гораздо удобнее, чем использовать `Equals()`:

```
if (vec1 == Vector.UnitX)
    Console.WriteLine($"{vec1} является ортом по оси X");
```

Ниже приведен расширенный код программы.

- В него добавлена перегрузка арифметических операторов и операций сравнения векторов на равенство и неравенство.
- Дублирующиеся операции скрыты в private-методы.
- Для разнообразия результат в начале накапливается в текстовую переменную и в конце выводится единым блоком.
- В условных проверках используются тернарные операторы.

Полный код программы

Глава 3\VectorSpace_4\Vector.cs

```
namespace VectorSpace
{
    public class Vector
    {
        public double X { get; set; }
        public double Y { get; set; }


        public Vector() { }

        public Vector(double x, double y)
        {
            X = x;
            Y = y;
        }


        public override string ToString() =>
            $"({X}; {Y})";


        public static Vector Zero => new Vector();
        public static Vector UnitX => new Vector(1,0);
        public static Vector UnitY => new Vector(0,1);

        public bool IsZero => (X == 0) && (Y == 0);
        public bool IsUnitX => (X == 1) && (Y == 0);
        public bool IsUnitY => (X == 0) && (Y == 1);
        public bool IsUnit => IsUnitX || IsUnitY;


        public double Length =>
            Math.Sqrt(X * X + Y * Y);

        private static Vector _Sum(Vector a, Vector b) =>
            new Vector(a.X + b.X, a.Y + b.Y);
        public static Vector Sum(Vector a, Vector b) =>
            _Sum(a, b);
        public static Vector operator +(Vector a, Vector b) =>
            _Sum(a, b);
```

```
        private static Vector _Sub(Vector a, Vector b) =>
            new Vector(a.X - b.X, a.Y - b.Y);
        public static Vector Sub(Vector a, Vector b) =>
            _Sub(a, b);
        public static Vector operator -(Vector a, Vector b) =>
            _Sub(a, b);


        private static Vector _Mul(Vector a, double k) =>
            new Vector(k * a.X, k * a.Y);
        public static Vector Mul(Vector a, double k) =>
            _Mul(a, k);
        public static Vector operator *(Vector a, double k) =>
            _Mul(a, k);
        public static Vector Mul(double k, Vector a) =>
            _Mul(a, k);
        public static Vector operator *(double k, Vector a) =>
            _Mul(a, k);


        public static double _DotProduct(Vector a, Vector b)
        =>
            a.X * b.X + a.Y * b.Y;
        public static double DotProduct(Vector a, Vector b) =>
            _DotProduct(a, b);
        public static double operator *(Vector a, Vector b) =>
            _DotProduct(a, b);


        public static bool operator ==(Vector a, Vector b) =>
            (a.X == b.X) && (a.Y == a.Y);

        public static bool operator !=(Vector a, Vector b) =>
            (a.X != b.X) || (a.Y != a.Y);

        public override bool Equals(object? obj)
        {
            if (obj == null || GetType() != obj.GetType())
            {
                return false;
            }
            else
            {
                Vector vec = (Vector) obj;
```

```
                return (X == vec.X) && (Y == vec.Y);
            }
        }

        public override int GetHashCode() =>
            HashCode.Combine(X, Y);
    }
}
```

