

O'REILLY®

Head First

Изучаем C#

Эндрю Стиллмен
Дженнифер Грин

Четвертое
издание



ПОДАРОК ДЛЯ МОЗГА



ОТЗЫВЫ О КНИГЕ

«Огромное спасибо! Ваши книги помогли мне начать карьеру».

— Райан Уайт, разработчик игр

«Если вы являетесь начинающим разработчиком C# (добро пожаловать на борт!), я настоятельно рекомендую вам эту книгу. Эндрю и Дженнифер написали лаконичное, авторитетное, а самое главное — увлекательное введение в разработку C#. Жаль, что у меня не было этой книги, когда я только изучал C#».

— Джон Галлоуэй, старший руководитель проектов группы .NET Community Team
в компании Microsoft

«Мало того что в книге изложены все нюансы, в которых я разобрался далеко не сразу, — в ней присутствует та самая магия серии Head First, благодаря которой она так легко читается».

— Джефф Кунц, старший разработчик C#

«“Head First. Изучаем C#“ — замечательная книга с занимательными примерами, благодаря которым обучение становится интересным».

— Линдси Бьеда, ведущий разработчик

«“Head First. Изучаем C#“ отлично подойдет как для начинающих разработчиков, так и разработчиков с опытом работы на Java (таких как я.) Авторы не делают никаких допущений относительно квалификации читателя, но уровень изложения растет достаточно быстро для тех, у кого уже есть опыт программирования, — выдержать такой баланс непросто. Книга помогла мне очень быстро выйти на требуемый уровень в моем первом крупномасштабном проекте разработки C# на работе — я настоятельно рекомендую ее».

— Шалева Одусанья, менеджер

«“Head First. Изучаем C#“ — превосходный, простой и интересный способ изучения C#. Это лучшая книга для новичков C#, которую я когда-либо видел, — примеры понятны, материал излагается кратко и хорошо написан. Мини-игры помогут закрепить новые знания в вашем мозгу. Превосходная книга для тех, кто предпочитает учиться на деле».

— Джонни Халиф, совладелец SOUTHWORKS

«“Head First. Изучаем C#“ — подробное руководство по изучению C#, которое читается как дружеская беседа. Многочисленные упражнения по программированию делают ее более интересной даже при изложении самых непростых концепций».

— Ребекка Данн-Кран, соучредитель Semaphore Solutions

«Я никогда не читал компьютерные книги от корки до корки, но эта книга удерживала мой интерес от первой страницы до последней. Если вы хотите глубоко изучить C#, да еще и получить удовольствие, это ТА САМАЯ книга, которая вам нужна».

— Энди Паркер, начинающий программист C#

Другие отзывы о Head First C#

«Трудно нормально изучать язык программирования без хороших, увлекательных примеров — в этой книге их полно! Книга поможет начинающим программистам всех сортов наладить длительные и продуктивные отношения с C# и .NET Framework».

— Крис Берроуз, разработчик

«Книга Эндрю и Дженни “Head First. Изучаем C#“ стала превосходным учебником для изучения C#. Она очень доступна, но при этом материал излагается весьма подробно и в неповторимом стиле. Если вас отпугнули более традиционные книги о C#, эта вам понравится».

— Джей Хильярд, директор и специалист по архитектуре программной безопасности, автор книги «C# 6.0 Cookbook»

«Я рекомендую эту книгу каждому, кто разыскивает хорошее введение в мир программирования и C#. С самой первой страницы авторы знакомят читателя с некоторыми нетривиальными концепциями C# в простой, доходчивой манере. В конце некоторых больших проектов/лабораторных работ читатель может оглянуться на свои программы и поразиться тому, чего он достиг».

— Дэвид Стерлинг, старший разработчик

«“Head First. Изучаем C#“ — весьма приятный учебник, полный запоминающихся примеров и увлекательных упражнений. Ее живой стиль изложения наверняка привлечет читателей — от примеров с юмористическими аннотациями до «Бесед у камина», в одной из которых абстрактный класс и интерфейс устраивают словесную перепалку! Для любого новичка в программировании просто не существует лучшего способа погрузиться в тему».

— Джозеф Албахари, создатель LINQPad, соавтор книг «C# 8.0 in a Nutshell» и «C# 8.0 Pocket Reference»

«“Head First. Изучаем C#“ хорошо читалась и была понятной. Я рекомендую эту книгу каждому разработчику, желающему погрузиться в воды C#. Я порекомендую ее разработчику, который хочет найти более эффективный способ объяснить, как работает C#, своим менее просвещенным друзьям-разработчикам».

— Джузеппе Туритто, технический директор

«Эндрю и Дженни создали еще один впечатляющий учебный курс из серии “Head First”. Берите карандаш, садитесь за компьютер и получайте удовольствие, напрягая свое левое полушарие мозга, правое полушарие и чувство юмора».

— Билл Метельски, системный аналитик

«Чтение книги принесло незабываемые впечатления. Я еще не встречал книжной серии, которая бы так хорошо учила... Определенно рекомендую эту книгу всем, кто хочет изучать C#».

— Кришна Пала, MCP

Отзывы о других книгах из серии Head First

«Я получил эту книгу вчера, начал читать ее... и не мог остановиться. Безусловно, это очень круто. Книга читается легко, но авторы излагают большой объем материала, и все написано по сути. Я под впечатлением».

— **Эрик Гамма, соавтор книги «Паттерны проектирования».**

«Одна из самых увлекательных и умных книг по проектированию программного обеспечения, которые мне когда-либо попадались».

— **Аарон Лаберг, старший вице-президент по технологиям и разработке продуктов, ESPN**

«То, что когда-то было долгим учебным процессом проб и ошибок, сократилось до увлекательной книги в мягкой обложке».

— **Майк Дэвидсон, бывший вице-президент по проектированию, Twitter, основатель Newsvine**

«Элегантный дизайн лежит в основе каждой главы, каждая концепция передается с равными дозами прагматизма и остроумия».

— **Кен Голдстейн, исполнительный вице-президент и директор-распорядитель, Disney Online**

«Обычно когда я читаю книгу или статью о паттернах проектирования, мне иногда приходится щипать себя, чтобы не отвлекаться от чтения... Но только не с этой книгой. Как бы странно это ни прозвучало, с этой книгой изучение паттернов проектирования становится интересным».

И если другие книги о паттернах проектирования говорят: «Расслабься... Тебе тепло... Твои веки тяжелеют...», эта книга во все горло кричит: «Взбодрись, парень!»».

— **Эрик Вьюлер**

«Я буквально влюблен в эту книгу. Я даже поцеловал ее на глазах у жены».

— **Сатиш Кумар**

Head First C#

A Learner's Guide to Real-World Programming with C# and .NET Core

4-th edition

Wouldn't it be dreamy if there was a C# book that was more fun than memorizing a dictionary? It's probably nothing but a fantasy...



Andrew Stellman &
Jennifer Greene

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First

Изучаем C#

4-е издание

Как бы было хорошо
найти книгу по C#, которая будет
веселее зазубривания словаря...
Об этом можно только мечтать...

Эндрю Стиллмен
Дженнифер Грин



 ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

ББК 32.973.2-018.1

УДК 004.43

Стиллмен Эндрю, Грин Дженнифер

C80 Head First. Изучаем C#. 4-е изд. / Пер. с англ. Е. Матвеева. — СПб.: Питер, 2022. — 768 с.: ил. — (Серия «Head First O'Reilly»).

ISBN 978-5-4461-3943-9

Серия Head First позволяет сразу приступить к созданию собственного кода на C#, даже если у вас нет никакого опыта программирования. Не нужно тратить время на изучение скучных спецификаций и примеров! Вы освоите необходимый минимум инструментов и сразу приступите к забавным и интересным программным проектам, от разработки 3D-игры до создания серьезного приложения и работы с данными. Четвертое издание книги было полностью обновлено и переработано, чтобы рассказать о возможностях современных C#, Visual Studio и .NET, оно будет интересно всем, кто изучает язык программирования C#. Особенностью данного издания является уникальный способ подачи материала, выделяющий серию «Head First» издательства O'Reilly в ряду множества скучных книг, посвященных программированию.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

Руководитель дивизиона *Ю. Сергиенко*
Литературный редактор *Н. Викторова*
Корректор *С. Беляева*
Художественный редактор *В. Мостипан*
Верстка *Л. Егорова, Е. Неволайнен*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес:
194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр.,
д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 02.2022. Наименование: книжная продукция.
Срок годности: не ограничен.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,
ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Налоговая льгота — общероссийский классификатор продукции
ОК 034-2014, 58.11.12.000 — Книги печатные
профессиональные, технические и научные.

Подписано в печать 13.01.22. Формат 84×108/16.
Бумага офсетная. Усл. п. л. 80,640.
Тираж 1200. Заказ 0000.

ISBN 978-1491976708 англ.
ISBN 978-5-4461-3943-9

Authorized Russian translation of the English edition of Head First C#, 4th Edition
ISBN 9781491976708 © 2021 Jennifer Greene, Andrew Stellman.

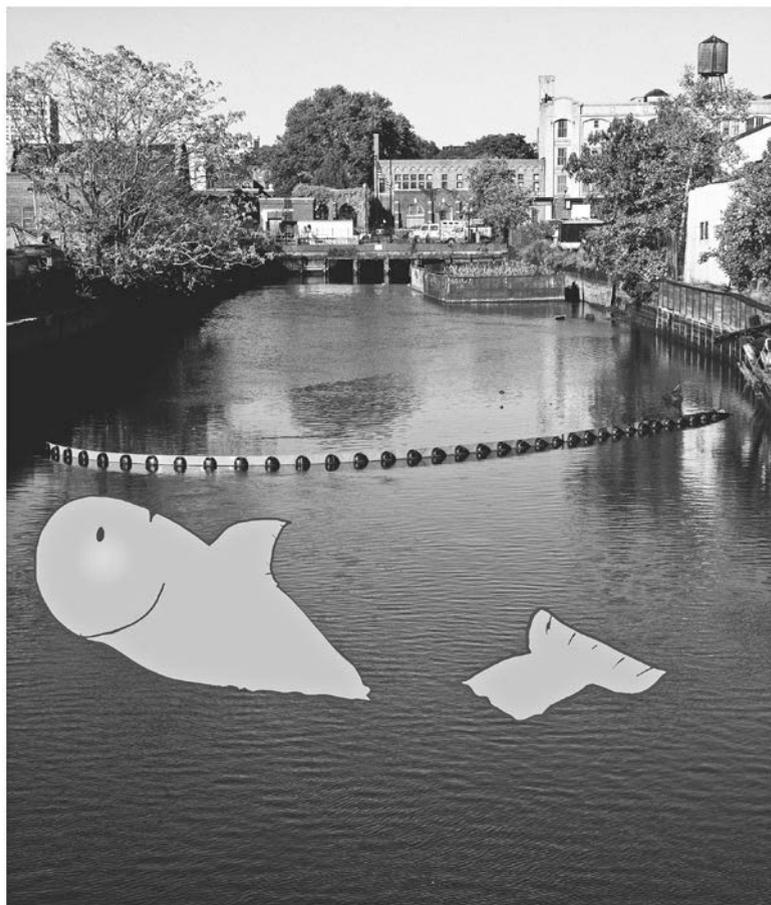
This translation is published and sold by permission of O'Reilly Media, Inc., which
owns or controls all rights to publish and sell the same.

© Перевод на русский язык ООО Издательство «Питер», 2022

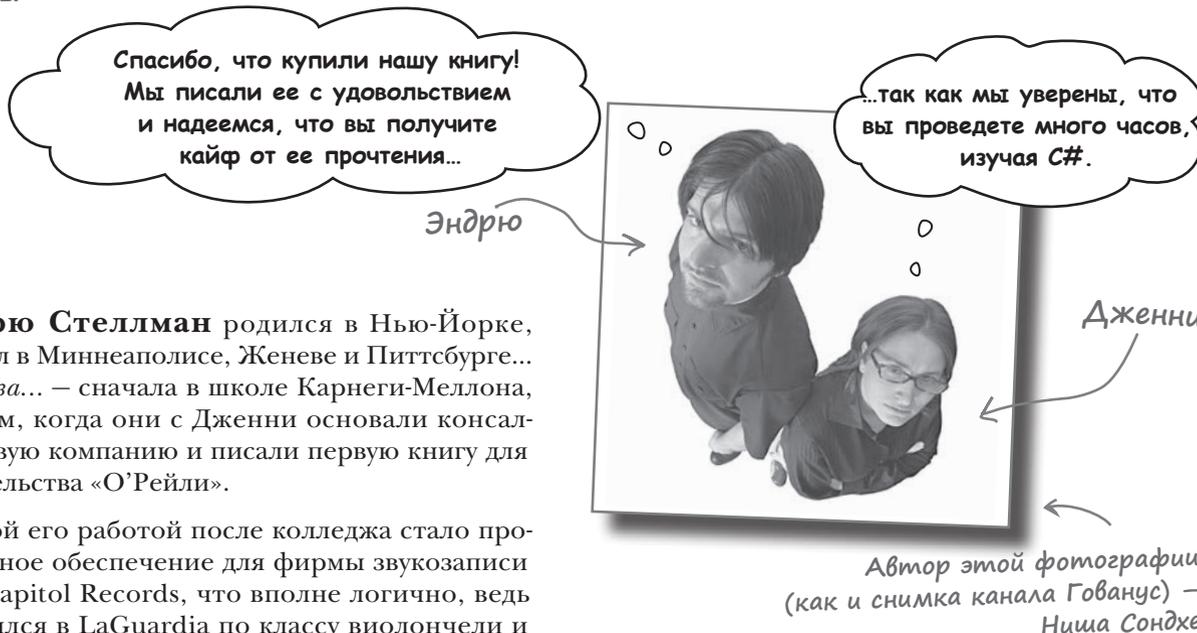
© Издание на русском языке, оформление ООО Издательство «Питер», 2022

© Серия «Head First O'Reilly», 2022

*Эта книга посвящается киту Сладжи,
который приплыл в Бруклин 17 апреля 2007 года*



*Ты пробыл в нашем канале всего день,
но навсегда останешься в наших
сердцах*



Эндрю Стеллман родился в Нью-Йорке, но жил в Миннеаполисе, Женеве и Питтсбурге... *два раза*... — сначала в школе Карнеги-Меллона, а затем, когда они с Дженни основали консалтинговую компанию и писали первую книгу для издательства «О’Рейли».

Первой его работой после колледжа стало программное обеспечение для фирмы звукозаписи EMI-Capitol Records, что вполне логично, ведь он учился в LaGuardia по классу виолончели и джазовой басс-гитары. Сначала они с Дженни работали в компании по производству финансового ПО на Уолл-стрит, где Эндрю руководил группой программистов. На протяжении многих лет он был вице-президентом крупного инвестиционно-го банка, конструировал масштабные серверные системы, управлял большими международными командами разработчиков ПО и консультировал фирмы, школы и организации, в том числе Microsoft, национальное бюро экономических исследований и Массачусетский технологический институт. За это время ему удалось поработать с замечательными программистами и многому от них научиться.

В свободное время Эндрю создает бесполезные (но забавные) программы, играет в компьютерные игры, практикует тайцзицюань и айкидо и заботится о своем карликовом шпице.

Дженни и Эндрю создают программы и пишут о разработке программного обеспечения с 1998 года. Их первая книга — *Applied Software Project Management* — вышла в издательстве «О’Рейли» в 2005 году. В этом же издательстве вышли *Beautiful Teams* (2009) и первая книга серии *Head First* *Head First PMP* (2007).

В 2003 году они основали компанию *Stellman & Greene Consulting*, чтобы разрабатывать программное обеспечение для ученых, изучающих последствия использования отравляющих веществ для ветеранов войны во Вьетнаме. Кроме программ и книг, эта компания оказывает консалтинговые услуги и выступает на конференциях и встречах разработчиков ПО, архитекторов и руководителей проектов.

С ними можно познакомиться в блоге *Building Better Software*: <http://www.stellman-greene.com> и в Twitter @AndrewStellman и @JennyGreene

Дженнифер Грин изучала в колледже философию и, как и многие ее однокурсники, не смогла найти работу по специальности. Но благодаря способностям к разработке программного обеспечения она начала работать в онлайн-службе.

В 1998 году Дженни переехала в Нью-Йорк и устроилась в фирму по разработке финансового ПО. Она управляла командами разработчиков, тестировщиков и программистов, занимавшихся разработкой ПО в медийной и финансовой областях.

Затем она много путешествовала по миру с различными командами разработчиков и реализовала целый ряд замечательных проектов.

Дженнифер обожает путешествия, индийское кино, комиксы, компьютерные игры и огромную сибирскую кошку.

Содержание (сводка)

	Введение	29
1	Начало работы с C#: <i>Быстро сделать что-то классное!</i>	41
2	Погружение в C#: <i>Команды, классы и код</i>	89
	<i>Лабораторный курс Unity № 1: Исследование C# с Unity</i>	127
3	Ориентируемся на объекты: <i>Написание осмысленного кода</i>	143
4	Типы и ссылки: <i>Данные и ссылки</i>	195
	<i>Лабораторный курс Unity № 2: Написание кода C# для Unity</i>	000
5	Инкапсуляция: <i>Умейте хранить секреты</i>	267
6	Наследование: <i>Генеалогическое древо объектов</i>	313
	<i>Лабораторный курс Unity № 3: Экземпляры GameObject</i>	383
7	Интерфейсы, приведение типов и is: <i>Классы должны держать обещания</i>	395
8	Перечисления и коллекции: <i>Организация данных</i>	445
	<i>Лабораторный курс Unity № 4: Пользовательские интерфейсы</i>	493
9	LINQ и лямбда-выражения: <i>Контроль над данными</i>	507
10	Чтение и запись файлов: <i>Прибереги последний байт для меня</i>	569
	<i>Лабораторный курс Unity № 5: Отслеживание лучей</i>	617
11	Капитан Великолепный: <i>Смерть объекта</i>	627
12	Обработка исключений: <i>Борьба с огнем надоедает</i>	663
	<i>Лабораторный курс Unity № 6: Перемещение по сцене</i>	691
I	Проекты ASP.NET Core Blazor: <i>Visual Studio для пользователей Mac</i>	703
II	Ката программирования: <i>Ката программирования для опытных и/или нетерпеливых</i>	765



Сделаем игру чуть более азартной! В нижней части окна выводится время, прошедшее с момента запуска игры. Показания таймера постоянно увеличиваются, а останавливается таймер только после нахождения последней пары.



Содержание (настоящее)

Введение

Ваш мозг и C#. Вы учитесь — готовитесь к экзамену. Или пытаетесь освоить сложную техническую тему. Ваш мозг хочет оказать вам услугу. Он старается сделать так, чтобы на эту очевидно несущественную информацию не тратились драгоценные ресурсы. Их лучше потратить на что-нибудь важное. Так как же заставить его изучить C#?

Для кого написана эта книга?	30
Кому эта книга не подойдет?	30
Мы знаем, о чем вы думаете	31
Метапознание: наука о мышлении	33
Вот что сделали МЫ	34
Что можете сделать ВЫ, чтобы заставить свой мозг повиноваться	35
Информация	36
Научные редакторы	38
Благодарности	39
И наконец...	39

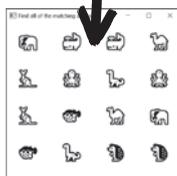




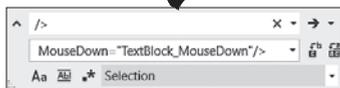
**СОЗДАНИЕ
ПРОЕКТА**



КОНСТРУИРОВАНИЕ ОКНА



**НАПИСАНИЕ
КОДА C#**



**ОБРАБОТКА
ЩЕЛЧКОВ**



**ДОБАВЛЕНИЕ
ТАЙМЕРА**

1

Начало работы с C#

Быстро сделать что-то классное!

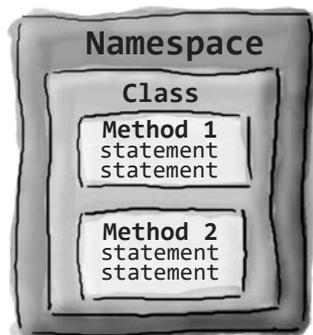
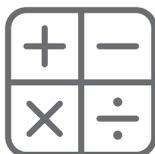
Хотите программировать быстро? C# — это мощный язык программирования. Благодаря Visual Studio вам не потребуется писать непонятный код, чтобы заставить кнопку работать. Вместо того чтобы запоминать параметры метода для имени и для ярлыка кнопки, вы сможете создать действительно классное приложение. Звучит заманчиво? Тогда переверните страницу и приступим к делу.

Зачем вам изучать C#	42
Visual Studio — инструмент для написания кода и изучения C#	43
Создание вашего первого проекта в Visual Studio	44
Давайте построим игру!	46
Как построить игру	47
Создание проекта WPF в Visual Studio	48
Построение окна с использованием XAML	52
Построение окна для игры	53
Определение размера окна и текста заголовка в свойствах XAML	54
Добавление строк и столбцов в сетку XAML	56
Выравнивание размеров строк и столбцов	57
Размещение элементов TextBlock в сетке	58
Теперь можно переходить к написанию кода игры	61
Генерирование метода для настройки игры	62
Завершение метода SetUpGame	64
Запуск программы	66
Добавление нового проекта в систему управления версиями	70
Следующий шаг построения игры — обработка щелчков	73
Реакция TextBlock на щелчки	74
Добавление кода TextBlock_MouseDown	77
Вызов обработчика события MouseDown остальными элементами TextBlock	78
Добавление таймера	79
Добавление таймера в код игры	80
Диагностика ошибок в отладчике	82
Добавьте оставшийся код и завершите построение игры	86
Обновление кода в системе управления версиями	87
Еще лучше, если...	88

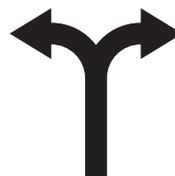
2 II. Глубокое погружение в C#

Команды, классы и код

Вы не просто пользователь IDE. Вы — разработчик. IDE может сделать за вас очень многое, и все же ее возможности небезграничны. Visual Studio — одна из самых совершенных систем разработки программного обеспечения, однако мощная IDE — только начало. Пришло время заняться углубленным изучением кода C#: какую структуру он имеет, как он работает, как управлять им... Потому что нет предела тому, что вы можете делать в ваших приложениях.



Присмотримся к файлам консольного приложения	90
Два класса могут находиться в одном пространстве имен (и файле!)	92
Команды являются структурными элементами приложений	95
Переменные используются в программах для работы с данными	96
Генерирование нового метода для работы с переменными	98
Добавление кода с использованием операторов	99
Использование отладчика для наблюдения за изменением переменных	100
Использование операторов для работы с переменными	102
Принятие решений в командах if	103
Циклы выполняют некоторые действия снова и снова	104
Используйте фрагменты кода для написания циклов	107
Элементы управления определяют механику ваших пользовательских интерфейсов	111
Создание приложения WPF для экспериментов с элементами управления	112
Добавление элемента TextBox в приложение	115
Добавление кода C# для обновления TextBlock	118
Добавление обработчика события, который разрешает вводить только числовые данные	119
Добавление ползунков в нижнюю строку сетки	123
Добавление кода C#, обеспечивающего работу элементов управления	124



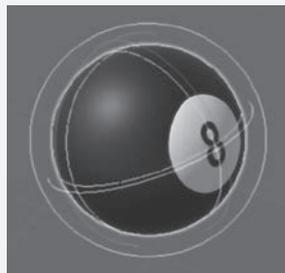
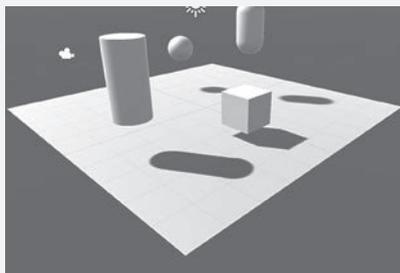
Лабораторный курс Unity № 1

Исследование C# с Unity

Добро пожаловать на первый урок **«Лабораторный курс Unity»**. Написание кода — навык, и, как и любой другой навык, он развивается за счет **практики** и **экспериментирования**. И в этом отношении Unity может стать очень полезным инструментом.

В первой лабораторной работе мы введем вас в курс дела. Вы начнете ориентироваться в редакторе Unity, а также создавать 3D-объекты и оперировать ими.

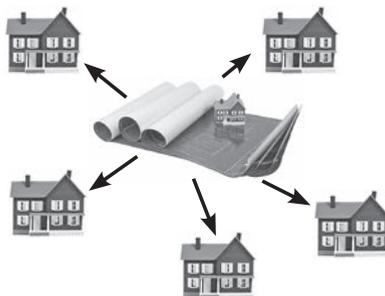
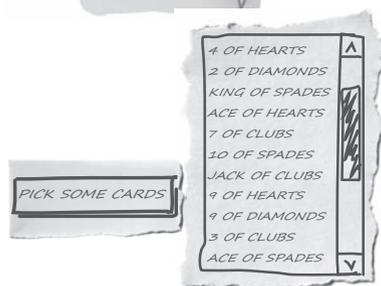
Unity — мощный инструмент для разработки игр	128
Загрузка Unity Hub	129
Использование Unity Hub для создания нового проекта	130
Управление макетом Unity	131
Сцена как 3D-среда	132
Игры Unity состоят из объектов GameObject	133
Использование инструмента Move для перемещения объектов GameObject	134
В окне Inspector выводятся компоненты GameObject	135
Добавление материала к объекту GameObject	136
Вращение сферы	139
Проявите фантазию!	142



3 Ориентируемся на объекты

Написание осмысленного кода

Каждая написанная вами программа решает некоторую задачу. Когда вы пишете программу, всегда желательно заранее подумать, какую задачу должна решать ваша программа. Вот почему объекты приносят такую пользу. Они позволяют сформировать структуру кода в соответствии с решаемой задачей, чтобы вы могли тратить время на задачу, над которой работаете, не отвлекаясь на механику написания кода. Если вы правильно используете объекты (и действительно хорошо продумали их при проектировании), получившийся код будет интуитивно понятным, будет легко читаться и изменяться.



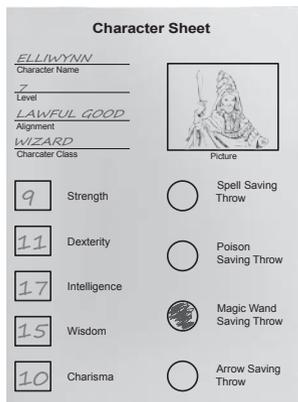
Если код полезен, он используется повторно	144
Некоторые методы получают параметры и возвращают значение	145
Программа для выбора карт	146
Создание консольного приложения PickRandomCards	147
Завершение метода PickSomeCards	148
Готовый класс CardPicker	150
Анна работает над следующей игрой	153
Игра Анны развивается...	154
Построение бумажного прототипа для классической игры	156
Следующий шаг: построение WPF-версии приложения для выбора карт	158
StackPanel — контейнер для наложения элементов	159
Повторное использование класса CardPicker в новом приложении WPF	160
Использование Grid и StackPanel для формирования макета главного окна	161
Формирование макета окна приложения Card Picker	162
Прототипы Анны выглядят замечательно...	165
Анна может воспользоваться объектами для решения своей задачи	166
Класс используется для построения объектов	167
Новый объект, созданный на базе класса, называется экземпляром этого класса	168
Хорошее решение для Анны (с объектами)	169
Экземпляры хранят данные в полях	173
Куча	176
Что на уме у вашей программы	177
Иногда код плохо читается	178
Использование содержательных имен классов и методов	180
Классы, парни и деньги	186
Простой способ инициализации объектов в C#	188
Используйте интерактивное окно C# для выполнения кода C#	194

4

Типы и ссылки

Данные и ссылки

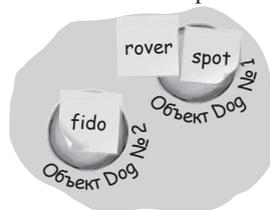
Чем были бы наши приложения без данных? Задумайтесь на минуту. Без данных наши программы... в общем, трудно представить, что кто-то станет писать код без данных. Вы запрашиваете информацию у ваших пользователей; эта информация используется для поиска данных или генерирования новой информации, которая возвращается пользователю. Собственно, практически все, что вы делаете в программировании, требует работы с данными в той или иной форме. В этой главе вы узнаете все тонкости типов данных и ссылок C#, узнаете, как работать с данными в программах, и даже узнаете кое-что новое об объектах (представьте, объекты — тоже данные!).



Создание ссылки выглядит так, словно вы пишете имя на наклейке и прикрепляете ее к объекту. Надпись становится своего рода «меткой», по которой вы можете обращаться к объекту в будущем.



Оуэну нужна наша помощь!	196
На листах персонажей хранятся разные виды данных	197
Тип переменной определяет, какие данные в ней могут храниться	198
В C# существует несколько типов для хранения целых чисел	199
Поговорим о строках	201
Литерал — значение, записанное непосредственно в вашем коде	202
Переменные как емкости для данных	205
Другие типы тоже могут иметь разные размеры	206
10 литров в 5-литровой банке	207
Приведение типов позволяет копировать значения, которые C# не может автоматически преобразовать к другому типу	208
C# выполняет некоторые преобразования автоматически	211
При вызове метода аргументы должны быть совместимы с типами параметров	212
Оуэн постоянно старается улучшить свою игру...	214
Поможем Оуэну в экспериментах с характеристиками	216
Использование компилятора C# для поиска проблемной строки кода	218
Использование ссылочных переменных для обращения к объектам	226
Ссылки напоминают наклейки на ваших объектах	227
Если ни одной ссылки не осталось, объект уничтожается сборщиком мусора	228
Множественные ссылки и их побочные эффекты	230
Две ссылки — ДВЕ переменные, по которым можно изменять данные одного объекта	237
Объекты используют ссылки для взаимодействия друг с другом	238
Массивы содержат группы значений	240
Массивы могут содержать ссылочные переменные	241
null означает, что ссылка не указывает ни на что	243
Тест-драйв со случайными числами	247
Добро пожаловать в забегаловку эконом-класса «У неторопливого Джо»!	248

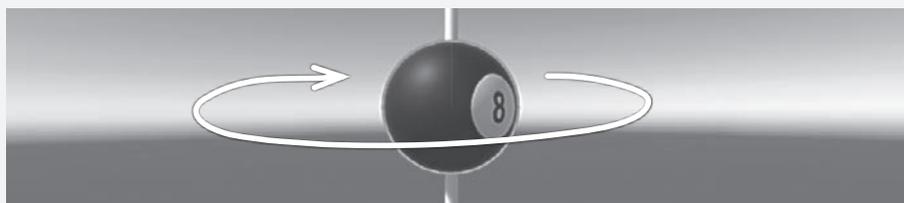


Лабораторный курс Unity № 2

Написание кода C# для Unity

Unity – не только мощный кросс-платформенный движок и редактор для построения 2D- и 3D-игр и моделирования. Также это **отличный способ потренироваться в написании кода C#**. В этой лабораторной работе мы начнем писать код для управления объектами GameObject.

Сценарии C# добавляют поведение к объектам GameObject	254
Добавление сценария C# к объекту GameObject	255
Написание кода C# для поворота сферы	256
Добавление точки прерывания и отладка игры	258
Использование отладчика для понимания Time.deltaTime	259
Добавление цилиндра для обозначения оси Y	260
Добавление полей для угла поворота и скорости	261
Debug.DrawRay и 3D-векторы	262
Запуск игры для отображения луча в представлении Scene	263
Поворот шара вокруг точки сцены	264
Эксперименты с поворотами и векторами в Unity	265
Проявите фантазию!	266



5

Инкапсуляция

Умейте хранить секреты

Вам когда-нибудь хотелось, чтобы посторонние не лезли в ваши личные дела? Вот и вашим объектам этого иногда хочется. И если вы не желаете, чтобы чужие люди читали ваш дневник или просматривали банковские выписки, хорошие объекты не позволяют другим объектам копаться в их полях. В этой главе вы узнаете о мощи инкапсуляции — приеме программирования, который делает ваш код более гибким. Такой код проще использовать и его труднее использовать некорректно. Данные вашего объекта объявляются приватными, и к ним добавляются свойства, защищающие обращения к этим данным.



SwordDamage
Roll
MagicMultiplier
FlamingDamage
Damage
CalculateDamage
SetMagic
SetFlaming



Поможем Оуэну реализовать броски на повреждения	268
Создание консольного приложения для вычисления повреждений	269
Разработка XAML для WPF-версии калькулятора повреждений	271
Код программной части для WPF-калькулятора повреждений	272
Разговор за столом (или, может, дискуссия о кубиках?)	273
Попробуем исправить ошибку	274
Использование Debug.WriteLine для вывода диагностической информации	275
Возможность некорректного использования объектов	278
Инкапсуляция подразумевает ограничение доступа к части данных класса	279
Применение инкапсуляции для управления доступом к методам и полям класса	280
Но ДЕЙСТВИТЕЛЬНО ЛИ поле RealName надежно защищено?	281
К приватным полям и методам могут обращаться только экземпляры того же класса	282
Для чего нужна инкапсуляция? Представьте объект в виде «черного ящика»...	287
Воспользуемся инкапсуляцией для улучшения класса SwordDamage	291
Инкапсуляция обеспечивает безопасность данных	292
Консольное приложение для тестирования класса PaintballGun	293
Свойства упрощают инкапсуляцию	294
Изменение метода Main для использования свойства Balls	295
Автоматически реализуемые свойства упрощают ваш код	296
Использование приватного set-метода для создания свойств, доступных только для чтения	297
А если потребуется изменить размер магазина?	298
Использование конструктора с параметрами для инициализации свойств	299
Передача аргументов при использовании ключевого слова "new"	300



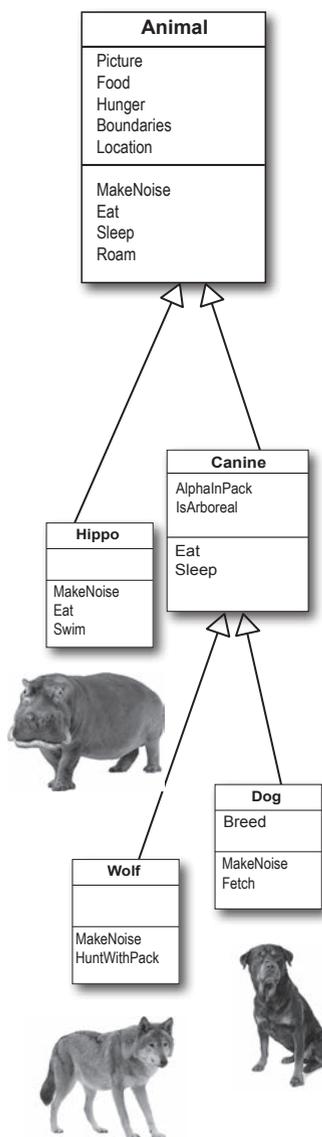
RealName: "Herb Jones"
 Alias: "Dash Martin"
 Password: "the crow flies at midnight"

6

Наследование

Генеалогическое древо объектов

Иногда люди ХОТЯТ быть похожими на своих родителей. Вы встречали объект, который действует почти так, как нужно? Думали ли вы о том, что при изменении всего нескольких элементов класс стал бы идеальным? Наследование позволяет расширять существующие классы, чтобы новый класс получал все поведение существующего — сохраняя при этом гибкость для внесения изменений, чтобы класс можно было адаптировать под любые конкретные требования. Наследование является одним из самых мощных инструментов C#: в частности, оно помогает избегать дублирования кода, более адекватно моделировать реальный мир и в конечном итоге упрощает их сопровождение и снижает риск ошибок.



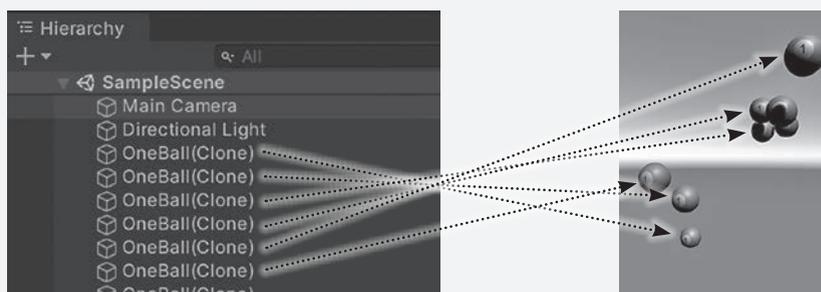
Вычисление повреждений для ДРУГИХ видов оружия	314
Команды switch для выбора из нескольких кандидатов	315
И еще... Можно ли вычислять повреждения от кинжала? От булавы? И шеста? И...	317
Если в ваших классах используется наследование, код достаточно написать только один раз	318
Постройте модель классов: начните с общего и переходите к конкретике	319
Как бы вы спроектировали симулятор зоопарка?	320
У разных животных разное поведение	322
Каждый subclass расширяет свой базовый класс	325
Расширение базового класса	330
Subclass может переопределять методы для изменения или замены унаследованных компонентов	332
Некоторые компоненты реализованы только в subclasse	337
Анализ переопределения в отладчике	338
Построение приложения для изучения virtual и override	340
Subclass может скрывать методы базового класса	342
Использование ключевых слов override и virtual для наследования поведения	344
Если базовый класс содержит конструктор, ваш subclass должен его вызвать	347
Subclass и базовый класс могут иметь разные конструкторы	348
Пора доделать приложение для Оуэна	349
Построение системы управления ульем	356
Модель классов системы управления ульем	357
Класс Queen: как матка управляет рабочими	358
Пользовательский интерфейс: добавление кода XAML главного окна	359
Обратная связь направляет работу системы управления ульем	368
Система управления ульем работает в пошаговом режиме...	
Преобразуем ее для работы в реальном времени	370
Экземпляры некоторых классов никогда не должны создаваться	372
Абстрактный класс — намеренно незавершенный класс	374
У абстрактных методов нет тела	377
Абстрактные свойства работают как абстрактные методы	378
Смертельный ромб	381

Лабораторный курс Unity № 3

Экземпляры GameObject

Unity – не только мощный кросс-платформенный движок и редактор для построения 2D- и 3D-игр и моделирования. Также это **отличный способ потренироваться в написании кода C#**. В этой лабораторной работе мы начнем писать код для управления объектами GameObject.

Построим игру в Unity!	384
Создайте новый материал в папке Materials	385
Создание бильярдного шара в случайной точке сцены	386
Применение отладчика для понимания Random.value	387
Преобразование объекта GameObject в заготовку	388
Создание сценария для управления игрой	389
Присоединение сценария к главной камере	390
Запустите свой код кнопкой Play	391
Работа с экземплярами GameObject в окне Inspector	392
Предотвращение перекрытия шаров	393
Проявите фантазию!	394



7 Интерфейсы, приведение типов и is

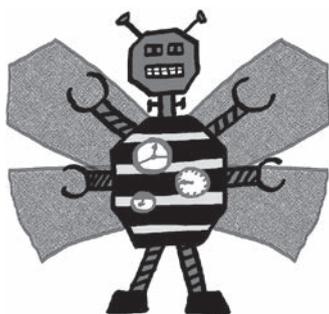
Классы должны держать обещания

Вам нужен объект для выполнения конкретной задачи? Используйте интерфейс. Иногда возникает необходимость сгруппировать объекты по выполняемым ими функциям, а не по классам, от которых они наследуют. На помощь приходят интерфейсы. Интерфейсы могут использоваться для определения конкретных задач. Любой экземпляр класса, реализующего интерфейс, гарантированно выполняет эту задачу независимо от того, с какими другими классами он связан. Чтобы эта схема работала, каждый класс, реализующий интерфейс, должен гарантировать выполнение всех своих обязательств... иначе программа компилироваться не будет.

Защищать
улей любой ценой.



Да,
повелительница!



Улей под атакой!	396
Мы можем воспользоваться приведением типов для вызова метода DefendHive...	397
Интерфейс определяет методы и свойства, которые должны быть реализованы классом...	398
Потренируемся в использовании интерфейсов	400
Создать экземпляр интерфейса невозможно, но можно получить ссылку на интерфейс	406
Ссылки на интерфейсы являются обычными ссылками на объекты	409
RoboBee 4000 может выполнять работу пчел без расхода драгоценного меда	410
Свойство Job в интерфейсе IWorker – костыль	414
Использование is для проверки типа объекта	415
Использование is для обращения к методам subclasses	416
А если мы захотим, чтобы другие животные плавали или охотились в стае?	418
Использование интерфейсов для работы с классами, выполняющими одну задачу	419
В C# также существует другой инструмент для безопасного преобразования типов: ключевое слово as	421
Пример повышающего приведения типа	423
Повышающее приведение преобразует CoffeeMaker в Appliance	424
Повышающие и понижающие приведения типов также работают и с интерфейсами	426
Интерфейсы могут наследовать от других интерфейсов	428
Интерфейсы могут содержать статические компоненты	435
Реализации по умолчанию определяют тело методов интерфейса	436
Добавление метода ScareAdults с реализацией по умолчанию	437
Связывание данных обеспечивает автоматическое обновление элементов WPF	439
Связывание данных в системе управления ульем	440
«Полиморфизм» означает, что один объект может существовать в разных формах	443



8

Перечисления и Коллекции

Организация данных

Данные не всегда бывают такими аккуратными и ухоженными, как нам хотелось бы. В реальном мире данные, как правило, не хранятся маленькими аккуратными кусочками. Нет, данные поступают вагонами, штабелями и кучами. Для их систематизации нужны мощные инструменты, и тут вам на помощь приходят перечисления и коллекции. Перечисления — типы, позволяющие определять значения для классификации ваших данных. Коллекции — специальные объекты, способные хранить и сортировать данные, которые обрабатывает программа, и управлять ими. В результате вы можете сосредоточиться на основной идее программирования, оставив задачу управления данных коллекциям.

Строки не всегда подходят для хранения категорий данных	446
Перечисления предназначены для работы с наборами допустимых значений	447
Для создания колоды карт можно воспользоваться массивом...	451
С массивами бывает неудобно работать	452
В списках можно хранить коллекции... чего угодно	453
Списки обладают большей гибкостью, чем массивы	454
Построим приложение для хранения обуви	457
В обобщенных коллекциях могут храниться любые типы	460
Инициализаторы коллекций похожи на инициализаторы объектов	466
Создание списка уток	467
Списки удобны, но с сортировкой могут возникнуть проблемы	468
IComparable<Duck> помогает списку List сортировать объекты Duck	469
Использование IComparer для определения порядка сортировки	470
Создание экземпляра компаратора	471
Компараторы могут выполнять сложные сравнения	472
Переопределение метода ToString позволяет объекту описать себя	475
Обновите циклы foreach, чтобы объекты Duck и Card выводили свои описания на консоль	476
Использование Dictionary для хранения ключей и значений	482
Краткая сводка функциональности Dictionary	483
Построение программы с использованием словаря	484
Другие разновидности коллекций...	485
Очередь работает по принципу FIFO — «первым вошел, первым вышел»	486
Стек работает по принципу LIFO — «последним вошел, первым вышел»	487
Упражнение: две колоды	492



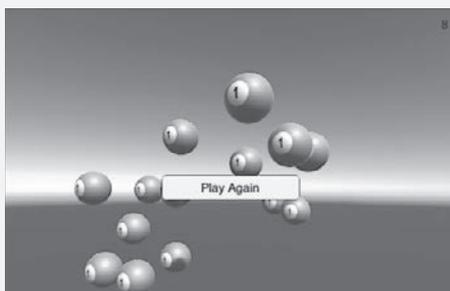
Карта «Герцог быков». В природе не встречается.

Лабораторный курс Unity № 4

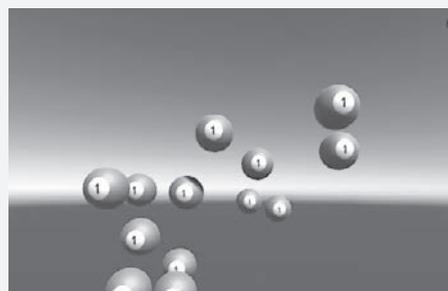
Пользовательские интерфейсы

В предыдущей лабораторной работе Unity вы начали строить игру. Мы использовали заготовку для создания экземпляров GameObject, которые появлялись в случайных точках трехмерного пространства игры и летали по кругу. В этой лабораторной работе мы продолжим с того места, на котором остановились в предыдущей главе; в ней вы сможете применить то, что узнали об интерфейсах C#, и многое другое.

Вывод текущего счета	494
Включение двух режимов в игру	495
Добавление игрового режима	496
Добавление пользовательского интерфейса к игре	498
Настройка объекта Text для вывода счета в UI	499
Кнопка для вызова метода, запускающего игру	500
Кнопка Play Again и текущий счет	501
Завершение кода игры	502
Проявите фантазию!	506



На этом снимке экрана показана игра в рабочем режиме. Шары добавляются в сцену, а игрок может щелкать на них, чтобы получать очки.



Когда на экране появится последний шар, игра переходит в режим завершения. На экране появляется кнопка Play Again, и новые шары перестают появляться.

9 LINQ и Лямбда-Выражения

Контроль над данными

Этим миром правят данные... И нам нужно знать, как в нем жить. Прошли те времена, когда можно было программировать днями и даже неделями, не имея дела с огромными объемами данных. В наши дни данные стали сутью любой программы. LINQ – технология C# и .NET, которая позволяет не только обращаться с запросами к данным в коллекциях .NET на интуитивно понятном уровне, но и группировать данные и выполнять слияние данных из разных источников. Модульные тесты помогут убедиться в том, что ваш код работает так, как предполагалось. А когда вы освоитесь с задачей разбиения данных на блоки, с которыми удобно работать, вы можете воспользоваться лямбда-выражениями, провести рефакторинг кода C# и сделать его еще более выразительным.



Джимми – фанат Капитана Великолепного...	508
Использование LINQ для управления коллекциями	510
LINQ работает с любыми реализациями IEnumerable<T>	512
Синтаксис запросов LINQ	515
LINQ работает с объектами	517
Использование запроса LINQ в приложении для Джимми	518
Ключевое слово var позволяет C# определить тип переменной за вас	520
Запросы LINQ выполняются только при обращении к результатам	527
Использование запросов group для разделения последовательности на группы	528
Использование запросов join для слияния данных из двух последовательностей	531
Использование ключевого слова new для создания анонимных типов	532
Модульные тесты помогают понять, как работает код	540
Добавление проекта модульного теста в приложение Джимми	542
Первый модульный тест	543
Написание модульного теста для метода GetReviews	545
Написание модульных тестов для обработки граничных случаев и аномальных данных	546
Оператор => и создание лямбда-выражений	548
Тест-драйв лямбда-выражений	549
Рефакторинг клоунов с использованием лямбда-выражений	550
Использование оператора ?: для принятия решений в лямбда-выражениях	553
Лямбда-выражения и LINQ	554
Запросы LINQ могут записываться в виде сцепленных вызовов методов LINQ	555
Использование оператора => для создания выражений switch	557
Исследование класса Enumerable	561
Ручное создание последовательности с поддержкой перебора	562
Упражнение: Go Fish	567

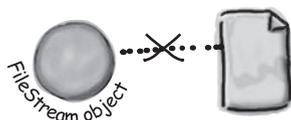


10

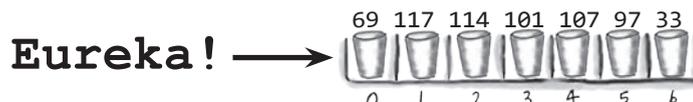
Чтение и запись файлов

Прибереги последний байт для меня

Иногда настойчивость окупается. Пока что все ваши программы жили недолго. Они запускались, некоторое время работали и закрывались. Но этого недостаточно, когда имеешь дело с важной информацией. Вы должны уметь сохранять свою работу. В этой главе мы поговорим о том, как записать данные в файл, а затем о том, как прочитать эту информацию. Вы познакомитесь с потоками данных, узнаете о сохранении объектов в файлах с использованием сериализации, а также освоите работу с шестнадцатеричными и двоичными данными и кодировку Юникод.



Для чтения и записи данных в .NET используются потоки данных	570
Различные потоки для разных данных	571
Объект FileStream читает и записывает байты в файл	572
Запись текста в файл за три простых шага	573
Дьявольский план Пройдохи	574
Использование StreamReader для чтения файла	577
Данные могут проходить через несколько потоков	578
Работа с файлами и каталогами с использованием статических классов File и Directory	582
Интерфейс IDisposable обеспечивает корректное закрытие объектов	585
Предотвращение ошибок файловой системы командами using	586
Потоки MemoryStream и хранение данных в памяти	587
При сериализации объекта также сериализуются все объекты, на которые он ссылается...	595
Использование JsonSerialization для сериализации объектов	596
JSON включает только данные, но не конкретные типы C#	599
Следующий шаг: углубленный анализ данных	601
Строки C# кодируются в Юникоде	603
Поддержка Юникода в Visual Studio	605
.NET использует Юникод для хранения символов и текста	606
C# может использовать массивы байтов для перемещения данных	608
Использование BinaryWriter для записи двоичных данных	609
Использование BinaryReader для чтения данных	610
Дамп позволяет просматривать байты в файлах	612
Использование StreamReader для вывода шестнадцатеричного дампа	613
Использование Stream.Read для чтения байтов из потока	614
Аргументы командной строки	615
Упражнение: Hide and Seek	616



Лабораторный курс Unity № 5

Отслеживание лучей

Создавая сцену в Unity, вы создаете виртуальный 3D-мир, в котором перемещаются персонажи вашей игры. Но в большинстве игр объекты окружающей обстановки не контролируются игроком напрямую. Как же эти объекты определяют свое место в сцене? В этой лабораторной работе мы построим сцену из объектов GameObject и используем навигацию для перемещения персонажей по сцене.

Создание нового проекта Unity и начало создания сцены	618
Настройка камеры	619
Создание объекта GameObject для игрока	620
Знакомство с системой навигации Unity	621
Создание сетки NavMesh	622
Автоматическая навигация в игровой области	623



Капитан Великолепный

Смерть объекта

Head First C#	
Четыре доллара	Глава 11



Жизнь и смерть объекта	630
Для принудительной сборки мусора используйте класс GC (осторожно!)	631
Когда именно выполняется финализатор?	633
Финализаторы не могут зависеть от других объектов	635
Структура похожа на объект...	639
...но не является объектом	639
Значения копируются, ссылки присваиваются	640
Структуры относятся к типам значений; объекты относятся к ссылочным типам	641
Стек и куча: подробнее о памяти	643
Параметры out и возвращение нескольких значений методом	646
Передача по ссылке с модификатором ref	647
Необязательные параметры и значения по умолчанию	648
Ссылка null не указывает ни на какой объект	649
Ссылочные типы, не допускающие null, помогут избежать NRE	650
Оператор объединения с null ??	651
Безопасная работа с типами значений, допускающими null	652
Капитан... уже не такой Великолепный	653
Методы расширения добавляют новое поведение в существующие классы	657
Расширение фундаментального типа: string	659

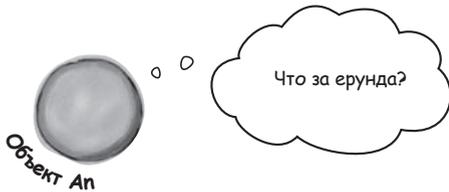


- У МЕНЯ... ОСТАЛОСЬ...
-ОХ! -
ОДНО... ПОСЛЕДНЕЕ... ДЕЛО...