Глава 3\VectorSpace_2\Program.cs

```
using System;

namespace VectorSpace
{
    class Program
    {
        static void Main(string[] args)
        {
            string? text_result = "РАБОТА С ВЕКТОРАМИ\n";

            Vector vec0 = new Vector();
            Vector vec1 = new Vector(-3, 6);
            Vector vec2 = new Vector(0, 1);

            Vector[] vectors = { vec0, vec1, vec2 };

            foreach (var vector in vectors)
            {
                text_result +=
                    $"Свойства вектора {vector}.\n" +
                    $"Длина: {vector.Length:F5}.\n" +
                    (vector.IsZero
                        ? "Является нуль-вектором."
                        : "Не является нуль-вектором.") +
                        "\n" +
                    (vector.IsUnit ?
                        "Является ортом." :
                        "Не является ортом.") +
                        "\n" +
                    (vector.IsUnitX
                        ? "Является ортом по оси Ox."
                        : "Не является ортом по оси Ox.") +
                        "\n" +
                    (vector.IsUnitY
```

```
                        ? "Является ортом по оси Oy."
                        : "Не является ортом по оси Oy.")
                        + "\n\n";
            }

            Vector sum = vec1 + vec2;
            Vector sub = vec1 - vec2;
            Vector mul = vec1 * 3.5;
            double scalar_prod = vec1 * vec2;
            Vector formula = (2 * vec1 + 1.5 * vec2) * 0.35;

            text_result += new string('-', 50) + "\n";

            text_result +=
                $"Сумма = {sum}.\n" +
                $"Разность = {sub}.\n" +
                $"Умножение на число = {mul}.\n" +
                $"Скалярное произведение = {scalar_prod}.\n" +
                $"Формула = {formula}.\n";

            text_result += new string('-', 50) + "\n";

            Vector vec3 = Vector.UnitX;

            text_result +=
                (vec3 == Vector.UnitX
                    ? $"{vec1} является ортом по оси Ox."
                    : $"{vec1} не является ортом по оси Ox.")
                    +
                    "\n" +
                (vec3.Equals(Vector.UnitX)
                    ? $"{vec1} является ортом по оси Ox."
                    : $"{vec1} не является ортом по оси Ox.");

            Console.WriteLine(text_result);
        }
    }
}
```

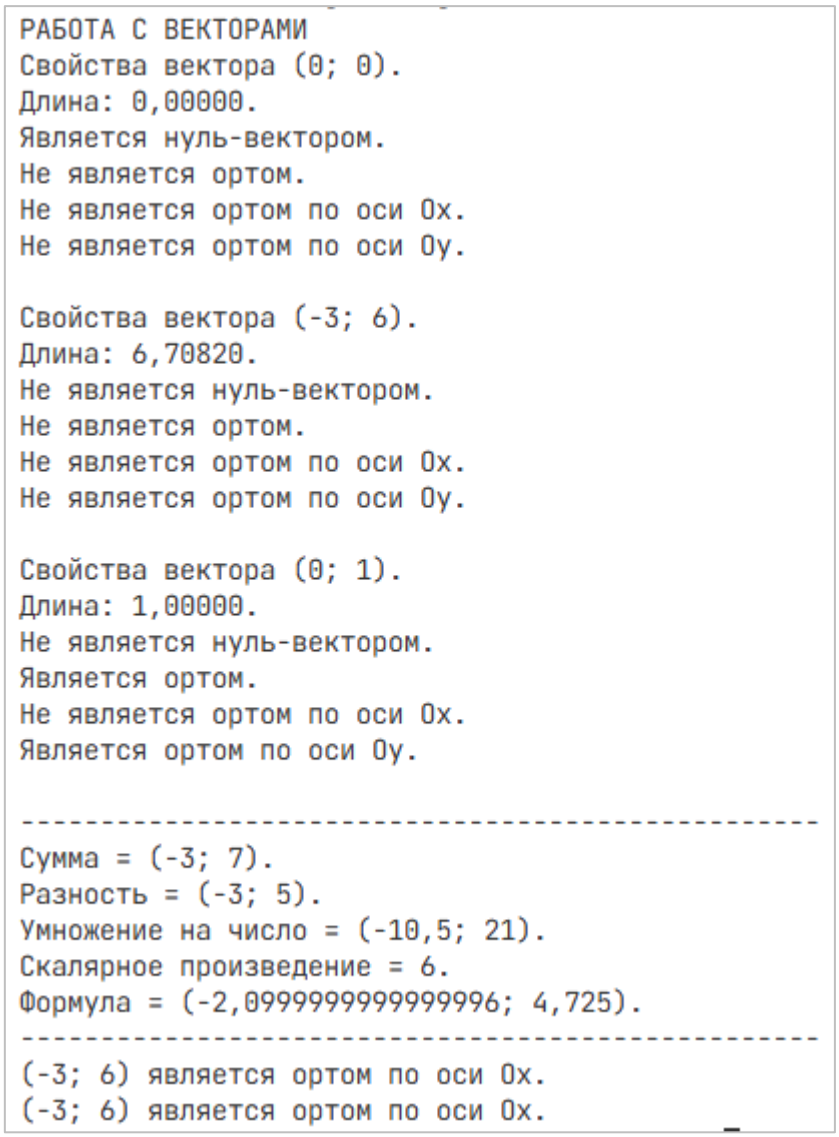


Рис. 3.87. Комплексное тестирование методов и свойств класса `Vector`

3.8.8. Рекомендации учителю

Знакомство с классами

- Важно подвести учащихся к необходимости использовать в программировании структуры данных, которые описывают объекты реального или абстрактного мира как информационные модели.
- Предложите учащимся описать собственные примеры объектов с их свойствами и функциями.
- Обобщите примеры тем, что объекты единой природы можно объединить классами объектов.

- Покажите, что в C# классы являются ключевой структурной единицей программы.
- Опишете пустой класс. Покажите, что для работы с классами важно указывать модификаторы доступа, как они влияют на допустимую область использования класса.

Изучение полей, методов, конструкторов

- Изучение следует начать примеров описания классов реальных сущностей. Покажите, что поля необходимы для хранения данных.
- Остановитесь подробнее на создании экземпляров класса. Обратите внимание на обращение к полям через оператор точку.
- Последовательно вводите в описание класса методы. Учащимся важно понять специфику работы метода внутри класса, управление данными полей.
- Постепенно демонстрируйте, что методы внутри класса могут взаимодействовать. Разберите ситуации, когда методы следует описывать закрытыми в классе.
- Рассмотрите роль конструктора класса и разные приемы его обработки.
- Проведите сравнительный анализ ситуаций, когда нужно использовать методы экземпляра и статические методы.

Методы доступа и свойства

- Обсудите с учащимися, какие проблемы и ограничения таят в себе открытые поля.
- Выработайте стратегию управления данными, подведя учащихся к необходимости использования методов доступа.
- Объясните, как правильно использовать методы доступа, рассмотрите разные варианты.
- Предложите проанализировать, почему работа с геттерами и сеттерами несколько избыточна.
- Рассмотрите синтаксис свойств как более совершенную модель методов доступа. Разберите реализованные ранее проекты, переписав методы доступа свойствами.

Вопросы для самопроверки

1. Почему геттеры и сеттеры вносят в некотором смысле противоречие между данными и логикой их использования?
2. Что представляют собой свойства и что у них общего с геттерами и сеттерами?
3. Как описываются свойства?
4. В каких случаях свойство рационально описать как автоматическое?
5. Нужно ли вычисляемому свойству собственное поле?
6. В чем особенности свойств только для чтения.

Практикум

Задание 1

1. Создайте проект *Properties*.
2. Последовательно повторите и воспроизведите рассмотренный в параграфе пример *Properties_2* с описанными свойствами.
3. В качестве эксперимента попробуйте добавить другие методы, которые ранее были описаны в рамках проектов *Institute*. Следует ли некоторые из них преобразовать в свойства?

Задание 2

1. Создайте проект *VectorSpace*.
2. Последовательно повторите и воспроизведите рассмотренный в параграфе пример *VectorSpaces_4*.
3. Внимательно изучите структуру кода, описанные свойства, их использование в конструкторе и других методах. Четко определите, в каких члучахся срабатывают `set` и `get`-методы.

Задание 3

1. Создайте проект *Account_14*.
2. За основу возьмите проект *Account_13* (или другой).

3. Далее:
    - организуйте доступ к закрытым полям через описание свойств;
    - ограничения на допустимые значения в полях `_Id` и `_Age` сохранить;
    - свойства `Login`, `FullName`, `AccessRights` опишите как автоматические.
4. Переделайте методы в вычисляемые свойства:
    - `IsAdult()` в одноименное свойство;
    - `IsAdmin()`, `IsManager()`, `IsEmployee()`, `IsGuest()` в одноименные свойства.
5. Имеет ли смысл переписать другие методы класса в виде свойств?
6. Продемонстрируйте пример изменения значений некоторых свойств и автоматический контроль их корректности.

```
var denis = new Account(
    10345, "YDA2023", "Якубович Денис Андреевич", 24, AccessRightsType.Admin
);

Console.WriteLine("Работаем со свойствами аккаунта:");
Console.WriteLine($"ID: {denis.Id}");
Console.WriteLine($"Login: {denis.Login}");
Console.WriteLine($"ФИО: {denis.FullName}");
Console.WriteLine($"Возраст: {denis.Age}");
Console.WriteLine($"Права доступа: {denis.AccessRights} ({denis.GetAccessRightsType()})\n");

Console.WriteLine($"Меняем права доступа:");
denis.AccessRights = AccessRightsType.Manager;
Console.WriteLine($"Права доступа: {denis.AccessRights} ({denis.GetAccessRightsType()})");
```