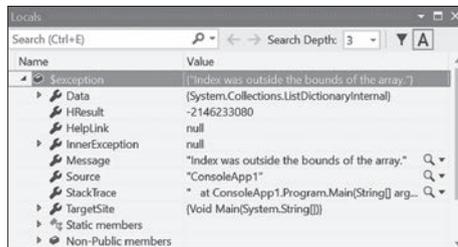
12 Обработка исключений

Борьба с огнем надоедает

Программисты не должны уподобляться пожарным. Вы усердно работали, штудировали справочники и руководства и, наконец, достигли вершины. Но вам до сих пор продолжают звонить с работы по ночам, потому что программа упала или работает не так, как должна работать. Ничто так не выбивает из колеи, как необходимость устранять странные ошибки... но благодаря обработке исключений вы сможете написать код, который сам будет разбираться с возможными проблемами. А еще лучше, что вы можете планировать такие проблемы и восстанавливать работоспособность программы при их возникновении.



```
int[] anArray = {3, 4, 1, 11};
int aValue = anArray[15];
```



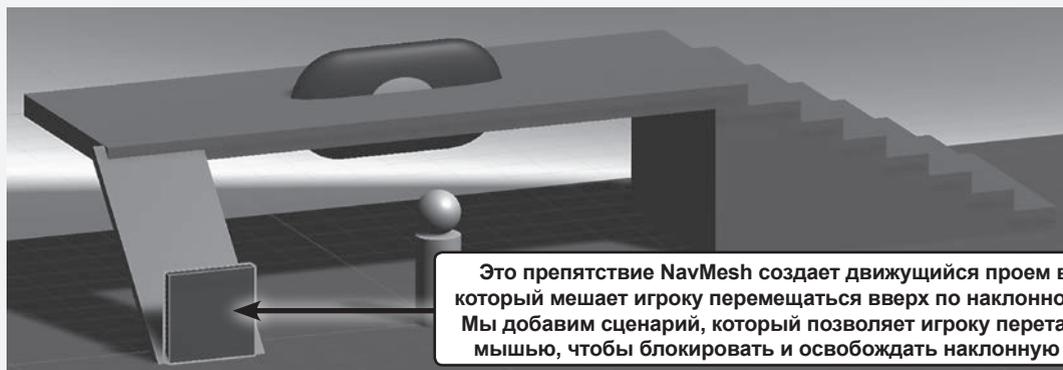
- Программа вывода шестнадцатеричного дампа читает имя файла из командной строки 664
- Когда ваша программа выдает исключение, CLR генерирует объект Exception 668
- Все объекты Exception наследуют от System.Exception 669
- Для некоторых файлов вывод дампа невозможен 672
- Что происходит при вызове небезопасного метода? 673
- Обработка исключений с try и catch 674
- Отслеживание передачи управления в try/catch 675
- Код блока finally выполняется всегда 676
- Перехват всех исключений 677
- Использование исключения, подходящего для конкретной ситуации 682
- Фильтры исключений повышают точность обработки исключений 686
- Наихудший блок catch: универсальный перехват с комментариями 688
- Временные решения допустимы (но только временно) 689

Лабораторный курс Unity № 6

Перемещение по сцене

В последней лабораторной работе Unity была создана сцена с полом (плоскость) и игроком (сфера с цилиндром). При этом использовался объект NavMesh, NavMesh Agent и отслеживание лучей, чтобы игрок следовал за щелчками в сцене. В этой работе мы воспользуемся навигационной системой Unity, чтобы объекты GameObject сами перемещались по сцене.

Продолжим с того места, на котором прервалась последняя лабораторная работа Unity	692
Добавление платформы в сцену	693
Изменение настроек предварительного построения	694
Включение лестницы и наклонной плоскости в NavMesh	695
Решение проблем с высотой в NavMesh	697
Добавление препятствия в сетку NavMesh	698
Добавление сценария для перемещения препятствия вверх и вниз	699
Проявите фантазию!	700

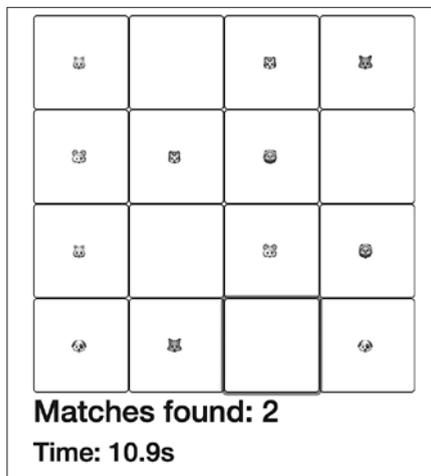


Это препятствие NavMesh создает движущийся проем в NavMesh, который мешает игроку перемещаться вверх по наклонной плоскости. Мы добавим сценарий, который позволяет игроку перетаскивать его мышью, чтобы блокировать и освобождать наклонную плоскость.

I

Приложение I. Проекты ASP.NET Core Blazor

Visual Studio для пользователей Mac



Зачем вам изучать C#	704
Создание вашего первого проекта в Visual Studio for Mac	706
Давайте построим игру!	710
Как построить игру	711
Создание проекта Blazor WebAssembly в Visual Studio	712
Запуск веб-приложения Blazor в браузере	714
Visual Studio помогает в написании кода C#	718
Завершение создания списка эмодзи и вывод их в приложении	720
Перестановка животных в случайном порядке	722
Добавление нового проекта в систему управления версиями	728
Добавление кода C# для обработки щелчков	729
Назначение обработчиков щелчков кнопкам	730
Тестирование обработчика события	732
Диагностика проблемы в отладчике	733
Отладка обработчика события	734
Поиск ошибки, породившей проблему...	736
Добавление кода для сброса игры при победе	738
Добавление таймера	741
Добавление таймера в код игры	742
Очистка меню навигации	744
Создание нового проекта Blazor WebAssembly App	747
Создание страницы с ползунком	748
Добавление текстового поля	750
Добавление селекторов для выбора цвета и даты	753
Построение Blazor-версии приложения для выбора карт	754
Страница состоит из строк и столбцов	756
Ползунок использует связывание данных для обновления переменной	757
Добро пожаловать в забегаловку эконом-класса «У нетерпеливого Джо»!	760

II

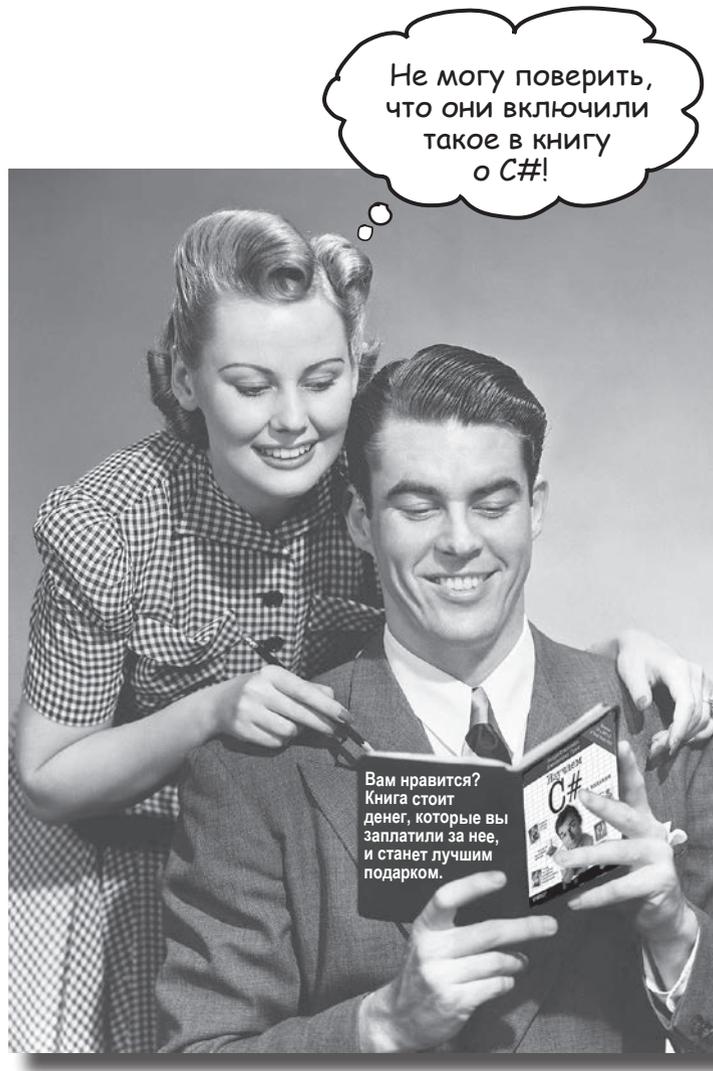
Приложение II. Ката программирования

Ката программирования для опытных и/или нетерпеливых



Как работать с этой книгой

Введение



В этом разделе мы ответим на насущный вопрос:
«Так почему они включили ТАКОЕ в книгу о C#?»»

Для кого написана эта книга?

Если на вопросы...

- 1 Вы хотите **изучать C#** (а попутно обзавестись начальными знаниями о разработке игр и Unity)?
- 2 Вы предпочитаете учиться практикуясь, а не просто читая текст?
- 3 Вы предпочитаете **оживленную беседу сухим, скучным академическим лекциям?**

...вы отвечаете положительно, то эта книга для вас.

Кому эта книга не подойдет?

Если вы ответите «да» на любой из следующих вопросов...

- 1 Вас больше интересует теория, чем практика?
- 2 Вы скучаете или раздражаетесь при мысли о том, что вам придется работать над проектами и писать код?
- 3 Вы боитесь **попробовать что-нибудь новое**? Считаете, что книга по такой серьезной теме, как программирование, должна быть неизменно серьезной?

КАТА ПРОГРАММИРОВАНИЯ

Вы **квалифицированный разработчик** с опытом работы на других языках и теперь хотите быстро освоить C# и Unity?

Вы предпочитаете **практику** и считаете, что лучше всего с ходу взяться за код?

Если вы ответили «ДА!» на оба вопроса, мы включили **ката программирования** специально для вас. Дополнительную информацию можно найти в разделе «Ката программирования» в конце книги.

Обязательно ли знать другой язык программирования, чтобы воспользоваться этой книгой?



Многие люди изучают C# как второй (третий, четвертый и т. д.) язык программирования, но вы не обязаны быть опытным программистом, чтобы получить пользу от чтения книги.

Если вы писали программы (даже маленькие!) на *любом* языке программирования, посещали вводные курсы в школе или интернете, писали сценарии для командной оболочки или пользовались языком баз данных, тогда вы **определенно** обладаете необходимой подготовкой для изучения этой книги и будете чувствовать себя как рыба в воде.

А если у вас **меньше практического опыта**, но вы все равно хотите изучать C#? Тысячи новичков — особенно тех, кто ранее строил веб-страницы или работал с функциями Excel, — воспользовались этой книгой для изучения C#. Но если вы *вообще* ничего не знаете о программировании, мы рекомендуем начать с книги Эрика Фримена (Eric Freeman) «Head First Learn To Code».

Мы знаем, о чем вы думаете

«Разве серьезные книги по программированию на C# *такие?*»

«И почему здесь столько рисунков?»

«Можно ли так чему-нибудь *научиться?*»

И мы знаем, о чем думает ваш мозг

Мозг жаждет новых впечатлений. Он постоянно ищет, анализирует, *ожидает* чего-то необычного. Он так устроен, и это помогает нам выжить.

Сегодня у вас меньше шансов стать закуской для тигра. Но ваш мозг все еще начеку, а вы просто не замечаете этого.

Как же наш мозг поступает со всеми обычными, повседневными вещами? Он всеми силами пытается оградиться от них, чтобы они не мешали его *настоящей* работе — запоминанию того, что действительно *важно*. Мозг не считает нужным сохранять скучную информацию. Она не проходит через фильтр, отсекающий «очевидно несущественное».

Но как же мозг *узнает*, что важно? Представьте, что вы отправились на прогулку и вдруг прямо перед вами появляется тигр. Что происходит в вашей голове и в теле?

Активизируются нейроны. Вспыхивают эмоции. *Происходят химические реакции*.

И тогда ваш мозг понимает...

Конечно, это важно! Не забывать!

А теперь представьте, что вы находитесь дома или в библиотеке, в теплом уютном месте, где тигры не водятся. Вы учитесь — готовитесь к экзамену. Или пытаетесь освоить сложную техническую тему, на которую вам выделили неделю... максимум десять дней.

И тут возникает проблема: ваш мозг пытается оказать вам услугу. Он старается сделать так, чтобы на эту *очевидно* несущественную информацию не тратились драгоценные ресурсы. Их лучше потратить на что-нибудь важное. На тигров, например. Или на то, что к огню лучше не прикасаться. Или на то, что ни в коем случае нельзя выкладывать фото с этой вечеринки на своей страничке в Facebook.

Нет простого способа сказать своему мозгу: «Послушай, мозг, я тебе, конечно, благодарен, но какой бы скучной ни была эта книга и пусть мой датчик эмоций сейчас на нуле, я *хочу* запомнить то, что здесь написано».

Ваш мозг считает, что ЭТО важно.



Замечательно. Еще 737 сухих скучных страниц.

Ваш мозг понимает, что ЭТО можно не запоминать.



Эта книга для тех, кто хочет учиться

Как мы что-то узнаем? Сначала нужно это «что-то» *понять*, а потом *не забыть*. Затолкать в голову побольше фактов недостаточно. Согласно новейшим исследованиям в области когнитивистики, нейробиологии и психологии обучения, для *усвоения материала* требуется нечто большее, чем просто прочитать текст. Мы знаем, как заставить ваш мозг работать.



Основные принципы серии «Head First»:

Наглядность. Графика запоминается гораздо лучше, чем обычный текст, и значительно повышает эффективность восприятия информации (до 89%, по данным исследований). Кроме того, материал становится более понятным.

Текст размещается на рисунках, к которым он относится, а не под ними или на соседней странице.

Разговорный стиль изложения. Недавние исследования показали, что при разговорном стиле изложения материала (вместо формальных лекций) улучшение результатов на итоговом тестировании составляло до 40%. Рассказывайте историю вместо того, чтобы читать лекцию. Не относитесь к себе слишком серьезно. Что скорее привлечет ваше внимание: занимательная беседа за столом или лекция?

Все элементы массива содержат ссылки. Сам массив является объектом.

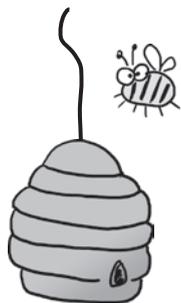


Я ОБЕДАЮ ТОЛЬКО «У НЕТОРОПЛИВОГО ДЖО»!



Активное участие читателя. Пока вы не начнете напрягать извилины, в вашей голове ничего не произойдет. Читатель должен быть заинтересован в результате; он должен решать задачи, формулировать выводы и овладевать новыми знаниями. А для этого необходимы упражнения и каверзные вопросы, в решении которых задействованы оба полушария мозга и разные чувства.

Привлечение (и сохранение) внимания читателя. Ситуация, знакомая каждому: «Я очень хочу изучить это, но засыпаю на первой же странице». Мозг обращает внимание на интересное, странное, притягательное, неожиданное. Изучение сложной технической темы не обязано быть скучным. Интересное запоминается намного быстрее.



Обращение к эмоциям. Известно, что наша способность запоминать в значительной мере зависит от эмоционального сопереживания. Мы запоминаем то, что нам безразлично. Мы запоминаем, когда что-то *чувствуем*. Нет, сентименты здесь ни при чем: речь идет о таких эмоциях, как удивление, любопытство, интерес и чувство «Да я крут!» при решении задачи, которую окружающие считают сложной, — а может быть, когда вы понимаете, что узнали столько *интересного и нового*, и теперь можете с пользой применять новые знания.



Даже испуг помогает информации закрепиться в вашем мозгу.



Метапознание: наука о мышлении

Если вы действительно хотите быстрее и глубже усваивать новые знания, задумайтесь над тем, как вы задумываетесь. Учитесь учиться.

Мало кто из нас изучает теорию метапознания во время учебы. Нам *положено* учиться, но нас редко этому *учат*.

Но раз вы читаете эту книгу, то, вероятно, хотите узнать, как программировать на C#, и по возможности быстрее. Вы хотите *запомнить* прочитанное и *применять* новую информацию на практике. Чтобы извлечь максимум пользы из учебного процесса, нужно заставить ваш мозг воспринимать новый материал как Нечто Важное. Критичное для вашего существования. Такое же важное, как тигр. Иначе вам предстоит бесконечная борьба с вашим мозгом, который всеми силами уклоняется от запоминания новой информации.

Как же УБЕДИТЬ мозг, что программирование на C# так же важно, как и тигр?

Есть способ медленный и скучный, а есть быстрый и эффективный. Первый основан на тупом повторении. Всем известно, что даже самую скучную информацию *можно* запомнить, если повторять ее снова и снова. При достаточном количестве повторений ваш мозг прикидывает: «Вроде бы несущественно, но раз одно и то же повторяется *столько раз...* Ладно, уговорил».

Быстрый способ основан на **повышении активности мозга**, и особенно на сочетании разных ее *видов*. Доказано, что все факторы, перечисленные на предыдущей странице, помогают вашему мозгу работать на вас. Например, исследования показали, что размещение слов *внутри* рисунков (а не в подписях, в основном тексте и т. д.) заставляет мозг анализировать связи между текстом и графикой, а это приводит к активизации большего количества нейронов. Больше нейронов — выше вероятность того, что информация будет сочтена важной и достойной запоминания.

Разговорный стиль тоже важен: обычно люди проявляют больше внимания, когда они участвуют в разговоре, так как им приходится следить за ходом беседы и высказывать свое мнение. Причем мозг совершенно *не интересуется*, что вы «разговариваете» с книгой! С другой стороны, если текст сух и формален, то мозг чувствует то же, что чувствуете вы на скучной лекции в роли пассивного участника. Его клонит в сон.

Но рисунки и разговорный стиль — это только начало.



Вот что сделали Мы

Мы использовали **рисунки**, потому что мозг лучше приспособлен для восприятия графики, чем текста. С точки зрения мозга рисунок стоит тысячи слов. А когда текст комбинируется с графикой, мы внедряем текст прямо в рисунки, потому что мозг при этом работает эффективнее.

Мы используем **избыточность**: повторяем одно и то же несколько раз, применяя *разные* средства передачи информации, обращаемся к *разным чувствам* — и все для повышения вероятности того, что материал будет закодирован в нескольких областях вашего мозга.

Мы используем концепции и рисунки несколько **неожиданным** образом, потому что мозг лучше воспринимает новую информацию. Кроме того, рисунки и идеи обычно имеют **эмоциональное содержание**, потому что мозг обращает внимание на биохимию эмоций. То, что заставляет нас *чувствовать*, лучше запоминается, будь то *шутка*, *удивление* или *интерес*.

Мы используем **разговорный стиль**, потому что мозг лучше воспринимает информацию, когда вы участвуете в разговоре, а не пассивно слушаете лекцию. Это происходит и при *чтении*.

В книгу включены многочисленные упражнения, потому что мозг лучше запоминает, когда вы работаете самостоятельно. Мы постарались сделать их непростыми, но интересными — то, что предпочитает большинство читателей.

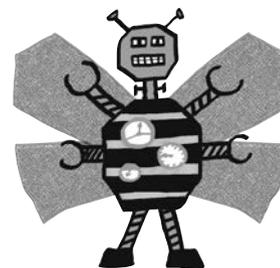
Мы совместили **несколько стилей обучения**, потому что одни читатели любят пошаговые описания, другие стремятся сначала представить «общую картину», а третьим хватает фрагмента кода. Независимо от ваших личных предпочтений полезно видеть несколько вариантов представления одного и того же материала.

Мы постарались задействовать **оба полушария вашего мозга**: это повышает вероятность усвоения материала. Пока одна сторона мозга работает, другая имеет возможность отдохнуть; это усиливает эффективность обучения в течение продолжительного времени.

А еще в книгу включены **истории** и упражнения, **отражающие другие точки зрения**. Мозг качественнее усваивает информацию, когда ему приходится оценивать и выносить суждения.

В книге часто встречаются **вопросы**, на которые не всегда можно дать простой ответ, потому что мозг быстрее учится и запоминает, когда ему приходится что-то делать. Невозможно накачать *мышцы*, наблюдая за тем, как занимаются *другие*. Однако мы позаботились о том, чтобы усилия читателей были приложены в *верном* направлении. Вам не придется ломать голову над невразумительными примерами или разбираться в сложном, перенасыщенном техническим жаргоном или слишком лаконичном тексте.

В историях, примерах, на картинках использованы **антропоморфные образы**. Ведь вы человек. И ваш мозг уделяет больше внимания *людям*, а не *вещам*.



КЛЮЧЕВЫЕ МОМЕНТЫ

Беседа у камина





Вырежьте и прикрепите на холодильник.

Что можете сделать Вы, чтобы заставить свой мозг повиноваться

Мы свое дело сделали. Остальное за вами. Эти советы станут отправной точкой; прислушайтесь к своему мозгу и определите, что вам подходит, а что не подходит. Пробуйте новое.

- 1 Не торопитесь. Чем больше вы поймете, тем меньше придется запоминать.**

Просто читать недостаточно. Когда книга задает вам вопрос, не переходите к ответу. Представьте, что кто-то действительно задает вам вопрос. Чем глубже ваш мозг будет мыслить, тем скорее вы поймете и запомните материал.
- 2 Выполняйте упражнения, делайте заметки.**

Мы включили упражнения в книгу, но выполнять их за вас не собираемся. И не *разглядывайте* упражнения. **Берите карандаш и пишите.** Физические действия *во время* обучения повышают его эффективность.
- 3 Читайте врезки.**

Это значит: читайте всё. **Врезки — часть основного материала!** Не пропускайте их.
- 4 Не читайте другие книги после этой перед сном.**

Часть обучения (особенно перенос информации в долгосрочную память) происходит *после* того, как вы откладываете книгу. Ваш мозг не сразу усваивает информацию. Если во время обработки поступит новая информация, часть того, что вы узнали ранее, может быть потеряна.
- 5 Пейте воду. И побольше.**

Мозгу нужна влага, так он лучше работает. Дегидратация (которая может наступить еще до того, как вы почувствуете жажду) снижает когнитивные функции.
- 6 Говорите вслух.**

Речь активизирует другие участки мозга. Если вы пытаетесь что-то понять или запомнить, произнесите вслух. А еще лучше — попробуйте объяснить кому-нибудь другому. Вы быстрее усвоите материал и, возможно, откроете что-то новое.
- 7 Прислушивайтесь к своему мозгу.**

Следите за тем, чтобы ваш мозг не уставал. Если вы стали поверхностно воспринимать материал или забываете только что прочитанное — пора сделать перерыв.
- 8 Чувствуйте!**

Ваш мозг должен знать, что материал книги действительно *важен*. Переживайте за героев наших историй. Придумывайте собственные подписи к фотографиям. Поморщиться над неудачной шуткой все равно лучше, чем не почувствовать ничего.
- 9 Пишите побольше кода!**

По-настоящему изучить язык C#, чтобы он закрепился в вашем сознании, можно только одним способом: **пишите побольше кода.** Именно этим вам предстоит заняться, читая книгу. Подобные навыки лучше всего закрепляются практикой. В каждой главе вы найдете упражнения. Не пропускайте их. Не бойтесь **подсмотреть** в решение задачи, если не знаете, что делать дальше! (Иногда можно застрять на элементарном.) Но все равно пытайтесь решать задачи самостоятельно. Пока ваш код не начнет работать, не стоит переходить к следующим страницам книги.

Информация

Это учебник, а не справочник. Мы намеренно убрали из книги все, что могло бы помешать изучению материала, над которым вы работаете. И при первом чтении книги начинать следует с самого начала, потому что книга предполагает наличие у читателя определенных знаний и опыта.

Упражнения ОБЯЗАТЕЛЬНЫ к выполнению.

Упражнения и задачи являются частью основного содержания книги, а не дополнительным материалом. Некоторые помогают запомнить новую информацию, некоторые — лучше понять ее, а какие-то — научиться применять ее на практике. Не пропускайте задачи. Необязательными являются только «Ребусы в бассейне», но следует помнить, что они безусловно помогают ускорить процесс обучения.

Повторение применяется намеренно.

У книг этой серии есть одна принципиальная особенность: мы хотим, чтобы вы действительно хорошо усвоили материал. И чтобы вы запомнили все, что узнали. Большинство справочников не ставят своей целью успешное запоминание, но это не справочник, а учебник, поэтому некоторые концепции излагаются в книге по несколько раз.

Выполняйте все упражнения!

Предполагается, что читатели этой книги хотят научиться программировать на C#. Что им не терпится приступить к написанию кода. И мы дали им массу возможностей сделать это. Во фрагментах, помеченных значком Упражнение!, демонстрируется пошаговое решение конкретных задач. А вот картинка с кроссовками сигнализирует о необходимости самостоятельного поиска решения. Не бойтесь подглядывать на страницу с ответом! Просто помните, что информация лучше всего усваивается, когда вы пытаетесь решать задачки без посторонней помощи.

Мы также включили весь исходный код, необходимый для выполнения упражнений. Он доступен на нашей странице GitHub: <https://github.com/head-first-csharp/fourth-edition>.

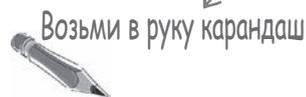
Упражнения «Мозговой штурм» не имеют ответов.

В некоторых из них правильного ответа вообще нет, в других вы должны сами оценить, насколько правильны ваши ответы (это является частью процесса обучения). В некоторых упражнениях «Мозговой штурм» приводятся подсказки, которые помогут вам найти нужное решение.

Для наглядного представления сложных понятий в книге используются диаграммы.



Выполняйте все упражнения этого раздела.



Не пропускайте эти упражнения, если вы действительно хотите изучить C#.



А этим значком помечены дополнительные упражнения для любителей логических задач. Если вам не нравится запутанная логика, скорее всего, эти задания вам тоже не понравятся.



При написании книги использовались C# 8.0, Visual Studio 2019 и Visual Studio 2019 для Mac.

Эта книга была написана для того, чтобы помочь вам в изучении C#. Группа Microsoft, которая занимается разработкой и сопровождением C#, периодически выпускает обновления языка. На момент публикации этой книги (речь идет об англоязычном издании. — *Примеч. ред.*) текущей версией языка была версия **C# 8.0**. Мы также интенсивно используем Visual Studio, интегрированную среду разработки (IDE) компании Microsoft, для изучения, преподавания и исследования возможностей C#. Снимки экранов, приведенные в книге, были получены в **последних версиях Visual Studio 2019 и Visual Studio 2019 для Mac** на момент публикации. Инструкции по установке Visual Studio приводятся в главе 1, а инструкции по установке Visual Studio for Mac — в приложении «Visual Studio для пользователей Mac».

Несмотря на выход версии C#9.0, в которой появились новые замечательные возможности, средства C#, описанные в данной книге, останутся без изменений, так что вы сможете использовать эту книгу с будущими версиями C#. Команды Microsoft, занимающиеся поддержкой Visual Studio и Visual Studio для Mac, периодически публикуют обновления, и эти изменения очень редко приводят к изменению внешнего вида снимков экрана.

В разделах «Лабораторный курс Unity» используется **Unity 2020.1** — новейшая версия Unity, доступная на момент сдачи книги в печать. Инструкции по установке Unity приведены в первом разделе «Лабораторный курс Unity».



Весь код в книге приводится на условиях лицензии Open Source, разрешающей его использование в ваших проектах. Его можно загрузить на странице GitHub (<https://github.com/head-first-csharp/fourth-edition>). Также вы можете загрузить документы в формате PDF, в которых рассматриваются возможности C#, не включенные в книгу (в том числе некоторые новейшие средства C#).

Разработка игр... и не только

Как в книге используются игры

В книге мы будем писать код для множества проектов, в том числе для игр. Мы сделали это не только потому, что мы любим игры. Игры — эффективный инструмент для **изучения и преподавания C#** по следующим причинам:

- Игры нам **знакомы**. Вам предстоит усвоить много новых концепций и идей. Их представление на знакомой основе сделает процесс обучения более спокойным.
- Игры упрощают **объяснение проектов**. Когда вы занимаетесь любым проектом в книге, прежде всего необходимо понять, что же именно вам предстоит сделать, — иногда это оказывается на удивление сложно. Когда в качестве проекта используются игры, это помогает вам быстрее определить, что от вас требуется, и погрузиться в код.
- Создавать игры **интересно!** Ваш мозг намного более восприимчив к новой информации, когда она вам интересна, поэтому включение проектов с построением игр — идея абсолютно естественная.

Мы используем игры для того, чтобы упростить изучение более широких концепций C# и программирования. Они являются важной частью книги. Обязательно выполняйте все игровые проекты в книге, даже если разработка игр не представляет для вас интереса. (Проекты «Лабораторный курс Unity» необязательны, но мы настоятельно рекомендуем вам выполнять их.)



Научные редакторы

Лиза Кельнер



Линдси Бьеда



Татьяна Мэк



Эшли Годболд



На фотографиях не изображены (замечательные) редакторы второго и третьего изданий: Ребекка Дан-Кран, Крис Барроус, Джонни Халиф и Дэвид Стерлинг.

А также редакторы первого издания: Джей Хилард, Дэниел Киннаэр, Айям Сингх, Теодор Кассер, Энди Паркер, Питер Ричи, Кристина Пала, Билл Метельски, Уэйн Брэдни, Дэйв Мэрдок, и особенно Бриджитт-Жули Ландерс.

Мы хотим особо поблагодарить наших замечательных читателей, прежде всего Алана Уэлетта, Джеффа Каунтса, Терри Грэхема, Сергея Кулагина, Уильяма Пива и Грега Комбоу, — которые сообщали нам о проблемах, обнаруженных при чтении книги, а также профессора Джо Варрасо из колледжа Мохаук, который включил нашу книгу в свой учебный курс.

Огромное спасибо всем!!!

«Если я видел дальше других, то потому, что стоял на плечах гигантов». — Исаак Ньютон

В книге, которую вы читаете, очень мало ошибок, и ее высокое качество в ЗНАЧИТЕЛЬНОЙ мере является заслугой наших замечательных научных редакторов — гигантов, которые любезно подставили нам свои плечи. Группе редакторов: мы невероятно благодарны за всю работу, которую вы проделали. Спасибо!

Линдси Бьеда — программист из Питтсбурга (штат Пенсильвания). В ее доме больше клавиатур, чем у любого другого человека. Когда Линдси не занимается программированием, она играет со своим котом по имени Дэш и пьет чай. С ее проектами и рассуждениями можно ознакомиться по адресу rarlindseysmash.com.

Татьяна Мэк — независимый американский инженер, работающий напрямую с организациями для построения логичных и стройных продуктов и систем. Она верит в то, что стремление к доступности, производительности и инклюзивности может улучшить нашу социальную среду как на цифровом, так и на физическом уровне. В этическом отношении она считает, что технологические специалисты могут избавляться от ограничительных систем в пользу инклюзивных и ориентированных на сообщества.

И мы полностью согласны с ней!

Доктор **Эшли Годболд** — программист, разработчик игр, автор, математик, преподаватель и мать. Она работает на полную ставку преподавателем программирования в крупной розничной сети, а также руководит небольшой инди-студией видеоигр Mouse Potato Games. Эшли является сертифицированным преподавателем Unity и ведет в колледжах учебные курсы по компьютерной теории, математике и разработке игры. Она является автором книг «Mastering Unity 2D Game Development (2nd Edition)» и «Mastering UI Development and Unity», а также разработчиком видеокурсов «2D Game Programming in Unity» и «Getting Started with Unity 2D Game Development».

И мы хотим от всей души поблагодарить **Лизу Кельнер** — это уже 12-я (!!!) книга, которую она редактирует для нас. Огромное спасибо!

Также хотим особо поблагодарить **Джо Албахари** и **Джона Скита** за их невероятную техническую квалификацию, предельно внимательное и продуманное редактирование первого издания, которое стало одной из составляющих успеха этой книги за прошедшие годы. Их информация нам сильно пригодилась — на самом деле намного сильнее, чем нам казалось в то время.



Благодарности

Редактор:

Прежде всего мы хотим поблагодарить нашего замечательного редактора **Николь Тачи** за все, что она сделала для нашей книги. Ты сделала все, чтобы книга увидела свет, и предоставила много полезной информации. Спасибо тебе!

Команда издательства O'Reilly:

Кэтрин Тозер



В издательстве O'Reilly очень много сотрудников, которых мы хотели бы поблагодарить, надеемся, что никого не забыли. Как всегда, хотелось бы выразить признательность Мэри Треслер, которая была с нами с первых дней нашего сотрудничества с O'Reilly. Спасибо выпускающему редактору Кэтрин Тозер, составителю алфавитного указателя Джоан Спротт и Рэйчел Хед за внимательную корректуру — все они помогли издать эту книгу за рекордно короткий срок. Большое сердечное спасибо Аманде Квинн, Оливии Макдональд и Мелиссе Даффилд, которые помогали поддерживать весь проект «на ходу». И еще вспомним наших других друзей из O'Reilly: Энди Орама, Джеффа Блайла, Майка Хендриксона и, конечно, Тима О'Рейли. За то, что вы сейчас читаете эту книгу, следует поблагодарить самую лучшую команду рекламистов: Марси Хэнон, Кэтрин Баррет, Сару Пейтон и остальных замечательных людей из города Севастополь, штат Калифорния.

И еще хочется упомянуть наших любимых авторов O'Reilly:

- Пэрис Баттфилд-Эддисон, Джона Мэннинга и Тима Нагента — их книга «Unity Game Development Cookbook» просто великолепна. Мы с нетерпением ожидаем книги «Head First Swift», написанной Пэрис и Джоном.
- Джозефа Албахари и Эрика Иохансена, написавших бесценную книгу «C# 8.0 in a Nutshell».

И наконец...

Большое спасибо Кэти Вайс из Indie Gamer Chick Fame за ее интересную информацию об эпилепсии, использованную в главе 10, а также за ее благородную борьбу за интересы людей, страдающих эпилепсией. Также благодарим Патрисию Аас за ее замечательное видео об изучении C# как второго языка, использованное в приложении «Ката программирования», и за полученную от нее информацию о том, как упростить изучение C# для опытных программистов.

Огромное спасибо нашим друзьям из Microsoft, помогавшим нам в работе над книгой, — ваша поддержка в этом проекте была просто замечательной. Мы благодарим Доминика Нахуса (поздравляем с рождением ребенка!), Джордана Мэттисена и Джона Миллера из команды Visual Studio для Mac, а также Коди Бейера, сыгравшего важную роль в организации нашего сотрудничества. Спасибо Дэвиду Стерлингу за великолепную редактуру третьего издания и Иммо Ландверту, который помог нам определиться с темами, включенными в это издание. Дополнительно благодарим Мэдса Торгерсена, руководителя программы разработки языка C#, за его великолепное руководство и советы за все прошедшие годы. Вы просто великолепны.

Мы особенно благодарны Джону Галлоуэю, который предоставил столько замечательного кода для проектов Blazor. Джон — старший руководитель программы разработки .NET Community Team. Он участвовал в написании нескольких книг о .NET, помогал проводить встречи .NET Community Standup и выступал соорганизатором подкаста «Herding Code». Огромное спасибо!

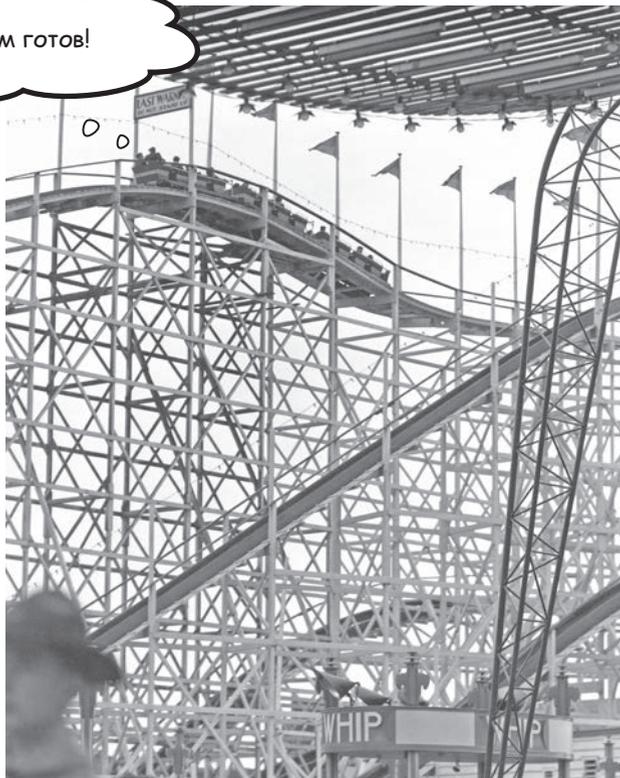
Джон Галлоуэй



1 Начало работы с C#

Быстро сделать что-то классное!

К диким гонкам готов!



Хотите программировать быстро? C# — это **мощный язык программирования**. Благодаря **Visual Studio** вам не потребуется писать непонятный код, чтобы заставить кнопку работать. Вместо того чтобы запоминать параметры метода для *имени* и для *ярлыка* кнопки, вы сможете **создать действительно классное приложение**. Звучит заманчиво? Тогда переверните страницу, и приступим к делу.

Зачем вам изучать C#

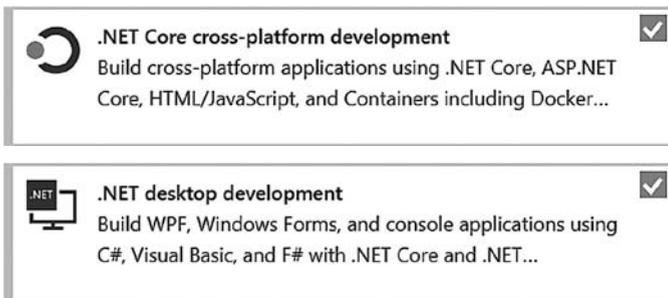
C# — простой современный язык, на котором можно решать совершенно невероятные задачи. Изучая C#, вы не ограничиваетесь только новым языком. C# делает доступным для вас целый мир .NET — невероятно мощной платформы с открытым кодом для построения самых разнообразных приложений.

Visual Studio станет вашим окном в C#

Если вы еще не установили Visual Studio 2019, самое время это сделать. Откройте страницу <https://visualstudio.microsoft.com> и загрузите версию **Visual Studio Community Edition**. (Если она уже установлена на вашем компьютере, запустите программу установки Visual Studio, чтобы обновить набор установленных компонентов.)

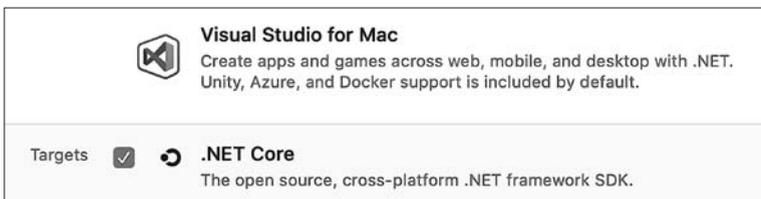
Если вы работаете в Windows...

Проследите за тем, чтобы у вас была установлена поддержка кросс-платформенной разработки .NET Core и настольных средств разработки .NET. Только не включайте вариант Game development with Unity — позднее в книге мы займемся разработкой 3D-игр на базе Unity, но поддержка Unity будет установлена отдельно.



Если вы работаете на Mac...

Загрузите и запустите программу установки Visual Studio для Mac. Проследите за тем, чтобы была установлена цель .NET Core.



Убедитесь в том, что вы устанавливаете Visual Studio, а не Visual Studio Code.

Visual Studio Code — прекрасный кросс-платформенный редактор с открытым кодом, но он не так хорошо подходит для разработки .NET, как Visual Studio. Вот почему мы будем использовать Visual Studio в книге как учебный и аналитический инструмент.

Проекты ASP.NET можно создавать и в системе Windows! Для этого проследите за тем, чтобы при установке Visual Studio был включен компонент «ASP.NET and web development».



В большинстве проектов этой книги создаются консольные приложения .NET Core, которые работают как в системе Windows, так и на Mac. В некоторых главах рассматриваются проекты (например, игра на поиск пар животных позднее в этой главе), являющиеся настольными проектами для Windows. Для этих проектов используйте приложение «Проекты ASP.NET Core Blazor». В нем приведена полная замена для главы 1, а также версии ASP.NET Core Blazor для других проектов WPF.

Visual Studio — инструмент для написания кода и изучения C#

Код C# можно писать и в Блокноте или в другом текстовом редакторе, но существует более удобный вариант. **IDE** (сокращение от Integrated Development Environment, т. е. «интегрированная среда разработки») включает в себя текстовый редактор, визуальный конструктор, менеджер файлов, отладчик... словно многофункциональный инструмент для всех операций, которые могут понадобиться при написании кода.

Перечислим лишь несколько задач, в решении которых вам поможет Visual Studio:

1 **БЫСТРОЕ создание приложения.** Язык C# гибок и прост в изучении, а Visual Studio позволяет автоматизировать многие операции, которые обычно приходится выполнять вручную. Примеры операций, которые Visual Studio делает за вас:

- ★ управление файлами проекта;
- ★ удобное редактирование кода проекта;
- ★ управление графикой, аудио, значками и другими ресурсами проекта;
- ★ отладка кода с возможностью выполнения в пошаговом режиме.

2 **Построение удобных интерфейсов.** Визуальный конструктор (Visual Designer) в Visual Studio IDE — одно из самых простых и удобных средств конструирования. Он делает за вас столько всего, что построение пользовательских интерфейсов для ваших программ становится одним из самых приятных аспектов разработки приложений C#. Чтобы строить полнофункциональные профессиональные программы, вам не придется тратить многие часы на возню с пользовательским интерфейсом (если только вы сами этого не захотите).

3 **Построение восхитительных в визуальном отношении программ.** Объединяя C# с XAML (язык визуальной разметки для проектирования пользовательских интерфейсов для настольных приложений WPF), вы используете один из самых эффективных инструментов для построения визуальных программ... и используете его для построения программ, которые выглядят так же великолепно, как и работают.

Пользовательский интерфейс (UI) любого приложения WPF строится на базе XAML (eXtensible Application Markup Language). Visual Studio содержит очень удобные средства для работы с XAML.

4 **Изучение и исследование C# и .NET.** Visual Studio представляет собой инструмент разработки мирового уровня, но к счастью для нас, это также великолепный учебный инструмент. Мы будем использовать **IDE** для исследования C#, что позволит вам быстро закрепить важные концепции программирования в вашем мозгу.

В книге Visual Studio будем называться просто «IDE».



Если вы работаете с Visual Studio на Mac, вы будете строить такие же великолепные приложения, но вместо XAML при этом будет использоваться комбинация C# с HTML.

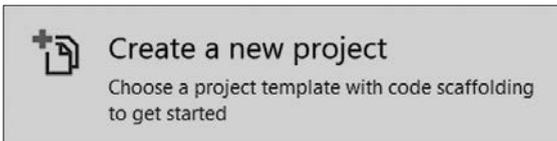
Visual Studio — замечательная среда разработки, но мы также будем использовать ее как учебный инструмент для изучения C#.

Создание вашего первого проекта в Visual Studio

Лучший способ изучения C# — непосредственное написание кода, поэтому мы воспользуемся Visual Studio для **создания нового проекта...** и немедленно начнем писать код!

1 Создайте новый проект по шаблону Console App (.NET Core).

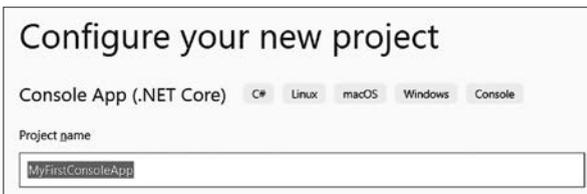
Запустите Visual Studio 2019. При первом запуске на экране появится окно Create a new project с несколькими вариантами. Выберите вариант Create a new project. Если вы закроете окно, не беспокойтесь: чтобы вызвать его обратно, достаточно выбрать в меню команду File>New>Project.



Выберите тип проекта **Console App (.NET Core)** и нажмите кнопку Next.



Введите имя проекта **MyFirstConsoleApp** и нажмите кнопку Create.



2 Просмотрите код нового приложения.

При создании нового проекта Visual Studio дает вам отправную точку для дальнейшей работы. Как только создание новых файлов для приложения будет завершено, на экране должен открыться файл с именем Program.cs со следующим кодом:

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

← При создании нового проекта консольного приложения Visual Studio автоматически добавляет класс с именем Program.

Класс изначально содержит метод с именем Main с одной командой для вывода строки текста на консоль. Классы и методы будут намного подробнее рассматриваться в главе 2.

← (Делайте это!)

Когда вы видите в тексте врезку (Делайте это! (или Добавьте!, или Принципы отладки), откройте Visual Studio и выполните приведенные инструкции. Мы точно расскажем, что следует делать и на что нужно обратить внимание, чтобы извлечь максимум пользы из приведенного примера.

Если вы используете Visual Studio для Mac, код этого проекта – и всех проектов .NET Core Console App в этой книге – останется неизменным, но некоторые функции IDE будут отличаться. За версией этой главы для Mac обращайтесь к приложению «Visual Studio for Mac Learner's Guide».

3 Запустите новое приложение.

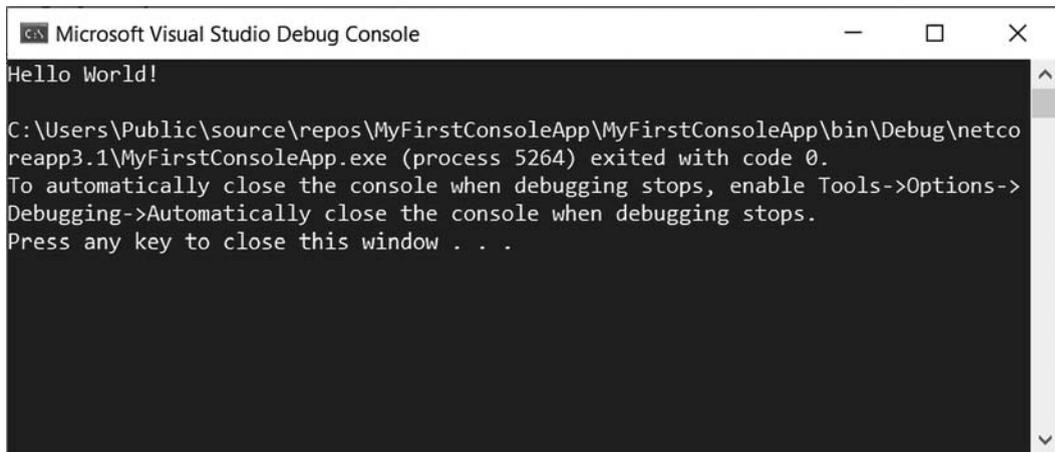
Приложение, которое среда Visual Studio создала за вас, готово к запуску. Найдите в верхней части Visual Studio IDE кнопку с зеленым треугольником и именем вашего приложения и нажмите ее:



4 Просмотрите результаты выполнения программы.

Когда вы запускаете вашу программу, на экране появляется окно отладочной консоли Microsoft Visual Studio с результатом выполнения программы:

Когда вы запускаете свое приложение, оно выполняет метод `Main`, который выводит эту строку текста на консоль.



Лучший способ изучить язык — писать на нем побольше кода, поэтому в этой книге мы будем строить множество программ. Многие из них будут проектами .NET Core Console App, поэтому давайте повнимательнее посмотрим, что же было сделано.

В верхней части окна выводятся выходные данные программы:

Hello World!

Далее происходит переход на следующую строку, за которой идет дополнительный текст:

```
C:\path-to-your-project-folder\MyFirstConsoleApp\MyFirstConsoleApp\bin\Debug\netcoreapp3.1\MyFirstConsoleApp.exe (process ####) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

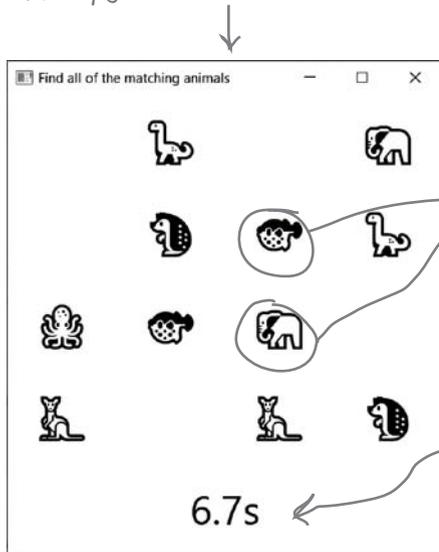
Это сообщение выводится в нижней части каждого окна отладочной консоли. Ваша программа вывела одну строку текста (`Hello world!`), после чего завершила работу. Visual Studio оставляет окно вывода открытым, ожидая, когда вы нажмете клавишу для его закрытия; это позволяет вам просмотреть результаты до того, как оно исчезнет с экрана.

Нажмите клавишу, чтобы закрыть окно. Затем снова запустите программу. Так вы будете запускать все проекты .NET Core Console App, которые будут создаваться в книге.

Давайте построим игру!

Вы только что построили свое первое приложение C#, и это замечательно! А теперь попробуем создать что-то посложнее. Мы построим игру, в которой игрок должен подбирать **пары животных**. На экране выводится квадратная сетка с 16 животными, а игрок щелкает на парах, чтобы они исчезали с экрана.

Игра с подбором пар животных, которую мы построим.



На экране выводятся восемь пар разных животных, случайным образом распределенных в окне. Игрок щелкает на двух животных: если эти животные одинаковы, то пара исчезает из окна.

Таймер следит за тем, сколько времени потребовалось игроку для завершения игры. Цель игрока — найти все пары за минимальное количество времени.

Игра является приложением WPF

Консольные приложения прекрасно подходят для ввода и вывода текста. Если вы хотите построить визуальное приложение, которое работает в окне, придется использовать другую технологию. Вот почему игра с подбором пар животных будет приложением **WPF**. WPF (Windows Presentation Foundation) позволяет создавать настольные приложения, которые могут работать в любой версии Windows. Во многих главах книги будет представлено одно приложение WPF. В этой главе кратко представлено WPF и описаны средства для построения как визуальных, так и консольных приложений.

Построение проектов разных типов является важной частью изучения C#. Мы выбрали WPF (Windows Presentation Foundation) для некоторых проектов в этой книге, потому что эта платформа предоставляет детализированные пользовательские интерфейсы, работающие во многих версиях Windows (даже в очень старых версиях, таких как Windows XP.)

Но язык C# не ограничивается приложениями Windows!

Вы работаете на Mac? Что же, вам повезло! Мы добавили для вас особый учебный план с описанием Visual Studio для Mac. Обращайтесь к приложению «Краткий курс Visual Studio для Mac» в конце книги. В нем приведена полная альтернативная версия этой главы, а также Mac-версии всех проектов WPF, представленных в книге.

В версиях проектов WPF для Mac используется ASP.NET Core. Проекты ASP.NET Core также могут строиться и для системы Windows.

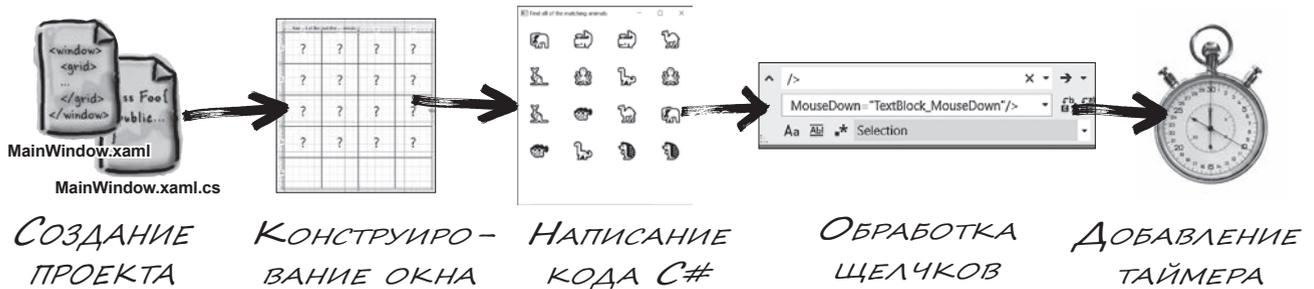
К тому времени, когда вы завершите этот проект, вы гораздо лучше освоите те инструменты, которые будут использованы в книге для изучения и исследования возможностей C#.

Как построить игру

В оставшейся части этой главы будет рассмотрен процесс построения игры, который состоит из нескольких шагов:

1. Сначала мы создадим новый проект настольного приложения в Visual Studio.
2. Затем мы воспользуемся XAML для построения окна.
3. Мы напишем код C# для размещения случайного эмодзи с животным в окне.
4. Игрок щелкает на парных эмодзи, чтобы удалить их из окна.
5. Наконец, чтобы игра стала более интересной, мы добавим в нее таймер.

На этот проект вам может потребоваться от 15 минут до часа (в зависимости от того, насколько быстро вы набираете текст). Мы лучше учимся, когда не спешим, так что не жалейте времени.



Обращайте внимание на врезки «Разработка игр... и не только», встречающиеся в тексте книги. Мы будем использовать принципы проектирования для изучения и исследования важных концепций и идей программирования, применимых в любых проектах, не только в видеоиграх.



Разработка игр... и не только

Что такое игра?

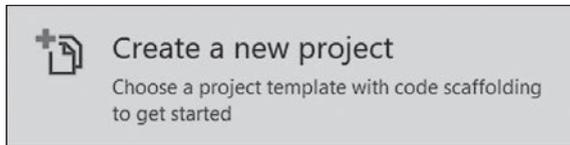
Вроде бы ответ на этот вопрос совершенно очевиден. Но задумайтесь на минутку — все не так просто, как кажется на первый взгляд.

- У всех ли игр есть **победитель**? У каждой ли игры есть конец? Не обязательно. Как насчет имитатора полета? Игры, в которой вы строите парк развлечений? Или таких игр, как The Sims?
- Игры всегда **интересны**? Не для всех. Некоторым игрокам нравится процесс «гринда», когда им приходится делать одно и то же раз за разом; другим это кажется ужасным.
- Всегда ли игры сопряжены с **принятием решений**, **конфликтами** или **решением задач**? Нет, не всегда. «Симуляторы ходьбы» — класс игр, в которых игрок просто исследует игровую среду, и часто в них вообще нет никаких головоломок или конфликтов.
- Обычно довольно трудно четко определить, что же именно следует считать игрой. В учебниках по разработке игр можно найти множество разных определений. Для наших целей определим смысл «игры» (по крайней мере для **хороших игр**) следующим образом:

Игра — программа, взаимодействовать с которой не менее увлекательно, чем создавать ее.

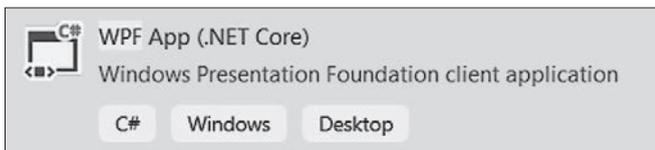
Создание проекта WPF в Visual Studio

Запустите новый экземпляр Visual Studio 2019 и создайте новый проект:

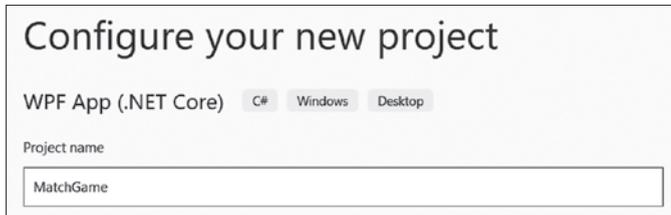


Мы закончили с проектом консольного приложения, созданным в первой части этой главы, поэтому тот экземпляр Visual Studio можно закрыть.

Игра будет строиться как настольное приложение с использованием WPF, поэтому выберите вариант **WPF App (.NET Core)** и щелкните на кнопке Next:



Visual Studio предложит вам настроить проект. **Введите имя проекта MatchGame** (при желании также можно изменить папку для создания проекта):



Этот файл содержит код XAML, определяющий пользовательский интерфейс главного окна.



MainWindow.xaml

Щелкните на кнопке Create. Visual Studio создает новый проект с именем MatchGame.

Visual Studio создает папку проекта с множеством файлов

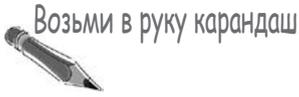
При создании нового проекта Visual Studio добавляет новую папку с именем MatchGame и заполняет ее файлами и папками, необходимыми для вашего проекта. Мы изменим два файла, *MainWindow.xaml* и *MainWindow.xaml.cs*.

В этом файле размещается код C#, который обеспечивает работоспособность вашей игры.



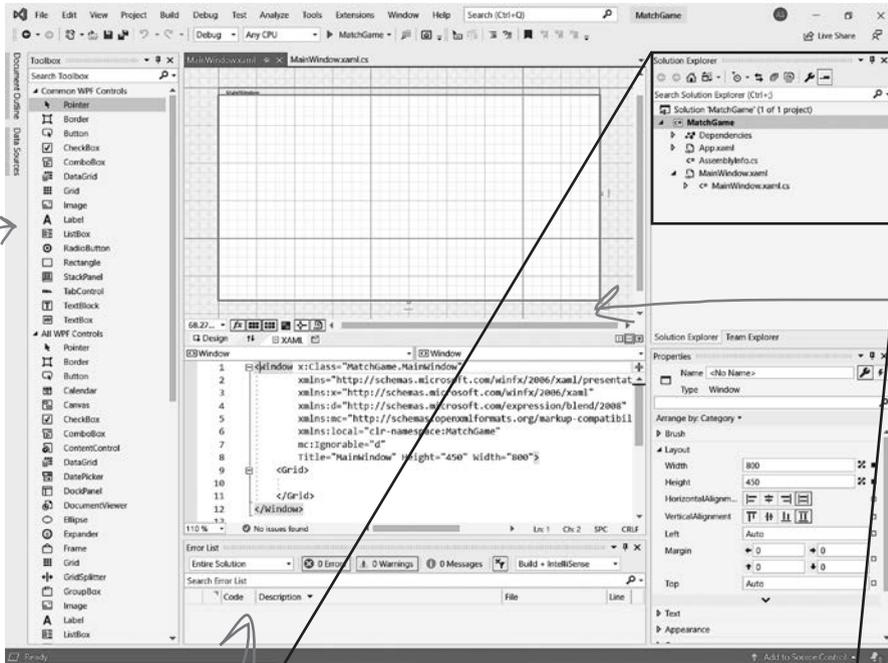
MainWindow.xaml.cs

Если вы столкнетесь с какими-либо трудностями во время реализации проекта, перейдите на страницу GitHub и откройте видеоролик по ссылке: <https://github.com/head-first-csharp/fourth-edition>



Упражнения «Возьми в руку карандаш» являются обязательными. Это важная часть обучения, тренировки и совершенствования ваших навыков в C#.

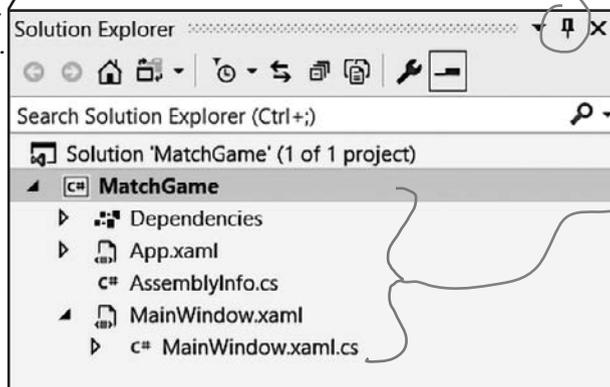
Настройте свою IDE в соответствии с приведенным снимком экрана. Сначала откройте файл **MainWindow.xaml** — сделайте двойной щелчок на нем в окне Solution Explorer. Затем откройте окно **Toolbox** и **Error List**, выбрав их в меню **View**. Чтобы понять предназначение многих окон и файлов, достаточно присмотреться к их именам. Попробуйте предположить, что делает каждая часть Visual Studio IDE. Мы привели один ответ, чтобы вам было проще начать работу. Попробуйте сделать обоснованные предположения для других частей IDE.



Конструктор... позволяет редактировать внешний вид интерфейса, перетаскивая элементы управления.

Вы заметили, что панель инструментов исчезает с экрана? Щелкните на этом значке, чтобы она оставалась на экране.

Мы выбрали тему **Light**, так как светлые снимки экрана лучше выглядят на бумаге. Чтобы выбрать другую тему, выберите команду «Options...» в меню **Tools** и выделите раздел **Environment**.



знайте ide

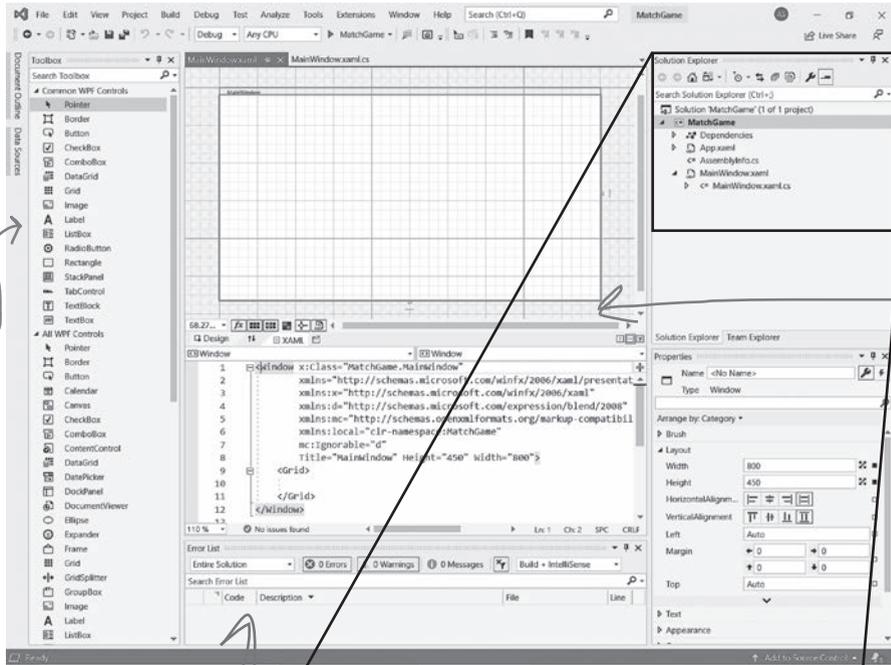


Возьми в руку карандаш
Решение

Мы привели описания разных частей Visual Studio C# IDE. Надеемся, вы правильно предположили, для чего предназначено каждое окно и каждый раздел IDE. Если ваши ответы в чем-то отличаются от наших, не огорчайтесь! Вы ОЧЕНЬ МНОГО узнаете об IDE во время практической работы.

И напомним еще раз: термины «Visual Studio» и «IDE» используются в книге как синонимы — в том числе и на этой странице.

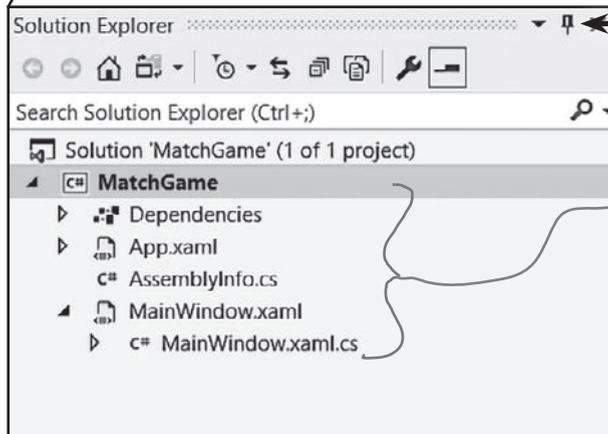
Панель инструментов. На ней размещаются визуальные элементы управления, которые можно перетаскивать в окно конструктора.



Конструктор позволяет редактировать внешний вид интерфейса, перетаскивая элементы управления.

В окне свойств выводятся свойства текущего выделенного элемента в конструкторе.

В окне Error List выводятся ошибки в коде. На этой панели отображается диагностическая информация.



Значок в виде кнопки переключает режим автоматического закрытия панели. Для окна Toolbox этот режим включен.

Файлы C# и XAML, которые создает IDE при добавлении нового проекта, появляются в окне Solution Explorer вместе с другими файлами приложения.

Окно Solution Explorer в IDE позволяет переходить от одного файла к другому.

Часть
Задаваемые
Вопросы

← Не пропускайте эти разделы. В них часто приводятся ответы на самые насущные вопросы, а также встречаются вопросы, которые возникают у других читателей. Кстати, многие из этих вопросов были заданы читателями предыдущих изданий книги!

В: Если код создается автоматически, не сводится ли изучение C# к изучению функциональности IDE?

О: Нет. IDE прекрасно генерирует шаблонный код, но не более того. Она неплохо справляется с такими задачами, как выбор исходного состояния или автоматическое изменение свойств элементов в UI, но понять, какую работу должна выполнять программа и как достичь поставленной цели, можете только вы. И хотя Visual Studio IDE считается одной из самых совершенных сред разработки, она не всесильна. Именно **вы**, а не IDE пишете код, непосредственно выполняющий работу приложения.

В: Что делать с ненужным кодом, автоматически созданным IDE?

О: Вы можете изменить или удалить его. IDE настроена так, чтобы генерировать код на основании наиболее распространенного использования элемента, но иногда это не то, что вам требуется. Все, что IDE делает за вас — каждую строку сгенерированного кода, каждый добавленный файл, — можно изменить вручную или удобными средствами пользовательского интерфейса IDE.

В: Почему вы предложили мне установить версию Visual Studio Community Edition? Вы уверены, что для полноценного использования материала книги мне не понадобится одна из платных версий Visual Studio?

О: В книге нет ничего, что нельзя было бы сделать в бесплатной версии Visual Studio (которую можно загрузить на сайте Microsoft). Различия между Community Edition и другими версиями не мешают вам писать код на C# и строить полнофункциональные, завершённые приложения.

В: В книге упоминается о комбинации C# и XAML. Что такое XAML и как его комбинировать с C#?

О: XAML (произносится «зэмл») — это **язык разметки**, позволяющий строить пользовательские интерфейсы для приложений WPF. XAML базируется на XML (так что если вам доводилось иметь дело с HTML, вам будет проще). Пример **тега** XAML, рисующего серый эллипс:

```
<Ellipse Fill="Gray"
  Height="100" width="75"/>
```

Если вы вернетесь к своему проекту и введете этот тег после <Grid> в коде XAML, в середине окна появится серый эллипс. На то, что это тег, указывает символ <, за которым следует слово («Ellipse»). Все вместе составляет **открывающий**, или **начальный**, тег. Тег Ellipse обладает тремя **свойствами**: одно задает серый цвет заливки, а два других определяют его ширину и высоту. Тег завершается символом />, но некоторые теги XAML могут включать в себя другие теги. Данный тег можно превратить в **контейнерный**, заменив /> на >, добавив другие теги (которые, в свою очередь, могут содержать внутренние теги) и завершив конструкцию **закрывающим**, или **конечным**, тегом: </Ellipse>.

В книге вы узнаете, как работает XAML, и освоите много других тегов XAML.

В: Мой экран отличается от вашего! Одних окон вообще нет, другие находятся в других местах. Я сделал что-то не так? Как вернуться к исходной настройке?

О: Чтобы вернуться к настройкам по умолчанию, выберите команду **Reset Window Layout** в меню Window — IDE восстановит стандартную конфигурацию окна. Затем при помощи команды View ► Other Windows можно **открыть окно Toolbox и Error List**, чтобы ваше окно IDE не отличалось от приведенного.

Visual Studio генерирует код, который может стать заготовкой для построения вашего приложения. Но только вы отвечаете за правильность работы этой программы.

Панель инструментов закрывается по умолчанию. Воспользуйтесь значком с изображением канцелярской кнопки в правом верхнем углу, чтобы она оставалась на экране.

использование xaml

В начале каждого раздела проекта будет приведена «карта», которая поможет вам представить место текущего этапа в общей картине.



Построение окна с использованием XAML

Итак, среда Visual Studio создала проект WPF. Пришло время поработать с XAML.

XAML (сокращение от **eXtensible Application Markup Language**) — чрезвычайно гибкий язык разметки, используемый разработчиками C# для построения пользовательских интерфейсов. При создании приложений используются две разновидности кода. Сначала разработчик строит пользовательский интерфейс (UI) на базе кода XAML, а затем добавляет код C#, обеспечивающий непосредственную работу игры.

Если вы когда-либо использовали HTML для создания веб-страниц, наверняка XAML покажется вам знакомым. Маленький пример создания окна в XAML:

```
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="This is a WPF window" Height="100" Width="400"> ①
  <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
    <TextBlock FontSize="18px" Text="XAML helps you design great user interfaces."/> ②
    <Button Width="50" Margin="5,10" Content="I agree!"/> ③
  </StackPanel>
</Window>
```

Мы добавили нумерацию в части XAML, в которых определяется текст.

Найдите соответствующие номера на следующем снимке.

А теперь посмотрим, как выглядит окно, когда WPF **отображает** (рисует) его на экране. Окно содержит два видимых **элемента управления**: элемент TextBlock для вывода текста и элемент Button, на котором пользователь может щелкать. Они размещаются на невидимом элементе StackPanel, который обеспечивает их отображение поверх друг друга. Посмотрите, как выглядят элементы на следующем снимке экрана, затем вернитесь к XAML и найдите теги TextBlock и Button.

Элемент TextBlock предназначен для вывода блока текста.

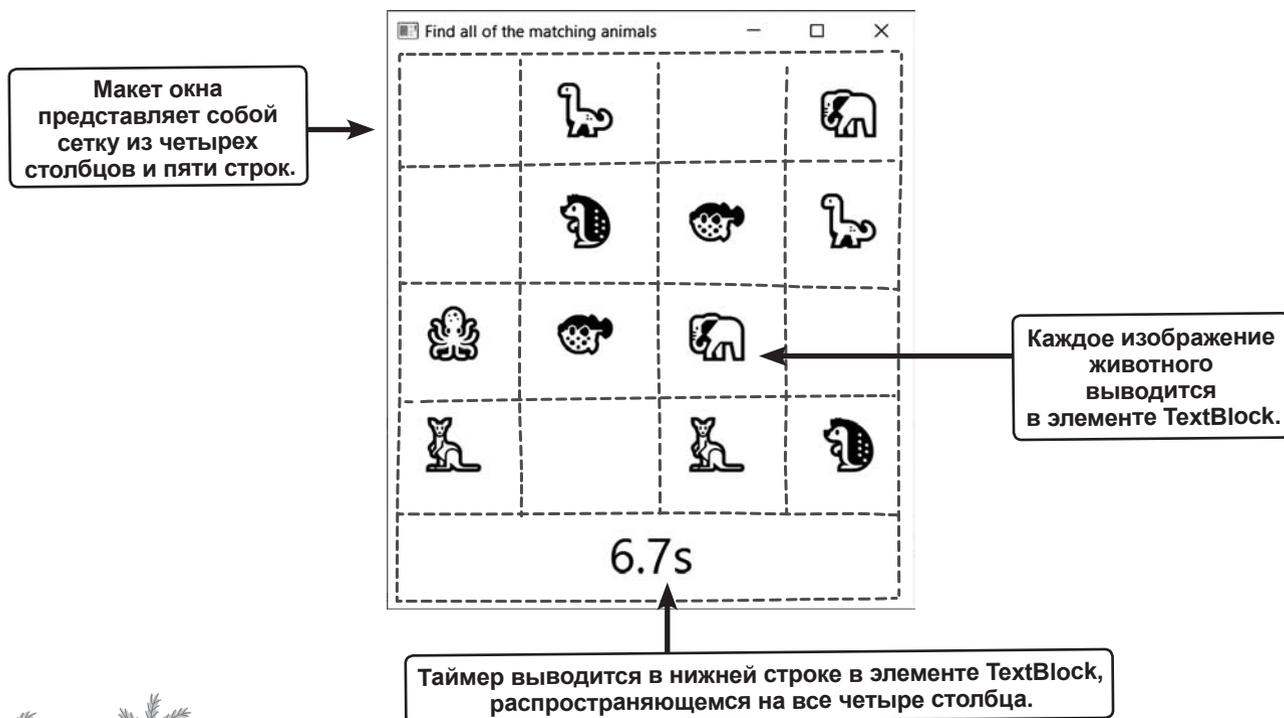


Номерами обозначены части UI, соответствующие нумерации в коде XAML.

Построение окна для игры

Для работы приложения понадобится графический интерфейс; объекты, обеспечивающие работу игры; и исполняемый файл, который можно будет запустить. На первый взгляд кажется, что нам предстоит грандиозная работа, но к концу этой главы все будет готово, а вы получите представление о том, как использовать Visual Studio для построения эффектных приложений WPF.

Окно приложения, которое мы собираемся построить, имеет следующую структуру:



Знание XAML — важный навык для разработчика C#.

Возможно, вы думаете: «Минутку! Книга называется “Head First. Изучаем C#”. Зачем тратить столько времени на XAML?»

Приложения WPF используют XAML для построения пользовательского интерфейса — как и другие виды проектов C#, XAML может использоваться не только в настольных приложениях, те же навыки могут применяться для построения мобильных приложений C# для Android и iOS на базе технологии Xamarin Forms, использующей разновидность XAML (с *немного* отличающимся набором элементов). Именно поэтому построение пользовательских интерфейсов на базе XAML является важным навыком для каждого разработчика C#, и в этой книге вы еще многое узнаете о XAML. Мы покажем, как строить код XAML **шаг за шагом** — средства конструктора XAML в Visual Studio 2019 позволяют построить пользовательский интерфейс в визуальном режиме, без ввода значительного объема кода. *Повторим для полной ясности:*

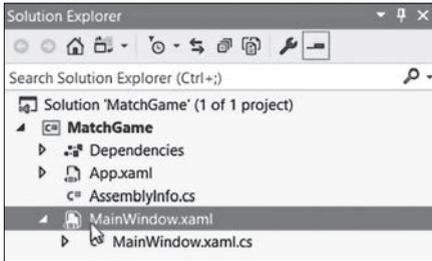
XAML — код, определяющий пользовательский интерфейс. **C#** — код, определяющий поведение.

Определение размера окна и текста заголовка в свойствах XAML

Начнем с построения пользовательского интерфейса для игры с подбором пар. Первое, что необходимо сделать, — уменьшить ширину окна и изменить его заголовок. Заодно вы познакомитесь с конструктором XAML Visual Studio — мощным инструментом для построения эффектных пользовательских интерфейсов ваших приложений.

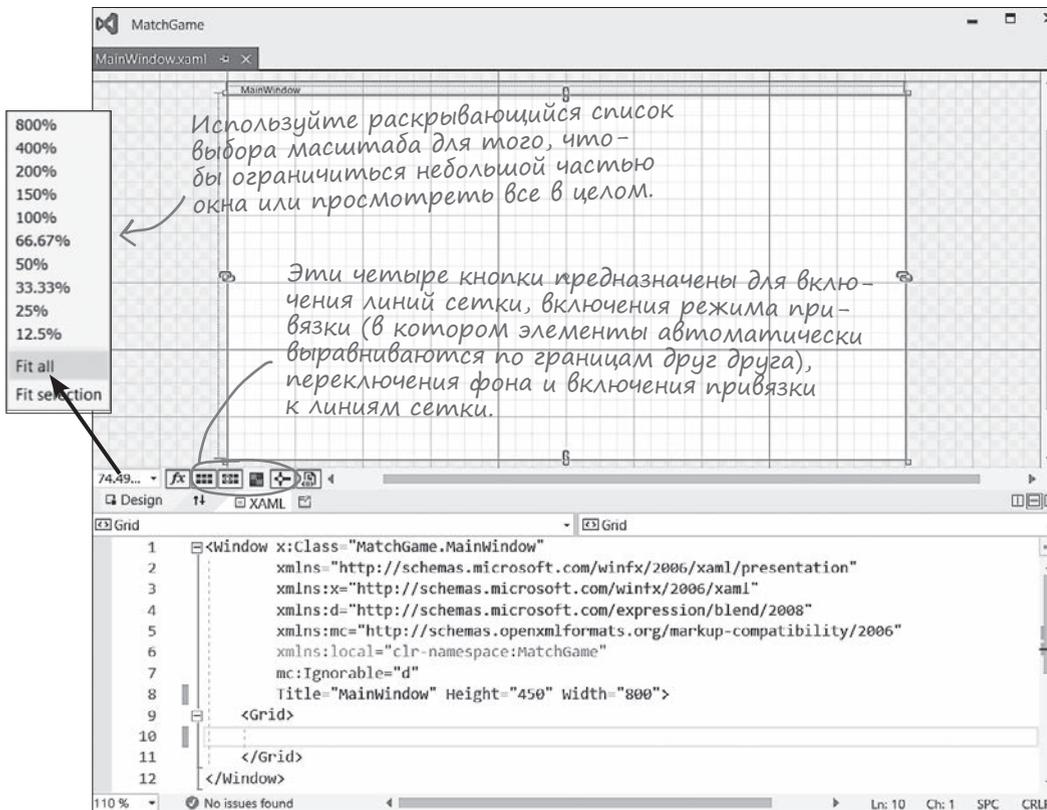
1 Выделите главное окно.

Сделайте двойной щелчок на файле MainWindow.xaml на панели Solution Explorer.



Двойной щелчок на файле панели Solution Explorer открывает этот файл в соответствующем редакторе. Файлы с кодом C# и расширением .cs открываются в редакторе кода. Файлы XAML с расширением .xaml открываются в конструкторе XAML.

Visual Studio немедленно открывает файл в конструкторе XAML.



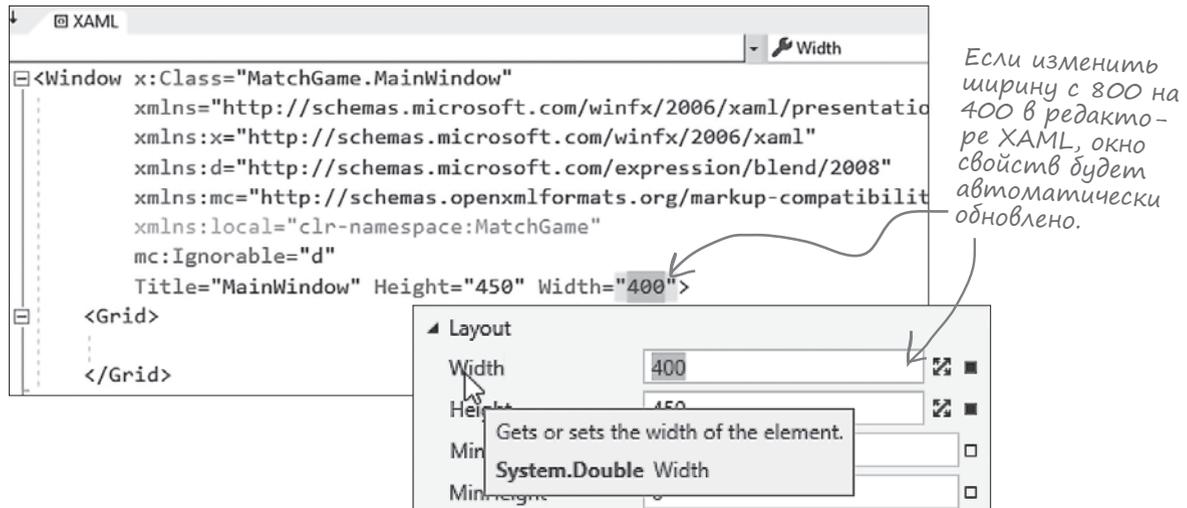
В конструкторе выводится предварительное изображение редактируемого окна. Любые изменения, которые вы вносите, отражаются в выводимом ниже коде XAML.

Также можно вносить изменения в код XAML и немедленно видеть результаты в верхней области предварительного просмотра.

2 Изменение размеров окна.

Переместите мышь в редактор XAML и щелкните в любой точке в пределах первых восьми строк кода XAML. Как только вы это сделаете, на панели Properties должен появиться перечень свойств.

Раскройте раздел Layout и **измените ширину (Width) на 400**. Окно на панели конструктора должно автоматически уменьшиться. Присмотритесь к коду XAML – свойство Width теперь равно 400.



3 Изменение текста заголовка окна.

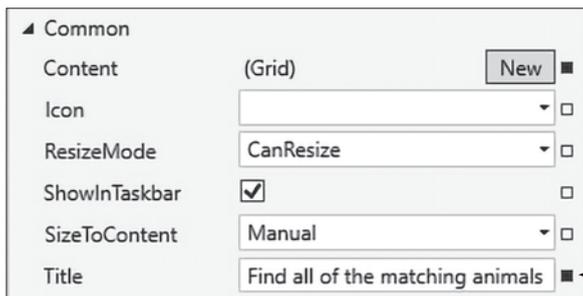
Найдите в конце тега Window следующую строку кода XAML:

```
Title="MainWindow" Height="450" Width="400">
```

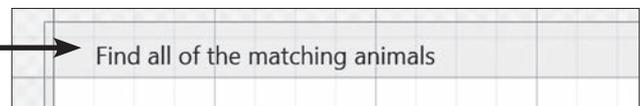
и замените текст заголовка на **Find all of the matching animals**:

```
Title="Find all of the matching animals" Height="450" Width="400">
```

Изменения отражаются в разделе Common окна свойств, – и что еще важнее, в полосе заголовка окна теперь выводится новый текст.



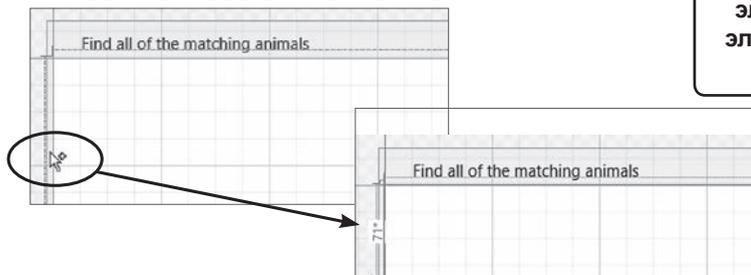
Когда вы изменяете свойства в тегах XAML, изменения автоматически отражаются в окне свойств. Когда вы используете окно свойств для модификации пользовательского интерфейса, IDE обновляет код XAML.



Добавление строк и столбцов в сетку XAML

Может показаться, что главное окно остается пустым, но присмотритесь повнимательнее к нижней части XAML. Видите строку `<Grid>`, за которой следует еще одна строка `</Grid>`? На самом деле окно содержит **сетку**, но в ней пока нет ни строк, ни столбцов. Давайте добавим строку.

Переместите указатель мыши в левую часть окна в конструкторе. Когда над указателем появится знак «+», щелкните кнопкой мыши, чтобы добавить строку.



Пользовательский интерфейс приложения WPF строится из **элементов управления**: кнопок, надписей, флажков и т. д. Сетка является **контейнером** — особой разновидностью элемента, которая может содержать другие элементы. Она использует строки и столбцы для определения макета.

Вы увидите число, за которым следует звездочка, и в окне появляется горизонтальная линия. Вы только что добавили новую строку в сетку! Добавим другие строки и столбцы:

- ★ Повторите операцию еще четыре раза, чтобы сетка содержала пять строк.
- ★ Наведите указатель мыши на верхнюю часть окна и щелкните, чтобы добавить четыре столбца. Окно должно выглядеть так, как показано ниже (у вас числа будут другими — это нормально).
- ★ Вернитесь к коду XAML. Теперь он содержит набор тегов **ColumnDefinition** и **RowDefinition**, соответствующих добавленным строкам и столбцам.

Такие врезки предупреждают вас о важных, но часто неочевидных моментах, которые могут сбить вас с толку или снизить темп обучения.

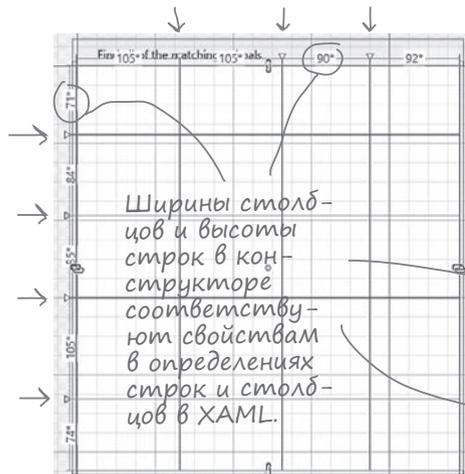


Будьте осторожны!

В вашей IDE что-то может выглядеть иначе.

Все снимки экрана были сделаны в Visual Studio Community 2019 для Windows. Пользователи издания Professional или Enterprise могут заметить ряд второстепенных отличий.

Не беспокойтесь, работать все будет точно так же.



```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="105*" />
  <ColumnDefinition Width="105*" />
  <ColumnDefinition Width="90*" />
  <ColumnDefinition Width="92*" />
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
  <RowDefinition Height="71*" />
  <RowDefinition Height="84*" />
  <RowDefinition Height="85*" />
  <RowDefinition Height="105*" />
  <RowDefinition Height="74*" />
</Grid.RowDefinitions>
```

Выравнивание размеров строк и столбцов

Животные, которых игрок должен распределять по парам, должны находиться на равных расстояниях. Каждое животное находится в ячейке сетки, а сетка автоматически подстраивается под размеры окна, поэтому все строки и столбцы должны иметь одинаковые размеры. К счастью, XAML позволяет легко изменять размеры строк и столбцов. **Щелкните на первом теге RowDefinition в редакторе XAML**, чтобы вывести его свойства в окне Properties:

Щелкните на этом тексте.

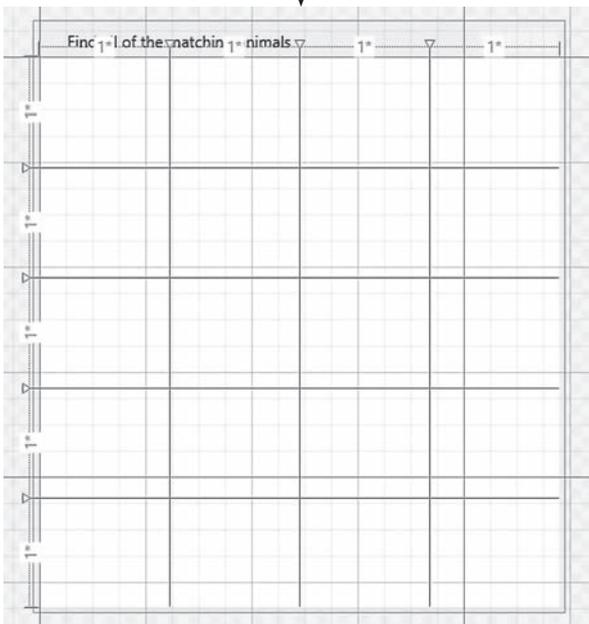
```
<Grid.RowDefinitions>
  <RowDefinition Height="71*" />
  <RowDefinition Height="84*" />
</Grid.RowDefinitions>
```



Когда этот квадратик заполнен, это означает, что свойство не имеет значения по умолчанию. Щелкните на квадратике и выберите в открывшемся меню команду Reset, чтобы вернуть ему значение по умолчанию.

Перейдите в окно свойств и щелкните на квадратике справа от свойства Height, после чего выберите в открывшемся меню команду Reset. Минутку, что происходит? Как только вы это делаете, строка исчезает из конструктора. Впрочем, на самом деле она не совсем исчезает — она просто становится очень узкой. Сбросьте свойство Height для всех строк. Затем сбросьте свойство Width для всех столбцов. После этого сетка должна состоять из четырех столбцов одинаковой ширины и пяти строк одинаковой высоты.

Вот что вы должны видеть в конструкторе:



Попробуйте прочитать код XAML. Если ранее вы никогда не работали с HTML или XML, он может показаться мешаниной <угловых скобок> и /косых черт. Чем внимательнее вы будете к нему присматриваться, тем более осмысленным он вам будет казаться.

А вот что вы должны видеть в редакторе XAML между открывающим тегом <Window...> и закрывающим тегом </Window>:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
</Grid>
```

Код XAML для создания сетки из четырех столбцов одинаковой ширины и пяти строк одинаковой высоты.

Размещение элементов TextBlock в сетке

WPF использует элементы **TextBlock** для вывода текста; мы воспользуемся ими для вывода изображений животных. Добавим такой элемент в окно.

Раскройте раздел Common WPF Controls на панели инструментов и перетащите элемент **TextBlock** в ячейку, находящуюся во втором столбце и второй строке. IDE добавляет тег **TextBlock** между начальным и конечным тегом **Grid**:

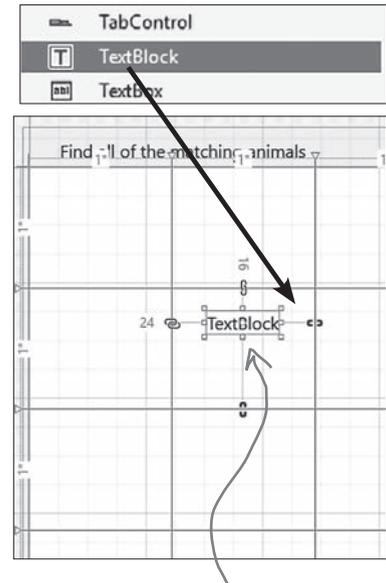
```
<TextBlock Text="TextBlock"
  HorizontalAlignment="Left" VerticalAlignment="Center"
  Margin="560,0,0,0" TextWrapping="Wrap" />
```

Код XAML для этого элемента **TextBlock** содержит пять свойств:

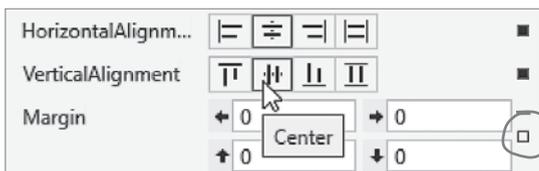
- ★ **Text** сообщает **TextBlock**, какой текст должен выводиться в окне.
- ★ **HorizontalAlignment** выбирает режим горизонтального выравнивания текста по левому краю, по правому краю или по центру.
- ★ **VerticalAlignment** выбирает режим вертикального выравнивания текста по верхнему краю, по нижнему краю или по центру.
- ★ **Margin** задает смещение от краев контейнера (верх, низ, стороны).
- ★ **TextWrapping** указывает, нужно ли добавлять разрывы строк для переноса текста.

У вас свойства могут следовать в другом порядке, а свойство **Margin** может содержать другие числа, потому что они зависят от того, в какое место ячейки вы перетаскивали элемент. Все эти свойства можно изменить или сбросить в окне свойств IDE.

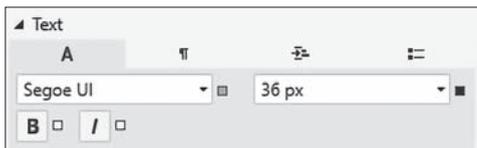
Изображения животных должны быть достаточно крупными, поэтому **раскройте раздел Text** в окне свойств **Layout** и **выберите размер шрифта 36 px**. Затем перейдите в раздел **Common** и задайте свойству **Text** значение **?**, чтобы в элементе выводился вопросительный знак.



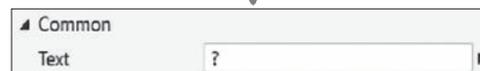
Когда вы перетаскиваете элемент с панели инструментов в ячейку, IDE добавляет в XAML тег **TextBlock** и задает номер строки, столбца и величины отступов.



Щелкните на этом квадратике и выберите команду **Reset**, чтобы сбросить поля.



Свойство **Text** (из раздела **Common**) задает текст элемента **TextBlock**.



Щелкните на поле поиска в верхней части окна свойств. Введите слово **wrap**, чтобы найти свойства с соответствующими именами. Значение свойства **TextWrapping** можно сбросить при помощи квадратика в правой части окна.



Часть
Задаваемые
Вопросы

В: Когда я сбросил значения высоты первых четырех строк, они исчезли, а потом вернулись, когда я сбросил высоту последней строки. Почему это произошло?

О: Вам кажется, что строки исчезают, потому что по умолчанию сетки WPF используют **пропорциональные размеры** строк и столбцов. Если высота последней строки была равна 74*, то при назначении первым четырем строкам высоты по умолчанию 1* размеры строки изменились таким образом, что первые четыре строки занимают 1/78 (или 1.3%) высоты строки, а последняя строка занимает 74/78 (или 94.8%) высоты, так что первые строки кажутся совсем маленькими. Как только вы возвращаете последней строке высоту по умолчанию 1*, размеры сетки автоматически изменяются так, чтобы каждая строка занимала 20% высоты.

В: Когда я назначаю окну ширину 400, в каких единицах задается это значение? 400 — это много или мало?

О: В WPF используются **аппаратно-независимые пиксели**, которые всегда занимают 1/96 дюйма. Это означает, что 96 пикселей всегда соответствуют 1 дюйму на экране без масштабирования. Но если вы возьмете линейку и измерите окно, может оказаться, что его ширина немного отличается от 400 пикселей (около 4.16 дюйма.) Дело в том, что в Windows используется полезная функциональность изменения масштаба экрана, чтобы приложения не казались слишком маленькими, если вместо монитора используется телевизор, на который вы смотрите из другого угла комнаты. Благодаря аппаратно-независимым пикселям приложения WPF хорошо смотрятся при любом масштабе.

Выходит, если я хочу, чтобы один столбец был вдвое шире других, я просто назначаю ему ширину 2*, а сетка сделает все остальное.



Такие упражнения еще не раз встретятся вам в книге. Они дают вам возможность потренироваться в написании кода. И помните: вы всегда можете заглянуть в решение!



Упражнение

Сетка содержит один элемент `TextBlock` — неплохое начало! Но для вывода всех животных понадобятся 16 элементов `TextBlock`. А вы сможете предположить, как добавить код XAML для включения идентичных элементов `TextBlock` во все ячейки первых четырех строк сетки?

Для начала просмотрите только что созданный тег XAML. Он должен выглядеть примерно так (свойства могут следовать в другом порядке, и мы добавили разрыв строки, чтобы код XAML лучше читался):

```
<TextBlock Text="?" Grid.Column="1" Grid.Row="1" FontSize="36"
    HorizontalAlignment="Center" VerticalAlignment="Center"/>
```

Ваша задача — продублировать элемент `TextBlock`, чтобы все 16 верхних ячеек сетки содержали одинаковые элементы. Чтобы выполнить это упражнение, необходимо добавить в приложение еще 15 элементов `TextBlock`. Несколько обстоятельств, о которых следует помнить:

- Нумерация строк и столбцов начинается с 0 — значения по умолчанию. Таким образом, если опустить свойство `Grid.Row` или `Grid.Column`, элемент `TextBlock` будет отображаться в левой ячейке строки или верхней ячейке столбца.
- Вы можете отредактировать пользовательский интерфейс в конструкторе или же скопировать и вставить код XAML. Опробуйте оба варианта — посмотрите, какой из них покажется вам более удобным!



Упражнение
Решение

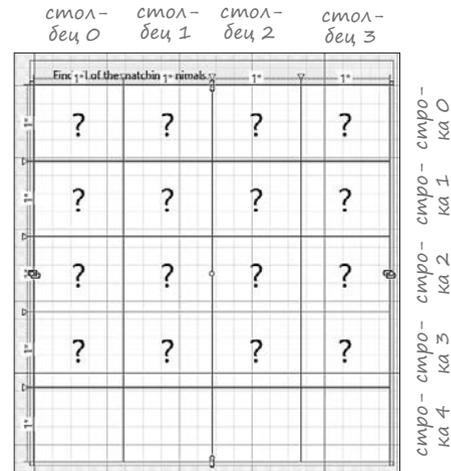
Ниже приведен код XAML для 16 элементов TextBlock с изображением животных — все они полностью идентичны, если не считать свойств Grid.Row и Grid.Column, которые размещают по одному элементу TextBlock в каждой из 16 ячеек сетки. (Ter Window остается тем же, поэтому мы его не приводим.)

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>

  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
```

Так окно выглядит в кон-
структуре Visual Studio
после добавления всех
элементов TextBlock.

А так выглядят
определения строк
и столбцов после вы-
равнивания размеров.



```
<TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center" Grid.Column="1"/>
<TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center" Grid.Column="2"/>
<TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center" Grid.Column="3"/>

<TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center" Grid.Row="1"/>
<TextBlock Text="?" FontSize="36" Grid.Row="1" Grid.Column="1"
  HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="1" Grid.Column="2"
  HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="1" Grid.Column="3"
  HorizontalAlignment="Center" VerticalAlignment="Center"/>

<TextBlock Text="?" FontSize="36" Grid.Row="2" HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="2" Grid.Column="1"
  HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="2" Grid.Column="2"
  HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="2" Grid.Column="3"
  HorizontalAlignment="Center" VerticalAlignment="Center"/>

<TextBlock Text="?" FontSize="36" Grid.Row="3" HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="3" Grid.Column="1"
  HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="3" Grid.Column="2"
  HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="3" Grid.Column="3"
  HorizontalAlignment="Center" VerticalAlignment="Center"/>
</Grid>
```

У этих четырех элементов TextBlock свойство Grid.Row равно 1, поэтому они находятся во второй строке сверху (так как номер первой строки равен 0).

Нормально, если вы включили свойства Grid.Row или Grid.Column со значением 0. Здесь они не указаны, потому что 0 является значением по умолчанию.

Вроде бы много кода, но на самом деле это одна и та же строка, повторенная 16 раз с небольшими вариациями. Каждая строка, начинающаяся с <TextBlock, содержит одинаковый набор из четырех свойств (Text, FontSize, HorizontalAlignment и VerticalAlignment). Различаются только свойства Grid.Row и Grid.Column. (Свойства могут следовать в любом порядке.)

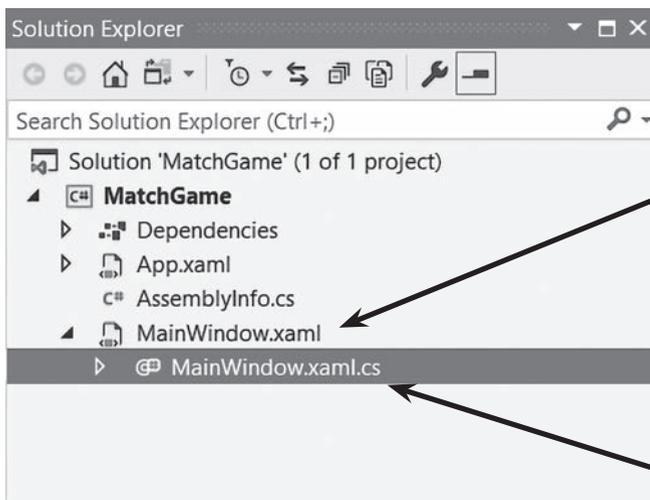
Вы здесь!



СОЗДАНИЕ ПРОЕКТА КОНСТРУИРОВАНИЕ ОКНА НАПИСАНИЕ КОДА C# ОБРАБОТКА ЦЕЛЧКОВ ДОБАВЛЕНИЕ ТАЙМЕРА

Теперь можно переходить к написанию кода игры

Конструирование главного окна завершено — хотя бы настолько, чтобы могла заработать следующая часть игры. Теперь добавим код C#.



Ранее мы редактировали код XAML в файле *MainWindow.xaml*. В нем размещаются все элементы структуры окна — этот код определяет внешний вид и макет окна.

Теперь мы начнем работать над кодом C# из файла *MainWindow.xaml.cs*. Он называется **кодом программной части окна**, потому что он объединяется с разметкой в файле XAML. Именно поэтому код хранится в файле с таким же именем, но с дополнительным расширением «.cs». Мы добавим в этот файл код C#, который определяет поведение игры, включая код добавления эмодзи в сетку, код обработки щелчков мышью и обеспечения работы таймера.



Будьте осторожны!

Когда вы вводите код C#, он должен быть **абсолютно** правильным.

Некоторые считают, что настоящим разработчиком становятся только после того, как впервые проведут несколько часов в поисках неправильно поставленной точки. Регистр символов важен: *SetUpGame* и *setUpGame* — разные имена. Лишние запятые, круглые скобки, точки с запятой и т. д. могут нарушить работоспособность вашего кода, или, что еще хуже, изменить ваш код так, что он будет успешно строиться, но работать совершенно не так, как предполагалось. Функция IDE IntelliSense помогает избежать подобных проблем... но и она не сможет сделать все за вас.

Генерирование метода для настройки игры

Итак, пользовательский интерфейс готов, теперь можно переходить к написанию кода самой игры. Для этого мы **сгенерируем метод** (сходный с методом Main, приведенным ранее) и добавим в него код.

1 Откройте файл **MainWindow.xaml.cs** в редакторе. Щелкните на кнопке с треугольником ▶ в файле MainWindow.xaml на панели Solution Explorer, а затем сделайте двойной щелчок на файле MainWindow.xaml.cs, чтобы открыть его в редакторе кода IDE. Нетрудно заметить, что в файле уже присутствует код. Visual Studio поможет добавить в него новый метод.

Если вы еще не на 100% понимаете, что такое метод, это вполне нормально.

2 Сгенерируйте **метод** с именем **SetUpGame**. Найдите в открывшемся коде следующий фрагмент:

```
public MainWindow();
{
    InitializeComponent();
}
```

Щелкните после строки `InitializeComponent();`, чтобы установить курсор непосредственно за символом `;`. Нажмите `Enter` дважды, а затем введите `SetUpGame();`.

Как только вы введете символ `;`, под SetUpGame появляется красная волнистая линия. Щелкните на слове SetUpGame — в левой части окна появляется изображение лампочки. Щелкните на нем, чтобы открыть меню **Quick Actions** и сгенерировать метод с его помощью.