Рис. 3.88. Пример кода в блоке `Main()`

```
Работаем со свойствами аккаунта:
ID: 10345
Login: YDA2023
ФИО: Якубович Денис Андреевич
Возраст: 24
Права доступа: Admin (администратор)

Меняем права доступа:
Права доступа: Manager (менеджер)
```

Рис. 3.89. Результат работы программы из задания 3

3.9. Примеры решения задач со свойствами класса

3.9.1. Проект «Геометрия круга»

Постановка задачи

Требуется задать класс `Circle`, описывающий свойства окружности заданного радиуса.

- Определить свойство `Radius`, хранящее радиус окружности. При попытке задать свойству отрицательное значение инициализировать его нулем.
- Задать конструкторы с параметром и без.
- Переопределить метод `ToString()` для возврата строки с данными по объекту.
- Определить свойство `Length`, возвращающее длину окружности.
- Определить свойство `Square`, возвращающее площадь круга.
- Оформить вывод в следующей форме:

```
Окружность: C(R = 2,4)
Радиус окружности: 2,400 ед
Длина окружности: 15,080 ед
Площадь круга: 18,096 ед²
```

Рис. 3.90. Ожидаемое оформление результата работы программы

Комментарии к реализации

Основным компонентом, характеризующим круг, является радиус. Поэтому начнем описание класса `Circle` с определения закрытого поля `_Radius` и свойства `Radius`.

```
namespace Geometry
{
    class Circle
    {
        private double _Radius;
        public double Radius
        {
            set => _Radius = (value > 0) ? value : 0;
            get => _Radius;
```

```
        }
    }
}
```

Следующим шагом определим конструкторы объектов:

```
namespace Geometry
{
    class Circle
    {
        // Поля и свойства выше ...

        public Circle() { }

        public Circle(double radius) =>
            Radius = radius;
    }
}
```

В приведенном примере конструктор с параметром за счет обращения к совйству `Radius` проконтролирует корректность записи значения.

Следующим шагом определим строковую запись об объекте:

```
namespace Geometry
{
    class Circle
    {
        // Поля, свойства и конструктор выше ...

        public override string ToString() =>
            $"C(R = {Radius})";
    }
}
```

Для вычисления длины окружности и площади круга потребуется обращение к числу π. Чтобы не обращаться к нему через класс `Math`, запишем его в закрытое поле нашего класса: это позволит писать более короткие формулы.

```
namespace Geometry
{
    class Circle
    {
        private const double PI = Math.PI;
```

```
        // Поля, свойства и конструктор выше ...

        public double Lenght =>
            2 * PI * Radius;

        public double Square =>
            PI * Radius * Radius;
    }
}
```

Свойства Length и Square описаны как вычисляемые свойства: они не имеют собственного поля и используют данные других свойств. Их рационально описать именно свойствами, поскольку в отличие от методов длина окружности и площадь круга описывают параметры самого объекта.

Программный код

Глава 3\Geometry_1\Circle.cs

```
namespace Geometry
{
    class Circle
    {
        private const double PI = Math.PI;

        private double _Radius;
        public double Radius
        {
            set => _Radius = (value > 0) ? value : 0;
            get => _Radius;
        }

        public Circle() { }

        public Circle(double radius) =>
            Radius = radius;

        public override string ToString() =>
            $"C(R = {Radius})";

        public double Lenght =>
            2 * PI * Radius;
```

```
        public double Square =>
            PI * Radius * Radius;
    }
}
```

Глава 3\Geometry_1\Program.cs

```
using System;

namespace Geometry
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Введите радиус: ");
            string? text_input = Console.ReadLine();

            if (
                !double.TryParse(
                    text_input, out double radius
                )
            )
                radius = 0;

            Circle circle = new Circle(radius);
            Console.WriteLine($"Окружность: {circle}");
            Console.WriteLine($"Радиус окружности:
                {circle.Radius:F3} ед");
            Console.WriteLine($"Длина окружности:
                {circle.Lenght:F3} ед");
            Console.WriteLine($"Площадь круга:
                {circle.Square:F3} ед²");
        }
    }
}
```

```
Введите радиус: 5,2
Окружность: C(R = 5,2)
Радиус окружности: 5,200 ед
Длина окружности: 32,673 ед
Площадь круга: 84,949 ед²
```

Рис. 3.91. Результат работы программы

Расширение функционала программы

Если известна длина окружности, несложно однозначно выразить и величину радиуса. Аналогично и для площади круга.

Это позволяет изменить описание свойств `Length` и `Square` таким образом, чтобы при записи в них значения автоматически обновлялся (т.е. менялся) радиус окружности. А при попытке записи отрицательных значений взять его равным нулю.

С технической точки зрения это означает, что свойства `Length` и `Square` нужно реализовать в полной форме, где логика обработки будет задана как `set`, так и `get`-методу.

Для свойства `Length`, как и ранее, метод get возвращает длину окружности. А метод set пересчитывает радиус (выведите формулу самостоятельно):

```
namespace Geometry
{
    class Circle
    {
        // Компоненты выше ...

        public double Lenght
        {
            set => Radius = value / (2 * PI);
            get => 2 * PI * Radius;
        }
    }
}
```

Обратите внимание, что нам не требуется дополнительно проверять, какое значение будет присвоено свойству `Length`. Если оно будет отрицательно, то выражение в правой части также отрицательно. Присвоение его свойству `Radius` вызовет метод `set` этого свойства, который и запишет нуль, что и требуется.

Аналогичным образом выражаем радиус через площадь:

```
namespace Geometry
{
    class Circle
    {
        // Компоненты выше ...

        public double Lenght
```

```
            public double Square
            {
                set => Radius = Math.Sqrt(value / PI);
                get => PI * Radius * Radius;
            }
        }
    }
```

Как и в случае длины окружности, подкоренное выражение может быть отрицательным. C# в этом случае помечает его как «не число» и присваивает свойству `Radius`. Однако в ходе проверки условия оно будет считаться ложным, поэтому радиус будет обновлен нулем, что и требовалось.

Заметим, что свойств `Length` и `Square` реализованы в полной форме, однако не имеют своих полей, которые контролируют: их задача управлять доступом и корректировать значения других свойств.

В следующем примере показана более гибкая реализация программы.

Глава 3\Geometry_2\Circle.cs

```
namespace Geometry
{
    class Circle
    {
        private const double PI = Math.PI;

        private double _Radius;
        public double Radius
        {
            set => _Radius = (value > 0) ? value : 0;
            get => _Radius;
        }

        public Circle() { }

        public Circle(double radius) =>
            Radius = radius;

        public override string ToString() =>
            $"C(R = {Radius})";

        public double Lenght
```

```
        {
            set => Radius = value / (2 * PI);
            get => 2 * PI * Radius;
        }

        public double Square
        {
            set => Radius = Math.Sqrt(value / PI);
            get => PI * Radius * Radius;
        }
    }
}
```

Глава 3\TriangleGeometry\Program.cs

```
using System;

namespace Geometry
{
    class Program
    {
        static void Main(string[] args)
        {
            Circle circle = new Circle(2.4);
            Console.WriteLine($"Окружность: {circle}");
            Console.WriteLine($"Радиус окружности:
                {circle.Radius:F3} ед");
            Console.WriteLine($"Длина окружности:
                {circle.Lenght:F3} ед");
            Console.WriteLine($"Площадь круга:
                {circle.Square:F3} ед²\n");

            Console.WriteLine("Установим длину = π и
                пересчитаем радиус: ");
            circle.Lenght = Math.PI;
            Console.WriteLine($"Окружность: {circle}");
            Console.WriteLine($"Радиус окружности:
                {circle.Radius:F3} ед");
            Console.WriteLine($"Длина окружности:
                {circle.Lenght:F3} ед");
            Console.WriteLine($"Площадь круга:
                {circle.Square:F3} ед²\n");

            Console.WriteLine("Установим площадь = 2π и
                 пересчитаем радиус: ");
```

```
            circle.Square = 2 * Math.PI;
            Console.WriteLine($"Окружность: {circle}");
            Console.WriteLine($"Радиус окружности:
                {circle.Radius:F3} ед");
            Console.WriteLine($"Длина окружности:
                {circle.Lenght:F3} ед");
            Console.WriteLine($"Площадь круга:
                {circle.Square:F3} ед²");
        }
    }
}
```

```
Окружность: C(R = 2,4)
Радиус окружности: 2,400 ед
Длина окружности: 15,080 ед
Площадь круга: 18,096 ед²

Установим длину = π и пересчитаем радиус:
Окружность: C(R = 0,5)
Радиус окружности: 0,500 ед
Длина окружности: 3,142 ед
Площадь круга: 0,785 ед²

Установим площадь = 2π и пересчитаем радиус:
Окружность: C(R = 1,4142135623730951)
Радиус окружности: 1,414 ед
Длина окружности: 8,886 ед
Площадь круга: 6,283 ед²
```