```
21 public partial class MainWindow : Window
22 {
23     0 references
24     public MainWindow()
25     {
26         InitializeComponent();
27         SetUpGame();
28     }
29 }
```

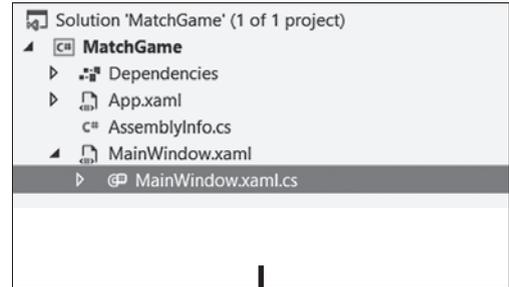
В окне Preview changes выводится описание ошибки, из-за которой появилась красная волнистая линия, а также предварительный вариант кода, предложенного действием для исправления ошибки.

Когда вы щелкаете на значке Quick Actions, на экране появляется контекстное меню со списком действий. Если какое-либо действие генерирует код, в IDE выводится предварительный вариант сгенерированного кода. Выберите действие **Generate Method**, чтобы сгенерировать новый метод с именем **SetUpGame**.

CS0103 The name 'SetUpGame' does not exist in the current context

```
private void SetUpGame()
{
    throw new NotImplementedException();
}
```

Генерируйте это!



Используйте вкладки в верхней части окна для переключения между редактором C# и конструктором XAML.





Quick Actions (Alt+Enter or Ctrl+.)

Каждый раз, когда в IDE появляется изображение лампочки, это означает, что для выбранного вами кода доступно быстрое действие, а следовательно, существует задача, которую Visual Studio может автоматизировать для вас. Чтобы просмотреть доступные быстрые действия, щелкните на лампочке или нажмите клавиши Alt+Enter или Ctrl+. (точка).

3 Попробуйте выполнить код.

Щелкните на кнопке в верхней части IDE, чтобы запустить программу (как это было сделано ранее для консольного приложения).



← Кнопка *Start Debugging* на панели инструментов в верхней части IDE запускает приложение. Также приложение можно запустить командой *Start Debugging* (F5) из меню *Debug*.

Что-то пошло не так. Вместо окна программа **выдает исключение**:

```

21 public partial class MainWindow : Window
22 {
23     0 references
24     public MainWindow()
25     {
26         InitializeComponent();
27         SetUpGame();
28     }
29     1 reference
30     private void SetUpGame()
31     {
32         throw new NotImplementedException();
33     }
34 }
35
                
```

Exception User-Unhandled

System.NotImplementedException: 'The method or operation is not implemented.'

View Details | Copy Details | Start Live Share session...

▸ Exception Settings

Может показаться, что наша программа сломана, но на самом деле произошло именно то, что должно было произойти! IDE приостанавливает вашу программу и выделяет последнюю выполненную строку кода. Присмотримся повнимательнее:

```
throw new NotImplementedException();
```

Метод, сгенерированный IDE, прямо приказывает C# выдать исключение. Обратите внимание на сообщение, выводимое с исключением: в нем говорится, что метод или операция не реализованы:

System.NotImplementedException: 'The method or operation is not implemented.'

И это вполне логично, потому что **вы сами должны реализовать метод**, сгенерированный IDE. Если вы забыли реализовать его, исключение напомним вам, что у вас осталась невыполненная работа. Если вы генерируете много методов, такое напоминание будет очень полезным!

Щелкните на кнопке Stop Debugging  панели инструментов (или выберите команду Stop Debugging (F5) в меню Debug), чтобы прервать работу программы и завершить реализацию метода SetUpGame.

Если вы запускаете приложение кнопкой IDE, кнопка Stop Debugging немедленно завершает его.

Завершение метода SetupGame

Это специальный метод, который называется конструктором. О том, как он работает, вы узнаете в главе 5.

Метод SetupGame размещается внутри метода public MainWindow(), потому что все содержимое этого метода вызывается сразу же после запуска приложения.

1 Начало добавления кода в метод SetupGame.

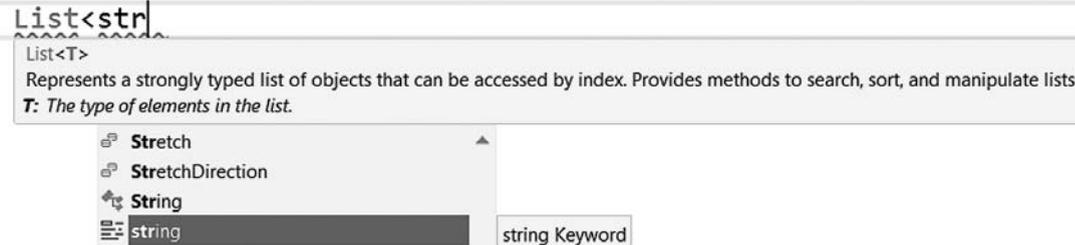
Метод SetupGame получает восемь пар эмодзи с изображением животных и случайным образом распределяет их между элементами TextBlock. Следовательно, первое, что понадобится вашему методу, — это список этих эмодзи, а IDE поможет написать для него код. Выделите команду throw, добавленную IDE, и удалите ее. Затем установите курсор в то место, где была эта команда, и введите List. IDE открывает окно IntelliSense с ключевыми словами, которые начинаются с «List»:

```
private void SetupGame()
{
    List
}
```



Выберите в списке IntelliSense строку List. Затем введите <str — на экране появляется еще одно окно IntelliSense с подходящими ключевыми словами:

```
private void SetupGame()
{
    List<str
}
```



Выберите строку string. Завершите ввод следующей строки кода, но пока не нажимайте клавишу Enter:

```
List<string> animalEmoji = new List<string>()
```

List — коллекция для хранения набора значений в определенном порядке. Коллекции рассматриваются в главах 8 и 9.

Ключевое слово «new» используется для создания списка List. О нем вы узнаете в главе 3.



Вскоре вы будете знать о методах гораздо больше.

Мы воспользовались IDE для добавления метода в приложение, но даже если вы еще не совсем понимаете, что такое метод, ничего страшного в этом нет. В следующей главе вы узнаете много нового о методах и о структуре кода C#.

2 **Добавление значений в List.**

Команда C# еще не закончена. Убедитесь в том, что курсор расположен после) в конце строки, после чего введите открывающую фигурную скобку { — IDE автоматически добавит парную закрывающую скобку, а курсор будет установлен между двумя скобками. **Нажмите Enter** — IDE добавит разрывы строк автоматически:

```
List<string> animalEmoji = new List<string>()
{
    ~
}
```

Пока панель эмодзи остается открытой, вы можете ввести слово (например, «осторус»), и оно будет замечено соответствующим эмодзи.

Используйте **панель эмодзи Windows** (нажмите клавишу с логотипом Windows+точка) или зайдите на свой любимый сайт с эмодзи (например, <https://emojipedia.org/nature>) и скопируйте отдельный символ эмодзи. Вернитесь к своему коду, введите ", вставьте символ, а за ним еще одну кавычку ", запятую, пробел, еще одну кавычку «, снова тот же символ эмодзи, последнюю кавычку " и запятую. Потом сделайте то же самое для семи других эмодзи, чтобы в итоге **в фигурных скобках были заключены восемь пар эмодзи с изображением животных**. Добавьте ; после завершающей фигурной скобки:

```
List<string> animalEmoji = new List<string>()
{
    "🐶", "🐱",
    "🐼", "🐨",
    "🐘", "🐘",
    "🐘", "🐘",
    "🐘", "🐘",
    "🐘", "🐘",
    "🐘", "🐘",
    "🐘", "🐘",
};
```

Задержите указатель мыши на точках под animalEmoji — IDE сообщит вам, что присвоенное значение нигде не используется. Предупреждение исчезнет сразу же после того, как список эмодзи будет использован далее в коде метода.



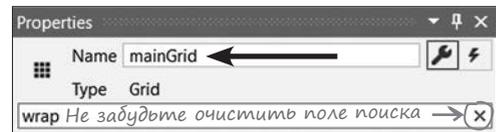
Панель эмодзи встроена в Windows 10. Чтобы вызвать ее на экран, нажмите клавишу с логотипом Windows + точка.

3 **Завершение метода.**

Теперь добавьте **остальной код** метода — будьте внимательны с точками, круглыми и фигурными скобками:

```
Random random = new Random();
foreach (TextBlock textBlock in mainGrid.Children.OfType<TextBlock>())
{
    int index = random.Next(animalEmoji.Count);
    string nextEmoji = animalEmoji[index];
    textBlock.Text = nextEmoji;
    animalEmoji.RemoveAt(index);
}
```

← Эта строка следует сразу же за закрывающей фигурной скобкой и точкой с запятой.



Красная волнистая линия под mainGrid в IDE указывает на ошибку: ваша программа не будет построена, потому что с этим именем в коде ничего не связано. **Вернитесь к редактору XAML** и щелкните на теге <Grid>, затем перейдите к окну свойств и введите mainGrid в поле Name.

Проверьте код XAML — в верхней части разметки сетки находится тег <Grid x:Name="mainGrid">. Сейчас никаких ошибок в коде быть не должно. Если они все же есть, **тщательно проверьте каждую строку** — совершить ошибку при вводе очень легко. ↴

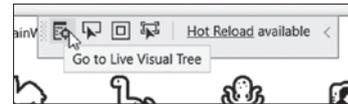
Если при запуске игры произойдет исключение, проверьте, что список animalEmoji содержит ровно 8 пар эмодзи, а в коде XAML содержатся 16 тегов <TextBlock.../>.

Запуск программы

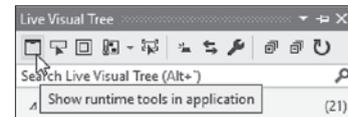
Щелкните на кнопке ▶ Start - панели инструментов, чтобы запустить программу. Открывается окно с восемью парами животных в случайных позициях:



При первом запуске программы в верхней части окна появляется панель инструментов времени выполнения:



Щелкните на первой кнопке панели, чтобы вызвать панель Live Visual Tree в IDE:



Затем щелкните на первой кнопке панели Live Visual Tree, чтобы отключить панель инструментов времени выполнения.

Во время выполнения программы IDE переходит в режим отладки: кнопка Start заменяется недоступной кнопкой Continue, а на панели инструментов появляются отладочные элементы ; эти кнопки предназначены для приостановки, остановки и перезапуска программы.

Остановите свою программу, щелкнув на кнопке со значком X в правом верхнем углу окна или на кнопке Stop (кнопка с квадратом) в отладочных элементах. Выполните программу несколько раз — животные будут каждый раз находиться в разных позициях.



Ого, игра уже сейчас неплохо выглядит!

Вы подготовили обстановку для следующей части, которую мы добавим в приложение.

В процессе построения новой игры вы не просто пишете код — вы также запускаете проект. В самом эффективном варианте реализации проект строится небольшими частями, и при каждом изменении вы запускаете проект и убеждаетесь в том, что работа движется в правильном направлении. При таком подходе вы сможете легко сменить курс, если произойдет что-то непредвиденное.

Еще одно упражнение, в котором вам придется поработать карандашом. Не жалейте времени на выполнение всех этих упражнений, потому что они помогут быстрее закрепить важные концепции C# в вашем мозгу.

начинаем программировать на C#

КТО И ЧТО ДЕЛАЕТ?

Поздравляем — вы создали работающую программу! Разумеется, программирование несколько сложнее простого копирования кода из книги. Но даже если вы никогда не писали код прежде, вас удивит, сколько всего вы уже понимаете. Соедините линией каждую команду C# в левом столбце с описанием того, что делает эта команда, в правом столбце. Мы привели решение для первой команды, чтобы вам было проще.

Команда C#

Что делает

```
List<string> animalEmoji = new List<string>()
{
    "🐶", "🐱",
    "🐼", "🐨",
    "🐘", "🐘",
    "🐘", "🐘",
    "🐘", "🐘",
    "🐘", "🐘",
    "🐘", "🐘",
    "🐘", "🐘",
    "🐘", "🐘",
};
```

Обновляет TextBlock случайным эмодзи из списка

Находит каждый элемент TextBlock в сетке и повторяет следующие команды для каждого элемента

Удаляет случайный эмодзи из списка

```
Random random = new Random();
```

```
foreach (TextBlock textBlock in
    mainGrid.Children.OfType<TextBlock>())
```

Создает список из восьми пар эмодзи

```
int index = random.Next(animalEmoji.Count);
```

Выбирает случайное число от 0 до количества эмодзи в списке и назначает ему имя «index»

```
string nextEmoji = animalEmoji[index];
```

Создает новый генератор случайных чисел

```
textBlock.Text = nextEmoji;
```

Использует случайное число с именем «index» для получения случайного эмодзи из списка

```
animalEmoji.RemoveAt(index);
```

КТО И ЧТО ДЕЛАЕТ? решение

Команда C#

```
List<string> animalEmoji = new List<string>()
{
    "🐶", "🐶",
    "🐱", "🐱",
    "🐼", "🐼",
    "🐨", "🐨",
    "🐼", "🐼",
    "🐼", "🐼",
    "🐼", "🐼",
    "🐼", "🐼",
};

Random random = new Random();

foreach (TextBlock textBlock in
    mainGrid.Children.OfType<TextBlock>())
{
    int index = random.Next(animalEmoji.Count);
    string nextEmoji = animalEmoji[index];
    textBlock.Text = nextEmoji;
    animalEmoji.RemoveAt(index);
}
```

Что делает

- Обновляет TextBlock случайным эмодзи из списка
- Находит каждый элемент TextBlock в сетке и повторяет следующие команды для каждого элемента
- Удаляет случайный эмодзи из списка
- Создает список из восьми пар эмодзи
- Выбирает случайное число от 0 до количества эмодзи в списке и назначает ему имя «index»
- Создает новый генератор случайных чисел
- Использует случайное число с именем «index» для получения случайного эмодзи из списка



Возьми в руку карандаш

Следующее упражнение поможет вам лучше понять код C#.

1. Возьмите лист бумаги и поверните его набок, чтобы он лежал в альбомной ориентации. Нарисуйте вертикальную линию в середине.
2. Запишите весь метод `SetUpGame` в левой части, оставляя свободное место между командами. (Особая точность с эмодзи не нужна.)
3. В правой части листа запишите каждый из ответов «Что делает» рядом с командой, с которой он соединен. Прочитайте обе стороны — код должен постепенно приобретать смысл.



Не уверена, нужны ли все эти упражнения. Разве не лучше просто дать мне код, который можно ввести в IDE?

Отработка навыков понимания кода повысит вашу квалификацию разработчика.

Письменные упражнения являются **обязательными**. Они предоставляют вашему мозгу новый способ усвоения информации. Однако они делают нечто еще более важное: предоставляют вам возможность *ошибаться*. Ошибки — важная часть обучения, и мы тоже допустили множество ошибок (возможно, вы даже найдете одну-две опечатки в этой книге!). Никто не пишет идеальный код с первого раза — действительно хорошие программисты всегда предполагают, что написанный сегодня код, возможно, потребуется изменить завтра. Позднее в книге вы узнаете о *рефакторинге* — приеме программирования, сутью которого становится улучшение вашего кода после того, как он будет написан.

В таких списках «ключевых моментов» приводятся краткие сводки многих идей и средств, которые были представлены в этой главе.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Visual Studio — интегрированная среда разработки (IDE) компании Microsoft, которая упрощает редактирование файлов с кодом C# и выполнение различных операций с ними.
- Консольные приложения .NET Core — кроссплатформенные приложения с текстовым вводом и выводом.
- Функция IDE IntelliSense помогает быстрее вводить код.
- WPF (или Windows Presentation Foundation) — технология, используемая для построения визуальных приложений в C#.
- Пользовательские интерфейсы WPF строятся на XAML (eXtensible Application Markup Language) — языке разметки на базе XAML, который использует теги и свойства для определения элементов управления в пользовательском интерфейсе.
- Тег Grid в XAML предоставляет структуру сетки, в которой могут содержаться другие элементы.
- Тег TextBlock в XAML добавляет элемент, который может содержать текст.
- Окно свойств IDE упрощает редактирование свойств элементов управления — например, изменение их структуры, текста или строки/столбца сетки, в котором они находятся.

Добавление нового проекта в систему управления версиями

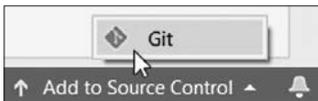
В этой книге мы будем строить много разных проектов. Только представьте, как удобно было бы иметь место, в котором их можно было бы сохранить или загружать позднее с любого устройства? А если вы допустите ошибку, разве не будет удобно вернуться к предыдущей версии вашего кода? Что ж, вам повезло! Именно эту задачу решает **система управления версиями**: она предоставляет простые средства для создания резервной копии всего кода и отслеживания всех вносимых изменений. Visual Studio позволяет легко добавлять проекты в систему управления версиями.

Git — популярная система управления версиями, и Visual Studio может публиковать ваш исходный код в любом **репозитории** Git. Мы считаем **GitHub** одним из самых удобных провайдеров Git. Для сохранения кода вам понадобится учетная запись GitHub. Если у вас еще нет учетной записи, зайдите на сайт <https://github.com> и создайте ее.

Найдите элемент **Add to Source Control** в строке состояния в нижней части IDE:



Щелкните на нем — Visual Studio предлагает добавить код в Git:



Выберите строку Git. Visual Studio запрашивает имя и адрес электронной почты. Строка состояния после этого должна выглядеть так:



Теперь ваш код находится под контролем системы управления версиями. Задержите указатель мыши над **↑ 2**:



IDE сообщает, что на вашем компьютере остаются две сохраненные версии вашего кода, которые не были сохранены в сетевом хранилище. При включении вашего проекта в систему управления версиями IDE открывает **окно Team Explorer** на одной панели с Solution Explorer. (Если вы не видите это окно, выберите его в меню View.) Окно Team Explorer предназначено для взаимодействия с системой управления версиями. С его помощью вы сможете опубликовать свой проект в **удаленном репозитории**. Когда на компьютере появляются локальные изменения, окно Team Explorer используется для отправки их в удаленный репозиторий. Для этого щелкните на кнопке **Publish to GitHub** в окне Team Explorer.



Включать проект в систему управления версиями не обязательно.

Возможно, вы работаете на компьютере офисной сети, которая не имеет доступа к GitHub (рекомендованному нами провайдеру Git). А может быть, вам просто не хочется это делать. Как бы то ни было, этот шаг можно пропустить — или же опубликовать код в приватном репозитории, если вы хотите хранить резервную копию, но не хотите, чтобы она была доступной для других.

Как только вы включите свой код в Git, строка состояния изменяется и показывает, что код проекта находится под контролем системы управления версиями. Git — очень популярная система управления версиями, и в Visual Studio включен полнофункциональный клиент Git. В папке проекта создается скрытая папка с именем `.git`, которая используется Git для отслеживания всех изменений, вносимых в код.

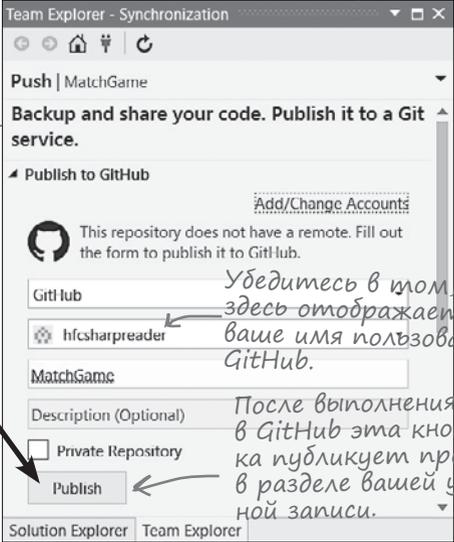


Git — система управления версиями, распространяемая с открытым кодом. Существует много сторонних сервисов, предоставляющих услуги Git (таких, как GitHub): выделение пространства для хранения кода, веб-доступ к вашим репозиториям и т. д. Чтобы больше узнать о Git, зайдите на сайт <https://git-scm.com>.

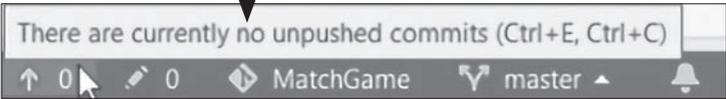
Когда вы нажимаете кнопку Publish to GitHub, Visual Studio открывает форму входа GitHub. Введите имя пользователя и пароль GitHub. (Если вы настроили двухфакторную аутентификацию, вам также будет предложено использовать ее.)



После того как IDE выполнит вход в GitHub, открывается форма Publish to GitHub. На ней можно выбрать провайдера GitHub, имя пользователя и имя проекта, а также указать, должен ли репозиторий быть приватным. Оставьте параметрам значения по умолчанию. Нажмите кнопку Publish, чтобы опубликовать проект в GitHub.

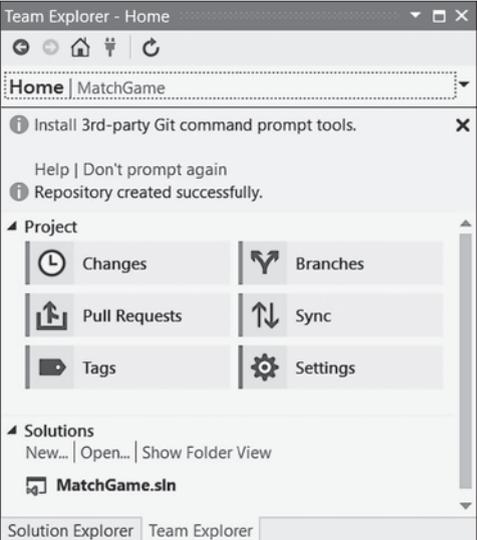


После публикации в GitHub статус Git в строке состояния обновляется; теперь строка состояния сообщает, что несохраненных изменений нет. Это означает, что ваш проект синхронизирован с репозиторием в учетной записи GitHub.



Git предоставляет мощные средства командной строки. Visual Studio может установить их для вас автоматически. Visual Studio упрощает работу с Git, и вы можете установить эти средства, но это необязательно.

После того как код будет опубликован в GitHub, вы сможете использовать Team Explorer для работы с репозиторием Git.



Чтобы просмотреть только что сохраненный код, откройте страницу <https://github.com/<ваше-имя-пользователя-github>/MatchGame>. Когда вы синхронизируете свой проект с удаленным репозиторием, обновления появляются в разделе Commits.

Часть Задаваемые Вопросы

В: XAML действительно является кодом?

О: Да, безусловно. Помните красную волнистую линию, которая появилась под mainGrid в коде C# и исчезла только при включении имени в тег Grid в XAML? Это объясняется тем, что мы уже изменяем код — после добавления имени в XAML он может использоваться вашим кодом C#.

В: Я считал, что XAML — что-то вроде разметки HTML, интерпретируемой браузером. Это не так?

О: Нет, XAML — это код, который строится параллельно с вашим кодом C#. В следующей главе вы узнаете о том, как использовать ключевое слово partial для разбиения класса на несколько файлов. Именно так происходит соединение XAML с C#: XAML определяет пользовательский интерфейс, C# определяет поведение, и они объединяются при помощи разделяемых (partial) классов.

Вот почему так важно рассматривать XAML как код, а хорошее знание XAML становится важным навыком для любого разработчика C#.

В: Я заметил МНОГО строк using в начале файла C#. Почему их так много?

О: Приложения WPF обычно используют код из разных *пространств имен* (о том, что такое пространство имен, вы узнаете в следующей главе.) Когда среда Visual Studio создает проект WPF за вас, она автоматически включает в начало файла *MainWindow.xaml.cs* директивы using для самых распространенных пространств. Собственно, вы уже применяете некоторые из них: IDE использует более светлый цвет символов для обозначения пространств имен, не используемых в коде.

В: Похоже, настольные приложения намного сложнее консольных. Они действительно работают одинаково?

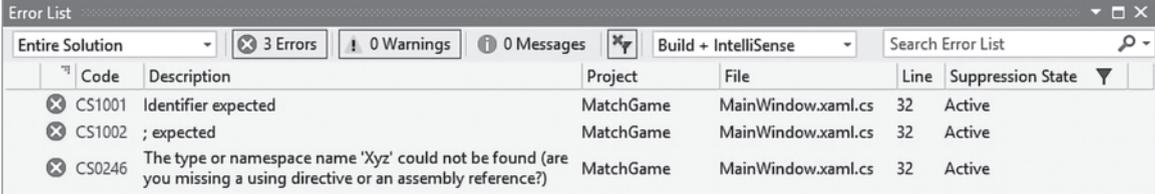
О: Да. Если разобраться, весь код C# работает по одному принципу: выполняется одна команда, потом другая, затем следующая и т. д. Настольные приложения кажутся намного более сложными из-за того, что некоторые методы вызываются только при выполнении определенных условий — скажем, при появлении окна на экране или нажатии кнопки пользователем. После того как метод будет вызван, дальше он работает по тем же правилам, что и консольное приложение.

Подсказка для IDE: список ошибок

Посмотрите на нижнюю часть редактора кода — в ней сейчас выводится сообщение . Это означает, что **построение** программы проходит успешно, это тот процесс, который используется IDE для преобразования кода в **двоичный формат**, который может выполняться вашей операционной системой. Давайте в порядке эксперимента сломаем программу.

Перейдите в первую строку кода в методе SetupGame. Нажмите клавишу Enter дважды, после чего введите в отдельной строке символы Xyz.

Снова проверьте сообщение в нижней части редактора кода — теперь в нем выводится уведомление . Если у вас в IDE не открыто окно Error List, откройте его командой Error List из меню View. В окне Error List выводятся описания трех ошибок:



Code	Description	Project	File	Line	Suppression State
CS1001	Identifier expected	MatchGame	MainWindow.xaml.cs	32	Active
CS1002	; expected	MatchGame	MainWindow.xaml.cs	32	Active
CS0246	The type or namespace name 'Xyz' could not be found (are you missing a using directive or an assembly reference?)	MatchGame	MainWindow.xaml.cs	32	Active

IDE выводит эти ошибки, потому что строка Xyz не является допустимым кодом C#, и это не позволяет IDE построить ваш код. Пока в коде остаются ошибки, он выполняться не может; удалите добавленную строку Xyz.



Следующий шаг построения игры — обработка щелчков

Итак, игра выводит животных, на которых должен щелкать пользователь. Теперь нужно добавить код, обеспечивающий работу самой игры. Игрок щелкает на животных, составляющих пару. Первое животное, на котором был сделан щелчок, исчезает. Если второе животное, на котором щелкнет игрок, совпадает с первым, то оно тоже исчезает. Если нет, первое животное появляется снова. Чтобы эта схема работала, мы добавим обработчик события, т. е. метод, вызываемый при выполнении некоторых операций (щелчков, двойных щелчков, изменения размеров окна и т. д.) в приложении.

Когда игрок щелкает на одном из животных, приложение вызывает метод с именем `TextBlock_MouseDown` для обработки щелчка. Вот что делает этот метод:



`TextBlock_MouseDown() {`

```

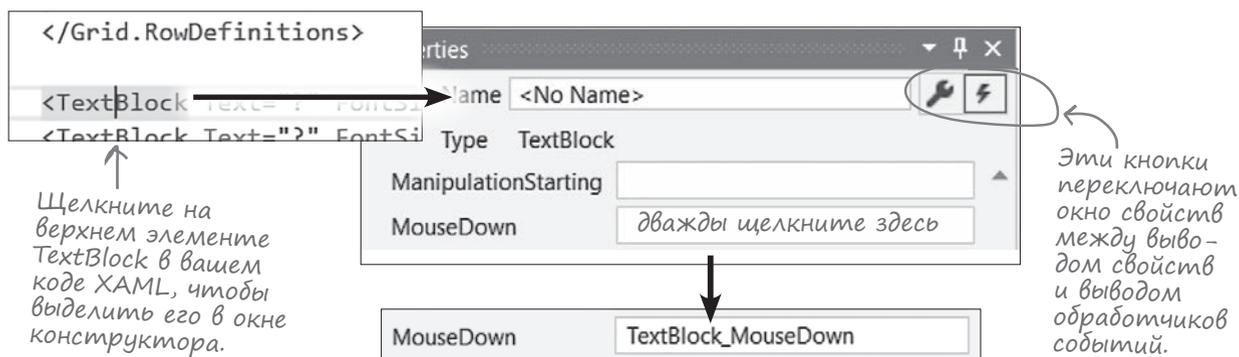
/* Если щелчок сделан на первом
 * животном в паре, сохранить
 * информацию о том, на каком
 * элементе TextBlock щелкнул
 * пользователь, и убрать животное
 * с экрана. Если это второе
 * животное в паре, либо убрать
 * его с экрана (если животные
 * составляют пару), либо вернуть
 * на экран первое животное (если
 * животные разные).
 */
}
    
```

Это комментарий. Весь текст между `/*` и `*/` игнорируется в C#. Мы добавили комментарий, чтобы объяснить, что должен делать метод `TextBlock_MouseDown`, а также чтобы показать, как выглядят комментарии.

Реакция TextBlock на щелчку

Наш метод `SetUpGame` настраивает элементы `TextBlock` так, чтобы в них выводились эмодзи с изображением животных. Этот пример показывает, как ваш код может изменять элементы в приложении. А теперь необходимо написать код, который идет в другом направлении — ваши элементы должны обращаться с вызовами к вашему коду, и IDE поможет вам в этом.

Вернитесь к редактору XAML и щелкните на первом теге `TextBlock` — IDE выделит этот тег в дизайнера, чтобы вы могли отредактировать его свойства. Затем перейдите в окно свойств и щелкните на кнопке **Event Handlers** (⚡). **Обработчиком события** называется метод, который вызывается при возникновении конкретного события. К числу таких событий относятся нажатия клавиш, перетаскивание мышью, изменение размеров окна и, конечно, перемещения указателя мыши и щелчки. Прокрутите окно свойств и просмотрите имена различных событий, для которых к `TextBlock` можно добавлять обработчики событий. **Сделайте двойной щелчок на поле справа от события `MouseDown`.**



IDE автоматически заполнила поле `MouseDown` именем метода `TextBlock_MouseDown`, а в коде XAML для `TextBlock` появляется свойство `MouseDown`:

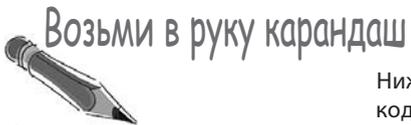
```
<TextBlock Text="?" FontSize="36" HorizontalAlignment="Center"
  VerticalAlignment="Center" MouseDown="TextBlock_MouseDown"/>
```

Возможно, вы этого не заметили, потому что среда IDE также **добавила новый метод** в код программной части (код, связанный с XAML) и немедленно переключилась на редактор C#, чтобы его вывести. Вы всегда можете вернуться обратно из редактора XAML; для этого щелкните правой кнопкой мыши на методе `TextBlock_MouseDown` в редакторе XAML и выберите команду `View Code`. Добавленный метод выглядит так:

```
private void TextBlock_MouseDown(object sender, MouseButtonEventArgs e)
{
  ...
}
```

Каждый раз, когда игрок щелкает на элементе `TextBlock`, приложение автоматически вызывает метод `TextBlock_MouseDown`. Таким образом, остается лишь добавить в него код. Затем остается связать все остальные элементы `TextBlock` с этим методом, чтобы они тоже вызывали его.

Обработчик событий — метод, который вызывается вашим приложением в ответ на такие события, как щелчок кнопкой мыши, нажатие клавиши, изменение размеров окна и т. д.



Ниже приведен код метода `TextBlock_MouseDown`. Прежде чем добавлять этот код в программу, прочитайте его и попробуйте понять, что он делает. Не огорчайтесь, если какие-то предположения оказались ошибочными! Наша цель — постепенно научить ваш мозг читать код C# и понимать его смысл.

```
TextBlock lastTextBlockClicked;
bool findingMatch = false;

private void TextBlock_MouseDown(object sender, MouseButtonEventArgs e)
{
    TextBlock textBlock = sender as TextBlock;
    if (findingMatch == false)
    {
        textBlock.Visibility = Visibility.Hidden;
        lastTextBlockClicked = textBlock;
        findingMatch = true;
    }
    else if (textBlock.Text == lastTextBlockClicked.Text)
    {
        textBlock.Visibility = Visibility.Hidden;
        findingMatch = false;
    }
    else
    {
        lastTextBlockClicked.Visibility = Visibility.Visible;
        findingMatch = false;
    }
}
```

1. Что делает *findingMatch*?

2. Что делает блок кода, начинающийся с *if (findingMatch == false)*?

3. Что делает блок кода, начинающийся с *else if (textBlock.Text == lastTextBlockClicked.Text)*?

4. Что делает блок кода, начинающийся с *else*?



Возьми в руку карандаш Решение

Ниже приведен код метода `TextBlock_MouseDown`. Прежде чем добавлять этот код в программу, прочитайте его и попробуйте понять, что он делает. Не огорчайтесь, если какие-то предположения оказались ошибочными! Наша цель — постепенно научить ваш мозг читать код C# и понимать его смысл.

```
TextBlock lastTextBlockClicked;
bool findingMatch = false;

private void TextBlock_MouseDown(object sender, MouseButtonEventArgs e)
{
    TextBlock textBlock = sender as TextBlock;
    if (findingMatch == false)
    {
        textBlock.Visibility = Visibility.Hidden;
        lastTextBlockClicked = textBlock;
        findingMatch = true;
    }
    else if (textBlock.Text == lastTextBlockClicked.Text)
    {
        textBlock.Visibility = Visibility.Hidden;
        findingMatch = false;
    }
    else
    {
        lastTextBlockClicked.Visibility = Visibility.Visible;
        findingMatch = false;
    }
}
```

А вот что делает весь код метода `TextBlock_MouseDown`. Чтение кода на новом языке программирования напоминает чтение нотной записи — это навык, который развивается тренировкой. Чем больше вы будете этим заниматься, тем лучше у вас будет получаться.

1. Что делает `findingMatch`?

Этот признак определяет, щелкнул ли игрок на первом животном в паре, и теперь пытается найти для него пару.

2. Что делает блок кода, начинающийся с `if (findingMatch == false)`?

Игрок только что щелкнул на первом животном в паре, поэтому это животное становится невидимым, а соответствующий элемент `TextBlock` сохраняется на случай, если его придется делать видимым снова.

3. Что делает блок кода, начинающийся с `else if (textBlock.Text == lastTextBlockClicked.Text)`?

Игрок нашел пару! Второе животное в паре становится невидимым (а при дальнейших щелчках на нем ничего не происходит), а признак `findingMatch` сбрасывается, чтобы следующее животное, на котором щелкнет игрок, снова считалось первым в паре.

4. Что делает блок кода, начинающийся с `else`?

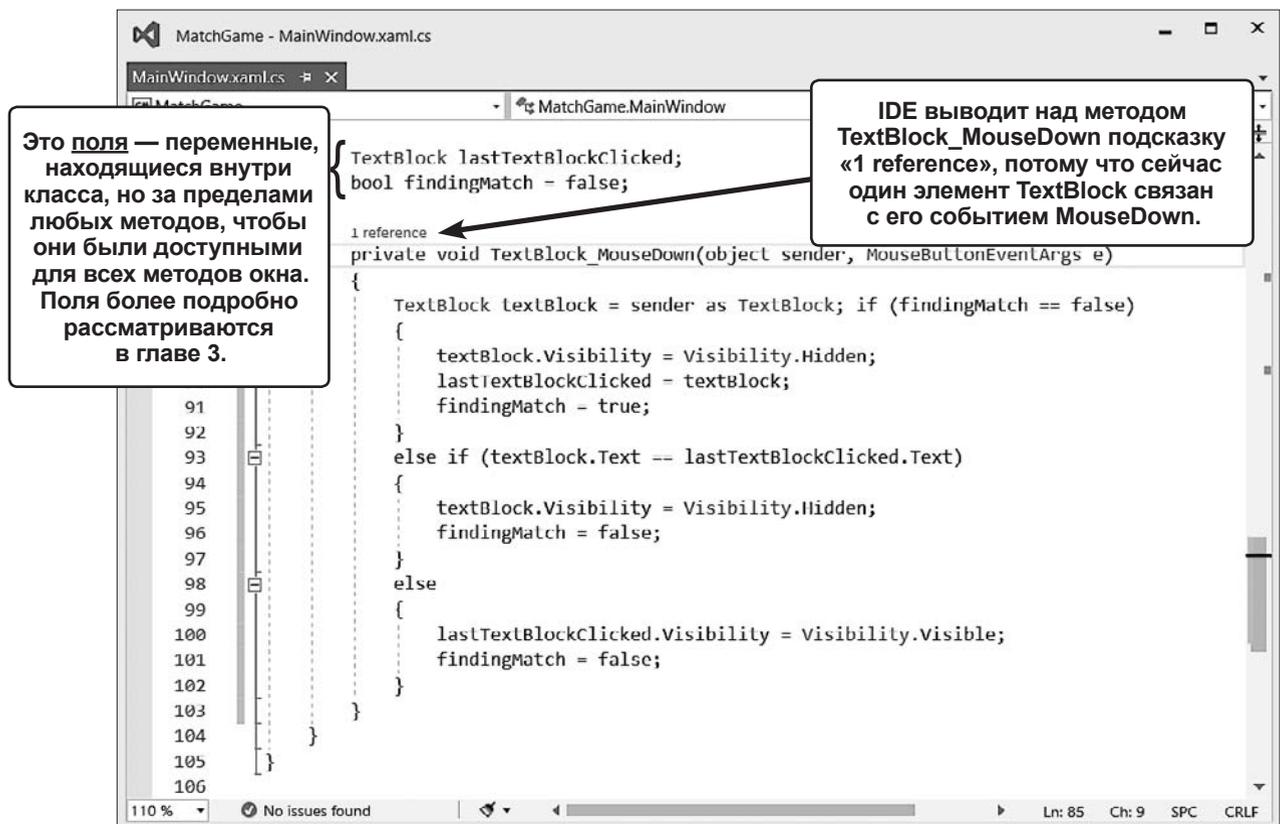
Игрок щелкнул на животном, которое не совпадает с первым, поэтому первое выбранное животное снова становится видимым, а признак `findingMatch` сбрасывается.

Добавление кода TextBlock_MouseDown

Теперь вы примерно представляете, как работает код TextBlock_MouseDown, и мы перейдем к добавлению его в программу. Вот что мы сделаем дальше:

1. Вставьте первые две строки с `lastTextBlockClicked` и `findingMatch` перед первой строкой метода `TextBlock_MouseDown`, добавленного IDE. Проследите за тем, чтобы они были вставлены между закрывающей фигурной скобкой в конце `SetUpGame` и новым кодом, добавленным IDE.
2. **Заполните код** `TextBlock_MouseDown`. Будьте внимательны со знаками равенства: `=` принципиально отличается от `==` (об этом вы узнаете в следующей главе).

А вот как это выглядит в IDE:



Вызов обработчика события `MouseDown` остальными элементами `TextBlock`

На данный момент только первый элемент `TextBlock` связан со своим событием `MouseDown`. Теперь необходимо проделать то же самое для остальных 15 элементов `TextBlock`. Для этого можно выбрать каждый элемент в визуальном конструкторе и ввести `TextBlock_MouseDown` в поле рядом с `MouseDown`. Вы уже знаете, как добавить свойство в код XAML; давайте воспользуемся этим приемом.

1 Выделите остальные 15 элементов `TextBlock` в редакторе XAML.

Перейдите в редактор XAML, щелкните слева от второго тега `TextBlock` и протащите указатель мыши по всем остальным элементам `TextBlock` до закрывающего тега `</Grid>`. В результате у вас должны быть выделены последние 15 элементов `TextBlock` (но не первый).

2 Воспользуйтесь заменой для добавления обработчиков событий `MouseDown`.

Выберите команду **Find and Replace >> Quick Replace** в меню Edit. Введите искомый текст `/>` и замените его текстом `MouseDown="TextBlock_MouseDown"/>` — проследите за тем, чтобы перед `MouseDown` находился пробел, а в разделе диапазона поиска было выбрано значение **Selection**, чтобы свойство добавлялось только к выделенным элементам `TextBlock`.



← Перед `MouseDown` ставится пробел, чтобы избежать случайного слияния с предыдущим свойством.

3 Выполните замену во всех 15 выделенных элементах `TextBlock`.

Щелкните на кнопке **Replace All** (🔄), чтобы добавить свойство `MouseDown` к элементам `TextBlock`, — IDE должна сообщить о выполнении 15 замен. Внимательно просмотрите код XAML и убедитесь в том, что каждый элемент содержит свойство `MouseDown`, полностью совпадающее с одноименным свойством первого элемента `TextBlock`.

Проверьте, что над методом теперь выводится подсказка «16 references» (выберите команду **Build Solution** из меню **Build**, чтобы обновить ее). Если в сообщении будут упомянуты 17 ссылок, вероятно, вы случайно присоединили обработчик события к `Grid`. Этого быть не должно — в противном случае при щелчке на животном будет происходить исключение.

Запустите программу и попробуйте щелкать на парах животных, чтобы они исчезали. Первое животное исчезает в любом случае. Если потом вы щелкнете на совпадающем животном, то оно тоже исчезнет. Если же второй щелчок будет сделан на неподходящем животном, то первое животное снова появится на экране. Когда все животные будут скрыты, перезапустите или закройте программу.

Когда вы встречаете врезку «Мозговой шторм», как следует поразмыслите над заданным вопросом.



Ваш проект достиг важной контрольной точки! Возможно, игра еще не закончена, но она уже работает, поэтому сейчас будет уместно ненадолго задержаться и подумать над тем, как бы улучшить ее. Какие изменения вы бы предложили для того, чтобы сделать программу более интересной?



Добавление таймера

Наша игра станет более интересной, если игрок сможет попытаться побить свой рекорд. Добавим **таймер**, который срабатывает с фиксированным интервалом и многократно вызывает метод.



Сделаем игру чуть более азартной! В нижней части окна выводится время, прошедшее с момента запуска игры. Показания таймера постоянно увеличиваются, а останавливается таймер только после нахождения последней пары.



Таймер срабатывает после истечения заданного интервала, а назначенный метод вызывается снова и снова. Мы используем таймер, который запускается вместе с запуском игры и перестает работать после нахождения последнего животного.

Добавление таймера в код игры



- Сначала найдите ключевое слово `namespace` в верхней части `MainWindow.xaml.cs` и добавьте прямо под ним строку `using System.Windows.Threading;`:

```
namespace MatchGame
{
    using System.Windows.Threading;
```

- Найдите строку `public partial class MainWindow` и **добавьте следующий код** сразу же за открывающей фигурной скобкой {:

```
public partial class MainWindow : Window
{
    DispatcherTimer timer = new DispatcherTimer();
    int tenthsOfSecondsElapsed;
    int matchesFound;
```

Добавьте эти три строки кода, чтобы создать новый таймер и добавить два поля для отслеживания прошедшего времени и количества найденных совпадений.

- Таймеру необходимо сообщить, с какой частотой он должен срабатывать и какой метод должен вызываться. Щелкните в начале строки, в которой вызывается метод `SetUpGame`, чтобы перевести курсор в эту позицию. Нажмите клавишу `Enter` и введите две строки на следующем снимке экрана, начинающиеся с `timer.`, — как только вы введете `+=`, IDE выведет сообщение:

```
0 references
public MainWindow()
{
    InitializeComponent();

    timer.Interval = TimeSpan.FromSeconds(.1);
    timer.Tick +=
    SetUpGame();
}
```

Затем добавьте эти две строки. Начните с ввода второй строки: `"timer.Tick+="`
Как только вы введете знак равенства, IDE выведет сообщение "Press TAB to insert".

- Нажмите клавишу `Tab`. IDE завершает строку кода и добавляет метод `Timer_Tick`:

```
0 references
public MainWindow()
{
    InitializeComponent();

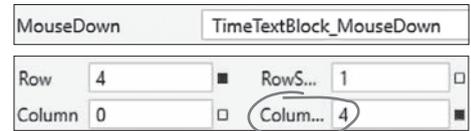
    timer.Interval = TimeSpan.FromSeconds(.1);
    timer.Tick += Timer_Tick;
    SetUpGame();
}

1 reference
private void Timer_Tick(object sender, EventArgs e)
{
    throw new NotImplementedException();
}
```

Когда вы нажимаете клавишу `Tab`, IDE автоматически вставляет метод, который должен вызываться таймером.

5 Метод `Timer_Tick` обновляет элемент `TextBlock`, распространяющийся на всю нижнюю строку сетки. Чтобы создать этот элемент, выполните следующие действия:

- ★ Перетащите элемент **TextBlock** в левый нижний квадрат.
- ★ В поле `Name` в верхней части окна свойств введите имя `timeTextBlock`.
- ★ Сбросьте отступы, выровняйте текст **по центру** (`Center`) ячейки, задайте свойству **FontSize** значение 36px, а свойству **Text** — значение «Elapsed time» (так же, как это делалось для других элементов управления).
- ★ Найдите свойство **ColumnSpan** и задайте ему значение 4.
- ★ Добавьте **обработчик события MouseDown** с именем `TimeTextBlock_MouseDown`.



Свойство `ColumnSpan` находится разделе `Layout` окна свойств. Используйте кнопки в верхней части окна для переключения между выводом свойств и событий.

Код XAML должен выглядеть так (внимательно сравните его с кодом в IDE):

```
<TextBlock x:Name="timeTextBlock" Text="Elapsed time" FontSize="36"
    HorizontalAlignment="Center" VerticalAlignment="Center"
    Grid.Row="4" Grid.ColumnSpan="4" MouseDown="TimeTextBlock_MouseDown" />
```

6 Когда вы добавляете обработчик события `MouseDown`, Visual Studio создает в коде программной части метод с именем `TimeTextBlock_MouseDown`, похожий на другие элементы `TextBlock`. Добавьте в него следующий код:

```
private void TimeTextBlock_MouseDown(object sender, MouseButtonEventArgs e)
{
    if (matchesFound == 8)
    {
        SetupGame();
    }
}
```

Сбрасывает игру, если были найдены все 8 пар (в противном случае ничего не делает, потому что игра еще продолжается).

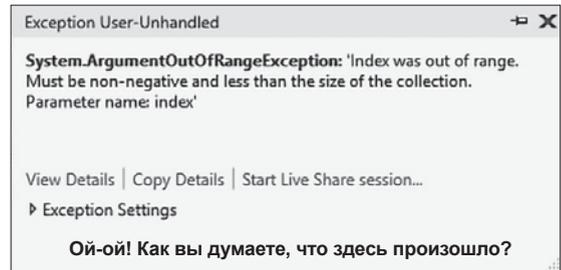
7 Теперь у вас есть все необходимое для завершения метода `Timer_Tick`, который обновляет новый элемент `TextBlock` истекшим временем и останавливает таймер после того, как игрок найдет все совпадения:

```
private void Timer_Tick(object sender, EventArgs e)
{
    tenthsOfSecondsElapsed++;
    timeTextBlock.Text = (tenthsOfSecondsElapsed / 10F).ToString("0.0s");
    if (matchesFound == 8)
    {
        timer.Stop();
        timeTextBlock.Text = timeTextBlock.Text + " - Play again?";
    }
}
```

И все же здесь что-то не так. Запустите программу... стоп! Происходит **исключение**.

Мы исправим эту ошибку, но сначала присмотритесь повнимательнее к сообщению об ошибке и выделенной строке в IDE.

А вы догадаетесь, из-за чего возникла ошибка?



Диагностика ошибок в отладчике

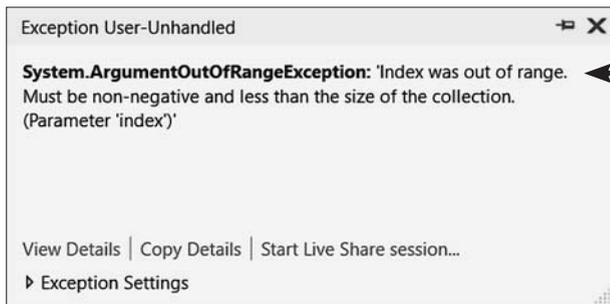
У каждой ошибки есть объяснение — в программе ничего не происходит без причин, но не каждую ошибку легко обнаружить.

Понимание ошибки — первый шаг к ее исправлению. К счастью, для этого существует превосходный инструмент — отладчик Visual Studio.

Принципы отладки



- 1 **Перезапустите свою игру несколько раз.** Первое, на что следует обратить внимание, — ваша программа всегда выдает исключение одного типа с одинаковым сообщением:



Исключения используются в C# для передачи информации о том, что во время выполнения кода что-то пошло не так. Каждое исключение относится к определенному типу: это конкретное исключение имеет тип `ArgumentOutOfRangeException`. Исключения также сопровождаются полезными сообщениями, которые помогут вам понять, что же именно произошло в программе. В сообщении нашего исключения сказано: «Индекс вышел за пределы диапазона». Воспользуемся этой информацией для поиска ошибки.

↑
Когда в программе происходит исключение, часто это становится хорошей новостью — в программе обнаружилась ошибка, и теперь вы можете заняться ее исправлением.

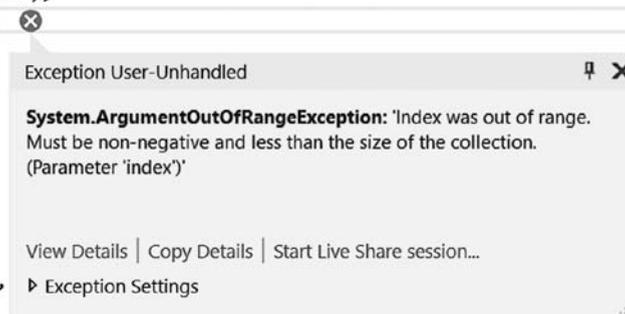
Если убрать окно исключения, вы увидите, что выполнение всегда прерывается в одной строке:

Строка, в которой выдается исключение.