Рис. 3.92.

3.9.2. Проект «Правильный треугольник»

Постановка задачи

Зададим класс Triangle, описывающий свойства правильного треугольника со стороной $\boldsymbol{a}$.

- Определить свойство `Size`, хранящее длину стороны треугольника (т.е. $\boldsymbol{a}$). При попытке задать свойству отрицательное значение инициализировать его нулем.
- Задать конструктор с параметром и без.
- Переопределить метод `ToString()` для возврата строки с данными по объекту.

- Определить свойство `Radius`, возвращающее радиус описанной вокруг треугольника окружности.
- Определить свойство `Square`, возвращающее площадь треугольника.
- При записи свойств `Radius` и `Square` осуществлять пересчет `Size`.

Комментарии к реализации

Начнем с описания свойства `Size`, которолирующего закрытое поле `_Size`.

```
namespace RegularTriangle
{
    public class Triangle
    {
        private double _Size;
        public double Size
        {
            set => _Size = (value >= 0) ? value : 0;
            get => _Size;
        }
    }
}
```

Реализуем конструкторы класса:

```
namespace RegularTriangle
{
    public class Triangle
    {
        // Компоненты выше ...

        public Triangle() { }

        public Triangle(double size) => Size = size;
    }
}
```

Строковое представление оформим следующим образом:

```
namespace RegularTriangle
{
    public class Triangle
    {
         // Компоненты выше ...
```

```
        public override string ToString() =>
            $"Δ = {Size}";
    }
}
```

Зная сторону правильного треугольника, нетрудно вычислить радиус описанной окружности и площадь треугольника. Верно и обратное – по извесному радиусу описанной окружности или площади треугольника можно вычислить сторону треугольника a.

$$R = \frac{a}{\sqrt{3}} \qquad a = \sqrt{3}R$$

$$S = \frac{\sqrt{3}}{4}a^2 \qquad a = \sqrt{\frac{4S}{\sqrt{3}}}$$

Рис. 3.93. Расчетные формулы

Это позволяет гибко реализовать свойства `Radius` и `Square`.

Заметим, что во всех формулах используется величина $\sqrt{3}$. Поэтому ее рационально записать в закрытое поле (ее нельзя сделать константой, потому что для расчета ее расчета используется функция `Sqrt()` класса `Math`).

```
namespace RegularTriangle
{
    public class Triangle
    {
        // Компоненты выше ...

        public double Radius
        {
            set => Size = SQRT3 * value;
            get => Size / SQRT3;
        }

        public double Square
        {
```

```
                set => Size = Math.Sqrt((4 * value) / SQRT3);
                get => (SQRT3 * Size * Size) / 4;
            }
        }
    }
```

Как и ранее, при попытке задать этим свойствам отрицательные значения set-метод свойства Radius исправит значение на нуль.

Программный код

Глава 3\RegularTriangle\Triangle.cs

```
namespace RegularTriangle
{
    public class Triangle
    {
        private double SQRT3 => Math.Sqrt(3);

        private double _Size;
        public double Size
        {
            set => _Size = (value >= 0) ? value : 0;
            get => _Size;
        }

        public Triangle() { }

        public override string ToString() =>
            $"Δ = {Size}";

        public Triangle(double size) => Size = size;

        public double Radius
        {
            set => Size = SQRT3 * value;
            get => Size / SQRT3;
        }

        public double Square
        {
            set => Size = Math.Sqrt(4 * value / SQRT3);
            get => (SQRT3 * Size * Size) / 4;
        }
    }
}
```

Глава 3\RegularTriangle\Program.cs

```
using System;

namespace RegularTriangle
{
    class Program
    {
        static void Main(string[] args)
        {
            Triangle triangle = new Triangle(4);
            Console.WriteLine($"Треугольник: {triangle}");
            Console.WriteLine($"Сторона:
                {triangle.Size:F3} ед");
            Console.WriteLine($"Радиус описанной окружности:
                {triangle.Radius:F3} ед");
            Console.WriteLine($"Площадь:
                {triangle.Square:F3} ед²\n");

            Console.WriteLine($"Установить радиус = 10
                и пересчитать стороны");
            triangle.Radius = 10;
            Console.WriteLine($"Треугольник: {triangle}");
            Console.WriteLine($"Сторона:
                {triangle.Size:F3} ед");
            Console.WriteLine($"Радиус описанной окружности:
                {triangle.Radius:F3} ед");
            Console.WriteLine($"Площадь:
                {triangle.Square:F3} ед²\n");

            Console.WriteLine($"Установить площадь = 1
                и пересчитать стороны");
            triangle.Square = 1;
            Console.WriteLine($"Треугольник: {triangle}");
            Console.WriteLine($"Сторона:
                {triangle.Size:F3} ед");
            Console.WriteLine($"Радиус описанной окружности:
                {triangle.Radius:F3} ед");
            Console.WriteLine($"Площадь:
                {triangle.Square:F3} ед²\n");
        }
    }
}
```

```
Треугольник: △ = 4
Сторона: 4,000 ед
Радиус описанной окружности: 2,309 ед
Площадь: 6,928 ед²

Установить радиус = 10 и пересчитать стороны
Треугольник: △ = 17,32050807568877
Сторона: 17,321 ед
Радиус описанной окружности: 10,000 ед
Площадь: 129,904 ед²

Установить площадь = 1 и пересчитать стороны
Треугольник: △ = 1,5196713713031853
Сторона: 1,520 ед
Радиус описанной окружности: 0,877 ед
Площадь: 1,000 ед²
```

Рис. 3.94. Результат работы программы

Практикум

Задание 1

1. Создайте проект *Geometry*.
2. Последовательно повторите и воспроизведите рассмотренный в параграфе пример *Geometry_2* с описанными свойствами.
3. Добавьте самостоятельно еще ряд свойств, характеризующих модель.

Задание 2

1. Создайте проект *RegularTriangle*.
2. Воспроизведите рассмотренный в параграфе пример *RegularTriangle*.
3. Добавьте в класс самостоятельно еще ряд свойств, характеризующих модель, например – радиус описанной окружности.

ЗАКЛЮЧЕНИЕ

Формирование компетенций в области программирования и разработки программного обеспечения − необходимый компонент в понимании принципов работы современных информационных и компьютерных систем. Навыки разработки помогают выпускникам вузов решать широкий спектр практических задач в профессиональной сфере.

Материал практикума направлен на системное и последовательное освоение фундаментальных возможностей платформы .NET и языка программирования C#, которые являются современной основой для формирования важнейших компетенций в области объектно-ориентированного программирования. Читателю представлен подробный и изложенный в простой форме материал по синтаксису языка, структурам данных, методам и классам, подкреплённый многочисленными примерами, иллюстрациями и практическими заданиями, что обеспечивает глубокое понимание концепций программирования. Много внимания уделено объяснению специфических деталей и оптимизации алгоритмов решения задач.

Предложенные в издании темы нацелены на формирование у студентов интереса и мотивации к активному изучению программирования как средства решения широкого спектра задач, в частности – обучении школьников основам современного программирования. Курс ориентирован на подготовку специалистов естественно-научного цикла, может быть использован в обучении информатике на базовом и профильном уровнях.