```
foreach (TextBlock textBlock in mainGrid.Children.OfType<TextBlock>())  
{  
    int index = random.Next(animalEmoji.Count);  
    string nextEmoji = animalEmoji[index];  
    textBlock.Text = nextEmoji;  
    animalEmoji.RemoveAt(index);  
}
```

```
TextBlock lastTextBlockClicked;  
bool findingMatch = false;
```

```
16 references  
private void TextBlock_MouseDown(object sender,  
{
```



Это исключение является **воспроизводимым**: вы можете стабильно заставить вашу программу выдавать одно и то же исключение, а также достаточно хорошо представляете, где кроется источник проблемы.



Анатомия отладчика

Когда ваше приложение приостанавливается в отладчике (это называется прерыванием), на панели инструментов появляются элементы отладки. В книге вы еще не раз потренируетесь в их использовании, поэтому запоминать их назначение сейчас не обязательно. Пока просто прочитайте описание и наведите указатель мыши на каждый элемент, чтобы увидеть название и эквивалентное сочетание клавиш.

Кнопка Break All приостанавливает приложение. Если приложение уже приостановлено, кнопка недоступна.

Кнопка Restart перезапускает приложение. Фактически приложение останавливается и запускается заново.

Кнопка Step Into выполняет следующую команду. Если эта команда является вызовом метода, то выполняется только первая команда в этом методе.

Кнопка Step Over также выполняет следующую команду, но если эта команда является вызовом метода, то будет выполнен весь метод в целом.



Эта кнопка продолжает выполнение приложения. Если вы нажмете ее сейчас, то снова будет выдано то же исключение.

Кнопка Stop Debugging уже использовалась для приостановки приложения.

Кнопка Show Next Statement переводит курсор к следующей команде, которая должна быть выполнена в программе.

Кнопка Step Out завершает выполнение текущего метода и прерывает программу в строке, следующей за той, из которой он был вызван.

2 Добавьте точку прерывания в строке, в которой выдается исключение.

Снова запустите программу, чтобы она была прервана при выдаче исключения. Прежде чем останавливать ее, выберите команду **Toggle Breakpoint (F9)** из меню Debug. Как только вы это сделаете, строка будет выделена красным цветом, а слева от нее появится красная точка. Теперь **снова остановите приложение** — выделение и точка останутся на своем месте:

```

67 | | | int index = random.Next(animalEmoji.Count);
68 | ● | | string nextEmoji = animalEmoji[index];
69 | | | textBlock.Text = nextEmoji;
    
```

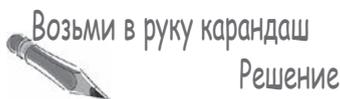
Вы только что установили в строке точку прерывания. Ваша программа будет останавливаться каждый раз, когда она будет достигать этой строки. Убедитесь в этом: снова запустите свое приложение. Программа остановится в этой строке, но на этот раз **исключение не выдается**. Нажмите кнопку Continue. Программа снова останавливается в той же строке. Нажмите Continue еще раз. Программа снова останавливается. Продолжайте, пока не появится исключение. Теперь остановите приложение.



Возьми в руку карандаш

Снова запустите приложение, но на этот раз наблюдайте за происходящим более внимательно. Ответьте на следующие вопросы.

1. Сколько раз останавливалось приложение перед исключением? _____
2. Во время отладки приложения появляется окно Locals. Как вы думаете, что оно делает? (Если окно Locals не появляется на экране, выберите в меню команду **Debug >> Windows >> Locals (Ctrl D, L)** .



Ваше приложение останавливалось 17 раз. После 17-го раза было выдано исключение.

В окне Locals выводятся текущие значения переменных и полей. В нем вы можете отслеживать изменения этих значений во время выполнения программы.

3 Соберите факты, которые помогут вам разобраться в причинах проблемы.

Вы заметили что-нибудь интересное в окне Locals при запуске приложения? Перезапустите его и пристально следите за переменной animalEmoji. Когда ваше приложение прервется в первый раз, в окне Locals должна выводиться следующая информация:



Нажмите кнопку Continue. Похоже, значение Count уменьшается на 1, с 16 до 15:



Приложение добавляет случайные эмодзи из списка animalEmoji в элементы TextBlock, а затем удаляет их из списка, так что значение Count должно каждый раз уменьшаться на 1. Все идет замечательно, пока список animalEmoji не останется пустым (так что Count содержит 0); в этот момент происходит исключение. Первый факт найден! Другой факт — все это происходит в **цикле foreach**. И последний факт заключается в том, что все началось *после добавления нового элемента TextBlock в окно*.

Пришло время побывать в роли Шерлока Холмса. А вы сможете выследить, что же становится причиной исключения?

Разновидность цикла, выполняемого для каждого элемента в коллекции.

Циклы предназначены для многократного выполнения блока кода. В нашем коде используется **цикл foreach**, или особый вид цикла, который выполняет один и тот же код для каждого элемента в коллекции (такой, как список animalEmoji). Пример использования цикла foreach со списком чисел:

```
List<int> numbers = new List<int>() { 2, 5, 9, 11 };
foreach (int aNumber in numbers)
{
    Console.WriteLine("The number is " + aNumber);
}
```

Цикл foreach выполняет команду Console.WriteLine для каждого числа в списке.

Этот цикл foreach создает новую переменную с именем aNumber. Затем он последовательно перебирает элементы списка number и выполняет Console.WriteLine для каждого элемента, присваивая aNumber очередное значение из списка List:

```
The number is 2
The number is 5
The number is 9
The number is 11
```

Цикл foreach снова и снова выполняет один код для каждого элемента коллекции, при этом переменной каждый раз присваивается следующий элемент. Таким образом, в данном случае переменной aNumber присваивается следующее число из списка, после чего эта переменная используется для вывода строки текста.

Здесь мы представляем новую концепцию — но только в общих чертах, чтобы вы хотя бы примерно понимали, как работает код. Циклы будут намного подробнее рассмотрены в главе 2. Затем в главе 3 мы вернемся к циклам foreach и напишем цикл, который имеет много общего с приведенным выше циклом. Даже если сейчас вам кажется, что мы движемся слишком быстро, к моменту возвращения к примеру в главе 3 все станет намного более понятным. По собственному опыту мы знаем, что повторное чтение кода, когда у вас появился более широкий контекст, сильно помогает закрепить новые знания в мозгу... так что не огорчайтесь, если сейчас что-то кажется слегка туманным.



За сценой



По следу

4 Поиск фактической причины ошибки.

Ошибка в программе происходит из-за того, что она пытается получить следующий эмодзи из списка `animalEmoji`, но список пуст. Из-за этого и происходит исключение `ArgumentOutOfRangeException`. Из-за чего же кончились эмодзи?

До внесения последнего изменения программа работала. Затем мы добавили `TextBlock...` и программа работать перестала. Ошибка возникает в цикле, перебирающем все элементы `TextBlock`. Наводит на размышления... очень, очень интересно.

Итак, когда вы запускаете свое приложение, оно *прерывается в этой строке для каждого элемента `TextBlock` в окне*. Для первых 16 элементов `TextBlock` все проходит нормально, потому что в коллекции содержится достаточно эмодзи:

```
foreach (TextBlock textBlock in mainGrid.Children.OfType<TextBlock>())
{
    int index = random.Next(animalEmoji.Count);
    string nextEmoji = animalEmoji[index];
    textBlock.Text = nextEmoji;
    animalEmoji.RemoveAt(index);
}
```

Отладчик выделяет команду, которая должна быть выполнена. Так выглядит программа непосредственно перед тем, как в ней будет выдано исключение.

Но после появления нового элемента `TextBlock` в нижней части окна прерывание происходит в 17-й раз, а поскольку коллекция `animalEmoji` содержала всего 16 эмодзи, она пуста:



Итак, перед внесением изменения у вас было 16 элементов `TextBlock` и список из 16 эмодзи; на каждый элемент `TextBlock` приходилось по одному эмодзи. Теперь в коллекции 17 `TextBlock`, а эмодзи только 16, поэтому программа не находит эмодзи для добавления в окно... и выдает исключение.

5 Исправление ошибки.

Так как исключение выдается из-за того, что в программе кончаются эмодзи в цикле, перебирающем элементы `TextBlock`, ошибку можно исправить, пропуская в цикле последний добавленный элемент `TextBlock`. Для этого можно проверить имя `TextBlock` и пропустить элемент для вывода времени. Удалите точку прерывания, повторно переключив ее состояние или выбрав команду **Delete All Breakpoints (Ctrl+Shift+F9)** из меню `Debug`.

```
foreach (TextBlock textBlock in mainGrid.Children.OfType<TextBlock>())
{
    if (textBlock.Name != "timeTextBlock")
    {
        textBlock.Visibility = Visibility.Visible;
        int index = random.Next(animalEmoji.Count);
        string nextEmoji = animalEmoji[index];
        textBlock.Text = nextEmoji;
        animalEmoji.RemoveAt(index);
    }
}
```

Добавьте этот код, чтобы исправить ошибку.

Добавьте эту команду `if` в цикл `foreach`, чтобы она пропускала элемент `TextBlock` с именем `timeTextBlock`.

Это не единственный способ исправления ошибки. Одна из истин, которые вы узнаете после того, как напишете много программ, — что у любой задачи существует много, ОЧЕНЬ много решений... и эта ошибка не является исключением (простите за каламбур).

Добавьте оставшийся код и завершите построение игры

Осталось решить еще одну проблему. Метод `TimeTextBlock_MouseDown` проверяет поле `matchesFound`, но это поле нигде не инициализируется. Добавьте следующие три строки в метод `SetUpGame` непосредственно после закрывающей фигурной скобки цикла `foreach`:

```

        animalEmoji.RemoveAt(index);
    }
}

timer.Start();
tenthsOfSecondsElapsed = 0;
matchesFound = 0;
}

```

Добавьте эти три строки в конец метода `SetUpGame`, чтобы запустить таймер и сбросить содержимое полей.

Затем добавьте следующую команду в средний блок `if/else` в `TextBlock_MouseDown`:

```

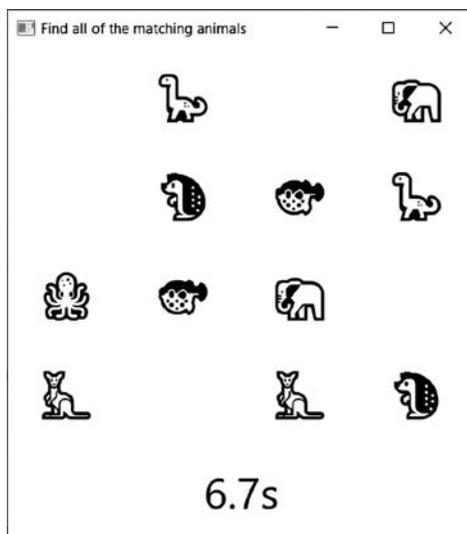
else if (textBlock.Text == lastTextBlockClicked.Text)
{
    matchesFound++;
    textBlock.Visibility = Visibility.Hidden;
    findingMatch = false;
}

```

Добавьте эту строку, чтобы значение `matchesFound` увеличивалось с каждой успешно найденной парой.

Теперь в игре работает таймер, который останавливается после того, как игрок завершит поиск пар, а после завершения игры вы можете щелкнуть на нем, чтобы сыграть снова. **Вы построили свою первую игру на C#. Поздравляем!**

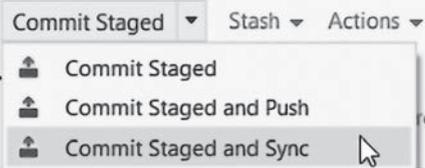
Теперь в вашей игре работает таймер, который отсчитывает время, необходимое игроку для нахождения всех пар. Сможете ли вы побить свой рекорд?

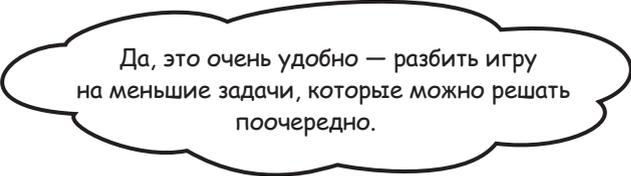
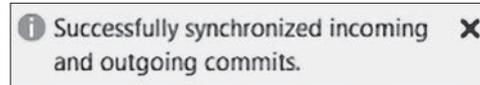
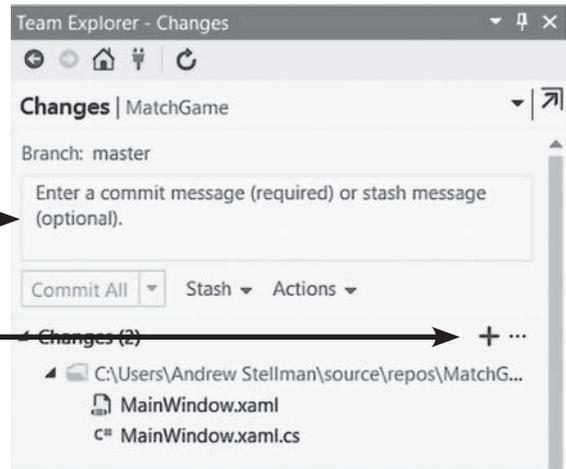


Чтобы просмотреть и загрузить полный код этого проекта, а также всех остальных проектов в книге, перейдите по адресу <https://github.com/head-first-csharp/fourth-edition/>

Обновление кода в системе управления версиями

Итак, ваша игра успешно работает. Сейчас самый подходящий момент для **сохранения изменений в Git**, и в Visual Studio это делается очень просто. От вас потребуется лишь *проиндексировать* изменения, ввести сообщение о сохранении, а затем синхронизировать проект с удаленным репозиторием.

- 1 Введите сообщение с кратким описанием изменений. 
- 2 Нажмите кнопку +, чтобы **проиндексировать** файлы, — тем самым вы сообщаете Git, что файлы готовы к сохранению. Если вы внесете изменения в файлы после того, как они были проиндексированы, в удаленном репозитории будут сохранены только проиндексированные изменения. 
- 3 Выберите команду **Commit Staged and Sync** из раскрывающегося списка (он находится прямо под полем сообщения о сохранении). Синхронизация может занять несколько секунд, после чего в окне Team Explorer появится сообщение об успехе: 



Сохранять ваш код в репозитории Git необязательно — но это определенно стоит делать!



Любой крупный проект всегда рекомендуется разбивать на меньшие части.

Один из самых полезных навыков программирования — умение взглянуть на большую и сложную задачу и разбить ее на ряд меньших, легко решаемых задач.

В самом начале большого проекта легко впасть в уныние: «Ого, да это бесконечная работа!» Но если вы сможете выделить меньшую подзадачу и начнете трудиться над ней, это станет отправной точкой для работы. А когда эта часть будет завершена, можно перейти к следующей меньшей части, потом к следующей, и т. д. Во время построения каждой части вы будете все больше узнавать о проекте в целом.

Еще лучше, если...

Игра получилась вполне достойной! Но любую игру — да, собственно, практически любую программу — можно усовершенствовать. Несколько предложений, которые, как нам кажется, могли бы улучшить нашу игру:

- ★ Добавьте больше видов животных, чтобы в игре не использовались одни и те же изображения.
- ★ Следите за лучшим временем игрока, чтобы он мог попытаться побить свой рекорд.
- ★ Реализуйте обратный отсчет времени, чтобы время игрока было ограничено.

МНТ Возьми в руку карандаш

А сможете ли вы предложить собственные улучшения для игры? Это очень полезное упражнение — подумайте несколько минут и запишите не менее трех улучшений для игры по поиску пар.

Мы абсолютно серьезно — не жалеете времени и сделайте это. Когда вы делаете шаг назад и размышляете о только что завершенном проекте, это отлично помогает закрепить в мозгу только что усвоенный материал.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Visual Studio отслеживает количество ссылок на метод в коде C# или XAML.
- **Обработчик события** представляет собой метод, который вызывается вашим приложением при возникновении некоторого события (щелчка кнопкой мыши, нажатия клавиши, изменения размеров окна и т. д.).
- IDE упрощает **добавление** методов обработчиков событий и **управление** ими.
- В окне **Error List** в IDE выводятся описания ошибок, которые препятствуют построению вашего приложения.
- **Таймеры** многократно выполняют обработчик события Tick с заданным интервалом.
- **foreach** — разновидность циклов для перебора коллекций элементов.
- Когда в вашей программе происходит исключение, соберите информацию и постарайтесь определить, что является его причиной.
- **Воспроизводимые** исключения проще устранять.
- Visual Studio упрощает использование систем управления версиями для резервного копирования кода и контроля вносимых вами изменений.
- Вы можете сохранять свой код в удаленном **репозитории Git**. Мы создали репозиторий с исходным кодом всех проектов в книге на GitHub.

На всякий случай напомним: в книге мы часто называем Visual Studio просто «IDE».

Отличная работа!



2 Погружение в C#

Команды, классы и код

Я слышала, что настоящие разработчики используют только механические клавиатуры «со щелчком». Это правда?



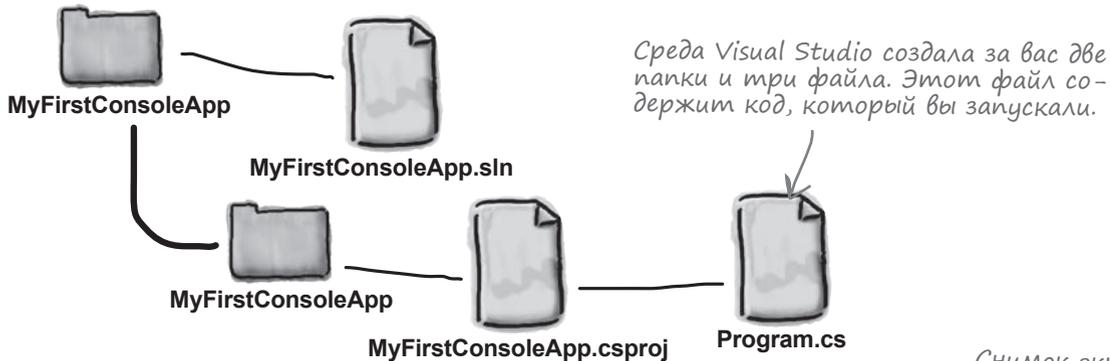
Вы не просто пользователь IDE. Вы — разработчик.

IDE может сделать за вас очень многое, и все же ее возможности не безграничны. Visual Studio — одна из самых совершенных систем разработки программного обеспечения, однако **мощная IDE** — только начало. Пришло время заняться **углубленным изучением кода C#**: какую структуру он имеет, как он работает, как управлять им... Потому что нет предела тому, что вы можете делать в ваших приложениях.

(И для ясности: вы можете быть **настоящим разработчиком** независимо от того, какие клавиатуры вы предпочитаете. Требование только одно: **писать качественный код!**)

Присмотримся к файлам консольного приложения

В последней главе мы создали проект консольного приложения .NET Core и присвоили ему имя MyFirstConsoleApp. Когда вы это сделали, среда Visual Studio создала две папки и три файла.



А теперь рассмотрим файл Program.cs. Откройте его в Visual Studio:

Снимок экрана сделан в Visual Studio для Windows. Если вы работаете в macOS, экран будет выглядеть немного иначе, но код останется тем же.

```

1  using System;
2
3  namespace MyFirstConsoleApp
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Hello World!");
10         }
11     }
12 }
13

```

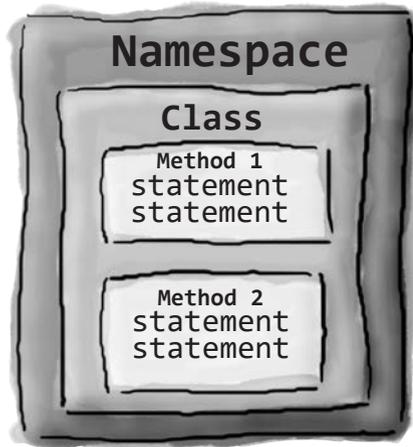
Метод с именем Main. При запуске консольное приложение ищет класс, содержащий метод с именем Main, и начинает выполнение с первой команды этого метода. Метод называется точкой входа, потому что именно в этом методе C# «входит» в программу.

- ★ В верхней части файла размещается **директива using**. Подобные строки using присутствуют во всех файлах с кодом C#.
- ★ Сразу же после директив using идет **ключевое слово namespace**. Ваш код принадлежит пространству имен с именем MyFirstConsoleApp. Сразу же после него следует открывающая фигурная скобка {, а в конце файла — закрывающая фигурная скобка }. Все, что находится между этими скобками, принадлежит пространству имен.
- ★ Внутри пространства имен находится **класс**. В вашей программе используется один класс с именем Program. Сразу же за объявлением класса следует открывающая фигурная скобка, а парная закрывающая скобка располагается в предпоследней строке файла.
- ★ Внутри класса находится метод с именем Main — за объявлением также следует пара фигурных скобок с содержимым.
- ★ Метод содержит одну **команду**: `Console.WriteLine("Hello World!");`



Анатомия отладчика

Код всех программ C# имеет одинаковую структуру. Во всех программах используются пространства имен, классы и методы, чтобы с кодом было удобнее работать.



При создании классов для них определяются пространства имен, чтобы эти классы существовали отдельно от классов, поставляемых с .NET.

Класс содержит часть вашей программы (хотя очень маленькие программы могут состоять из одного класса).

Класс содержит один или несколько методов. Методы всегда должны принадлежать классу. Методы состоят из команд (таких, как команда `Console.WriteLine`, используемая приложением для вывода строки на консоль).

Порядок следования методов в файле класса роли не играет. Метод 2 с таким же успехом может размещаться перед методом 1.

Команда выполняет одно действие

Каждый метод состоит из **команд**, таких как команда `Console.WriteLine`. Когда ваша программа вызывает метод, она выполняет первую команду, затем следующую, затем следующую и т. д. Когда будет выполнена последняя команда метода (или программа достигнет команды `return`), метод завершается и выполнение программы продолжается с точки после команды, из которой был вызван метод.

Часть Задаваемые Вопросы

В: Я понимаю, для чего нужен файл `Program.cs` — здесь хранится код моей программы. Но зачем нужны два других файла и папки?

О: Когда вы начинаете новый проект в Visual Studio, среда создает **решение** (solution). Решение представляет собой контейнер для проекта. Файл решения имеет суффикс `.sln` и содержит список проектов, входящих в решение, а также незначительный объем дополнительной информации (например, версию Visual Studio, которая использовалась для создания решения). **Проект** хранится в папке внутри папки решения. Для проекта выделяется отдельная папка, потому что некоторые решения могут содержать несколько проектов, но в нашем решении проект только один и его имя совпадает с именем решения (`MyFirstConsoleApp`). Папка проекта вашего приложения содержит два файла: файл `Program.cs` с кодом и **файл проекта** с именем `MyFirstConsoleApp.csproj`. Файл проекта содержит всю информацию, необходимую Visual Studio для **построения** кода, т. е. преобразования его в форму, которая может выполняться на компьютере. В будущем вы увидите в папке проекта **еще две папки**: папка `bin/` содержит используемые файлы, построенные на основе вашего кода C#, а в папке `obj` хранятся временные файлы, использованные при построении.

Два класса могут находиться в одном пространстве имен (и файле!)

Взгляните на файлы с кодом C# из программы с именем PetFiler2. Они содержат три класса: Dog, Cat и Fish. Так как все они принадлежат одному пространству имен PetFiler2, команды метода Dog.Bark смогут вызывать методы Cat.Meow и Fish.Swim без включения директивы using.

Если метод помечен ключевым словом `public`, это означает, что он может использоваться другими классами.

MoreClasses.cs

```
namespace PetFiler2 {  
  
    public class Fish {  
        public void Swim() {  
            // команды  
        }  
    }  
  
    public partial class Cat {  
        public void Purr() {  
            // statements  
        }  
    }  
}
```

SomeClasses.cs

```
namespace PetFiler2 {  
  
    public class Dog {  
        public void Bark() {  
            // здесь размещаются команды  
        }  
    }  
  
    public partial class Cat {  
        public void Meow() {  
            // другие команды  
        }  
    }  
}
```

Класс может охватывать несколько файлов, но при его объявлении должно использоваться ключевое слово `partial`. Неважно, как разные пространства имен и классы распределяются по файлам. При выполнении они работают точно так же.

Класс может быть разбит по разным файлам только при использовании ключевого слова `partial`. Возможно, в коде, написанном для этой книги, эта возможность будет использоваться не так часто, но она еще встретится вам в этой главе, и мы хотим избежать неприятных сюрпризов.



Выходит, IDE действительно сильно упрощает работу. Среда и генерирует код, и помогает мне с поиском проблем в моем коде.

IDE помогает разработчику написать правильный код.

Давным-давно программистам приходилось вводить код в простых текстовых редакторах, таких как Блокнот для Windows или TextEdit для macOS. В то время некоторые возможности таких редакторов были невероятно передовыми (например, поиск с заменой или Ctrl+G для перехода к строке с заданным номером в Блокноте). Нам приходилось использовать много хитроумных приложений командной строки для построения, запуска, отладки и развертывания кода.

За прошедшие годы Microsoft (и будем откровенными — многие другие компании, а также отдельные разработчики) придумала много других полезных функций: выделение ошибок, IntelliSense, визуальное редактирование пользовательского интерфейса в режиме WYSIWYG, автоматическое генерирование кода и т. д.

После многих лет эволюции среда Visual Studio стала одной из самых совершенных систем редактирования кода. И к счастью для нас, она также стала *отличным инструментом для изучения C# и исследования процесса разработки приложений.*

В: Я уже видел фразу «Hello World». У нее есть какой-то особый смысл?

О: «Hello World» — программа, которая делает только одно: она выводит фразу «Hello World», чтобы показать, что программа действительно успешно запускается и выполняется. Часто она становится первой программой, которую вы пишете на новом языке, а для многих из нас — первым кодом, который мы пишем на любом языке.

В: Фигурных скобок очень много — в них трудно ориентироваться. Без них действительно не обойтись?

О: В C# фигурные скобки (некоторые люди называют их «усами», но мы этот термин не используем) группируют команды внутри блоков. Фигурные скобки всегда образуют пары. Закрывающая фигурная скобка может встретиться в программе только после открывающей. IDE помогает находить парные фигурные скобки — щелкните на одной скобке, и она вместе со своей парной скобкой изменит цвет. Также можно сворачивать/разворачивать блоки в фигурных скобках при помощи кнопки  в левой части редактора.

В: Что же такое «пространства имен» и для чего они нужны?

О: Пространства имен помогают упорядочить различные средства, необходимые вашей программе. Чтобы вывести строку текста, приложение использует класс Console, являющийся частью .NET Core — кроссплатформенного фреймворка с открытым кодом, который содержит многочисленные вспомогательные классы для построения приложений. И «многочисленные» следует понимать буквально — эти классы исчисляются тысячами, поэтому .NET использует пространства имен для их упорядочения. Класс Console принадлежит пространству имен с именем System, поэтому чтобы вы могли использовать его в программе, в начале кода должна располагаться директива using System;.

В: Я плохо понимаю, что такое точка входа. Можете объяснить еще раз?

О: Ваша программа состоит из множества команд, и эти команды не могут выполняться одновременно. Программа начинает с первой команды, выполняет ее, переходит к следующей, затем к следующей за ней и т. д. Команды обычно распределяются по классам.

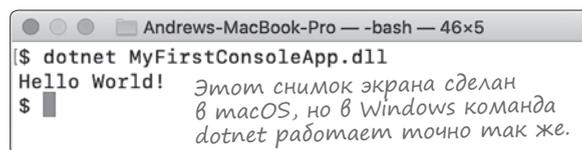
Итак, вы запускаете свою программу. Как ей узнать, с какой команды нужно начать выполнение? Именно для этого и нужна точка входа. Чтобы ваш код успешно построился, в нем должен присутствовать **ровно один метод с именем Main**. Он называется точкой входа, потому что программа начинает выполнение (т. е. входит в код) с первой команды метода Main.

В: Значит, мои консольные приложения .NET Core будут работать в других операционных системах?

О: Да! .NET Core является кроссплатформенной реализацией .NET (который включает такие классы, как List и Random), так что ваши приложения могут запускаться на любом компьютере с системой Windows, macOS или Linux.

Попробуйте сделать это прямо сейчас. Вам понадобится .NET Core. Программа установки Visual Studio **устанавливает .NET Core автоматически**, но .NET Core также можно загрузить по адресу <https://dotnet.microsoft.com/download>.

После того как поддержка .NET Core будет установлена, найдите папку проекта — щелкните правой кнопкой мыши на проекте MyFirstConsoleApp в IDE и выберите команду Open Folder in File Explorer (Windows) или Reveal in Finder (macOS). Перейдите в соответствующий **подкаталог bin/Debug/** и скопируйте все файлы на компьютер, на котором должна выполняться программа. После этого программу можно будет запустить, и она будет работать на **любом** компьютере с системой Windows, Mac или Linux с установленной средой .NET Core:



```
Andrews-MacBook-Pro — bash — 46x5
[$ dotnet MyFirstConsoleApp.dll
Hello World!
$
```

Этот снимок экрана сделан в macOS, но в Windows команда dotnet работает точно так же.

В: Обычно я запускаю программы двойным щелчком, но с файлом .dll у меня ничего не получается. Возможно ли создать обычный исполняемый файл Windows или macOS, который можно запустить напрямую?

О: Да. Команда dotnet позволяет публиковать **исполняемые двоичные файлы** для других платформ. Откройте окно командной строки или терминал, перейдите в папку, в которой находится файл .sln или .csproj, и выполните эту команду, чтобы сгенерировать исполняемый файл Windows, — этот прием работает в любой операционной системе с установленной командой dotnet, не только в Windows:

```
dotnet publish -c Release -r win10-x64
```

Последняя строка вывода должна содержать текст MyFirstConsoleApp->, за которым следует имя папки. Папка содержит файл MyFirstConsoleApp.exe (и несколько DLL-файлов, необходимых для его работы). Также можно строить исполняемые программы для других платформ. Замените win10-x64 на osx-x64, чтобы опубликовать автономное приложение для macOS:

```
dotnet publish -c Release -r osx-x64
```

или укажите linux-x64 для публикации приложения Linux. Этот параметр называется **идентификатором среды выполнения** (RID) — полный список RID доступен по адресу <https://docs.microsoft.com/en-us/dotnet/core/rid-catalog>.

Команды являются структурными элементами приложений

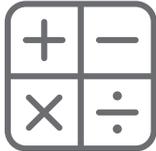
Ваше приложение состоит из классов, классы содержат методы, а методы состоят из команд. А значит, если вы хотите построить приложение, которое решает много разных задач, вам понадобится **много разных видов команд**. Одну разновидность команд вы уже видели:

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

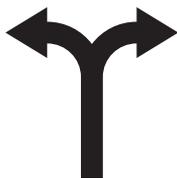
Эта команда **вызывает метод**, а именно метод `Console.WriteLine`, который выводит строку текста на консоль. В этой и в других главах книги будут описаны другие разновидности команд. Несколько примеров:



Переменные и объявления переменных позволяют приложению хранить данные и работать с ними.



Во многих программах задействованы математические вычисления, поэтому математические операторы используются для вычитания, умножения, деления и т. д.



Условные команды позволяют нам выбирать между вариантами, чтобы в программе выполнялся либо один блок кода, либо другой.



Благодаря циклам один блок программы выполняется снова и снова, пока не будет выполнено некоторое условие.

Переменные используются в программах для работы с данными

Любая программа, независимо от ее размера, работает с данными. Эти данные могут быть представлены в форме документа, картинки в видео-игре, обновления в социальной сети, и все равно они остаются данными. Здесь-то в игру и вступают **переменные**. Они используются программой для хранения данных.



Объявление переменных

При **объявлении** переменной вы сообщаете программе ее тип и имя. Зная тип вашей переменной, C# сможет выдавать ошибки при попытке выполнения бессмысленных операций – например, при попытке вычитания "Fido" из 48353. Такие ошибки препятствуют построению программы. Несколько примеров объявления переменных:

```
// Let's declare some variables  
int maxWeight;  
string message;  
bool boxChecked;
```

Типы переменных. Для C# тип определяет, какие данные могут храниться в переменной.

Имена переменных. С точки зрения C# неважно, какие имена присвоены переменным, они нужны только для вас.

Вот почему так важно выбирать содержательные и понятные имена переменных.

Любая строка, начинающаяся с //, является **комментарием** и не выполняется программой. Комментарии можно использовать для добавления примечаний, которые помогут людям разобраться в вашем коде и понять его логику.

Каждый раз, когда вашей программе требуется работать с числами, текстом, значениями «истина / ложь» или любыми другими видами данных, для хранения этих данных используются **переменные**.

Переменные могут изменяться

В разные моменты выполнения программы переменная может содержать разные значения. Другими словами, значение переменной **изменяется** (собственно, поэтому они и называются «переменными»). И это очень важный момент, потому что эта идея лежит в основе любой написанной вами программы. Допустим, ваша программа присваивает переменной `myHeight` значение 63:

```
int myHeight = 63;
```

Каждый раз, когда `myHeight` встречается в коде, C# заменяет ее имя текущим значением 63. Допустим, позднее переменной будет присвоено новое значение 12:

```
myHeight = 12;
```

С этого момента C# будет заменять `myHeight` значением 12 (пока переменная снова не изменится), но переменная по-прежнему будет называться `myHeight`.

Перед использованием переменной необходимо присвоить значение

Попробуйте ввести следующие команды непосредственно перед командой вывода «Hello World» в новом консольном приложении:

```
string z;
string message = "The answer is " + z;
```

Попробуйте сделать это прямо сейчас. Вы получите сообщение об ошибке, и IDE не сможет построить ваш код. Дело в том, что IDE проверяет каждую переменную и следит за тем, чтобы перед использованием ей было присвоено значение. Чтобы вы не забыли присвоить значение переменной перед ее использованием, проще всего объединить объявление переменной с командой, присваивающей ей значение:

```
int maxWeight = 25000;
string message = "Hi!";
bool boxChecked = true;
```

Эти значения присваиваются переменным. Вы можете объявить переменную и присвоить ей начальное значение в одной команде (хотя это и необязательно).

Несколько полезных типов

У каждой переменной имеется тип, который сообщает C#, какого рода данные в ней могут храниться. Различные типы C# будут подробно рассмотрены в главе 4, а пока мы сосредоточимся на трех самых популярных типах. Тип `int` предназначен для хранения целых чисел, в типе `string` хранится текст, а в типе `bool` — логические значения «истина/ложь».

пе-ре-мен-ная, суц.
элемент или фактор, который с большой вероятностью может измениться.

Сделайте это!

Если вы пишете код с использованием переменных, которым не было присвоено значение, ваш код строиться не будет. Этой ошибки можно легко избежать, для этого достаточно объединить объявление переменной с присваиванием в одну команду.

После того как переменной будет присвоено значение, это значение можно в любой момент изменить. А значит, присваивание переменной исходного значения при объявлении не создает никаких неудобств.

Генерирование нового метода для работы с переменными

В предыдущей главе вы узнали, что Visual Studio умеет **генерировать код за вас**. Это весьма удобно, когда вы пишете код, а также может быть очень полезным учебным средством. Давайте воспользуемся тем, что вы узнали ранее, и присмотримся повнимательнее к генерированию методов.

← (Делайте это!

1 Добавьте метод в новый проект MyFirstConsoleApp.

Откройте проект консольного приложения, созданный в последней главе. IDE создала ваше приложение с методом Main, который содержит ровно одну команду:

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

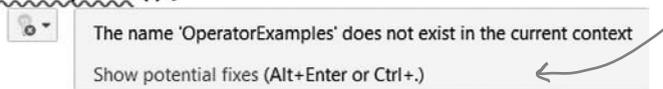
Замените ее командой, в которой вызывается другой метод:

```
OperatorExamples();
```

2 Получите информацию о проблеме от Visual Studio.

Как только вы завершите замену, Visual Studio подчеркнет вызов метода красной волнистой линией. Наведите на нее указатель мыши. В IDE появляется временное окно с подсказкой:

```
OperatorExamples();
```



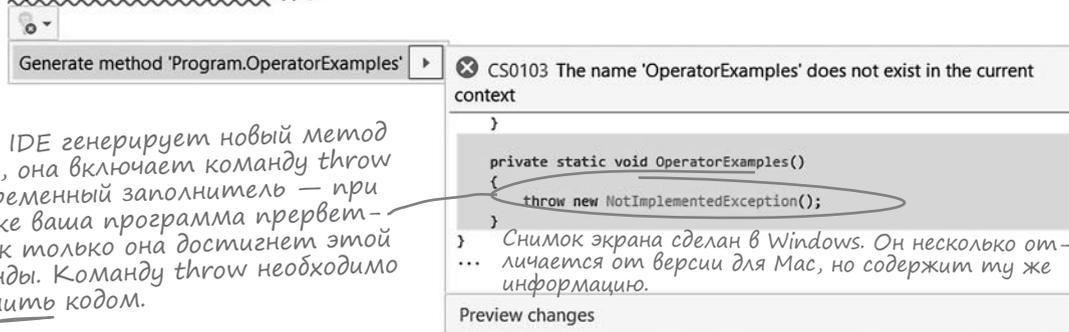
На Mac щелкните на ссылке или нажмите Option+Return, чтобы вывести список предлагаемых решений.

Visual Studio сообщает вам две вещи: что в программе существует проблема — вы пытаетесь вызвать несуществующий метод (из-за чего построение кода невозможно) и что у среды имеются предложения по ее исправлению.

3 Сгенерируйте метод OperatorExamples.

В системе **Windows** временное окно предлагает нажать Alt+Enter или Ctrl+ для просмотра возможных исправлений. В **macOS** присутствует ссылка «Show potential fixes» — нажмите Option+Return. Нажмите одну из этих комбинаций клавиш (или щелкните на раскрывающемся списке слева от временного окна).

```
OperatorExamples();
```

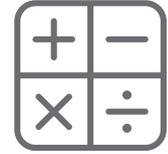


Когда IDE генерирует новый метод за вас, она включает команду throw как временный заполнитель — при запуске ваша программа прервется, как только она достигнет этой команды. Команду throw необходимо заменить кодом.

Снимок экрана сделан в Windows. Он несколько отличается от версии для Mac, но содержит ту же информацию.

У IDE есть решение: она сгенерирует метод с именем OperatorExamples в классе Program. **Щелкните на ссылке Preview Changes**, чтобы открыть окно с потенциальным решением IDE — добавлением нового метода. Затем **щелкните на кнопке Apply**, чтобы включить метод в ваш код.

Добавление кода с использованием операторов



Итак, в переменной хранятся данные. Что теперь с ними можно сделать? Если это число, его можно с чем-нибудь сложить или умножить. Если это строка, ее можно объединить с другими строками. В этом вам помогут **операторы**. Ниже приведено тело нового метода `OperatorExample`. **Включите этот код в свою программу** и прочитайте *комментарии*, чтобы узнать об использованных в нем операторах.

```
private static void OperatorExamples()
{
    // Эта команда объявляет переменную и присваивает ей значение 3
    int width = 3;

    // Оператор ++ инкрементирует переменную (увеличивает ее на 1)
    width++;

    // Объявляем еще две переменные int для хранения чисел
    // и используем операторы + и * для сложения и умножения значений
    int height = 2 + 4;
    int area = width * height;
    Console.WriteLine(area);

    // Следующие две команды объявляют строковые переменные
    // и объединяют их оператором + (эта операция называется конкатенацией)
    string result = "The area";
    result = result + " is " + area;
    Console.WriteLine(result);

    // Логическая переменная может содержать
    // либо true, либо false
    bool truthValue = true;
    Console.WriteLine(truthValue);
}
```

Строковые переменные используются для хранения текста. Когда вы используете оператор + со строками, эти строки объединяются, так что сложение "abc"+"def" дает строку "abcdef". Такое объединение строк называется конкатенацией.

MINI Возьми в руку карандаш

Команды, только что добавленные в ваш код, выводят на консоль три строки: каждая команда `Console.WriteLine` выводит отдельную строку. Прежде чем запускать программу, постарайтесь разобраться, что будет выведено, и запишите результаты. И не пытайтесь подсмотреть решение — его здесь нет! Чтобы проверить свои ответы, просто запустите программу.

Подсказка: при преобразовании bool в строку может быть получен результат False или True.

Строка 1: _____

Строка 2: _____

Строка 3: _____

Использование отладчика для наблюдения за изменением переменных

Когда вы запускали свою программу ранее, она выполнялась в отладчике — невероятно полезном инструменте, который помогает понять, как работают программы. Вы можете использовать **точки прерывания** для приостановки программы в тот момент, когда она достигает определенных команд, и следить за значениями переменных в **окне просмотра**. Воспользуемся отладчиком, чтобы увидеть код в действии. Нам потребуются три функции отладчика, которые вы найдете на панели инструментов:

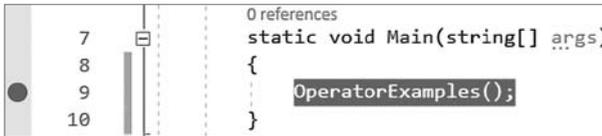


Если программа окажется в состоянии, которого вы не ожидали, просто перезапустите отладчик кнопкой Restart .

← Делайте это!

1 Установите точку прерывания и запустите программу.

Наведите указатель мыши на вызов метода, добавленный к методу Main вашей программы, и **выберите команду Toggle Breakpoint (F9) из меню Debug**. Строка должна выглядеть так:

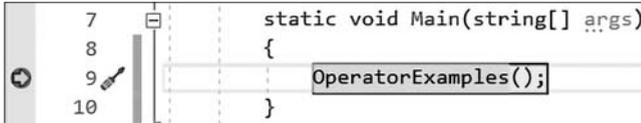


Нажмите кнопку  `MyFirstConsoleApp`, чтобы выполнить программу в отладчике, как это делалось ранее.

На Mac для управления отладкой используются комбинации клавиш **Step Over** ($\uparrow \&\#O$), **Step Into** ($\uparrow \&\#I$) и **Step Out** ($\uparrow \&\#U$). Экраны будут незначительно отличаться, но отладчик работает точно так же, как показано в «Руководстве для пользователя Mac» (приложение 1).

2 Выполните программу в пошаговом режиме с входом в метод.

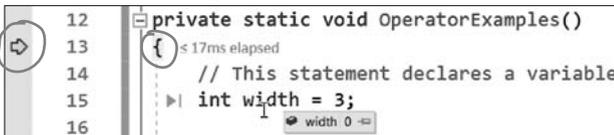
Ваш отладчик останавливается в точке прерывания, установленной на команде с вызовом метода `OperatorExamples`.



Нажмите кнопку **Step Into (F11)** — отладчик заходит в метод, а затем останавливается перед выполнением первой команды.

3 Проверьте значение переменной width.

При **пошаговом выполнении кода** отладчик приостанавливается после каждой выполненной команды. Разработчик получает возможность проверить значения переменных. Наведите указатель мыши на переменную `width`.

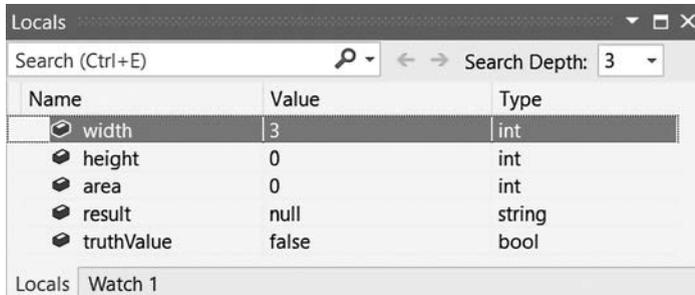


Выделенная фигурная скобка и стрелка на левом поле означают, что код был приостановлен перед выполнением первой команды метода.

IDE открывает временное окно с текущим значением переменной — в настоящее время оно равно 0. Теперь **нажмите кнопку Step Over (F10)** — управление пропускает комментарий и переходит к первой команде, которая выделяется цветом. Мы хотим выполнить команду, поэтому снова **нажмите кнопку Step Over (F10)**. Еще раз задержите указатель мыши на `width`. Теперь переменная содержит значение 3.

4 В окне Locals выводятся значения переменных.

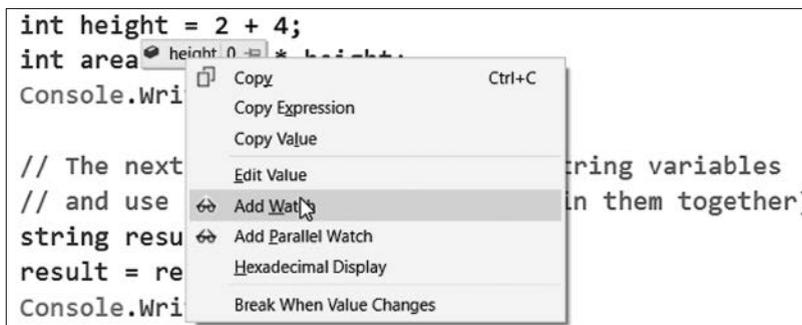
Объявленные переменные являются **локальными** по отношению к методу OperatorExamples — это означает, что они существуют только внутри метода и могут выполняться командами в методе. В процессе отладки Visual Studio выводит значения переменных в окне Locals в нижней части IDE.



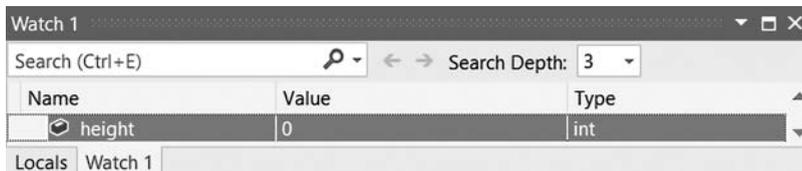
Окна Locals и Watch в Visual Studio для Mac выглядят немного иначе, чем в Windows, но содержат ту же информацию. Отслеживаемые значения в версиях Visual Studio для Windows и Mac добавляются одним и тем же способом.

5 Добавьте отслеживание для переменной height.

Одной из самых полезных возможностей отладчика является **окно Watch**, которое обычно располагается на той же панели, что и окно Locals в нижней части IDE. Когда вы наводите указатель мыши на переменную, вы можете добавить ее для отслеживания — щелкните правой кнопкой мыши на имени переменной во временном окне и выберите команду **Add Watch**.



Переменная height появляется в окне Watch.



6 Выполните остальной код метода в пошаговом режиме.

Выполните в пошаговом режиме каждую команду OperatorExamples. В ходе пошагового выполнения метода обращайте внимание на окна Locals и Watch и следите за изменением значений. В системе Windows нажмите Alt+Tab до и после команд Console.WriteLine, чтобы переключаться на отладочную консоль и обратно для просмотра вывода. В macOS вывод направляется в окно терминала, так что вам не придется переключаться между окнами.

Отладчик — один из самых полезных инструментов Visual Studio, а заодно и прекрасный инструмент для анализа работы ваших программ.

= *против* ==

Использование операторов для работы с переменными

Хорошо, данные сохранены в переменной. Что с ними можно сделать? Часто в программе требуется выполнить некоторые действия в зависимости от значения. И здесь начинают играть важную роль операторы **проверки равенства, операторы отношения и логические операторы**:

Операторы проверки равенства

Оператор == сравнивает два значения и возвращает результат true, если они равны.

Оператор != очень похож на ==, но он возвращает true, если два сравниваемых значения не равны.

Операторы отношения

Операторы > и < используются для сравнения чисел: они проверяют, что число в одной переменной больше или меньше числа в другой переменной.

Также можно использовать оператор >= для проверки того, что одно значение больше либо равно другому, или оператор <= для проверки того, что одно значение меньше либо равно другому.

Логические операторы

Отдельные условия объединяются в одно составное условие при помощи оператора && (**И**) и оператора || (**ИЛИ**).

Вот как можно проверить, что *i* равно 3 или *j* меньше 5:
(i == 3) || (j < 5)



**Будьте
осторожны!**

**Не путайте операторы =
и ==!**

Оператор = (один знак равенства) присваивает значение переменной, а оператор == (два знака равенства) проверяет два значения на равенство. Вы не поверите, сколько ошибок в программах — даже у опытных программистов! — встречается из-за использования = вместо ==. Если IDE начинает жаловаться на то, что «не удается преобразовать тип 'int' в 'bool'», то, скорее всего, именно это произошло в вашей программе.

Операторы для сравнения двух переменных int

Простые проверки проверяют значение переменной при помощи оператора сравнения. Примеры сравнения двух переменных int, *x* и *y*:

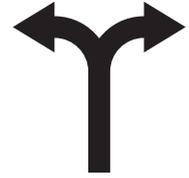
x < *y* (меньше)

x > *y* (больше)

x == *y* (равно – обратите внимание на два знака равенства)

Именно такие проверки вы будете использовать чаще всего.

Принятие решений в командах if



Команды `if` в вашей программе выполняют некоторые действия только в том случае, если заданные условия истинны (или ложны). Команда `if` *проверяет условие* и выполняет код, если проверка дает результат `true`. Очень часто команды `if` проверяют два значения на равенство. В таких ситуациях используется оператор `==` (напомним: он отличается от оператора `=`, который используется для присваивания).

```
int someValue = 10;
string message = "";

if (someValue == 24)
{
    message = "Yes, it's 24!";
}
```

Каждая команда `if` начинается с условия в круглых скобках. Затем следует блок команд в фигурных скобках, который выполняется в случае истинности условия.

Команды в фигурных скобках выполняются только в том случае, если условие истинно.

Команды `if/else` также выполняют некоторые действия, если условие не истинно

Команды `if/else` расширяют логику `if`: если условие истинно, они делают что-то одно, а если нет — что-то другое. Команда `if/else` представляет собой команду `if`, за которой следуют ключевое слово `else` и второй набор выполняемых команд. Если условие истинно, программа выполняет команды в первой паре фигурных скобок, а если нет — выполняются команды из второй пары.

```
if (someValue == 24)
{
    // Фигурные скобки могут содержать
    // сколько угодно команд
    message = "The value was 24.";
}
else
{
    message = "The value wasn't 24.";
}
```

ПОМНИТЕ: для проверки равенства двух значений всегда используются два знака равенства.

Циклы выполняют некоторые действия снова и снова



У многих программ (и *особенно* игр!) есть одна странная особенность: они почти всегда требуют повторения некоторых действий. Для этого в программах используются **циклы** — они приказывают вашей программе выполнять заданный набор команд, пока некоторое условие остается истинным (или ложным).

Циклы `while` выполняются, пока условие остается истинным

В **циклах** `while` все команды в фигурных скобках выполняются, пока условие в круглых скобках остается истинным.

```
while (x > 5)
{
    // Команды в этих фигурных скобках выполняются,
    // пока значение x больше 5
}
```

Циклы `do/while` сначала выполняют команды, а потом проверяют условие

Цикл `do/while` очень похож на цикл `while`, но с одним отличием. Цикл `while` сначала проверяет условие, а **затем** выполняет свои команды только в том случае, если условие истинно. Таким образом, цикл `do/while` хорошо подходит для тех случаев, в которых набор команд должен быть гарантированно выполнен хотя бы один раз.

```
do
{
    // Команды в фигурных скобках будут выполнены один раз,
    // а затем цикл будет продолжаться, пока
    // выполняется условие x > 5
} while (x > 5);
```

Циклы `for` выполняют свой блок при каждом проходе

Цикл `for` выполняет команду при каждом выполнении (итерации).

Заголовок цикла `for` состоит из трех команд. Первая команда задает начальное состояние переменной цикла. Цикл продолжает выполняться, пока вторая команда остается истинной. Наконец, третья команда продолжает выполняться после каждого прохода цикла.

```
for (int i = 0; i < 8; i = i + 2)
{
    // Все команды, заключенные в фигурные
    // скобки, будут выполнены 4 раза
}
```

Части заголовка `for` называются **инициализатором** (`int i=0`), **условием цикла** (`i<8`) и **итератором** (`i=i+2`). Каждый проход цикла `for` (и вообще любого цикла) называется **итерацией**.

Условие цикла всегда проверяется в начале каждой итерации, а итератор всегда выполняется в конце итерации.



Циклы for под увеличительным стеклом

Циклы `for` устроены сложнее (но при этом более гибки) по сравнению с простыми циклами `while` или `do`. Самая распространенная разновидность цикла `for` просто выполняется заданное количество раз. **Фрагмент кода `for`** заставляет IDE создать такую разновидность цикла `for`:

```
for (int i = 0; i < length; i++)
{
    ...
}
```

Когда вы используете фрагмент `for`, нажатие `Tab` используется для переключения между `i` и `length`. Если вы измените имя переменной `i`, фрагмент автоматически изменит два других вхождения этого имени.

Цикл `for` состоит из четырех частей: инициализатора, условия, итератора и тела:

```
for (инициализатор; условие; итератор) {
    тело
}
```

В большинстве случаев инициализатор используется для объявления новой переменной — например, инициализатор `int i = 0` в приведенном фрагменте кода `for` объявляет переменную с именем `i`, которая может использоваться только внутри цикла `for`. После этого цикл выполняет тело (это может быть одна команда или блок команд в фигурных скобках), пока условие остается истинным. В конце каждой итерации цикл `for` выполняет итератор. Таким образом, следующий цикл `for`:

```
for (int i = 0; i < 10; i++) {
    Console.WriteLine("Iteration #" + i);
}
```

будет выполнен 10 раз, а на консоль будут выведены сообщения `Iteration #0`, `Iteration #1`, ..., `Iteration #9`.

Возьми в руку карандаш



Ниже приведены примеры циклов. Запишите, будет ли каждый цикл повторяться бесконечно или со временем завершится. Если он завершится, то сколько раз он будет выполнен? Также ответьте на вопросы в комментариях в циклах 2 и 3.

```
// Цикл #1
int count = 5;
while (count > 0) {
    count = count * 3;
    count = count * -1;
}
```

Помните: цикл `for` всегда проверяет условие перед выполнением блока, а итератор — в конце блока.

```
// Цикл #4
int i = 0;
int count = 2;
while (i == 0) {
    count = count * 3;
    count = count * -1;
}
```

```
// Цикл #2
int j = 2;
for (int i = 1; i < 100; i = i * 2)
{
    j = j - 1;
    while (j < 25)
    {
        // Сколько раз будет
        // выполнена
        // следующая команда?
        j = j + 5;
    }
}
```

```
// Цикл #5
while (true) { int i = 1;}
```

```
// Цикл #3
int p = 2;
for (int q = 2; q < 32; q = q * 2)
{
    while (p < q)
    {
        // Сколько раз будет
        // выполнена следующая
        // команда?
        p = p * 2;
    }
    q = p - q;
}
```

Подсказка: значение `p` изначально равно 2. Подумайте над тем, когда будет выполняться итератор "`p = p * 2`".

Когда мы включаем в книгу письменное упражнение, решение обычно приводится на следующей странице.



 Возьми в руку карандаш
Решение

Ниже приведены примеры циклов. Запишите, будет ли каждый цикл повторяться бесконечно или со временем завершится. Если он завершится, то сколько раз он будет выполнен? Также ответьте на вопросы в комментариях в циклах 2 и 3.