Авторы надеются, что представленные в практикуме учебные материалы оказались интересны и полезны читателю.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Албахари, Дж.* C# 7.0. Справочник. Полное описание языка / Дж. Албахари, Б. Албахари – М. : ДМК Пресс, 2018. – 1040 с.
2. *Алешкин, А. В.* Программирование на языке C#. Практикум / А. В. Алекшкин – Киров : ВятГУ, 2013. – 66 с.
3. *Арораа, Г.* Паттерны проектирования для C# и платформы .NET Core. / Г. Арораа, Д. Чилберто – М. : ДМК Пресс, 2021. – 448 с.
4. *Вагнер, Б.* Эффективное программирование на C#. 50 способов улучшения кода / Б. Вагнер – М. : ДМК Пресс, 2019. – 320 с.
5. *Васильев, А.* Программирование на C# для начинающих: учеб. пособие / А. Васильев – М., 2023. – 320 с.
6. *Гриффитс, И.* Программируем на C# 8.0 / И. Гриффитс – М. : ДМК Пресс, 2021. – 608 с.
7. *Гуриков, С. Р.* Введение в программирование на языке Visual C# : учеб. пособие / С. Р. Гуриков – М. : Форум : ИНФРА-М, 2013. – 448 с.
8. *Канцедал, С. А.* Алгоритмизация и программирование : учеб. пособие / С. А. Канцедал. – М. : ИД ФОРУМ : ИНФРА-М, 2014. – 352 с.
9. *Мюллер, Д. П.* C# для чайников / Д. П. Мюллер – М. : Диалектика, 2022. – 480 с.
10. *Пахомов, Б. И.* C# для начинающих / Б. И. Пахомов. – СПб. : БХВ-Петербург, 2014. – 432 с.
11. *Прайс, М.* C# 8 и .NET Core. Разработка и оптимизация / М. Прайс – СПб.: Питер, 2020. – 1236 с.
12. *Прайс, М.* C# 9 и .NET 5. Разработка и оптимизация. – М. : Прайс ; СПб. : Питер, 2022. – 832 с.
13. *Рихтер, Д.* CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. 4-е изд. / Д. Рихтер – СПб. : Питер, 2013. – 896 с.
14. *Скит, Дж.* C# для профессионалов. Тонкости программирования. 4-е изд. / Дж. Скит – М. : Вильямс, 2020. – 528 с.

15. *Стеллман, Э.* Head First. Изучаем C# / Э. Стеллман, Дж. Грин – М. : О'Рейли, 2021. – 720 с.

16. *Троелсен, Э.* Язык программирования C# 9 и платформа .NET 5. Т. 1. / Э. Троелсен, Ф. Джепикс – М. : Вильямс, 2021. – 770 с.

17. *Хокинг, Д.* Unity в действии. Мультиплатформенная разработка на C#. 3-е междунар. изд. / Д. Хокинг — М. : ДМК Пресс, 2023. – 464 с.

18. *Хорев, П. Б.* Объектно-ориентированное программирование с примерами на C# : учеб. пособие / П. Б. Хорев – М. : Форум : ИНФРА-М, 2016. – 200 с.

19. *Шакин, В. Н.* Базовые средства программирования на Visual Basic в среде Visual Studio Net. : практикум / В. Н. Шакин – М. : Форум : ИНФРА-М, 2015. – 288 с.

20. *Якубович, Д. А.* Основы WEB-разработки : учеб.-метод. пособие для проведения лаб. занятий / Д. А. Якубович, Е. С. Еропова, И. А. Еропов ; Владим. гос. ун-т им. А. Г. и Н. Г. Столетовых. – Владимир : Шерлок-пресс, 2017. – 102 с.

ОБ АВТОРАХ

Якубович Денис Андреевич – старший преподаватель кафедры физико-математического образования и информационных технологий Владимирского государственного университета.

Сфера научных интересов: дифференциальные уравнения в математической физике.

Преподаваемые дисциплины: «Информационные технологии в образовании»; «Информационные технологии в профессиональной деятельности»; «Практикум по решению задач на ЭВМ», «Прикладная информатика».

E-mail: yakubovich.studylib@mail.ru

Еропова Елена Станиславовна – кандидат педагогических наук, доцент кафедры физико-математического образования и информационных технологий Владимирского государственного университета.

Сфера научных интересов: проблемы информационных технологий в образовании.

Преподаваемые дисциплины: «Информационные технологии в образовании»; «Информационные технологии в профессиональной деятельности»; «Основы математической обработки информации».

E-mail: eropova13061962@mail.ru

Учебное электронное издание

ЯКУБОВИЧ Денис Андреевич
ЕРОПОВА Елена Станиславовна

ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ
НА БАЗЕ ПЛАТФОРМЫ .NET И ЯЗЫКА C#

Издается в авторской редакции

Системные требования: Intel от 1,3 ГГц; Windows XP/7/8/10; Adobe Reader; дисковод CD-ROM.

Тираж 10 экз.

Издательство Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых.
600000, Владимир, ул. Горького, 87.