```
// Цикл #1
int count = 5;
while (count > 0) {
    count = count * 3;
    count = count * -1;
}
```

Цикл 1 будет выполнен один раз.

Помните: $count = count * 3$ умножает $count$ на 3, после чего сохраняет результат (15) в той же переменной $count$.

```
// Цикл #4
int i = 0;
int count = 2;
while (i == 0) {
    count = count * 3;
    count = count * -1;
}
```

Цикл 4 будет выполняться бесконечно.

```
// Цикл #2
int j = 2;
for (int i = 1; i < 100;
     i = i * 2)
{
    j = j - 1;
    while (j < 25)
    {
        // Сколько раз будет
        // выполнена следующая
        // команда?
        j = j + 5;
    }
}
```

Цикл 2 будет выполнен 7 раз.
Команда $j = j + 5$ выполняется 6 раз.

```
// Loop #5
while (true) { int i = 1; }
Цикл 5 также является бесконечным.
```

```
// Цикл #3
int p = 2;
for (int q = 2; q < 32;
     q = q * 2)
{
    while (p < q)
    {
        // Сколько раз будет
        // выполнена следующая
        // команда?
        p = p * 2;
    }
    q = p - q;
}
```

Цикл 3 выполняется 8 раз.
Команда $p = p * 2$ выполняется 3 раза.



Не жалейте времени и постарайтесь по-настоящему разобраться в том, как работает цикл 3. Это идеальная возможность опробовать отладчик на практике! Установите точку прерывания на команде $q = p - q$; после чего воспользуйтесь окном Locals для наблюдения за изменениями p и q при пошаговом выполнении цикла.

Используйте фрагменты кода для написания циклов



В этой книге вы будете часто писать циклы, и Visual Studio может ускорить эту работу при помощи **фрагментов** (snippets) — простых шаблонов, используемых для добавления кода. Воспользуемся фрагментами для включения нескольких циклов в метод `OperatorExamples`.

Если код все еще выполняется, выберите команду **Stop Debugging (Shift+F5)** из меню `Debug` (или нажмите кнопку `Stop`  на панели инструментов). Затем найдите `Console.WriteLine(area)`; в методе `OperatorExamples`. Щелкните в конце строки, чтобы курсор был установлен после точки с запятой, после чего несколько раз нажмите `Enter`, чтобы добавить немного свободного места. Теперь начинайте вводить фрагмент. **Введите `while` и дважды нажмите клавишу `Tab`**. IDE добавляет в код шаблон для цикла `while`, в котором выделено условие:

```
while (true)
{
    ...
}
```

Введите `area < 50` — IDE заменит `true` введенным текстом. **Нажмите `Enter`**, чтобы завершить фрагмент. Добавьте между фигурными скобками две команды:

```
while (area < 50)
{
    height++;
    area = width * height;
}
```

Затем воспользуйтесь **фрагментом цикла `do/while`** и добавьте другой цикл сразу же после только что добавленного цикла `while`. **Введите `do` и дважды нажмите клавишу `Tab`**. IDE добавляет фрагмент:

```
do
{
    ...
} while (true);
```

Введите `area > 25` и нажмите `Enter`, чтобы завершить фрагмент. Добавьте между фигурными скобками две команды:

```
do
{
    width--;
    area = width * height;
} while (area > 25);
```

Теперь **воспользуйтесь отладчиком**, чтобы хорошо разобраться в том, как работают циклы:

1. Щелкните на строке непосредственно перед первым циклом и выберите команду **Toggle Breakpoint (F9)** из меню `Debug`, чтобы добавить точку прерывания. Запустите свой код и нажмите `F5`, чтобы перейти к новой точке прерывания.
2. Выполните два цикла в пошаговом режиме при помощи кнопки **Step Over (F10)**. Следите за изменением значений `height`, `width` и `area` в окне `Locals`.
3. Остановите программу и измените условие цикла `while` на `area < 20`, чтобы оба цикла имели ложные условия. Снова проведите отладку программы. Цикл `while` сначала проверяет условие и пропускает цикл, а цикл `do/while` выполняется однократно, после чего проверяется условие.

Подсказка для IDE: скобки

Если в программе существуют непарные фигурные скобки, программа строиться не будет. К счастью, IDE поможет вам в этом! Установите курсор у фигурной скобки, и IDE автоматически выделит ее пару.

Возьми в руку карандаш



Немного потренируемся в работе с условными командами и циклами. Обновите метод Main в консольном приложении, чтобы он соответствовал новому методу Main, приведенному ниже, после чего добавьте методы TryAnIf, TryAnIfElse и TrySomeLoops. Прежде чем запускать код, попробуйте ответить на вопросы. Затем выполните код и проверьте свои ответы.

```
static void Main(string[] args)
```

```
{
    TryAnIf();
    TrySomeLoops();
    TryAnIfElse();
}
```

Что выведет на консоль метод TryAnIf?

```
private static void TryAnIf()
```

```
{
    int someValue = 4;
    string name = "Bobbo Jr.";
    if ((someValue == 3) && (name == "Joe"))
    {
        Console.WriteLine("x is 3 and the name is Joe");
    }
    Console.WriteLine("this line runs no matter what");
}
```

```
private static void TryAnIfElse()
```

```
{
    int x = 5;
    if (x == 10)
    {
        Console.WriteLine("x must be 10");
    }
    else
    {
        Console.WriteLine("x isn't 10");
    }
}
```

Что выведет на консоль метод TryAnIfElse?

```
private static void TrySomeLoops()
```

```
{
    int count = 0;
    while (count < 10)
    {
        count = count + 1;
    }
    for (int i = 0; i < 5; i++)
    {
        count = count - 1;
    }
    Console.WriteLine("The answer is " + count);
}
```

Что выведет на консоль метод TrySomeLoops?

Ответы для этого упражнения в книге не приводятся. Чтобы проверить свои ответы, просто запустите программу.

Несколько полезных советов по поводу кода C#

- ★ **Не забывайте, что все команды должны завершаться символом ; (точка с запятой).**

```
name = "Joe";
```

- ★ **Чтобы добавить комментарий в код, начните строку с двух символов //.**

```
// Этот текст игнорируется
```

- ★ **Используйте /* и */ для обозначения начала и конца комментариев, которые могут включать разрывы строк.**

```
/* Этот комментарий
 * занимает несколько строк */
```

- ★ **Объявление переменной начинается с *типа*, за которым следует *имя*.**

```
int weight;
// Переменная с типом int и именем weight
```

- ★ **Во многих случаях лишние пробелы игнорируются.**

```
Таким образом, команда      int      j      =      1234      ;
в точности эквивалентна     int j = 1234;
```

- ★ **Вся суть команд if/else, while, do и for заключается в их условиях.**

Каждый цикл, встречавшийся нам до сих пор, выполнялся до тех пор, пока его условие оставалось истинным.

Что-то здесь не так! А что произойдет с циклом, если его условие никогда не становится ложным?

Тогда ваш цикл будет работать бесконечно.

Каждый раз, когда ваша программа проверяет условие, результат условия может быть равен true или false. Если он равен true, то программа выполнит тело цикла еще один раз. После того как код цикла будет выполнен достаточное количество раз, условие в какой-то момент должно вернуть false. В противном случае цикл будет продолжаться бесконечно, пока вы не прервете программу или не перезагрузите компьютер!

Это называется бесконечным циклом. И разумеется, возможны ситуации, когда вы действительно хотите использовать такой цикл в своем коде.



А вы можете предположить, для чего может быть нужен цикл, который никогда не останавливается?





Разработка игр... и не только

Механика

К игровой **механике** относятся аспекты игры, составляющие реальный игровой процесс: правила, возможные действия игрока и поведение игры в ответ на эти действия.

- Начнем с классической видеоигры. К механикам игры **Pac-Man** относится управление игровым персонажем с джойстика, количество очков за точки и «энергетические таблетки», поведение призраков, продолжительность их перехода в синее (пассивное) состояние, изменение их поведения после того, как игрок съест энергетическую таблетку, условие получения дополнительных жизней игроком, замедление призраков при движении по туннелю — все правила, определяющие игру.
- Говоря о механике, разработчики игры часто имеют в виду отдельный режим взаимодействия или управления: например, двойной прыжок в платформенной игре или защита, способная получить определенное количество попаданий в шутере. Часто бывает полезно выделить отдельную механику для проверки и усовершенствования.
- **Настольные игры** предоставляют отличные возможности для понимания концепций механики. Генераторы случайных чисел (кубики, карты и т. д.) являются отличными примерами конкретных механик.
- Вам уже встречался хороший пример механики: **таймер**, добавленный в игру с поиском пар, полностью изменяет впечатление от игры. Таймеры, препятствия, враги, карты, гонки, призовые очки... все это механики.
- Разные механики комбинируются различными способами, и это может оказать большое влияние на опыт игрока. «Монополия» — хороший пример игры, объединяющей два разных генератора случайных чисел (карты и кубики), чтобы сделать игровой процесс более интересным и нетривиальным.
- К игровым механикам также относятся особенности **структурирования данных и организации кода**, работающего с данными, даже если эти механики не были введены намеренно! Вспомните легендарную *ошибку 256-го уровня Pac-Man*. На этом уровне из-за ошибки в коде экран наполовину заполнялся «мусором», а игра становилась невозможной. Эта ошибка стала частью игровых механик.
- Таким образом, когда мы говорим о механиках игры C#, в эту категорию также **включаются классы и код**, потому что они управляют работой игры.



Наверняка концепция механики может помочь мне в проектах любого типа, не только в играх.



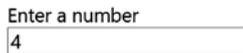
Безусловно! В каждой программе используются свои механики.

Механики существуют на всех уровнях проектирования программных продуктов. Их проще обсуждать и понять в контексте видеоигр. Мы воспользуемся этим обстоятельством для более глубокого понимания механик, что может быть полезно для проектирования и построения любых видов проектов.

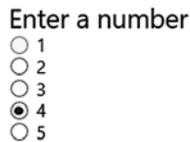
Пример: механики игры определяют, насколько сложно или просто играть в нее. Заставьте героя Pac-Man перемещаться быстрее или замедлите призраков — и игра становится проще. Она не обязательно становится лучше или хуже; она просто становится другой. И знаете что? Эта идея применима и к проектированию ваших классов! Вы можете задуматься над тем, **как проектировать методы и поля** как механики классов. Решения, которые вы принимаете относительно разбиения кода на методы или использования полей, упрощают или усложняют работу с ними.

Элементы управления определяют механику ваших пользовательских интерфейсов

В предыдущей главе для построения игры использовались **элементы управления** TextBox и Grid. Однако существует много разных способов использования элементов, а решения, принятые вами при выборе элементов, могут очень сильно изменить ваше приложение. Звучит неожиданно? На самом деле все это очень похоже на принятие решений при проектировании игр. Если вы создаете настольную игру, которой нужен генератор случайных чисел, вы можете воспользоваться кубиками или картами. Если вы работаете над игрой-платформером, вы можете решить, что игровой персонаж должен уметь прыгать, делать двойной прыжок, делать прыжок от стены или летать (или выполнять разные действия в разное время). То же относится к приложениям: если вы разрабатываете приложение, в котором пользователь должен ввести число, вы можете выбрать для этой цели разные элементы — **и от вашего выбора зависят впечатления пользователя при работе с приложением.**



- ★ В **текстовом поле** пользователь может ввести любой текст по своему усмотрению. При этом в данном примере необходимо проверить, что пользователь вводит только числа, а не произвольный текст.



- ★ **Переключатели** ограничивают выбор пользователя несколькими фиксированными вариантами (например, их можно использовать для выбора чисел). Вы можете разместить их так, как считаете нужным.



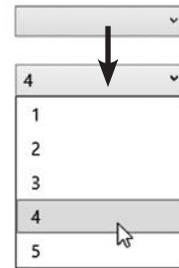
- ★ Другие элементы управления на этой странице могут использоваться для других типов данных, но **ползунки** предназначены исключительно для выбора чисел. В принципе, телефонные номера тоже являются обычными числами, так что формально ползунок может использоваться для выбора телефона. Как вы думаете, хорошая ли это мысль?



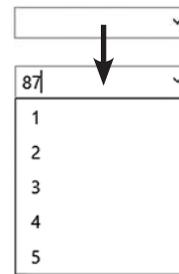
- ★ **Список** предоставляет пользователю возможность выбрать элемент из нескольких вариантов. Длинные списки обычно снабжаются полосой прокрутки, чтобы пользователю было проще найти нужное значение.

Элементы управления — компоненты пользовательского интерфейса (UI), строительные блоки для построения UI. От выбора элементов управления зависит механика вашего приложения.

Позаимствуем идею механики из видеоигр. Она поможет лучше понять возможные варианты и принять правильные решения в любых приложениях (не только видеоиграх).



- ★ **Поле со списком** объединяет поведение списка и текстового поля. Оно выглядит как обычное текстовое поле, но когда пользователь щелкает на списке, под ним открывается список.



В редактируемом поле со списком пользователь может либо выбрать значение из списка, либо ввести собственное значение.

7,183,876,962

В оставшейся части этой главы рассматривается проект для построения настольного приложения WPF для Windows. За соответствующим проектом для macOS обращайтесь к приложению «Visual Studio для пользователей Mac».

Создание приложения WPF для экспериментов с элементами управления



Если вы заполняли форму на веб-странице, то вы уже видели все элементы управления с предыдущей страницы (даже если вы не знали их официальные названия). Теперь **создадим проект WPF**, чтобы немного потренироваться в использовании этих элементов. Приложение будет очень простым — оно предлагает пользователю ввести число, после чего выводит выбранное число.

Элемент `TextBox` предназначен для ввода текста. Мы добавим код, чтобы в текстовом поле можно было вводить только числовые данные.

Элемент `ListBox` позволяет выбрать число из списка.

Эти два ползунка предназначены для выбора чисел. Верхний ползунок позволяет выбрать число от 1 до 5. Нижний ползунок позволяет выбрать телефонный номер — просто чтобы доказать, что это возможно.

Шесть разных переключателей (элементы `RadioButton`). При установке любого переключателя элемент `TextBox` обновляется соответствующим числом.

Элемент `TextBlock` — такой же, как в игре с поиском пар. Каждый раз, когда вы используете любой другой элемент для выбора числа, этот элемент `TextBlock` будет обновляться выбранным числом.

Элемент `ComboBox` тоже позволяет выбрать число из списка, но список открывается только по щелчку.

Еще один элемент `ComboBox`. Он отличается от предыдущего, потому что является редактируемым. Это означает, что пользователь может либо выбрать число из списка, либо ввести его самостоятельно.



Не старайтесь запомнить разметку XAML для этих элементов управления.

Предназначение врезки (сделайте это!) и упражнений — немного потренироваться в использовании XAML для построения UI с элементами. Вы всегда можете вернуться к ним, когда мы будем использовать эти элементы позднее в книге.



Упражнение

В главе 1 мы добавляли определения строк и столбцов сетки в приложение WPF, а именно была создана сетка с пятью строками равной высоты и четырьмя столбцами равной ширины. То же самое будет сделано и в этом приложении.

Создайте новый проект WPF

Запустите Visual Studio 2019 и **создайте новый проект WPF**, как было сделано в игре с поиском пар в главе 1. Выберите команду Create a new project и выберите вариант WPF App (.NET Core).



Присвойте проекту имя **ExperimentWithControls**.

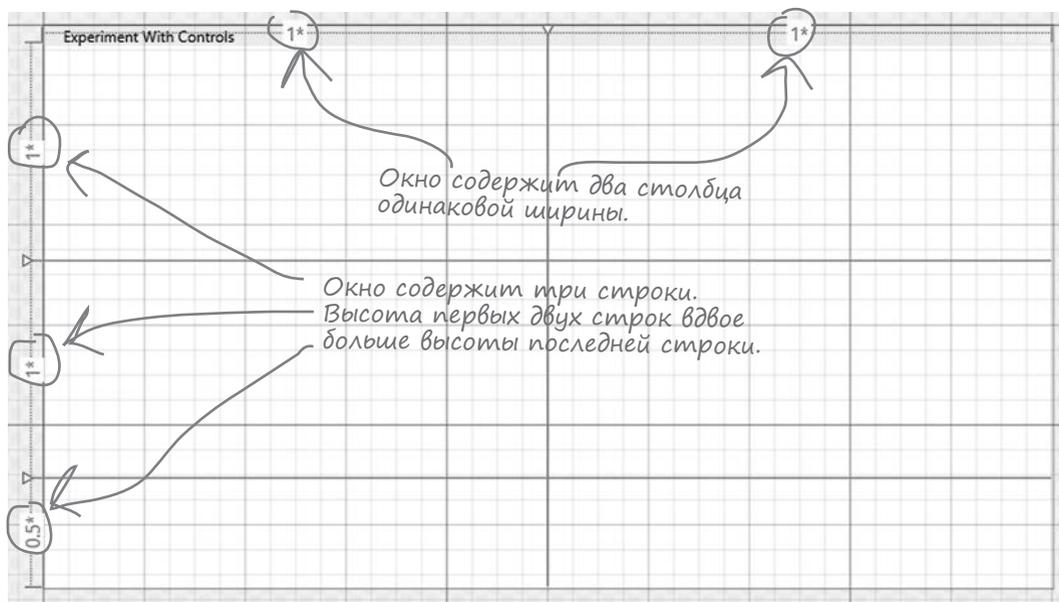
Выберите текст заголовка окна

Измените свойство Title тега <Window> и задайте текст заголовка окна Experiment With Controls.

Добавьте строки и столбцы

Добавьте три строки и два столбца. Высота каждой из первых двух строк должна быть вдвое больше высоты третьей, а два столбца должны иметь одинаковую ширину.

А вот как окно должно выглядеть в конструкторе:





Упражнение Решение

Ниже приведен код XAML главного окна. Мы использовали более светлый шрифт для кода XAML, который был сгенерирован Visual Studio за вас и который вам не пришлось изменять. Свойство Title в теге <Window> было изменено, после чего были добавлены разделы <Grid.RowDefinitions> и <Grid.ColumnDefinitions>.

```
<Window x:Class="ExperimentWithControls.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:ExperimentWithControls"
  mc:Ignorable="d"
  Title="Experiment With Controls" Height="450" Width="800">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
      <RowDefinition Height=".5*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
  </Grid>
</Window>
```

↑
Измените свойство Title окна,
чтобы задать текст заголовка.

↖
В результате назначения нижней строке высоты .5* ее высота будет равна половине высоты каждой из двух других строк. Также можно было задать двум другим строкам высоту 2* (или задать двум верхним строкам высоту 4*, а нижней высоту 2*, или задать двум верхним строкам высоту 1000*, а нижней строке 500*, и т. д.).



Уверен, сейчас самое время добавить проект в систему управления версиями...

«Сохраняйтесь пораньше, сохраняйтесь почаще».

Эта старая поговорка существовала еще до того, как в видеоиграх появилась функция автосохранения, когда для создания резервной копии проекта в компьютер приходилось вставлять одну из таких штук... Но это отличный совет! Visual Studio упрощает добавление проектов в систему управления версиями и поддержание их в актуальном состоянии, чтобы вы всегда могли вернуться к старой версии и просмотреть все изменения, внесенные с тех пор.



Добавление элемента TextBox в приложение

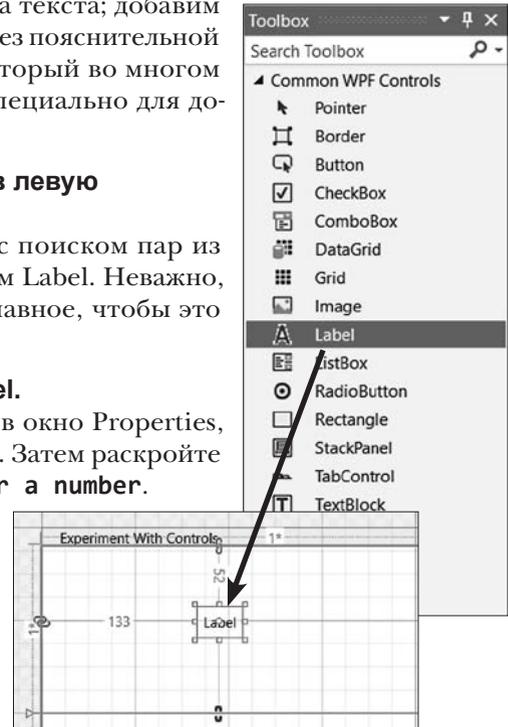
Элемент TextBox предоставляет пользователю поле для ввода текста; добавим его в ваше приложение. Но нам не хотелось бы иметь TextBox без пояснительной надписи, поэтому сначала будет добавлен элемент Label (который во многом похож на TextBox, не считая того, что он предназначен специально для добавления пояснений к другим элементам).

1 Перетащите элемент Label с панели инструментов в левую верхнюю ячейку сетки.

Именно так добавлялись элементы TextBox в игре с поиском пар из главы 1, но только на этот раз это делается с элементом Label. Неважно, в какую именно позицию ячейки вы ее перетащите, главное, чтобы это была левая верхняя ячейка.

2 Задайте размер текста и содержимое элемента Label.

Пока элемент Label остается выделенным, перейдите в окно Properties, раскройте раздел Text и выберите размер шрифта **18px**. Затем раскройте раздел Common и задайте свойству Content текст **Enter a number**.

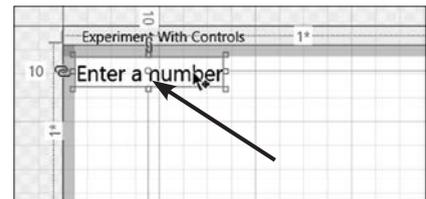


3 Перетащите элемент Label в левый верхний угол ячейки.

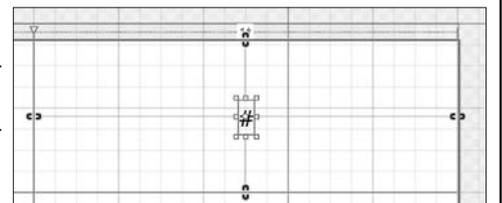
Щелкните на элементе Label в конструкторе и перетащите его в левый верхний угол. Когда он будет находиться в пределах 10 пикселей от левого или верхнего края ячейки, серые полосы исчезнут и элемент будет привязан к отступу размером 10 px.

В коде XAML должен появиться элемент Label:

```
<Label Content="Enter a number" FontSize="18"
      Margin="10,10,0,0"
      HorizontalAlignment="Left"
      VerticalAlignment="Top" />
```



В главе 1 мы добавили элементы TextBox во многие ячейки сетки и поместили в каждый из них символ ?. Также элементу Grid и каждому из элементов TextBox будет присвоено имя. Для этого проекта **добавьте один элемент TextBox**, присвойте ему имя **number**, назначьте текст # с размером шрифта **24px** и выровняйте его по центру правой верхней ячейки сетки.





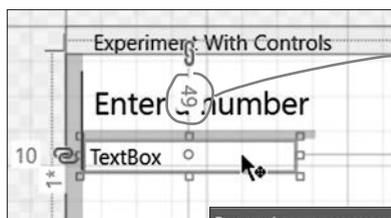
Ниже приведен код XAML элемента `TextBlock`, располагающегося в правой верхней ячейке сетки. Вы можете воспользоваться визуальным конструктором или ввести XAML вручную. Главное — проследите за тем, чтобы свойства вашего элемента `TextBlock` точно соответствовали свойствам в приведенном решении, но, как и прежде, у вас свойства могут следовать в другом порядке.

```
<TextBlock x:Name="number" Grid.Column="1" Text="#" FontSize="24"
  HorizontalAlignment="Center" VerticalAlignment="Center" TextWrapping="Wrap" />
```

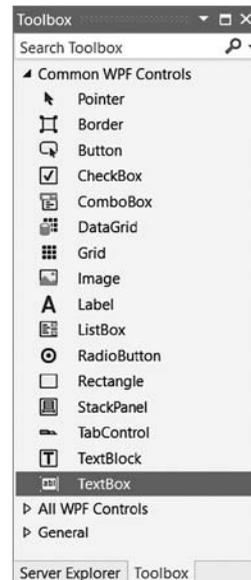
4

Перетащите элемент `TextBox` в левую верхнюю ячейку сетки.

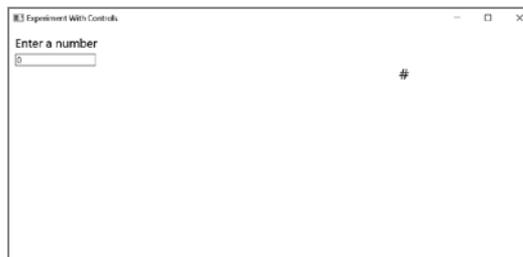
В вашем приложении элемент `TextBox` будет располагаться под `Label`, чтобы пользователь мог вводить числа. Перетащите его так, чтобы он располагался у левого края ячейки под `Label`, — появятся те же серые полосы для размещения его непосредственно под `Label` с левым отступом 10 px. Назначьте элементу имя `numberTextBox`, размер шрифта **18px** и текст **0**.



Когда вы используете серые полосы для размещения элемента управления, он фиксируется в позиции с отступом 10px под элементом, расположенным выше. Вы увидите, как левые и верхние отступы изменяются при перетаскивании.



Ваше окно сейчас должно выглядеть так:



А код XAML, появляющийся внутри `<Grid>` за определениями строки и столбца, но до `</Grid>`, должен выглядеть так:

```
<Label Content="Enter a number" FontSize="18" Margin="10,10,0,0"
  HorizontalAlignment="Left" VerticalAlignment="Top" />
```

```
<TextBox x:Name="numberTextBox" FontSize="18" Margin="10,49,0,0" Text="0" Width="120"
  HorizontalAlignment="Left" TextWrapping="Wrap" VerticalAlignment="Top" />
```

```
<TextBlock x:Name="number" Grid.Column="1" Text="#" FontSize="24"
  HorizontalAlignment="Center" VerticalAlignment="Center" TextWrapping="Wrap" />
```

Помните: свойства могут следовать в другом порядке или быть разбитыми на строки.

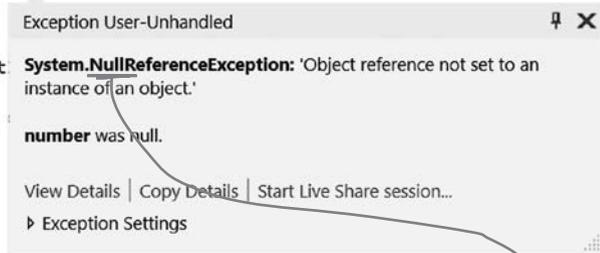
Теперь запустите приложение. Стоп! Что-то пошло не так — программа выдает исключение.

Хороший разработчик занимается не только написанием кода! Вы получили другое исключение, которое вам придется расследовать подобно тому, как это было сделано в главе 1. Выяснение причин и исправление подобных проблем также является очень важным навыком программиста.

```

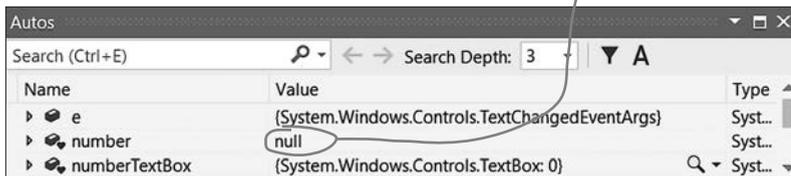
1 reference
private void numberTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    number.Text = numberTextBox.Text;
}

1 reference
private void numberTextBox_PreviewText...
{
    e.Handled = !int.TryParse(e.Text,
}
    
```



Взгляните в нижнюю часть IDE. В ней находится окно Autos, в котором выводятся все определенные переменные.

Для элемента TextBlock number выводится "null" — и это же слово присутствует в типе исключения NullReferenceException.



Что же происходит и, что еще важнее, как с этим справиться?

По следу

В результате перемещения тега TextBlock в XAML так, чтобы он располагался над TextBox, элемент TextBlock будет инициализироваться первым.

В окне Autos выводятся переменные, используемые в команде, которая выдала исключение: number и numberTextBox. Переменная numberTextBox содержит {System.Windows.Controls.TextBox: 0}, и именно так нормальный элемент TextBox должен выглядеть в отладчике. Но значение number — элемента TextBlock, в который должен копироваться текст, — равно null. Позднее вы узнаете, что означает null.

И это исключительно важная подсказка: IDE сообщает вам, что **элемент TextBlock number не инициализирован**.

Проблема в том, что код XAML для TextBox содержит команду Text="0", поэтому при запуске приложение инициализирует TextBox и пытается задать текст. При этом срабатывает обработчик события TextChanged, который пытается скопировать текст в TextBlock. Но ссылка на TextBlock все еще равна null, поэтому приложение выдает исключение.

Итак, чтобы исправить ошибку, необходимо позаботиться о том, чтобы элемент TextBlock инициализировался до TextBox. При запуске приложения WPF элементы **инициализируются в порядке их следования в XAML**. Следовательно, ошибку можно исправить **простым изменением порядка** элементов в XAML.

Поменяйте местами элементы TextBlock и TextBox, чтобы элемент TextBlock предшествовал TextBox:

```

<Label Content="Enter a number" ... />
<TextBlock x:Name="number" Grid.Column="1" ... />
<TextBox x:Name="numberTextBox" ... />
    
```

Выделите тег TextBlock в редакторе XAML и переместите его над TextBox, чтобы он инициализировался первым.

Приложение должно выглядеть в конструкторе точно так же, как прежде, и это логично, потому что оно содержит те же элементы управления. Теперь снова запустите свое приложение. На этот раз оно запустится, и элемент TextBox будет принимать только числовой ввод.

Добавление кода C# для обновления TextBlock

В главе 1 мы добавили **обработчики событий** (методы, которые вызываются при возникновении определенного **события**) для обработки щелчков мышью в игре с поиском пар. Теперь мы добавим обработчик события в код программной части, который вызывается каждый раз при вводе текста в TextBox и копирует этот текст в элемент TextBlock, добавленный в правую верхнюю ячейку в нашем мини-упражнении.

1 Сделайте двойной щелчок на элементе TextBlock для добавления метода.

Как только вы сделаете двойной щелчок на TextBox, IDE **автоматически добавляет метод-обработчик события C#**, связанный с событием TextChanged этого элемента. IDE генерирует пустой метод и присваивает ему имя, состоящее из имени элемента (numberTextBox), символа подчеркивания и имени обрабатываемого события — numberTextBox_TextChanged:

```
private void numberTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
}
}
```

2 Добавьте код в новый обработчик события TextChanged.

Каждый раз, когда пользователь вводит текст в TextBox, приложение должно скопировать его в элемент TextBlock, добавленный в правый верхний угол сетки. Так как элементу TextBlock было присвоено имя (number) и у TextBox оно тоже имеется (numberTextBox), для копирования содержимого элемента достаточно одной строки кода:

```
private void numberTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    number.Text = numberTextBox.Text;
}
}
```

Эта строка кода задает текст элемента TextBlock так, чтобы он совпадал с текстом элемента TextBox. Она вызывается каждый раз, когда пользователь изменяет текст в TextBox.

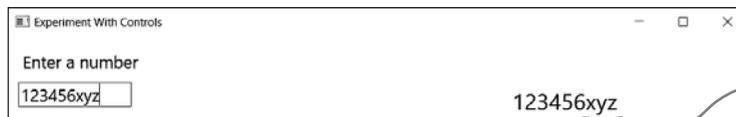
3 Запустите приложение и проверьте, как работает TextBox.

Запустите свое приложение при помощи кнопки Start Debugging (или командой Start Debugging (F5) из меню Debug), как это делалось в игре с поиском пар в главе 1. (Если появится панель инструментов времени выполнения, ее можно отключить, как было описано в главе 1.) Введите любое число в TextBox, и оно будет скопировано.



Когда вы вводите число в TextBox, обработчик события TextChange копирует его в TextBlock.

И все же что-то пошло не так — в TextBox можно ввести любой текст, не только числа!



Должен быть какой-то способ ограничить ввод только числовыми данными! Как вы думаете, что нужно сделать?

Когда вы делаете двойной щелчок на элементе TextBox, IDE добавляет обработчик события для события TextChanged. Этот обработчик вызывается каждый раз, когда пользователь изменяет текст в поле. Двойной щелчок на других типах элементов может добавлять другие обработчики событий, а в некоторых случаях (например, с TextBlock) вообще не добавляет никакие обработчики.

Добавление обработчика события, который разрешает вводить только числовые данные

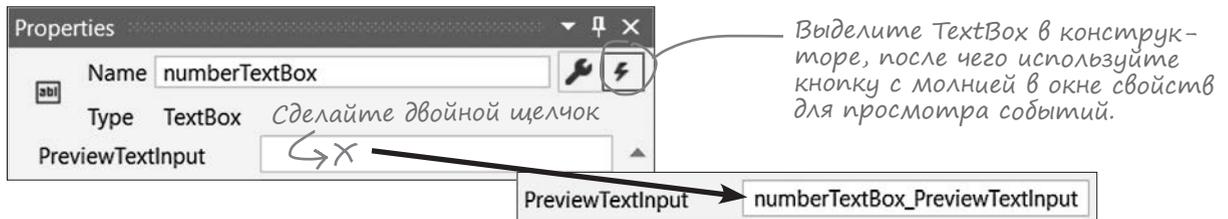
Чтобы добавить обработчик события `MouseDown` к элементу `TextBlock` в главе 1, мы воспользовались кнопкой в правом верхнем углу окна свойств для переключения между свойствами и событиями. Теперь мы сделаем то же самое, только на этот раз событие `PreviewTextInput` будет использоваться для принятия ввода, состоящего только из числовых данных, и отклонения всех остальных вводимых символов.

Если ваше приложение выполняется в настоящий момент, остановите его. Перейдите в отладчик, щелкните на элементе `TextBox`, чтобы выделить его, и перейдите в окно свойств, чтобы просмотреть его события. Прокрутите список и сделайте двойной щелчок в поле рядом с `PreviewTextInput`, чтобы IDE сгенерировала метод-обработчик события.

Кнопка с гаечным ключом в правом верхнем углу окна свойств выводит свойства выделенного элемента. Кнопка с молнией возвращается к выводу обработчиков событий.



Сделайте это!



Ваш новый обработчик события должен состоять из одной команды:

```
private void numberTextBox_PreviewTextInput(object sender, TextCompositionEventArgs e)
{
    e.Handled = !int.TryParse(e.Text, out int result);
}
```

Вы узнаете все об `int.TryParse` позднее в этой книге — пока просто введите код точно в таком виде, в каком он приведен здесь.

Этот обработчик события работает по следующей схеме:

1. Обработчик события вызывается, когда пользователь вводит текст в `TextBox`, но до обновления `TextBox`.
2. Специальный метод `int.TryParse` используется для проверки того, что введенный пользователем текст является числом.
3. Если пользователь ввел число, `e.Handled` присваивается `true`; тем самым вы указываете WPF игнорировать ввод.

Прежде чем выполнять код, вернитесь и просмотрите тег XAML для `TextBox`:

```
<TextBox x:Name="numberTextBox" FontSize="18" Margin="10,49,0,0" Text="" Width="120"
    HorizontalAlignment="Left" TextWrapping="Wrap" VerticalAlignment="Top"
    TextChanged="numberTextBox_TextChanged"
    PreviewTextInput="numberTextBox_PreviewTextInput" />
```

Теперь элемент связан с двумя обработчиками событий: событие `TextChanged` связывается с методом `numberTextBox_TextChanged`, а прямо под ним событие `PreviewTextInput` связывается с методом `numberTextBox_PreviewTextInput`.

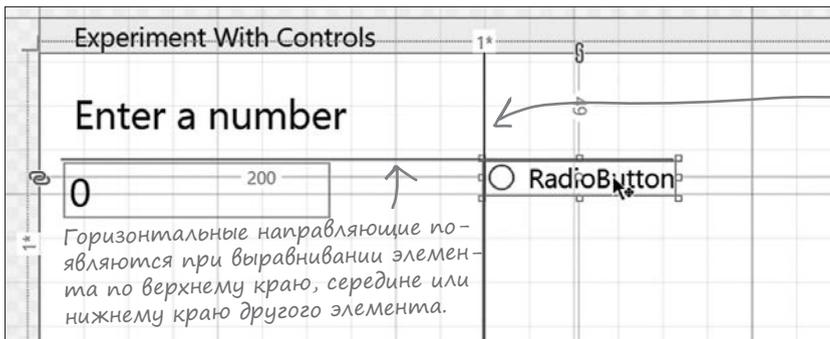


Упражнение

Добавьте остальные элементы XAML для приложения ExperimentWithControls: переключатели, список, две разновидности переключателей и два ползунка. Каждый элемент обновляет содержимое элемента TextBlock в правой верхней ячейке сетки.

Добавление переключателей в левую верхнюю ячейку рядом с TextBox

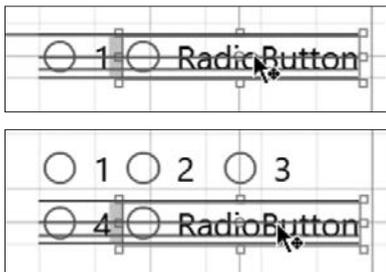
Перетащите элемент RadioButton с панели инструментов в левую верхнюю ячейку сетки. Перемещайте его, пока левый край не будет выровнен по центру ячейки, а верхний — по верхнему краю TextBox. Во время перетаскивания элементов в конструкторе появляются **направляющие**, которые помогают аккуратно выровнять элементы; положение элемента фиксируется по этим направляющим.



Вертикальная направляющая появляется в тот момент, когда левый край перетаскиваемого элемента выравнивается по центру ячейки.

Раскройте раздел Common окна свойств и задайте свойству Content элемента RadioButton значение 1.

Затем добавьте еще пять элементов RadioButton, выровняйте их и задайте их свойства Content. Но на этот раз не перетаскивайте их с панели инструментов. Вместо этого щелкните на RadioButton на панели инструментов, а затем щелкните внутри ячейки. (Дело в том, что если вы перетаскиваете элемент с панели инструментов в тот момент, когда выделен элемент RadioButton, IDE вложит новый элемент в RadioButton. Вложение элементов рассматривается позднее в книге.)



← При добавлении каждого переключателя можно использовать полосы и направляющие для выравнивания его по другим элементам.

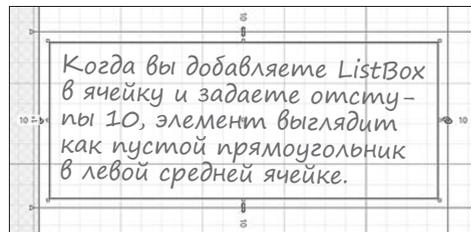
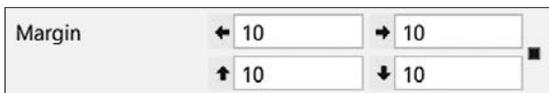
В окне свойств отображаются обработчики событий, а не свойства? Воспользуйтесь кнопкой



для вывода свойств; если вы использовали поле поиска, не забудьте очистить его содержимое.

Добавление списка в левую среднюю ячейку сетки

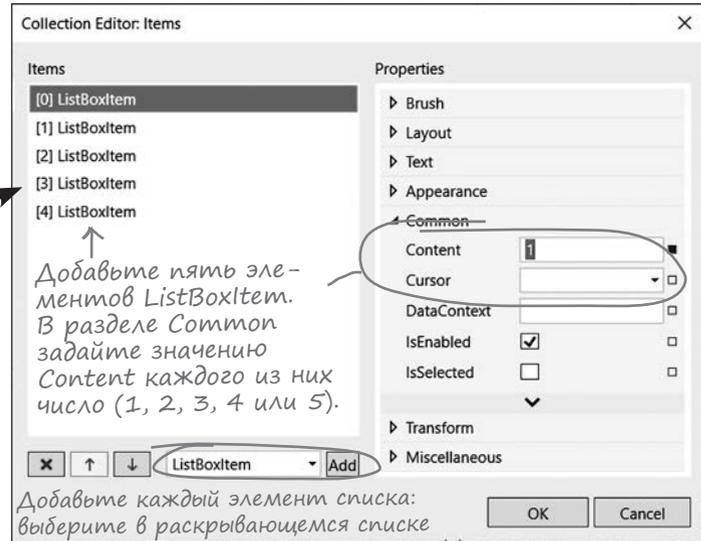
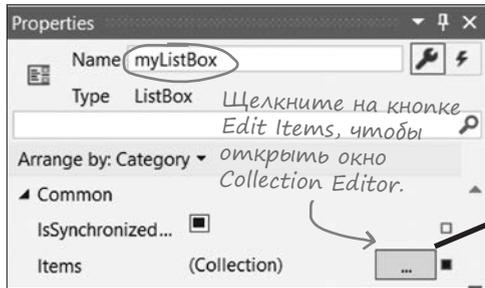
Щелкните на элементе ListBox панели инструментов, затем щелкните внутри левой средней ячейки для добавления элемента. В разделе Layout задайте всем отступам размер 10.





Присвойте элементу `ListBox` имя `myListBox` и добавьте в него элементы `ListBoxItem`

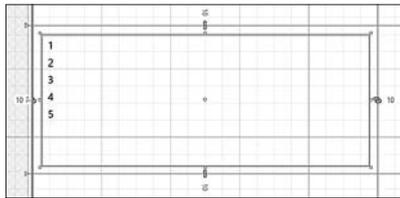
Элемент `ListBox` предназначен для выбора вариантов. Для этого в список необходимо добавить варианты. Выберите элемент `ListBox`, раскройте раздел `Common` в окне свойств и щелкните на кнопке **Edit Items** рядом с пунктом `Items` (...). **Добавьте пять элементов `ListBoxItem`** и присвойте их значениям `Content` числа от 1 до 5.



Добавьте пять элементов `ListBoxItem`. В разделе `Common` задайте значению `Content` каждого из них число (1, 2, 3, 4 или 5).

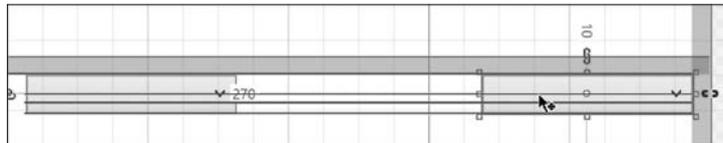
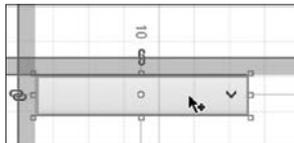
Добавьте каждый элемент списка: выберите в раскрывающемся списке `ListBoxItem` и щелкните на кнопке `Add`.

Элемент `ListBox` должен выглядеть так:



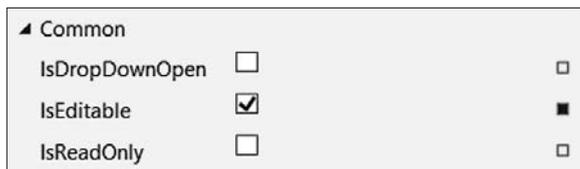
Добавьте два элемента `ComboBox` в левую среднюю ячейку сетки

Щелкните на кнопке элемента `ComboBox` на панели инструментов, затем щелкните в правой средней ячейке, чтобы **добавить элемент `ComboBox`**, и присвойте ему имя `readOnlyComboBox`. Перетащите элемент в левый верхний угол и при помощи серых полос назначьте ему левый и верхний отступ 10. Затем **добавьте еще один элемент `ComboBox`** с именем `editableComboBox` в ту же ячейку и выровняйте его по правому верхнему углу.



Воспользуйтесь окном `Collection Editor` для **добавления тех же элементов `ListBoxItem`** с числами 1, 2, 3, 4 и 5 в оба элемента `ComboBox` — сначала это следует проделать с первым элементом `ComboBox`, затем со вторым.

Наконец, **разрешите редактирование для элемента `ComboBox` справа** — раскройте раздел `Common` в окне свойств и установите флажок `IsEditable`. Теперь пользователь может ввести свое число в элементе `ComboBox`.



Редактируемый элемент `ComboBox` отличается от нередатируемого, чтобы пользователь знал, что он может либо ввести собственное значение, либо выбрать значение из списка.



Упражнение Решение

Ниже приведен код XAML для элементов `RadioButton`, `ListBox` и двух элементов `ComboBox`, добавленных в упражнение. Код XAML должен находиться в самом низу содержимого сетки — вы найдете эти строки рядом с закрывающим тегом `</Grid>`. Как и в другом коде XAML, который встречался вам ранее, в вашем коде свойства в теге могут следовать в другом порядке или разрывы строк могут находиться в других местах.

```
<RadioButton Content="1" Margin="200,49,0,0"
  HorizontalAlignment="Left" VerticalAlignment="Top"/>
<RadioButton Content="2" Margin="230,49,0,0"
  HorizontalAlignment="Left" VerticalAlignment="Top"/>
<RadioButton Content="3" Margin="265,49,0,0"
  HorizontalAlignment="Left" VerticalAlignment="Top"/>
<RadioButton Content="4" Margin="200,69,0,0"
  HorizontalAlignment="Left" VerticalAlignment="Top"/>
<RadioButton Content="5" Margin="230,69,0,0"
  HorizontalAlignment="Left" VerticalAlignment="Top"/>
<RadioButton Content="6" Margin="265,69,0,0"
  HorizontalAlignment="Left" VerticalAlignment="Top"/>
```

← IDE добавляет свойства, определяющие отступы и выравнивание, при перетаскивании каждого элемента `RadioButton`.

```
<ListBox x:Name="myListBox" Grid.Row="1" Margin="10,10,10,10">
  <ListBoxItem Content="1"/>
  <ListBoxItem Content="2"/>
  <ListBoxItem Content="3"/>
  <ListBoxItem Content="4"/>
  <ListBoxItem Content="5"/>
</ListBox>
```

Когда вы используете окно `Collection Editor` для добавления элементов `ListBoxItem` в элемент `ListBox` или `ComboBox`, оно создает закрывающий тег `</ListBox>` или `</ComboBox>` и добавляет теги `<ListBoxItem>` между открывающим и закрывающим тегами.

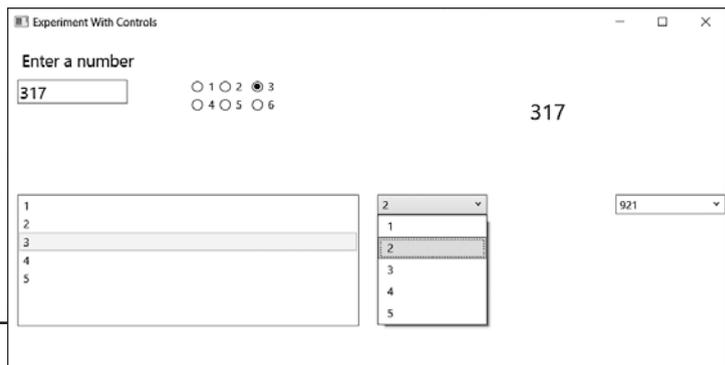
```
<ComboBox x:Name="readOnlyComboBox" Grid.Column="1" Margin="10,10,0,0" Grid.Row="1"
  HorizontalAlignment="Left" VerticalAlignment="Top" Width="120">
  <ListBoxItem Content="1"/>
  <ListBoxItem Content="2"/>
  <ListBoxItem Content="3"/>
  <ListBoxItem Content="4"/>
  <ListBoxItem Content="5"/>
</ComboBox>
```

Убедитесь в том, что элементу `ListBox` и двум элементам `ComboBox` присвоены правильные имена. Они будут использоваться в коде C#.

Два элемента `ComboBox` отличаются только свойством `IsEditable`.

```
<ComboBox x:Name="editableComboBox" Grid.Column="1" Grid.Row="1" IsEditable="True"
  HorizontalAlignment="Left" VerticalAlignment="Top" Width="120" Margin="270,10,0,0">
  <ListBoxItem Content="1"/>
  <ListBoxItem Content="2"/>
  <ListBoxItem Content="3"/>
  <ListBoxItem Content="4"/>
  <ListBoxItem Content="5"/>
</ComboBox>
```

Когда вы запускаете свою программу, она должна выглядеть примерно так. Вы можете использовать все элементы управления, но только `TextBox` обновляет значение наверху справа.



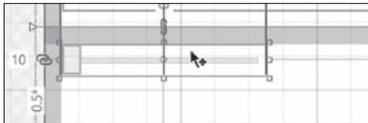
Чтобы найти элемент Slider на панели инструментов, необходимо раскрыть раздел All WPF Controls и прокрутить его почти до самого конца.

Добавление ползунков в нижнюю строку сетки

Добавим два ползунка в нижнюю строку, а затем назначим им обработчики событий, чтобы они обновляли элемент TextBox в правом верхнем углу.

1 Добавьте ползунок в приложение.

Перетащите элемент Slider с панели инструментов в правую нижнюю ячейку. Разместите его в левом верхнем углу ячейки и воспользуйтесь серыми полосами, чтобы установить для него левый и верхний отступ размером 10.



Используйте раздел Common окна свойств, чтобы присвоить AutoToolTipPlacement значение TopLeft, Maximum — значение 5 и Minimum — значение 1. Присвойте элементу имя smallSlider. Затем сделайте двойной щелчок на ползунке, чтобы добавить этот обработчик:

```
private void smallSlider_ValueChanged(
    object sender, RoutedEventArgs<double> e)
{
    number.Text = smallSlider.Value.ToString("0");
}
```

Значение элемента Slider представляет собой дробное число. Этот символ «0» преобразует его в целое число.

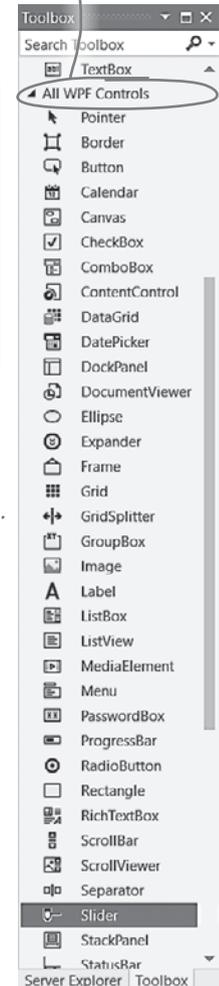
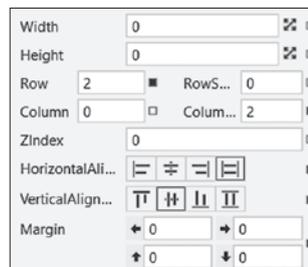
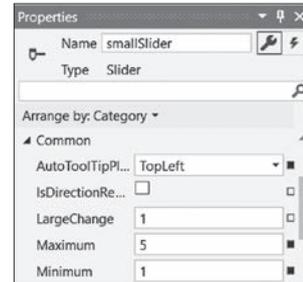
2 Добавьте ползунок для выбора телефонных чисел.

Есть старая поговорка: «Даже если идея ужасная и, возможно, дурацкая, это еще не значит, что от нее нужно отказаться». Так что мы сделаем нечто такое, что выглядит довольно глупо: добавим ползунок для выбора телефонных номеров.

Перетащите другой ползунок в нижнюю строку. Используйте раздел Layout окна свойств, чтобы сбросить его ширину, задайте свойству ColumnSpan значение 2, задайте всем отступам значение 10, выберите вертикальное выравнивание Center и горизонтальное выравнивание Stretch. Затем в разделе Common задайте свойству AutoToolTipPlacement значение TopLeft, свойству Minimum — значение 11111111, свойству Maximum — значение 999999999 и свойству Value — значение 7183876962. Присвойте элементу имя bigSlider, затем сделайте на нем двойной щелчок и добавьте следующий обработчик события ValueChanged:

```
private void bigSlider_ValueChanged(
    object sender, RoutedEventArgs<double> e)
{
    number.Text = bigSlider.Value.ToString("000-000-0000");
}
```

Нули и дефисы нужны для того, чтобы метод преобразовывал любое число из 10 знаков в формат телефонного номера США.



Добавление кода C#, обеспечивающего работу элементов управления

Каждый элемент управления в нашем приложении должен делать одно и то же: обновлять TextBlock в правом верхнем углу числом, так что при установке переключателя или выборе элемента из ListBox или ComboBox элемент TextBlock обновляется выбранным значением.

1 Добавьте обработчик события Checked к первому элементу управления RadioButton.

Сделайте двойной щелчок на первом элементе RadioButton. IDE добавляет новый метод-обработчик события с именем RadioButton_Checked (так как вы еще не присвоили элементу имя, для генерирования метода используется тип элемента). Добавьте следующую строку кода:

```
private void RadioButton_Checked(
    object sender, RoutedEventArgs e)
{
    if (sender is RadioButton radioButton) {
        number.Text = radioButton.Content.ToString();
    }
}
```

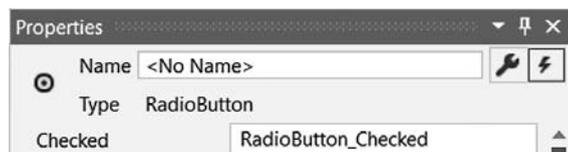
Проверить
код



Эта команда использует ключевое слово `is`, о котором вы узнаете в главе 7. А пока просто аккуратно введите команду точно так, как она приведена на странице (и сделайте то же самое для остальных методов-обработчиков).

2 Назначьте тот же обработчик события другим элементам RadioButton.

Внимательно присмотритесь к коду XAML только что измененного элемента RadioButton. IDE добавляет свойство `Checked="RadioButton_Checked"` — именно так к элементу подключались другие обработчики событий. **Скопируйте свойство в другие теги RadioButton**, чтобы они имели одинаковые свойства `Checked`, — *в результате все они связываются с одним обработчиком события Checked*. Вы можете воспользоваться режимом просмотра событий в окне свойств, чтобы убедиться в том, что обработчики всех элементов RadioButton были назначены правильно.



Если переключить окно свойств в режим событий, вы можете выбрать любой из элементов RadioButton и убедиться в том, что у всех элементов событие `Checked` связано с обработчиком события `RadioButton_Checked`.

3 Заставьте ListBox обновлять TextBlock в правой верхней ячейке.

Выполняя упражнение, вы присвоили своему элементу ListBox имя `myListBox`. Теперь добавим обработчик события, который будет срабатывать каждый раз, когда пользователь выбирает элемент в списке и использует имя для получения выбранного числа.

Сделайте двойной щелчок внутри пустого пространства в ListBox под элементами, чтобы IDE добавила метод-обработчик для события `SelectionChanged`. Добавьте следующую команду:

```
private void myListBox_SelectionChanged(
    object sender, SelectionChangedEventArgs e)
{
    if (myListBox.SelectedItem is ListBoxItem listBoxItem) {
        number.Text = listBoxItem.Content.ToString();
    }
}
```

Проследите за тем, чтобы щелчок был сделан в пустом пространстве под элементами списка. Если вы щелкнете на элементе списка, то обработчик будет добавлен только для этого элемента, а не для всего элемента ListBox.

4 Заставьте поле со списком, доступное только для чтения, обновлять TextBox.

Сделайте двойной щелчок на элементе ComboBox, доступном только для чтения, чтобы среда Visual Studio добавила обработчик для события SelectionChanged, которое выдается при выборе нового значения в ComboBox. Ниже приведен код — он очень похож на код ListBox:

```
private void readOnlyComboBox_SelectionChanged(
    object sender, SelectionChangedEventArgs e)
{
    if (readOnlyComboBox.SelectedItem is ListBoxItem listBoxItem)
        number.Text = listBoxItem.Content.ToString();
}
```

Вы также можете воспользоваться окном свойств для добавления события SelectionChanged. Если вы случайно сделаете это, нажмите кнопку отмены (но проследите за тем, чтобы это было сделано в обоих файлах).

5 Заставьте редактируемое поле со списком обновлять TextBox.

Редактируемое поле со списком напоминает гибрид ComboBox и TextBox. Вы можете выбирать значения из списка, но также можете ввести собственный текст. Так как элемент работает как TextBox, стоит добавить обработчик события PreviewTextInput и ограничить ввод числами, как это было сделано для TextBox. Собственно, можно даже **повторно использовать обработчик**, уже добавленный для TextBox.

Перейдите к коду XAML редактируемого элемента ComboBox, установите курсор непосредственно перед закрывающей угловой скобкой > и начните **вводить имя PreviewTextInput**. На экране появляется окно IntelliSense, которое помогает завершить имя события. Затем **добавьте знак =** — как только вы это сделаете, IDE предложит добавить новый обработчик или выбрать уже добавленный. Выберите существующий обработчик события.

```
<ComboBox x:Name="editableComboBox" Grid.Column="1" Grid.Row="1" IsEditable="True"
    HorizontalAlignment="Left" VerticalAlignment="Top" Width="120" Margin="270,10,0,0"
    PreviewTextInput="" >
    <New Event Handler>
    numberTextBox_PreviewTextInput
```

Предыдущие обработчики событий использовали элементы списка для обновления TextBox. Однако в ComboBox пользователь может ввести произвольный текст, поэтому на этот раз мы добавим **другой обработчик события**.

Снова отредактируйте код XAML и добавьте новый тег под ComboBox. На этот раз введите TextBoxBase.; как только вы введете точку, автозамена предложит возможные варианты. Выберите TextBoxBase.TextChanged и введите знак =. Выберите в раскрывающемся списке <New Event Handler>.

```
TextBoxBase.>
    SelectionChanged
    TextChanged
    TextBoxBase.TextChanged="" >
    <New Event Handler>
    numberTextBox_TextChanged
```

IDE добавляет новый обработчик события в код программной части. Он выглядит так:

```
private void editableComboBox_TextChanged(object sender, TextChangedEventArgs e)
{
    if (sender is ComboBox comboBox)
        number.Text = comboBox.Text;
}
```

Теперь запустите свою программу. Все элементы должны работать. Превосходно!



Столько разных способов выбора чисел!
У меня появляется НЕВЕРОЯТНО много вариантов
при разработке приложения.

Элементы управления дают вам гибкость для того, чтобы упростить жизнь пользователей.

Когда вы строите UI для приложения, вам приходится принимать много разных решений: какие элементы использовать, где разместить каждый элемент, что делать с входными данными. Выбор того или иного элемента дает неявный сигнал относительно того, как пользоваться вашим приложением. Например, при виде набора переключателей вы знаете, что нужно выбрать одно значение из небольшого набора, тогда как редактируемое поле со списком говорит о том, что выбор практически не ограничен. Не думайте, что задачей проектирования UI является поиск «правильных» решений вместо «неправильных». Вместо этого считайте, что вы должны упростить жизнь вашим пользователям, насколько это возможно.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Приложение состоит из **классов**, классы содержат **методы**, а методы состоят из **команд**.
- Каждый класс принадлежит некоторому **пространству имен**. Некоторые пространства имен (например, `System.Collections.Generic`) содержат классы `.NET`.
- Классы могут содержать **поля**, которые существуют за пределами методов. Разные методы могут работать с одним полем.
- Если метод помечается ключевым словом **public**, это означает, что он может вызываться из других классов.
- **Консольные приложения .NET Core** представляют собой кроссплатформенные программы, не имеющие графического интерфейса.
- IDE **строит** ваш код, чтобы преобразовать его в **двоичный файл**, т. е. файл, который можно запустить для выполнения.
- Кроссплатформенное консольное приложение `.NET Core` можно преобразовать программой командной строки `dotnet`, чтобы **построить двоичные файлы** для разных операционных систем.
- Метод **`Console.WriteLine`** выводит строку на консоль.
- Переменные должны быть **объявлены**, прежде чем их можно будет использовать в программе. При объявлении переменной можно присвоить исходное значение.
- Отладчик Visual Studio позволяет **приостановить приложение** и проанализировать значения переменных.
- Элементы управления **выдают события** для многих изменений: щелчков мыши, изменений выделения, ввода текста и т. д. Иногда говорят, что события **иницируются** или **генерируются**, — это то же самое.
- **Обработчики событий** — методы, которые вызываются при выдаче события для реакции на это событие (обработки).
- Элементы `TextBox` могут использовать событие **`PreviewTextInput`** для подтверждения или отклонения введенного текста.
- **Ползунки** отлично подходят для ввода числовых данных, но для выбора телефонных номеров их лучше не использовать.

Лабораторный курс Unity № 1

Исследование C# с Unity

Добро пожаловать на первый урок «Лабораторный курс Unity». Написание кода – навык, и, как и любой другой навык, он развивается за счет **практики** и **экспериментирования**. И в этом отношении Unity может стать очень полезным инструментом.

Unity – кроссплатформенный инструмент разработки игр, который может использоваться для создания игр профессионального уровня, симуляторов и многих других приложений. Также Unity предоставляет интересные и приятные возможности **потренироваться в применении средств и идей C#**, представленных в книге. Мы разработали эти короткие, целенаправленные лабораторные работы для **закрепления** только что описанных концепций и приемов, которые помогут вам отточить навыки C#.

Лабораторные работы необязательны, но они станут полезной практикой, **даже если вы не планируете использовать C# для построения игр**.

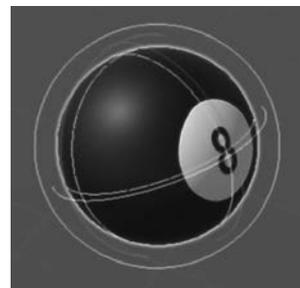
В первой лабораторной работе мы введем вас в курс дела. Вы начнете ориентироваться в редакторе Unity, а также создавать 3D-объекты и оперировать ими.

Unity — мощный инструмент для разработки игр

Добро пожаловать в мир Unity — полнофункциональной системы для создания игр профессионального уровня (как двумерных (2D), так и трехмерных (3D)), а также моделирования, разработки и проектирования. Unity включает ряд мощных функций, в том числе...

Кроссплатформенный игровой движок

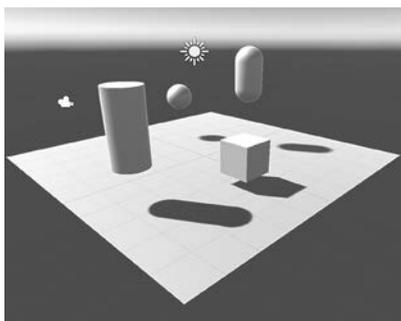
Игровой движок обеспечивает отображение графики, управление 2D- или 3D-персонажами, обнаружение столкновений, моделирование поведения реальных физических объектов и многое, многое другое. Unity реализует все эти возможности для 3D-игр, которые будут построены в книге.



Мощный редактор сцен

Вы проведете немало времени в редакторе Unity. Он позволяет редактировать уровни, наполненные 2D- и 3D-фигурами, а также предоставляет средства для построения полноценных игровых миров в ваших играх. Игры Unity используют C# для определения своего поведения, а редактор Unity интегрируется с Visual Studio, чтобы предоставить в ваше распоряжение эффективную среду разработки игр.

Хотя «Лабораторные работы Unity» направлены на разработку C# в Unity, если вы являетесь дизайнером или специалистом по компьютерной графике, редактор Unity содержит много средств, предназначенных для вас. С ними можно ознакомиться по адресу <https://unity3d.com/unity/features/editor/art-and-design>.



Экосистема для создания игр

Кроме невероятно мощных средств для создания игр, Unity также представляет экосистему, которая поможет вам в изучении и построении приложений. На странице Learn Unity (<https://unity.com/learn>) приведены полезные учебные ресурсы для самостоятельного изучения, а форумы Unity (<https://forum.unity.com>) позволят связаться с другими разработчиками игр и задать им вопросы. Unity Asset Store (<https://assetstore.unity.com>) предоставляет как платные, так и бесплатные ресурсы (персонажи, фигуры и эффекты), которые вы можете использовать в своих проектах Unity.

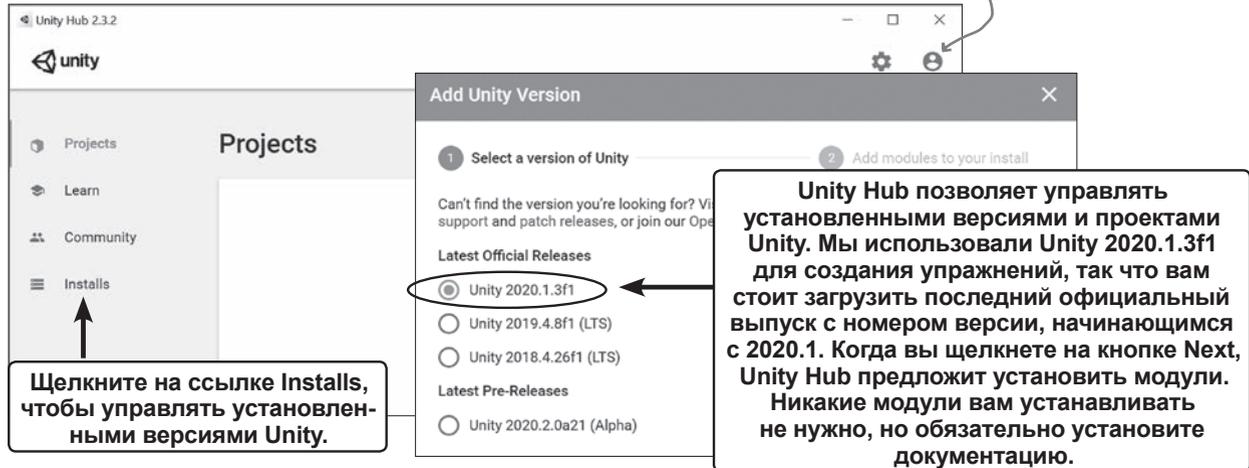
В лабораторной работе Unity мы будем рассматривать Unity как средство для изучения C#, а также для практического применения инструментов и идей C#, о которых вы узнали в книге.

Учебный стиль «Лабораторных работ Unity» полностью ориентирован на разработчика. Их цель — помочь вам как можно быстрее войти в курс дела. При этом мы уделяем первоочередное внимание легкости изложения, которое применяется в книге, чтобы вы прошли целенаправленную и эффективную тренировку в применении концепций и приемов C#.

Загрузка Unity Hub

Unity Hub — приложение для управления проектами Unity и установленными экземплярами Unity, которое также становится отправной точкой для создания нового проекта Unity. Начните с загрузки Unity Hub по адресу <https://store.unity.com/download>, затем установите и запустите программу.

Все снимки экрана в этой книге были сделаны в бесплатном издании Unity Personal Edition. Вы должны ввести в Unity Hub имя пользователя и пароль своей учетной записи unity.com, чтобы активировать лицензию.



Unity Hub позволяет управлять несколькими версиями Unity на одном компьютере, поэтому вам следует установить ту же версию, которая использовалась для построения лабораторных работ. Щелкните на ссылке Official Releases и установите последнюю версию, которая начинается с Unity 2020.1, — это та же версия, которая использовалась для создания снимков экрана в лабораторных работах. После того как версия будет установлена, проследите за тем, чтобы она была назначена приоритетной.

Программа установки Unity может предложить установить другую версию Visual Studio. На одном компьютере также допускается установка нескольких версий Visual Studio, но если у вас уже установлена одна версия Visual Studio, добавлять другую из программы установки Unity не нужно.

За дополнительной информацией об установке Unity Hub в Windows, macOS и Linux обращайтесь по адресу <https://docs.unity3d.com/2020.1/Documentation/Manual/GettingStartedInstallingHub.html>.

Unity Hub позволяет установить несколько версий Unity на одном компьютере. Таким образом, даже если у вас доступна новая версия Unity, вы можете воспользоваться Unity Hub для установки версии, которую мы использовали в лабораторных работах.



Будьте осторожны!

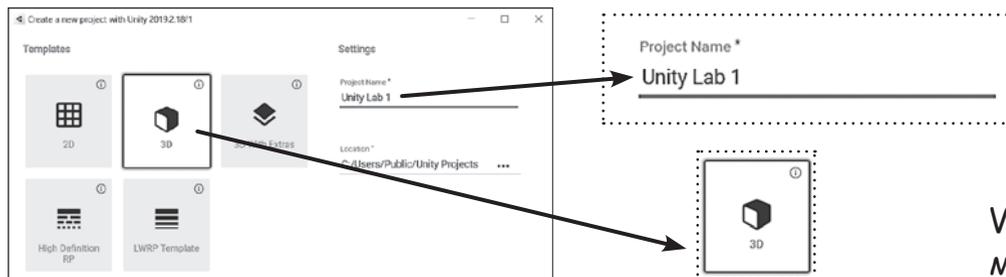
Unity Hub может выглядеть немного по-другому.

Снимки экрана, приведенные в книге, были сделаны в Unity 2020.1 (Personal Edition) и Unity Hub 2.3.2. При помощи Unity Hub можно установить несколько разных версий Unity на одном компьютере, но установить можно только новейшую версию Unity Hub. Группа разработки Unity постоянно улучшает Unity Hub и редактор Unity, и может оказаться, что увиденное вами будет отличаться от

иллюстраций на этой странице. Мы будем обновлять «Лабораторные работы Unity» для новых изданий «Head First C#». PDF-файлы обновленных лабораторных работ будут доступны на нашей странице GitHub: <https://github.com/head-first-csharp/fourth-edition>.

Использование Unity Hub для создания нового проекта

Щелкните на кнопке **NEW** на странице Project в Unity Hub для создания нового проекта Unity. Пристройте ему имя **Unity Lab 1**, убедитесь в том, что выбран шаблон 3D, а проект создается в подходящем месте (обычно в папке Unity Projects в домашнем каталоге).



Щелкните на ссылке Create Project, чтобы создать новую папку с проектом Unity. При создании нового проекта Unity генерирует множество файлов (как и при создании новых проектов в Visual Studio). Для создания всех файлов нового проекта Unity может потребоваться одна-две минуты.

Выбор Visual Studio в качестве редактора сценариев Unity

Редактор Unity тесно взаимодействует с Visual Studio IDE, чтобы упростить редактирование и отладку кода ваших игр. Таким образом, первым делом мы примем меры к тому, чтобы связать Unity с Visual Studio. **Выберите команду Preferences из меню Edit** (или из меню Unity на Mac), чтобы открыть окно Unity Preferences. Щелкните на категории External Tools на левой панели и **выберите Visual Studio** в окне External Script Editor.

*В некоторых старых версиях Unity может присутствовать флажок **Editor Attaching** — в таком случае проследите за тем, чтобы он был установлен (это позволит вам проводить отладку кода Unity в IDE).*

Visual Studio может использоваться для отладки кода Unity. Для этого достаточно выбрать Visual Studio в качестве внешнего редактора сценариев в настройках Unity.



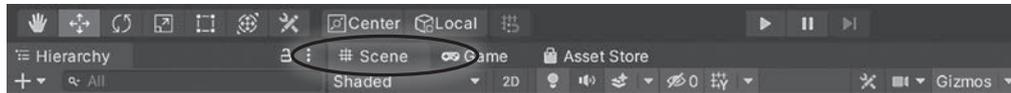
Если вы не видите Visual Studio в раскрывающемся списке External Script Editor, выберите Browse... и перейдите к Visual Studio. В системе Windows это обычно файл devenv.exe из папки C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\Common7\IDE\. На Mac это обычно приложение Visual Studio из папки Applications.

Отлично! Все готово для построения вашего первого проекта Unity.

Управление макетом Unity

Редактор Unity представляет собой аналог IDE для всех частей проекта Unity, которые не являются кодом C#. Он будет использоваться для работы со сценами, редактирования трехмерных объектов, создания материалов и т. д. Как и в Visual Studio, окна и панели редактора Unity можно переупорядочить, создавая разные варианты макетов окна.

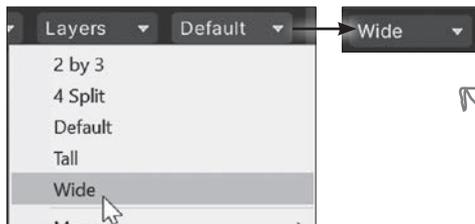
Найдите вкладку Scene в верхней части окна. Щелкните на вкладке и перетащите ее, чтобы открепить окно:



Попробуйте закрепить его внутри или рядом с другими панелями, а затем перетащите его во внутреннюю часть редактора, чтобы окно оставалось плавающим.

Выберите макет Wide

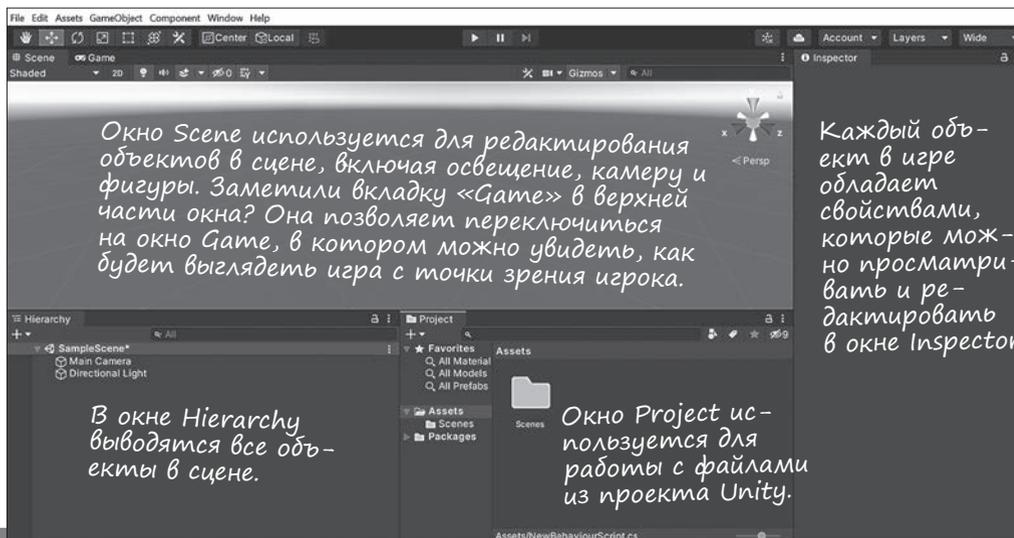
Мы выбрали макет Wide, потому что он хорошо подходит для снимков экрана в этих лабораторных работах. Найдите раскрывающийся список и выберите команду Wide, чтобы ваш редактор Unity выглядел так же, как на наших иллюстрациях.



После того как вы измените макет при помощи раскрывающегося списка Layout справа от панели инструментов, метка раскрывающегося списка может измениться в соответствии с выбранным макетом.

Представление Scene — основное интерактивное представление создаваемого вами мира. Оно используется для размещения 3D-фигур, камер, источников света и всех остальных объектов в игре.

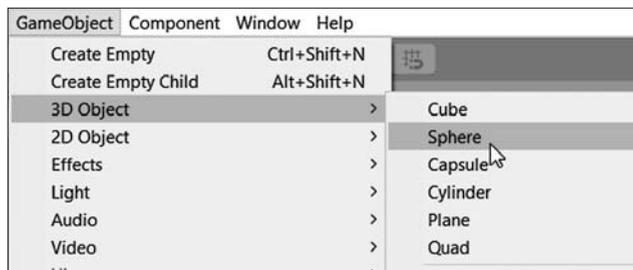
В макете Wide редактор Unity должен выглядеть так:



Сцена как 3D-среда

Сразу же после запуска редактора вы оказываетесь в режиме редактирования **сцены**. Сцены можно рассматривать как уровни в играх Unity. Каждая игра Unity состоит из одной или нескольких сцен. Каждая сцена содержит отдельную 3D-среду с собственным набором источников света, фигур и других 3D-объектов. Когда вы создаете проект, Unity добавляет сцену с именем SampleScene и сохраняет ее в файле с именем SampleScene.unity.

Добавьте в сцену сферу командой меню **GameObject>>3D Object>>Sphere**.



Так называемые примитивные объекты Unity. Мы будем часто использовать их в лабораторных работах Unity.

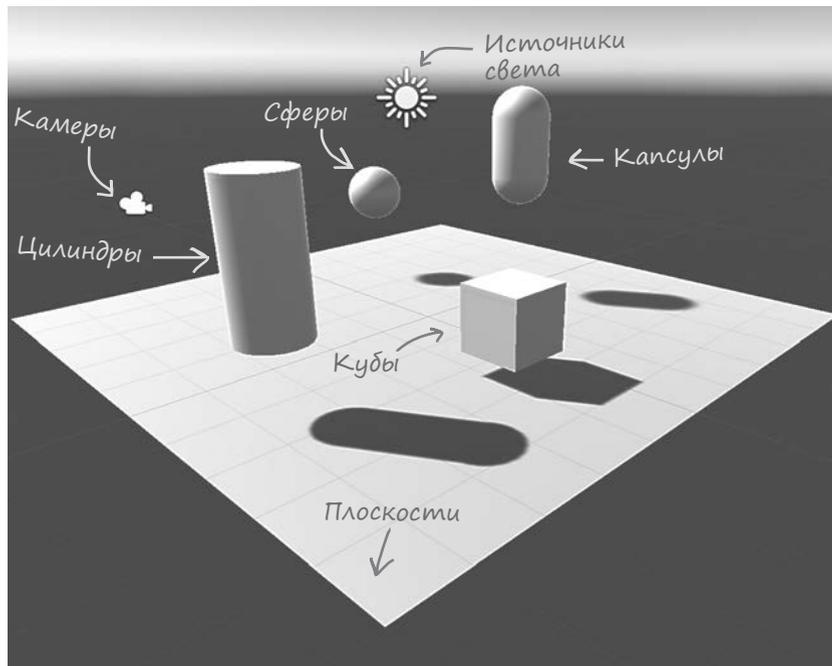
В окне Scene появляется сфера. Все, что вы видите в окне Scene, показывается с точки зрения **камеры Scene**, которая «рассматривает» сцену и воспроизводит увиденное.



Игры Unity состоят из объектов GameObject

Когда вы добавили сферу в свою сцену, тем самым вы создаете новый объект **GameObject**. Объекты **GameObject** являются одной из фундаментальных концепций Unity. Каждый предмет, фигура, персонаж, источник света, камера и специальный эффект в игре Unity является объектом **GameObject**. Все декорации, персонажи и элементы окружения в игре представляются объектами **GameObject**.

В лабораторных работах Unity мы будем строить разные виды объектов **GameObject**, включая:



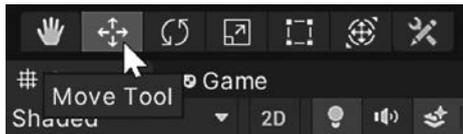
GameObject — фундаментальная разновидность объектов в Unity, а **компоненты** являются основными структурными элементами их поведения. В окне Inspector выводится информация о каждом объекте **GameObject** в сцене и его компонентах.

Каждый объект **GameObject** состоит из нескольких компонентов, которые определяют его форму, задают его позицию и наделяют его поведением. Пример:

- ★ *Компоненты преобразований* определяют позицию и угол поворота **GameObject**.
- ★ *Компоненты материалов* изменяют способ **визуализации** **GameObject**, т. е. способ его прорисовки Unity, за счет изменения цвета, отражений, уровня гладкости и т. д.
- ★ *Компоненты сценариев* используют сценарии Unity для определения поведения **GameObject**.

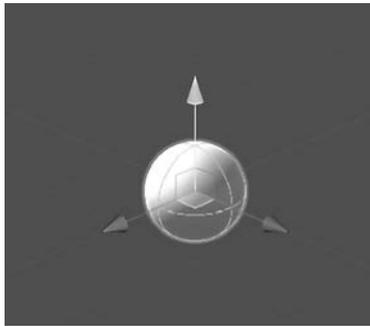
Использование инструмента Move для перемещения объектов GameObject

Панель инструментов в верхней части редактора Unity позволяет выбрать инструменты Transform. Если инструмент Move не выбран, выберите его нажатием соответствующей кнопки.



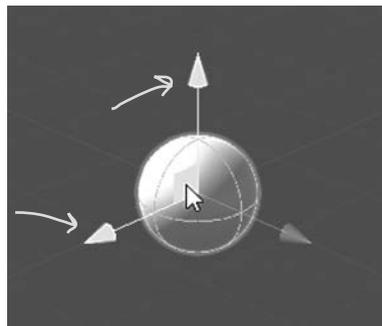
← Кнопка в левой части панели инструментов позволяет выбирать такие инструменты преобразования, как инструмент Move, который выводит манипулятор Move в виде стрелок и куба поверх текущего объекта GameObject.

Инструмент Move предоставляет возможность перемещать объекты GameObject в трехмерном пространстве при помощи **манипулятора Move**. В окне появляются три стрелки (красная, зеленая и синяя), а также куб. Это манипулятор Move, который может использоваться для перемещения выделенного объекта по сцене.



Перемещайте указатель мыши над кубом в центре манипулятора Move — заметили, как каждая из граней куба подсвечивается при перемещении над ней указателя мыши? Щелкните на левой верхней грани и перетащите сферу. Сфера будет перемещаться в плоскости X-Y.

Когда вы щелкаете на левой верхней грани куба в середине манипулятора Move, стрелки X и Y подсвечиваются, и вы можете перетаскивать сферу в плоскости X-Y вашей сцены.



Манипулятор Move позволяет перемещать объекты GameObject вдоль любой оси или любой плоскости трехмерного пространства вашей сцены.

Попробуйте перемещать сферу по сцене, чтобы получить представление о том, как работает манипулятор Move. Щелкните на каждой грани куба и перетащите объект по всем трем плоскостям. Обратите внимание на то, как сфера уменьшается при отдалении от вас (а на самом деле от камеры сцены) и увеличивается при приближении.

В окне Inspector выводятся компоненты GameObject

Во время перемещения сферы в трехмерном пространстве наблюдайте за **окном Inspector**, расположенным в правой части окна Unity (при использовании макета Wide). Просмотрите содержимое окна Inspector — вы увидите, что сфера состоит из четырех компонентов с именами Transform, Sphere (Mesh Filter), Mesh Renderer и Sphere Collider.

Каждый объект GameObject состоит из набора компонентов, которые предоставляют основные структурные блоки его поведения; кроме того, каждый объект GameObject содержит **компонент Transform**, который управляет его местонахождением, поворотом и масштабированием.

Чтобы понаблюдать за компонентом Transform в действии, воспользуйтесь манипулятором Move для перемещения сферы в плоскости X–Y. Проследите за тем, как при перемещении сферы изменяются числа X и Y в строке Position компонента Transform.

Если вы случайно снимете выделение с объекта GameObject, просто щелкните на нем повторно. Если объект не виден в сцене, его можно выделить в окне Hierarchy, в котором перечислены все объекты GameObject в сцене. Когда вы переключаетесь на макет Wide, окно Hierarchy располагается в левом нижнем углу редактора Unity.



А вы заметили сетку в трехмерном пространстве? Удерживайте клавишу Ctrl во время перетаскивания сферы. В этом случае перемещаемый объект GameObject привязывается к сетке. Вы увидите, что числа в компоненте Transform изменяются с целыми приращениями (вместо малых дробных приращений).

Попробуйте пощелкать на двух других гранях куба манипулятора Move, чтобы перемещать сферу в плоскостях X–Z и Y–Z. Затем щелкайте на красной, зеленой и синей стрелках и перетаскивайте сферу вдоль осей X, Y и Z. Вы увидите, что значения X, Y и Z в компоненте Transform изменяются при перемещении сферы.

Теперь **нажмите клавишу Shift**, чтобы превратить куб в середине манипулятора в квадрат. Щелкните на квадрате и перетащите указатель мыши, чтобы переместить сферу в плоскости, параллельной камере сцены.

После того как вы вдоволь поэкспериментируете с манипулятором Move, воспользуйтесь контекстным меню компонента Transform для сброса компонента к значениям по умолчанию. Щелкните на кнопке контекстного меню  в верхней части панели Transform и выберите из меню команду Reset.



Воспользуйтесь контекстным меню для сброса компонента. Чтобы вызвать контекстное меню, щелкните либо на кнопке с тремя точками, либо правой кнопкой мыши в любой точке верхней строки панели Transform в окне Inspector.

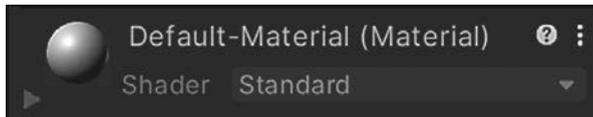
Позиция сферы возвращается к координатам [0, 0, 0].

Добавление материала к объекту GameObject

Unity использует материалы для предоставления цветов, узоров, текстур и других визуальных эффектов. Ваша сфера сейчас выглядит довольно уныло — трехмерный объект отображается в простом белом цвете. Попробуем придать ей вид бильярдного шара.

1 Выделите сферу.

Когда сфера будет выделена, ее материал отображается в виде компонента в окне Inspector:



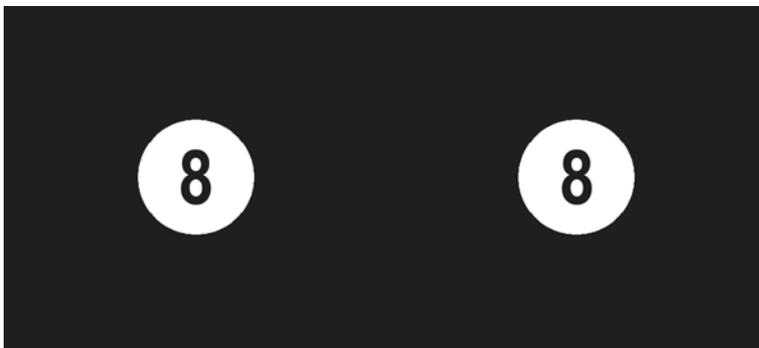
Чтобы сфера выглядела более интересно, добавим **текстуру** — простой графический файл, который накладывается на трехмерный объект (так, как если бы вы напечатали изображение на листе резины и завернули в него объект).

2 Перейдите на нашу страницу текстур на GitHub.

Перейдите по адресу <https://github.com/head-first-csharp/fourth-edition> и щелкните на ссылке *Billiard Ball Textures*, чтобы просмотреть папку, содержащую полный набор файлов текстур бильярдного шара.

3 Загрузите текстуру бильярдного шара с «восьмеркой».

Щелкните на файле *8 Ball Texture.png*, чтобы просмотреть текстуру шара с «восьмеркой». Это обычный графический файл 1200 × 600 в формате PNG, который вы можете открыть в своей любимой программе просмотра графических файлов.



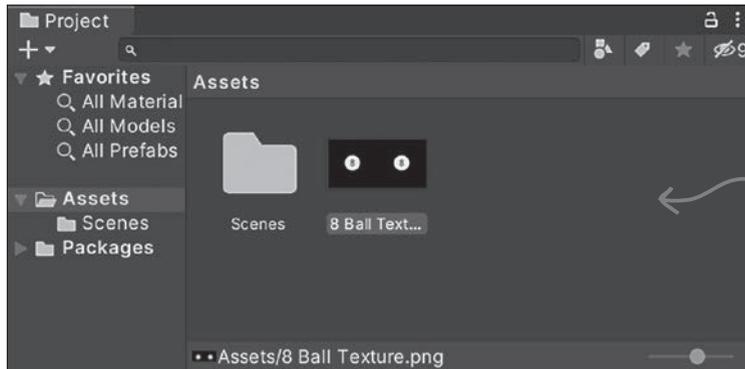
← Мы спроектировали этот графический файл так, чтобы он был похож на бильярдный шар с «восьмеркой», когда Unity «заворачивает» в него сферу.

Загрузите файл в папку на своем компьютере.

*(Чтобы сохранить файл, щелкните правой кнопкой мыши на кнопке *Download* или воспользуйтесь кнопкой *Download*, чтобы открыть файл, а затем сохранить его, — конкретный способ зависит от браузера.)*

④ **Импортируйте текстуру в свой проект Unity.**

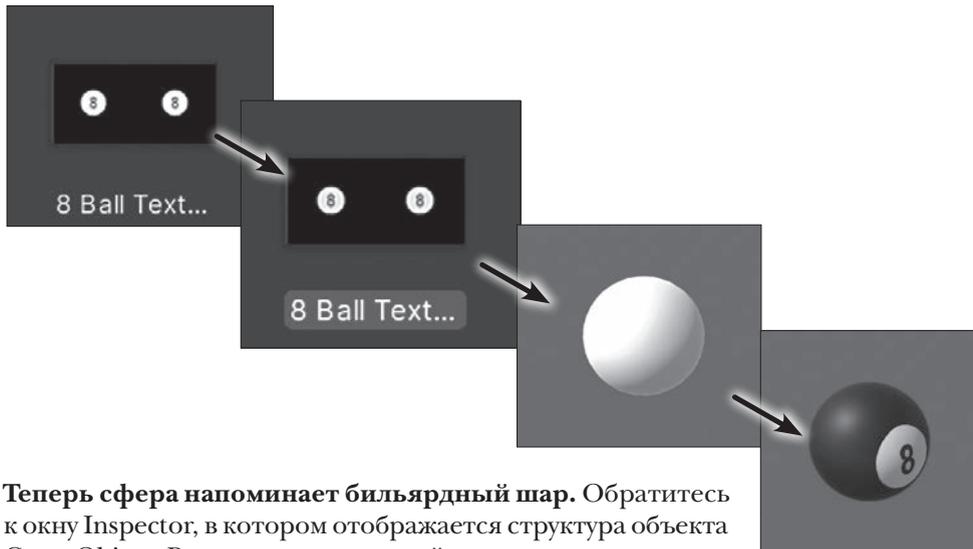
Щелкните правой кнопкой мыши на папке Assets в окне Project, выберите команду **Import New Asset...** и импортируйте файл текстуры. Теперь текстура должна появляться в списке, когда вы щелкаете на папке Assets в окне Project.



Чтобы импортировать новый ресурс, вы щелкнули правой кнопкой мыши на папке Assets в окне Project, поэтому текстура будет добавлена в эту папку.

⑤ **Добавьте текстуру к сфере.**

Теперь необходимо взять текстуру и «завернуть» в нее сферу. Щелкните на текстуре 8 Ball Texture в окне Project, чтобы выделить ее. После того как текстура будет выделена, перетащите ее на сферу.



Теперь сфера напоминает бильярдный шар. Обратитесь к окну Inspector, в котором отображается структура объекта GameObject. В нем появился новый компонент материала:



Я изучаю C# для работы, а не для того, чтобы создавать видеоигры. Зачем мне отвлекаться на Unity?



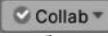
Unity помогает действительно хорошо усвоить C#.

Программирование — это навык, и чем больше практики у вас будет в написании кода C#, тем лучше вы будете программировать. Вот почему мы проектировали лабораторные работы Unity в этой книге специально для того, чтобы **помочь вам в отработке навыков C#** и укрепить ваш уровень владения инструментами и концепциями C#, представленными в каждой главе. Чем больше кода C# вы напишете, тем лучше у вас будет получаться, и это действительно эффективный способ стать квалифицированным программистом C#. Нейробиология утверждает, что мы более эффективно учимся при экспериментировании, поэтому мы разработали лабораторные работы Unity с большим количеством вариантов и экспериментов, а также приводим рекомендации относительно того, как проявить творческие наклонности при выполнении каждой лабораторной работы.

Но что еще важнее, Unity помогает закрепить важные концепции и приемы C# в вашем мозгу. Во время изучения нового языка программирования очень полезно видеть, как работает язык на разных платформах и технологиях. Вот почему в основной материал главы включаются как консольные приложения, так и приложения WPF, а в некоторых случаях один и тот же проект даже строится с применением обеих технологий. Добавление Unity предоставляет третье направление, которое может действительно ускорить ваше понимание C#.

Расширение GitHub for Unity (<https://unity.github.com>) позволяет сохранять ваши проекты в Unity. Вот как это делается:

- **Установка GitHub for Unity:** откройте страницу <https://assetstore.unity.com> и добавьте GitHub for Unity к ресурсам. Вернитесь к Unity, **выберите команду Package Manager** в меню Window, выберите расширение GitHub for Unity из категории My Assets и импортируйте его. GitHub нужно будет импортировать в каждый новый проект Unity.
- **Сохранение изменений в репозитории GitHub:** выберите команду GitHub из меню Window. Каждый проект Unity хранится в отдельном репозитории, связанном с учетной записью GitHub, поэтому щелкните на кнопке Initialize, чтобы инициализировать новый локальный репозиторий (вам будет предложено ввести учетные данные для входа на GitHub). Щелкните **на кнопке Publish**, чтобы создать новый репозиторий для вашей учетной записи GitHub вашего проекта. Каждый раз, когда вы захотите сохранить изменения на GitHub, **перейдите на вкладку Changes** в окне GitHub, выберите вариант **All**, введите краткое описание сохранения (подойдет любой текст) и щелкните на кнопке **Commit at** в нижней части окна GitHub. Затем щелкните **на кнопке Push(1)** в верхней части окна GitHub, чтобы сохранить изменения на GitHub.

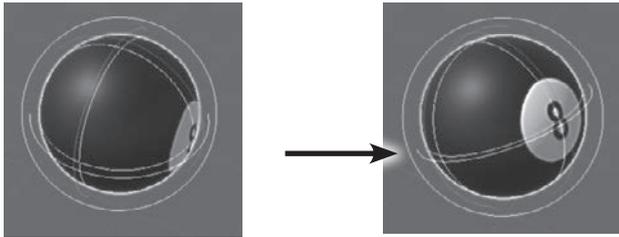
Также для резервного копирования и распространения ваших проектов Unity можно воспользоваться сервисом **Unity Collaborate**, который позволяет публиковать проекты в облачном хранилище. Ваша учетная запись Unity Personal бесплатно получает 1 Гбайт облачного хранилища; этого достаточно для всех проектов лабораторных работ Unity в этой книге. Unity даже будет отслеживать историю вашего проекта (она не учитывается в хранилище). Чтобы опубликовать свой проект, **щелкните на кнопке Collab** () на панели инструментов, после чего щелкните на кнопке Publish. Та же кнопка используется для публикации обновлений. Чтобы просмотреть список опубликованных проектов, перейдите по адресу <https://unity3d.com>, просмотрите свою учетную запись по соответствующей ссылке, а затем щелкните на ссылке Projects на странице со сводкой учетной записи.

Вращение сферы

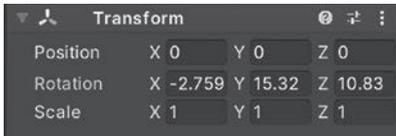
Щелкните на **инструменте Rotate** на панели инструментов. Клавиши Q, W, E, R, T и Y могут использоваться для быстрого переключения между инструментами Transform — при помощи клавиш E и W вы будете переключаться между инструментами Rotate и Move.



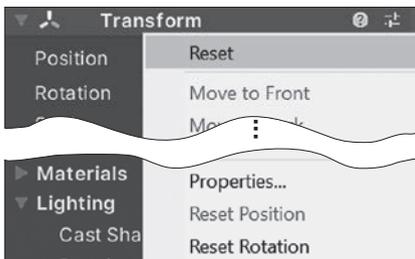
- Щелкните на **сфере**. Unity выводит каркасную модель сферы — манипулятор Rotate с красным, синим и зеленым кругом. Щелкните на красном круге и перетащите его, чтобы повернуть сферу по оси X.



- Щелкните и перетащите **зеленый и синий круги**, чтобы выполнить поворот по осям Y и Z. Внешний белый круг поворачивает сферу вокруг оси, исходящей из камеры Scene. Проследите за изменением чисел Rotation в окне Inspector.



- Откройте **контекстное меню с панели Transform в окне Inspector**. Щелкните на ссылке Reset так же, как это делалось ранее. Все содержимое компонента Transform сбрасывается до значений по умолчанию — в данном случае углы поворота сферы возвращаются к [0, 0, 0].



Щелкните на кнопке с тремя точками (или щелкните правой кнопкой мыши в любой точке заголовка панели Transform), чтобы открыть контекстное меню. Команда Reset в начале меню возвращает компонент к значениям по умолчанию.

Эти команды контекстного меню используются для сброса позиции и углов поворота объекта GameObject.

Чтобы сохранить сцену **прямо сейчас**, выполните команду **File>>Save** или нажмите **Ctrl+S /⌘S**.
Сохраняйтесь пораньше, сохраняйтесь почаще!



Окна и камеры легко возвращаются к значениям по умолчанию.

Если вы изменили представление Scene так, что сфера не видна, или если вы перетащили окна из привычного положения, просто воспользуйтесь раскрывающимся списком в правом верхнем углу, чтобы **вернуть редактор Unity к макету Wide**. При этом происходит сброс макета окна, а камера сцены возвращается к позиции по умолчанию.

Перемещение камеры сцены инструментом Hand и манипулятором Scene

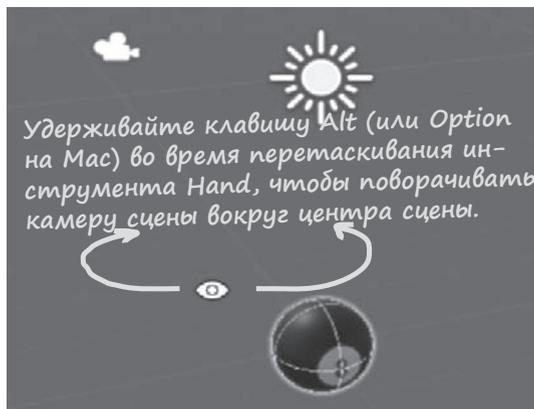
Колесо мыши или прокрутка с сенсорной панели используются для увеличения/уменьшения изображения, а также переключения между манипуляторами Move и Rotate. Обратите внимание: размеры сферы при этом изменяются, но размеры манипуляторов остаются неизменными. Окно Scene в редакторе показывает представление из виртуальной **камеры**, а функция прокрутки приближает и отдаляет камеру.

Нажмите клавишу Q, чтобы выбрать **инструмент Hand**, или выберите этот инструмент на панели инструментов. Указатель мыши принимает вид руки.



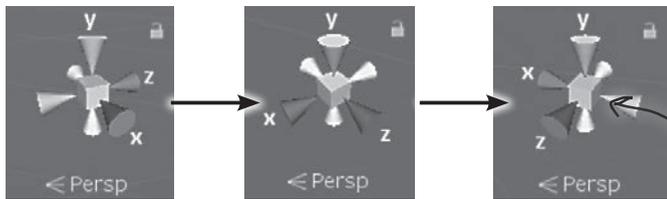
Если удерживать нажатой клавишу Alt (или Option на Mac) во время перетаскивания, инструмент Hand заменяется изображением глаза, а представление поворачивается относительно центра окна.

Инструмент Hand выполняет панорамирование сцены посредством изменения позиции и поворота камеры сцены. Пока инструмент Hand остается выбранным, щелчок в любой точке осуществляет панорамирование.



Пока остается выбранным инструмент Hand, камеру сцены можно **панорамировать** — щелкните и перетащите указатель мыши. Также можно **поворачивать** камеру, удерживая клавишу Alt (или Option) при перетаскивании. **Колесо мыши** используется для приближения/отдаления камеры. Удерживание **правой кнопки мыши** позволяет произвольно перемещаться («летать») по сцене при помощи клавиш W-A-S-D.

Поворачивая камеру сцены, следите за **манипулятором Scene** в правом верхнем углу окна Scene. Манипулятор Scene всегда отображает ориентацию камеры — следите за тем, когда вы используете инструмент Hand для перемещения камеры сцены. Щелкайте на конусах X, Y и Z, чтобы привязать камеру к оси.



Щелкайте на конусах манипулятора Scene, чтобы привязать камеру к оси. Перетаскивайте их, чтобы поворачивать камеру.

В руководстве Unity приведены полезные советы по навигации в сценах:

<https://docs.unity3d.com/2020.1/Documentation/Manual/SceneViewNavigation.html>.

Часть
Задаваемые
Вопросы

В: Мне все еще не совсем понятно, что такое «компонент». Что он делает и чем отличается от объекта `GameObject`?

А: Объект `GameObject` сам по себе практически ничего не делает. В действительности `GameObject` только служит контейнером для компонентов. Когда вы использовали меню `GameObject` для добавления сферы в сцену, редактор Unity создал новый объект `GameObject` и добавил в него все компоненты, определяющие сферы, включая компонент `Transform` для позиционирования, поворотов и масштаба; компонент `Material`, который окрасил сферу в простой белый цвет; и еще несколько компонентов, которые определяют его форму и помогают игре определить, когда он сталкивается с другими объектами. Именно эти компоненты формируют сферу.

В: Означает ли это, что я могу добавить любой компонент в `GameObject`, и тот получит соответствующее поведение?

О: Да, именно так. Когда редактор Unity создавал нашу сцену, он добавил два объекта `GameObject`: один назывался `Main Camera`, а другой — `Directional Light`. Если щелкнуть на объекте `Main Camera` в окне `Hierarchy`, вы увидите, что он состоит из трех компонентов: `Transform`, `Camera` и `Audio Listener`. Если задуматься, то камера должна делать именно это: где-то находиться, получать визуальную информацию и аудио. Объект `GameObject Directional Light` содержит всего два компонента: `Transform` и `Light`, который освещает другие объекты `GameObject` в сцене.

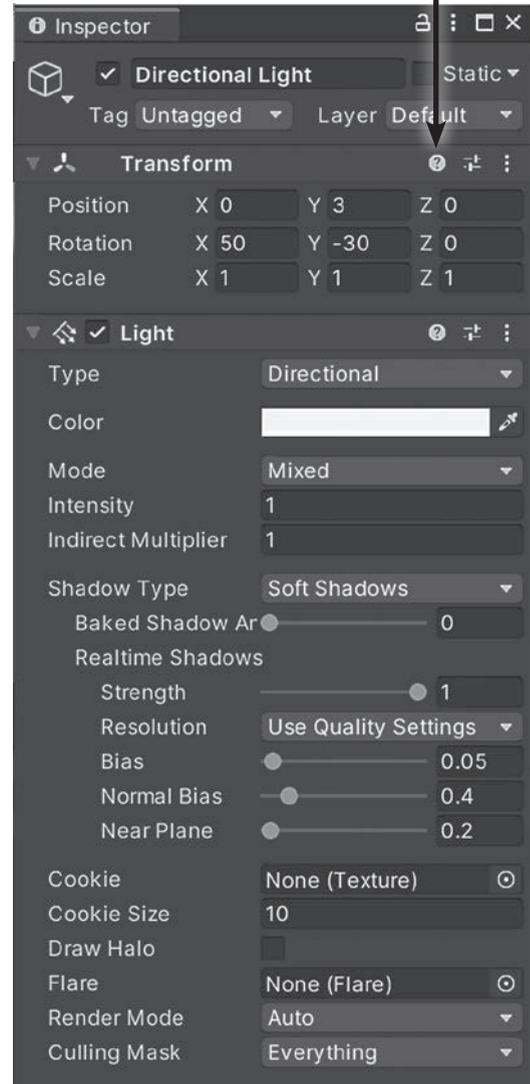
В: Если я добавлю компонент `Light` к любому объекту `GameObject`, он становится источником света?

О: Да! Источником света становится любой объект `GameObject` с компонентом `Light`. Если щелкнуть на кнопке `Add Component` в нижней части окна `Inspector` и добавить компонент `Light` к бильярдному шару, он начнет излучать свет. Если добавить к сцене еще один объект `GameObject`, то он будет отражать этот свет.

В: Вы как-то слишком осторожно выражаетесь. Почему вы говорите об излучении и отражении света? Почему бы просто не сказать, что он светится?

О: Потому что объект `GameObject`, излучающий свет, и светящийся объект `GameObject` — совсем не одно и то же. Если добавить компонент `Light` к шару, он начнет излучать свет, но внешний вид его не изменится, потому что `Light` влияет только на другие объекты `GameObject`, которые излучают свет. Если вы хотите, чтобы объект `GameObject` светился, необходимо сменить его материал или использовать другой компонент, влияющий на его визуализацию.

Щелкните на значке `Help` любого компонента, чтобы открыть соответствующую страницу руководства Unity.



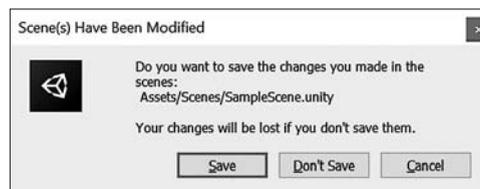
↑
Когда вы щелкаете на объекте `GameObject Directional Light` в окне `Hierarchy`, в окне `Inspector` выводится перечень его компонентов. Их всего два: компонент `Transform`, определяющий его позицию и углы поворота, и компонент `Light`, который непосредственно излучает свет.

Проявите фантазию!

Мы создали эти лабораторные работы Unity, чтобы предоставить вам платформу для самостоятельных экспериментов с C#, потому что это самый эффективный способ стать хорошим разработчиком C#. В конце каждой лабораторной работы Unity мы будем приводить несколько предложений относительно того, что вы можете попробовать сделать самостоятельно. Не жалейте времени и поэкспериментируйте со всем, что вы узнали, прежде чем переходить к следующей главе:

- ★ Добавьте еще несколько сфер к своей сцене. Попробуйте использовать другие текстуры бильярдных шаров. Их можно загрузить из той же папки, из которой вы загрузили файл *8 Ball Texture.png*.
- ★ Попробуйте добавить другие фигуры: выберите в меню `GameObject>>3D Object` варианты `Cube`, `Cylinder` или `Capsule`.
- ★ Поэкспериментируйте с различными изображениями в качестве текстур. Посмотрите, что произойдет с фотографиями людей или пейзажами при использовании их для создания текстур и добавления к различным фигурам.
- ★ Сможете ли вы создать интересную 3D-сцену из фигур, текстур и источников света?

Когда вы будете готовы перейти к следующей главе, не забудьте сохранить проект, потому что мы вернемся к нему в следующей лабораторной работе. Unity предложит вам сохранить сцену при выходе из редактора. →



Чем больше кода C# вы напишете, тем лучше у вас будет получаться, и это действительно эффективный способ стать квалифицированным программистом C#. Мы создали эти лабораторные работы Unity, чтобы предоставить вам платформу для практики и экспериментов.

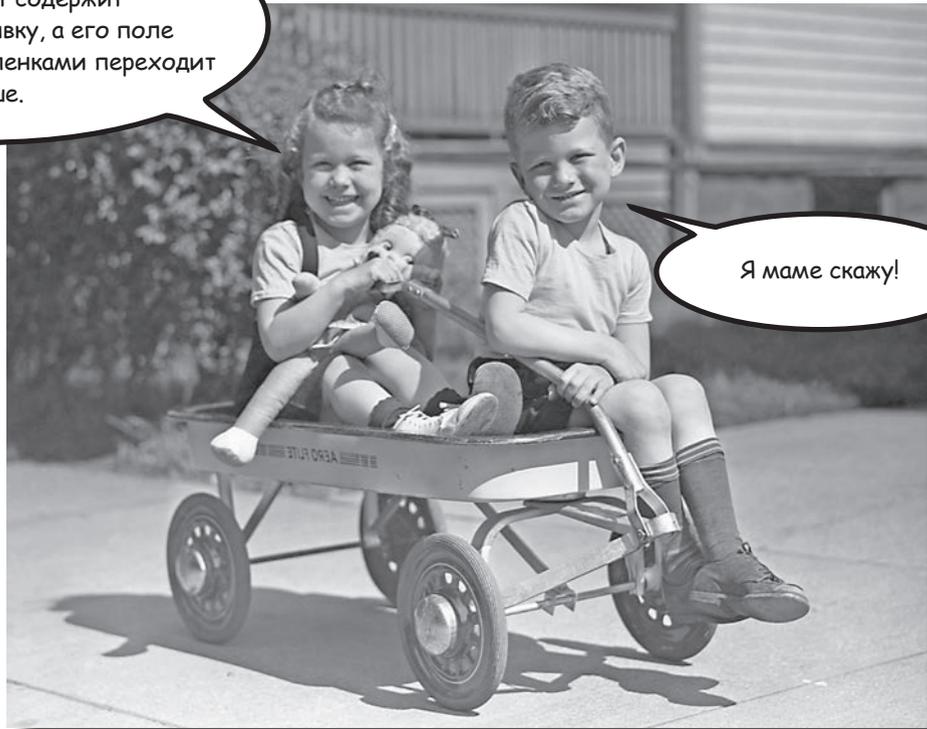
КЛЮЧЕВЫЕ МОМЕНТЫ

- Представление **Scene** — основное интерактивное представление мира, который вы создаете.
- Манипулятор **Move** перемещает объекты по сцене. Манипулятор **Scale** позволяет изменять масштаб объектов `GameObject`.
- Манипулятор **Scene** всегда отображает ориентацию камеры.
- Unity использует **материалы** для определения цветов, узоров, текстур и других визуальных эффектов.
- Некоторые материалы используют **текстуры** (графические файлы, наложенные на фигуры).
- Декорации, персонажи, элементы окружения, камеры и источники света строятся из **объектов `GameObject`**.
- Объекты `GameObject` — фундаментальная разновидность объектов в Unity, а **компоненты** являются основными структурными элементами их поведения.
- Каждый объект `GameObject` содержит **компонент `Transform`**, определяющий его позицию, углы поворота и масштаб.
- **Окно `Project`** отображает представление ресурсов вашего проекта, включая сценарии C# и текстуры, в виде иерархии папок.
- В **окне `Hierarchy`** представлен список всех объектов `GameObject` в сцене.
- **GitHub for Unity** (<https://unity.github.com>) упрощает сохранение проектов Unity в GitHub.
- **Unity Collaborate** также позволяет создавать резервные копии проектов в бесплатном облачном пространстве, прилагаемом к учетной записи Unity Personal.

3 Ориентируемся на объекты

Написание осмысленного кода

...и поэтому мой объект
МладшийБрат содержит
метод СъестьКозявку, а его поле
ПтахнетГрязнымиТеленками переходит
в true.



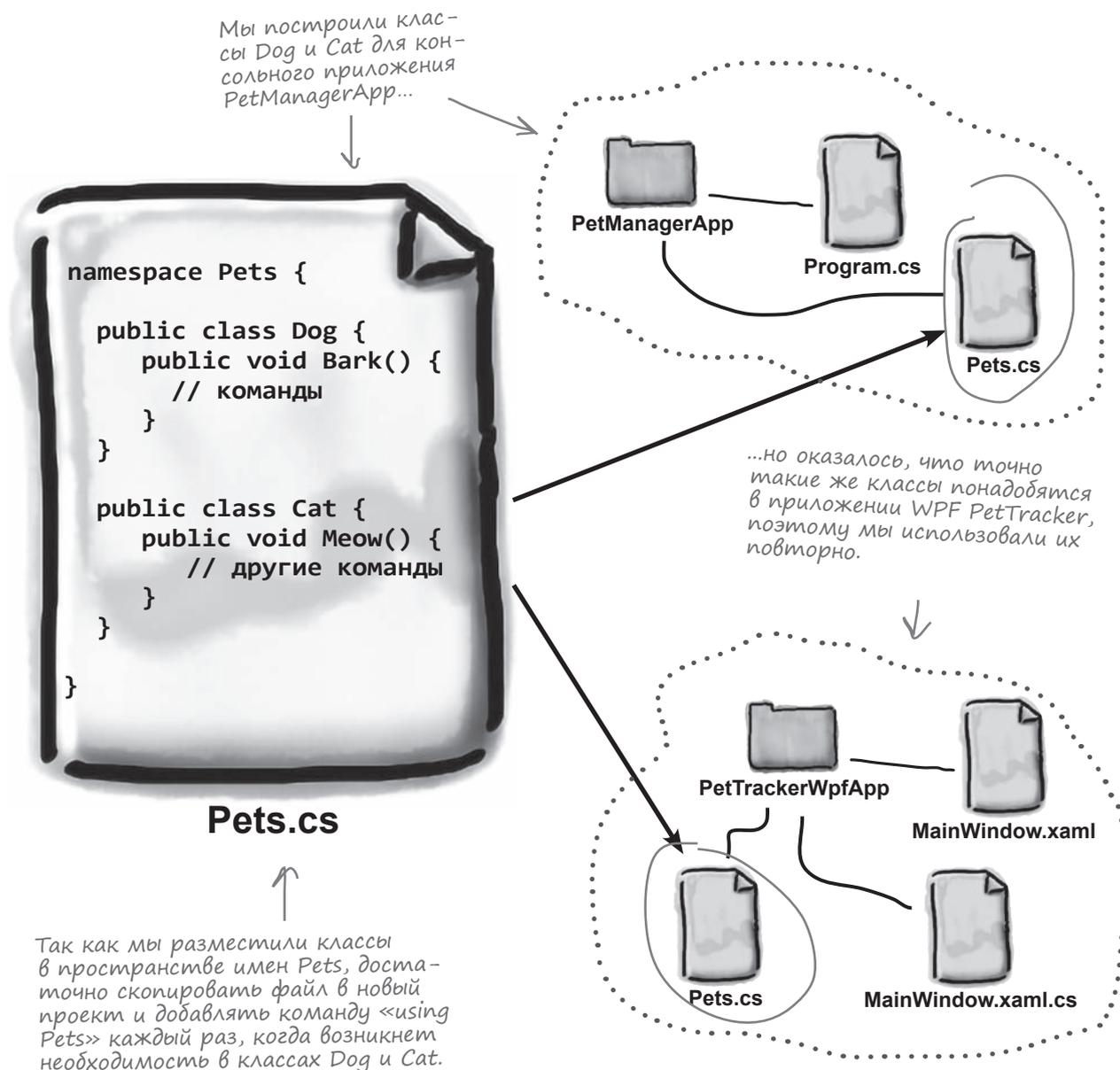
Я маме скажу!

Каждая написанная вами программа решает некоторую задачу.

Когда вы пишете программу, всегда желательно заранее подумать, какую задачу должна решать ваша программа. Вот почему объекты приносят такую пользу. Они позволяют сформировать структуру кода в соответствии с решаемой задачей, чтобы вы могли тратить время на задачу, над которой работаете, не отвлекаясь на механику написания кода. Если вы правильно используете объекты (и действительно хорошо продумали их при проектировании), получившийся код будет интуитивно понятным, будет легко читаться и изменяться.

Если код полезен, он используется повторно

Разработчики стремились к повторному использованию кода с первых дней программирования, и это вполне понятно. Если вы написали класс для одной программы и у вас имеется другая программа, для которой нужен код, делающий то же самое, будет вполне логично **повторно использовать** тот же класс в новой программе.



Некоторые методы получают параметры и возвращают значение

Вы видели методы, которые выполняют конкретные операции (как, например, метод `SetUpGame` в главе 1, который выполняет подготовку вашей игры). Но методы способны на большее: они могут использовать **параметры** для получения ввода, что-то сделать с полученными данными, а затем сгенерировать выходные данные в **возвращаемом значении**, которое может быть использовано командой, вызвавшей метод.



Параметры представляют собой значения, используемые методом в качестве входных данных. Они объявляются как переменные, включаемые в объявление метода (в круглых скобках). Возвращаемое значение вычисляется или генерируется внутри метода и возвращается команде, в которой был вызван метод. Тип возвращаемого значения (например, `string` или `int`) называется **возвращаемым типом**. Если метод имеет возвращаемый тип, то он должен содержать команду **return**.

Пример метода с двумя параметрами `int` и возвращаемым типом `int`:

```

Возвращается тип int, поэтому метод должен возвращать значение int командой return.
int Multiply(int factor1, int factor2)
{
    int product = factor1 * factor2;
    return product;
}
Метод получает два параметра int с именами factor1 и factor2. Они рассматриваются как переменные int.
Команда return передает значение команде, вызвавшей метод.
  
```

Метод получает два параметра с именами `factor1` и `factor2`. Он использует оператор умножения `*` для вычисления результата, который возвращается ключевым словом **return**.

Этот код вызывает метод `Multiply` и сохраняет результат в переменной с именем `area`:

```

int height = 179;
int width = 83;
int area = Multiply(height, width);
Методам можно передать непосредственные значения (например, 3 и 5 — Multiply(3, 5)), но при вызове методов также можно использовать переменные. При этом имена переменных могут отличаться от имен параметров.
  
```

Сделайте это! → Сейчас мы начнем создавать методы, возвращающие значения, поэтому самое время написать код и воспользоваться отладчиком для того, чтобы действительно понять, как работает команда **return**.

- ★ Что произойдет, когда метод завершит выполнение всех своих команд? Посмотрите сами — откройте одну из программ, написанных до настоящего момента, установите точку прерывания внутри метода и выполните метод в пошаговом режиме.
- ★ Когда будет выполнена последняя команда в методе, управление **возвращается** команде, из которой метод был вызван, и выполнение программы продолжается со следующей команды.
- ★ Метод также может содержать команду **return**, которая заставляет метод немедленно вернуть управление без выполнения других команд. Попробуйте включить дополнительную команду **return** в середину метода, после чего продолжите выполнение в пошаговом режиме.

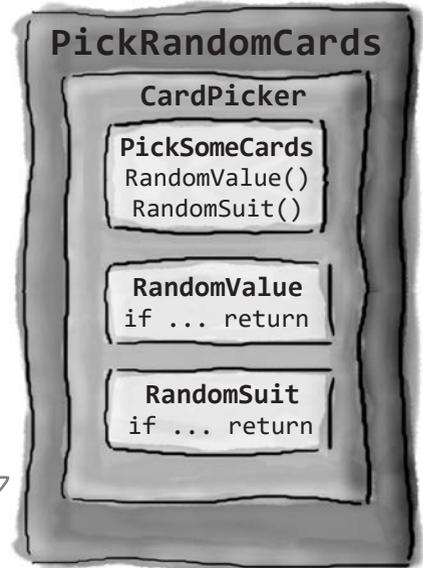
Программа для выбора карт

В первой программе этой главы мы построим консольное приложение .NET Console с именем PickRandomCards, которое позволяет выбирать случайные игровые карты. Структура приложения выглядит так:

Когда вы создаете консольное приложение в Visual Studio, в пространство имен с именем, совпадающим с именем проекта, добавляется класс с именем Program. Класс содержит метод Main, который становится точкой входа.



Мы добавим еще один класс CardPicker с тремя методами. Метод Main будет вызывать метод PickSomeCards нового класса.



Метод PickSomeCards будет использовать строковые значения для представления карт. Если вы хотите получить пять карт, вызов будет выглядеть так:

```
string[] cards = PickSomeCards(5);
```

Переменная cards имеет тип, который вам пока неизвестен. Квадратные скобки [] означают, что это массив строк. Массивы позволяют использовать одну переменную для хранения многих значений — в данном случае строк с игровыми картами. Пример массива строк, который может быть возвращен методом PickSomeCards:

```
{ "10 of Diamonds",  
  "6 of Clubs",  
  "7 of Spades",  
  "Ace of Diamonds",  
  "Ace of Hearts" }
```



Массив из пяти строк. Наше приложение будет создавать такие массивы для представления случайно выбранных карт.

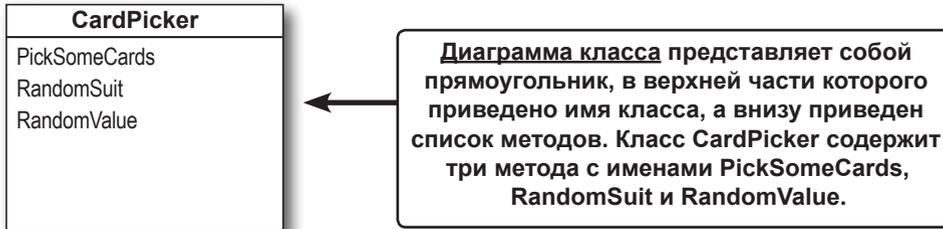


После того как массив будет сгенерирован, воспользуйтесь циклом foreach для вывода его содержимого на консоль.

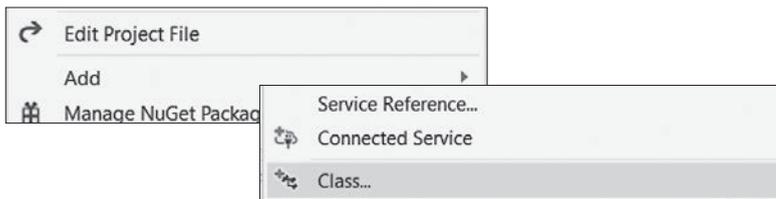
Создание консольного приложения PickRandomCards

Сделайте это!

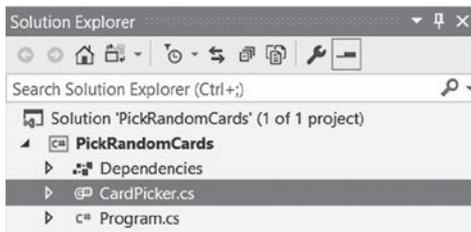
Воспользуемся тем, что узнали в этой главе, и создадим программу для выбора нескольких случайных карт. Откройте Visual Studio и создайте новый проект консольного приложения с именем PickRandomCards. В вашу программу будет входить класс с именем CardPicker. Диаграмма класса, на которой обозначено его имя и методы, выглядит так:



Щелкните правой кнопкой мыши на проекте PickRandomCards в окне Solution Explorer и **выберите команду Add>>Class...** в Windows (или Add>>New Class... в macOS) в контекстном меню. Visual Studio запрашивает имя класса — выберите *CardPicker.cs*.



Visual Studio создает в проекте новый класс с именем CardPicker:



Новый класс пока пуст — он начинается с объявления `class CardPicker` и состоит из пары фигурных скобок, в которых ничего нет. **Добавьте новый метод с именем PickSomeCards.** Новый класс должен выглядеть так:

```
class CardPicker
{
    public static string[] PickSomeCards(int numberOfCards)
    {
    }
}
```

Обязательно включите ключевые слова *public* и *static*. Они будут описаны позднее в этой главе.

Если вы тщательно ввели это объявление метода точно так, как оно приведено здесь, под *PickSomeCards* должна появиться красная волнистая линия. Как вы думаете, что это значит?



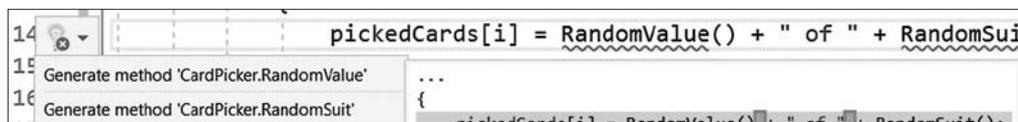
Завершение метода PickSomeCards

- 1 Метод `PickSomeCards` должен содержать команду `return`; добавьте ее. Введите оставшийся код метода — после того, как в нем появляется команда `return`, возвращающая строковый массив, ошибка исчезает:

```
class CardPicker
{
    public static string[] PickSomeCards(int numberOfCards)
    {
        string[] pickedCards = new string[numberOfCards];
        for (int i = 0; i < numberOfCards; i++)
        {
            pickedCards[i] = RandomValue() + " of " + RandomSuit();
        }
        return pickedCards;
    }
}
```

← Когда в методе появляется команда `return`, которая возвращает значение с типом, соответствующим возвращаемому типу метода, красная волнистая линия исчезает.

- 2 Сгенерируйте недостающие методы. Теперь в коде появляются другие ошибки, потому что в нем нет метода `RandomValue` или `RandomSuit`. Сгенерируйте эти методы так, как это было сделано в главе 1. Используйте кнопку быстрых действий на левом поле редактора кода: если щелкнуть на ней, открывается меню с командами генерирования обоих методов:



Сгенерируйте методы. В классе должны появиться методы `RandomValue` и `RandomSuit`:

```
class CardPicker
{
    public static string[] PickSomeCards(int numberOfCards)
    {
        string[] pickedCards = new string[numberOfCards];
        for (int i = 0; i < numberOfCards; i++)
        {
            pickedCards[i] = RandomValue() + " of " + RandomSuit();
        }
        return pickedCards;
    }

    private static string RandomValue()
    {
        throw new NotImplementedException();
    }

    private static string RandomSuit()
    {
        throw new NotImplementedException();
    }
}
```

Для генерирования методов была использована IDE. Если методы следуют в другом порядке, это нормально — порядок методов в классе роли не играет.

3

Используйте команду `return` для построения методов `RandomSuit` и `RandomValue`. Метод может содержать более одной команды `return`, и при выполнении одной из команд он немедленно возвращает управление — никакие другие команды в методе не выполняются.

Ниже приведен пример использования команд `return` в программе. Допустим, вы строите карточную игру и вам нужны методы для генерирования случайных карточных мастей и номиналов. Начнем с создания генератора случайных чисел по аналогии с тем, который использовался в игре с поиском пар в главе 1. Добавьте его под объявлением класса:

```
class CardPicker
{
    static Random random = new Random();
```

Теперь добавьте в метод `RandomSuit` код, использующий команды `return` для остановки выполнения метода сразу же после обнаружения совпадения. Метод `Next` генератора случайных чисел может получать два параметра: `random.Next(1, 5)` возвращает число от 1 (включительно) до 5 (не включая), другими словами, случайное число от 1 до 4. Он будет использоваться методом `RandomSuit` для выбора случайной карточной масти:

```
private static string RandomSuit()
{
    // получить случайное число от 1 до 4
    int value = random.Next(1, 5);
    // если это 1, вернуть строку Spades
    if (value == 1) return "Spades";
    // если это 2, вернуть строку Hearts
    if (value == 2) return "Hearts";
    // если это 3, вернуть строку Clubs
    if (value == 3) return "Clubs";
    // если выполнение продолжается, вернуть строку Diamonds
    return "Diamonds";
}
```

Мы добавили комментарии, чтобы объяснить, что же здесь происходит.

А вот метод `RandomValue`, генерирующий случайный номинал карты. Попробуйте разобраться в том, как он работает:

```
private static string RandomValue()
{
    int value = random.Next(1, 14);
    if (value == 1) return "Ace";
    if (value == 11) return "Jack";
    if (value == 12) return "Queen";
    if (value == 13) return "King";
    return value.ToString();
}
```

А вы заметили, что метод возвращает `value.ToString()`, а не просто `value`? Дело в том, что `value` является переменной `int`, а метод `RandomValue` был объявлен со строковым возвращаемым типом, поэтому `value` необходимо преобразовать в строку. Добавление `.ToString()` к любой переменной или значению преобразует их в строку.

Команда `return` заставляет метод немедленно прервать выполнение и вернуться к команде, из которой он был вызван.

Готовый класс CardPicker

Ниже приведен код готового класса CardPicker. Он должен принадлежать пространству имен, соответствующему имени вашего проекта:

```
class CardPicker
{
    static Random random = new Random();

    public static string[] PickSomeCards(int numberOfCards)
    {
        string[] pickedCards = new string[numberOfCards];
        for (int i = 0; i < numberOfCards; i++)
        {
            pickedCards[i] = RandomValue() + " of " + RandomSuit();
        }
        return pickedCards;
    }

    private static string RandomValue()
    {
        int value = random.Next(1, 14);
        if (value == 1) return "Ace";
        if (value == 11) return "Jack";
        if (value == 12) return "Queen";
        if (value == 13) return "King";
        return value.ToString();
    }

    private static string RandomSuit()
    {
        // получить случайное число от 1 до 4
        int value = random.Next(1, 5);
        // если это 1, вернуть строку Spades
        if (value == 1) return "Spades";
        // если это 2, вернуть строку Hearts
        if (value == 2) return "Hearts";
        // если это 3, вернуть строку Clubs
        if (value == 3) return "Clubs";
        // если выполнение продолжается, вернуть строку Diamonds
        return "Diamonds";
    }
}
```

Статическое поле с именем "random", используемое для генерирования случайных чисел.



Мы еще не говорили о полях... почти.

Класс CardPicker содержит **поле** с именем **random**. Поля уже встречались вам в игре с поиском пар из главы 1, но мы с ними еще не работали. Не беспокойтесь — поля и ключевое слово **static** будут намного подробнее рассмотрены в этой главе.

Мы добавили комментарии, чтобы помочь вам понять, как работает метод RandomSuit. Попробуйте добавить в метод RandomValue аналогичные комментарии, объясняющие, как он работает.



Ключевые слова **public** и **static** использовались при добавлении метода **PickSomeCards**. Visual Studio оставляет **static** при генерировании метода, но объявляет их с ключевым словом **private**, а не **static**. Как вы думаете, что делают эти ключевые слова?



Упражнение

Теперь ваш класс `CardPicker` содержит метод для выбора случайных карт, и у вас появилось все необходимое для завершения консольного приложения, для чего необходимо **завершить метод `Main`**. Нам понадобится лишь несколько полезных методов, при помощи которых консольное приложение сможет получить данные от пользователя, и воспользоваться ими для выбора карт.

Полезный метод 1: `Console.WriteLine`

Метод `Console.WriteLine` вам уже известен. Его родственник `Console.Write` выводит текст на консоль, но не добавляет разрыв строки в конце. Он используется для вывода сообщения для пользователя:

```
Console.WriteLine("Enter the number of cards to pick: ");
```

Полезный метод 2: `Console.ReadLine`

Метод `Console.ReadLine` читает строку текста и возвращает строку. С помощью этого метода пользователь сможет указать программе, сколько карт нужно выбрать:

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

Полезный метод 3: `int.TryParse`

Метод `CardPicker.PickSomeCards` получает параметр `int`. Входные данные, полученные от пользователя, представляют собой строку, которую необходимо как-то преобразовать в `int`. Для этой цели используется метод `int.TryParse`:

```
if (int.TryParse(line, out int numberOfCards))
{
    // этот блок выполняется в том случае, если строка МОЖЕТ БЫТЬ преобразована в int
    // значение, сохраняемое в новой переменной, называется numberOfCards
}
else
{
    // этот блок выполняется, если строка НЕ МОЖЕТ БЫТЬ преобразована в int
}
```

В главе 2 метод `int.TryParse` использовался в обработчике события `TextBox`, чтобы он допускал ввод только числовых данных. Задержитесь на минуту и вспомните, как работает этот обработчик событий.

Все вместе

Ваша задача — взять все три новых фрагмента и объединить их в новом методе `Main` консольного приложения. Измените файл `Program.cs` и замените строку с выводом «Hello World!» в методе `Main` кодом, который решает следующие задачи:

- ★ Метод `Console.WriteLine` запрашивает у пользователя количество выбираемых карт.
- ★ Метод `Console.ReadLine` читает строку ввода в строковую переменную с именем `line`.
- ★ Метод `int.TryParse` пытается преобразовать ее в переменную типа `int` с именем `numberOfCards`.
- ★ Если ввод **преобразуется** в значение `int`, используйте свой класс `CardPicker` для выбора количества карт, заданного пользователем: `CardPicker.PickSomeCards(numberOfCards)`. Используйте переменную `string[]` для сохранения результатов, а затем воспользуйтесь циклом `foreach`, чтобы вызвать `Console.WriteLine` для каждой карты в массиве. Вернитесь к главе 1, чтобы увидеть пример цикла `foreach` — он будет использоваться для перебора всех элементов массива. Первая строка цикла: `foreach (string card in CardPicker.PickSomeCards(numberOfCards))`.
- ★ Если ввод **не может быть преобразован**, используйте `Console.WriteLine` для вывода сообщения, в котором говорится, что число недействительно.

Пока вы работаете над методом `Main` программы, обратите внимание на его возвращаемый тип. Как вы думаете, что здесь происходит?



Упражнение Решение

Ниже приведен метод `Main` вашего консольного приложения. Он запрашивает у пользователя количество выбираемых карт, пытается преобразовать полученную строку в `int`, после чего использует метод `PickSomeCards` из класса `CardPicker` для выбора соответствующего количества карт. `PickSomeCards` возвращает все выбранные карты в виде массива строк, поэтому для вывода всех карт на консоль используется цикл `foreach`.

```
static void Main(string[] args)
{
    Console.Write("Enter the number of cards to pick: ");
    string line = Console.ReadLine();
    if (int.TryParse(line, out int numberOfCards))
    {
        foreach (string card in CardPicker.PickSomeCards(numberOfCards))
        {
            Console.WriteLine(card);
        }
    }
    else
    {
        Console.WriteLine("Please enter a valid number.");
    }
}
```

Этот метод `Main` заменяет метод, который выводит "Hello World!" (этот метод был создан для вас Visual Studio в файле `Program.cs`).

Цикл `foreach` выполняет `Console.WriteLine(card)` для каждого элемента массива, возвращенного `PickSomeCards`.

Ваш метод `Main` использует возвращаемый тип `void`, чтобы сообщить C# об отсутствии возвращаемого значения. Метод с возвращаемым типом `void` не обязан содержать метод `return`.

А вот что вы увидите при запуске консольного приложения:

```
Microsoft Visual Studio Debug Console
Enter the number of cards to pick: 13
5 of Spades
3 of Hearts
9 of Diamonds
King of Clubs
5 of Diamonds
4 of Diamonds
6 of Spades
King of Diamonds
King of Diamonds
4 of Diamonds
Jack of Hearts
6 of Clubs
6 of Spades

C:\Users\Public\source\repos\PickRandomCards\PickRandomCards\bin\Debug\netcoreapp3.1\PickRandomCards.exe (process 8068) exited with code 0.
```

Не торопитесь и постарайтесь по-настоящему разобраться в том, как работает эта программа, — вам предоставляется отличная возможность исследовать ваш код в отладчике Visual Studio. Установите точку прерывания в первой строке метода `Main`, после чего воспользуйтесь функцией `Step Into` (F11) для пошагового выполнения всей программы. Добавьте отслеживание для переменной `value` и продолжайте следить за ней во время пошагового выполнения методов `RandomSuit` и `RandomValue`.

Анна работает над следующей игрой

Знакомьтесь: это Анна, разработчик инди-игр. Ее последняя игра была распродана тысячами экземпляров, и теперь она начинает работу над следующей.

Анна начала работать над **прототипами**. Она работала над кодом противников-инопланетян для одной захватывающей части игры, когда игрок пытается выбраться из своего убежища, а захватчики его ищут. Анна написала несколько методов, определяющих поведение врагов: они обыскивают последнее место, в котором был замечен игрок, через какое-то время бросают поиски, если найти игрока не удалось, и захватывают игрока, если им удалось подобраться слишком близко.



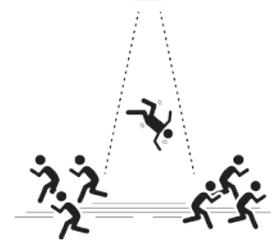
В моей следующей игре игрок защищает город от инопланетного вторжения.



```
SearchForPlayer();
```



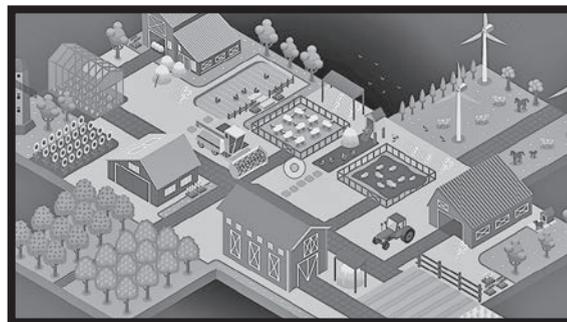
```
if (SpottedPlayer()) {
    CommunicatePlayerLocation();
}
```



```
CapturePlayer();
```

Игра Анны развивается...

Люди против инопланетян – классная идея, но Анна не на сто процентов уверена, что ей захочется пойти именно в этом направлении. Она также думает об игре, в которой игроку-капитану нужно будет уклоняться от пиратов. А может быть, это будет игра про зомби на заброшенной ферме. В этих трех случаях враги будут иметь разную графику, но их поведение может определяться одними и теми же методами.



Наверняка эти методы могут подойти и для других игр.



...Как же Анна может упростить свою задачу?

Анна не уверена, в каком направлении пойдет игра, поэтому она хочет построить несколько прототипов, и все они должны использовать для врагов один и тот же код с методами SearchForPlayer, StopSearching, SpottedPlayer, CommunicatePlayerLocation и CapturePlayer.



А вы можете предложить хороший способ использования одних и тех же методов для врагов из разных прототипов?



Я поместила все методы поведения врагов в один класс Enemy. Смогу ли я повторно использовать класс в каждом из трех разных прототипов игры?

Enemy
SearchForPlayer
SpottedPlayer
CommunicatePlayerLocation
StopSearching
CapturePlayer



Прототипы

Разработка игр... и не только

Прототип представляет собой раннюю версию игры, которую можно использовать для игры, тестирования, изучения и улучшения. Прототип может стать чрезвычайно ценным инструментом, упрощающим внесение ранних изменений. Прототипы особенно полезны тем, что они позволяют быстро поэкспериментировать с разными идеями, прежде чем принимать долгосрочные решения.

- Первым прототипом часто становится **бумажный прототип**: все основные элементы игры раскладываются на листе бумаги. Например, чтобы достаточно много узнать о своей игре, можно нарисовать уровни или игровые области на больших листах, использовать наклейки или карточки для представления разных элементов игры и перемещать их вручную.
- Еще одно преимущество прототипов заключается в том, что они позволяют очень быстро **перейти от идеи к работоспособной, действующей игре**. Больше всего информации об игре (и вообще о любой программе) можно получить, передав рабочий продукт в руки игроков (или пользователей).
- Многие игры проходят через **несколько прототипов**. Так вы получаете возможность опробовать много разных идей и поучиться на них. Даже если что-то пошло не так, считайте это экспериментом, а не ошибкой.
- Построение прототипов — это **навык**. Как и любой другой навык, он совершенствуется с применением на практике. К счастью, строить прототипы также интересно, и это отличный способ стать сильнее в написании кода C#.

Прототипы применяются не только для игр! При построении программы любого типа часто стоит начать с построения прототипа, чтобы поэкспериментировать с разными идеями.



Построение бумажного прототипа для классической игры

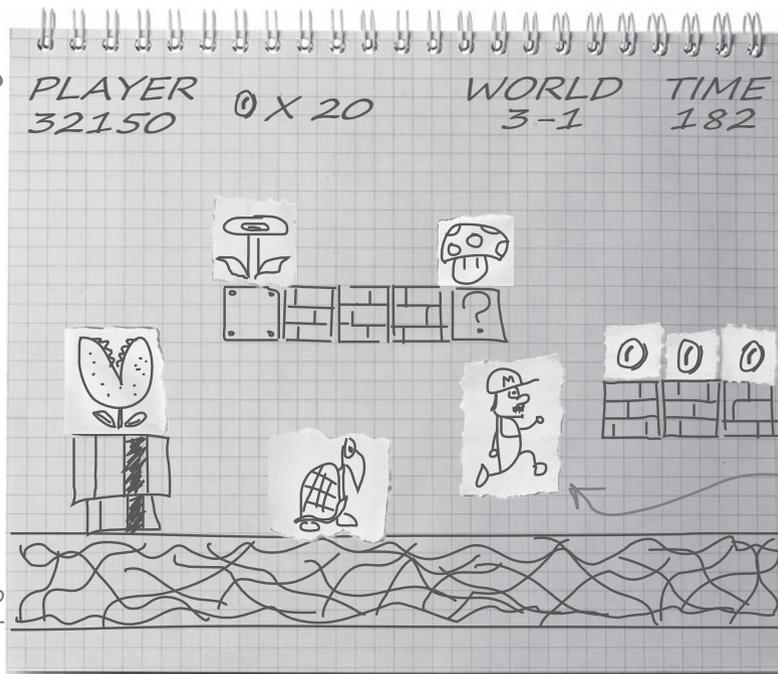
Бумажные прототипы помогут вам продумать, как должна работать игра, еще до начала ее построения, а это сэкономит вам уйму времени. Построение начнется практически мгновенно — вам понадобится лишь бумага и карандаш. Попробуйте выбрать свою любимую классическую игру. Платформенные игры подходят особенно хорошо, поэтому мы выбрали одну из **самых популярных, самых узнаваемых** классических игр... но вы можете выбрать любую игру по своему вкусу! Теперь необходимо сделать следующее.

Нарисуйте



- 1 **Нарисуйте фон на листе бумаги.** Построение прототипа начинается с создания фона. В нашем прототипе земля, кирпичи и трубы не двигаются, поэтому мы нарисовали их на бумаге. Также в верхней части листа добавлен счет, время и прочий текст.
- 2 **Оторвите маленькие клочки бумаги и нарисуйте подвижные части.** В нашем прототипе персонажи, хищное растение, гриб, огненный цветок и монетки рисуются на отдельных листках. Если вы не сильны в рисовании — ничего страшного! Просто нарисуйте схематичных человечков и приблизительные контуры фигур. В конце концов, никто не увидит вашего творчества!
- 3 **Немного «поиграйте».** Это самая интересная часть! Попробуйте смоделировать движение игрока. Перемещайте игрока по странице. Также заставляйте двигаться других персонажей. Очень полезно несколько минут провести за игрой, а затем вернуться к прототипу и посмотреть, сможете ли вы как можно точнее воспроизвести движение. (Поначалу это кажется немного странным, но это нормально!)

Текст в верхней части экрана называется HUD (Head-Up Display).



Земля, кирпичи и трубы не двигаются, поэтому мы нарисовали их на фоновой бумаге. Не существует жестких правил относительно того, что должно находиться на фоне, а что двигаться.

Когда игрок подбирает гриб, он увеличивается в размерах, поэтому мы также нарисовали отдельного маленького персонажа на отдельном клочке бумаги.



Механика прыжков игрового персонажа была тщательно спроектирована создателями игры. Моделирование прыжков на бумажном прототипе — полезное учебное упражнение.



Похоже, бумажные прототипы пригодятся не только в играх. Уверен, что их можно будет использовать и в других проектах.

Все инструменты и идеи, представленные в разделе «Разработка игр... и не только», относятся к числу важных навыков программирования, которые выходят за рамки простой разработки игр. Тем не менее наш опыт показывает, что их становится проще осваивать после того, как вы сначала опробуете их на играх.

Да! Бумажный прототип станет отличным первым шагом для любого проекта.

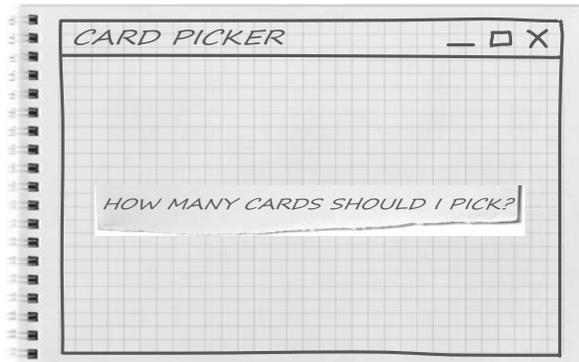
Если вы строите настольное приложение, мобильное приложение или любой другой проект, у которого есть пользовательский интерфейс, построение бумажного прототипа станет хорошей отправной точкой. Иногда приходится построить несколько бумажных прототипов, прежде чем вы начнете понимать, что к чему. Собственно, именно поэтому мы начали с бумажного прототипа для классической игры... потому, что он наглядно демонстрирует, как строить бумажные прототипы. **Построение прототипов — исключительно важный навык для любых разработчиков**, не только для разработчиков игр.

Возьми в руку карандаш



В следующем проекте мы создадим приложение WPF, которое использует класс CardPicker для генерирования случайного набора карт. В этом письменном упражнении мы построим бумажный прототип нашего приложения, чтобы опробовать различные варианты проектирования интерфейса.

Начнем с рисования контура окна на большом листе бумаги и надписи на меньшем листке.



Где-то в окне приложения должен размещаться список с картами и кнопка «Pick some cards».

PICK SOME CARDS

4 OF HEARTS
2 OF DIAMONDS
KING OF SPADES
ACE OF HEARTS
7 OF CLUBS
10 OF SPADES
JACK OF CLUBS
9 OF HEARTS
9 OF DIAMONDS
3 OF CLUBS
ACE OF SPADES

Теперь нарисуем несколько разных элементов управления на еще меньших клочках бумаги. Перемещайте их в окне и экспериментируйте с их совместным размещением. Как вы думаете, какой вариант работает лучше всего? Единственно правильного ответа не существует — любое приложение можно спроектировать множеством разных способов.

Ваше приложение должно предоставить возможность выбора количества карт. Попробуйте нарисовать поле ввода, в котором пользователь может напрямую ввести число в приложении.

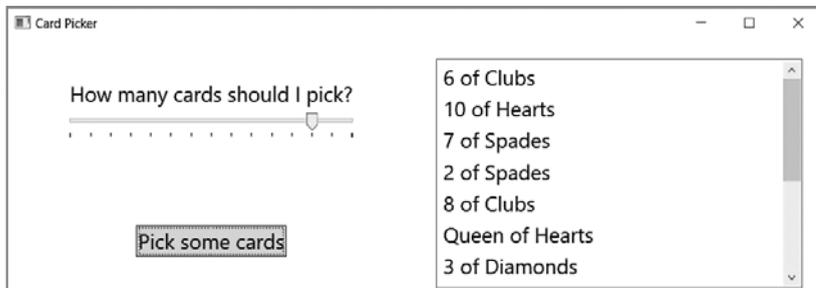


Также попробуйте использовать ползунок и набор переключателей. А сможете ли вы предложить другие элементы, которые ранее уже использовались для ввода чисел в приложении? Может, раскрывающийся список? Проявите фантазию!



Следующий шаг: построение WPF-версии приложения для выбора карт

В следующем проекте мы построим приложение WPF с именем PickACardUI. Оно будет выглядеть примерно так:



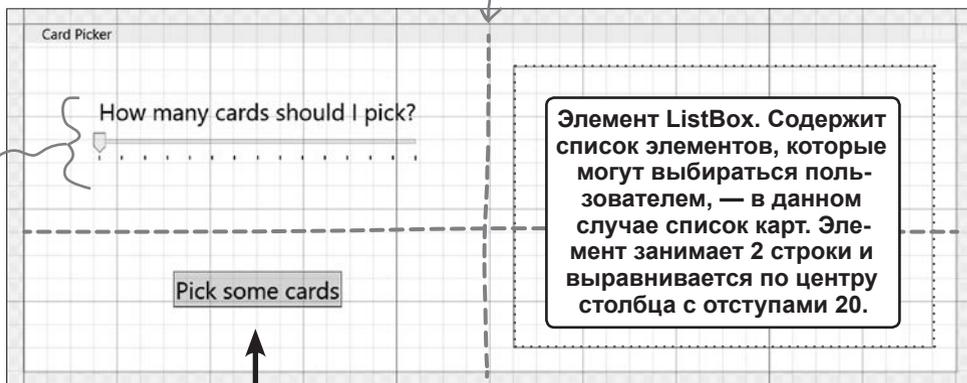
Мы решили использовать ползунок для выбора числа карт. Но конечно, это не единственный вариант построения интерфейса! Может, вы предложили другой вариант в своем бумажном прототипе? Это нормально! Любое приложение можно спроектировать множеством разных способов, и почти никогда не существует однозначно правильного (или ошибочного) решения.

В приложении PickACardUI элемент Slider используется для выбора количества случайных карт. После выбора количества карт вы щелкаете на кнопке, чтобы приложение выбрало их и добавило в ListBox.

Окно будет выглядеть примерно так:

Окно делится на две строки и два столбца. Элемент ListBox в правом столбце занимает обе строки.

В ячейке расположены два элемента, Label и Slider. Вскоре мы более внимательно разберемся в том, как это работает.



Обработчик события Button будет вызывать метод вашего класса, который возвращает список карт. Все возвращенные карты будут добавляться в ListBox.

Мы не будем напоминать вам о том, что проекты следует добавлять в систему управления версиями, но мы все равно считаем, что вам стоит создать учетную запись GitHub и публиковать под ней все ваши проекты!

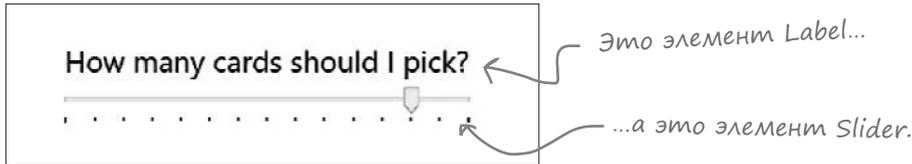
Здесь вы найдете версии всех WPF-проектов из книги для ASP.NET Core со снимками экрана из Visual Studio для Mac.

↑ Add to Source Control ▲

Версия этого проекта для Mac доступна в приложении «Visual Studio для пользователей Mac».

StackPanel — контейнер для наложения элементов

Ваше приложение WPF будет использовать элемент Grid для размещения элементов по аналогии с тем, как это делалось в игре с поиском пар. Но прежде чем переходить к написанию кода, стоит повнимательнее присмотреться к двум элементам в левой верхней ячейке сетки:

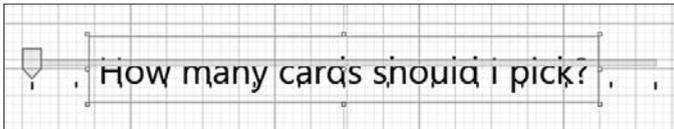


Как же связать эти элементы друг с другом? *Можно* попытаться разместить их в одной ячейке сетки:

```
<Grid>
  <Label HorizontalAlignment="Center" VerticalAlignment="Center" Margin="20"
    Content="How many cards should I pick?" FontSize="20"/>
  <Slider VerticalAlignment="Center" Margin="20"
    Minimum="1" Maximum="15" Foreground="Black"
    IsSnapToTickEnabled="True" TickPlacement="BottomRight" />
</Grid>
```

Код XAML элемента Slider. Он будет более подробно проанализирован, когда мы займемся построением формы.

Но в этом случае элементы просто будут перекрываться:

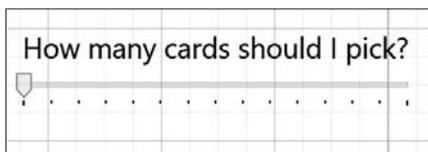


На помощь приходит элемент **StackPanel**. StackPanel является контейнером — как и Grid, он предназначен для размещения других элементов и обеспечения их правильной позиции в окне. Если Grid позволяет выстраивать свои элементы по строкам и столбцам, StackPanel выстраивает элементы в *горизонтальный или вертикальный ряд*.

Возьмем те же элементы Label и Slider, но на этот раз воспользуемся StackPanel для размещения их таким образом, чтобы элемент Label располагался поверх Slider. Обратите внимание на то, что свойства выравнивания и отступов были перемещены в StackPanel — по центру должна выравниваться сама панель вместе с прилегающими к ней отступами:

```
<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center" Margin="20" >
  <Label Content="How many cards should I pick?" FontSize="20" />
  <Slider Minimum="1" Maximum="15" Foreground="Black"
    IsSnapToTickEnabled="True" TickPlacement="BottomRight" />
</StackPanel>
```

При использовании StackPanel элементы в ячейке выглядят именно так, как требовалось:



*Так должен работать наш проект.
Перейдем к его построению!*

Повторное использование класса CardPicker в новом приложении WPF

Если вы написали класс для одной программы, часто бывает возможно использовать то же поведение в другой программе. Одно из главных преимуществ классов как раз и заключается в том, что они упрощают **повторное использование** кода. Давайте создадим для приложения красивый новый интерфейс, но сохраним прежнее поведение за счет повторного использования класса CardPicker.

Повторите это

1 Создайте новое приложение WPF с именем PickACardUI.

Выполните те же действия, что и для игры с поиском пар в главе 1:

- ★ Откройте Visual Studio и создайте новый проект.
- ★ Выберите шаблон **WPF App (.NET Core)**.
- ★ Присвойте новому приложению имя **PickACardUI**. Visual Studio создает проект и добавляет в него файлы *MainWindow.xaml* и *MainWindow.xaml.cs*, входящие в пространство имен PickACardUI.

2 Добавьте класс CardPicker, созданный для проекта консольного приложения.

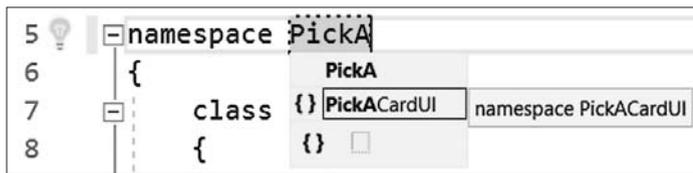
Щелкните правой кнопкой мыши на имени проекта, выберите в меню команду **Add>>Existing Item...**



Перейдите в папку с вашим консольным приложением и выберите файл *CardPicker.cs*, чтобы добавить его в проект. В проекте WPF появляется копия файла *CardPicker.cs* из консольного приложения.

3 Измените пространство имен класса CardPicker.

Сделайте двойной щелчок на файле *CardPicker.cs* в окне Solution Explorer. Файл все еще принадлежит пространству имен из консольного приложения. **Измените пространство имен** и приведите его в соответствие с именем проекта. Подсказка IntelliSense предложит пространство имен PickACardUI — **нажмите клавишу Tab, чтобы согласиться с предложением:**



Мы изменяем пространство имен в файле *CardPicker.cs* и заменяем его пространством имен, использованным Visual Studio при создании файлов нового проекта. Это делается для того, чтобы класс *CardPicker* мог использоваться в коде нового проекта.

Теперь класс CardPicker должен принадлежать пространству имен PickACardUI:

```
namespace PickACardUI
{
    class CardPicker
    {
```

Поздравляем, вы повторно использовали класс CardPicker! Класс должен появиться в окне Solution Explorer, и вы сможете использовать его в коде своего приложения WPF.

Использование Grid и StackPanel для формирования макета главного окна

В главе 1 элемент Grid использовался для формирования макета окна игры с поиском пар. Вернитесь к той части главы, в которой создавалась структура сетки, потому что нам предстоит проделать то же самое для формирования макета окна нового приложения.

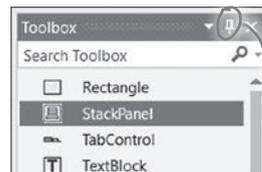
- 1 Определите строки и столбцы.** Выполните действия, описанные в главе 1, чтобы **добавить в сетку две строки и два столбца**. Если все было сделано правильно, определения строк и столбцов должны появиться под тегом <Grid> в XAML:

```
<Grid.RowDefinitions>
  <RowDefinition/>
  <RowDefinition/>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
  <ColumnDefinition/>
  <ColumnDefinition/>
</Grid.ColumnDefinitions>
```

← Используйте конструктор Visual Studio для добавления двух строк одинаковой высоты и двух столбцов одинаковой ширины. Если у вас возникнут проблемы, введите код XAML прямо в редакторе.

- 2 Добавьте элемент StackPanel.** Работать с пустым элементом StackPanel в визуальном конструкторе XAML немного неудобно, потому что на нем трудно щелкнуть. Поэтому мы выполним операцию в редакторе кода XAML. **Сделайте двойной щелчок на элементе StackPanel на панели инструментов**, чтобы добавить пустой элемент StackPanel на сетку. Это должно выглядеть примерно так:

```
</Grid.ColumnDefinitions>
<StackPanel/>
</Grid>
</Window>
```



Чтобы вам было проще перетаскивать элементы с панели инструментов, воспользуйтесь кнопкой в правом верхнем углу панели инструментов, чтобы закрепить ее в окне.

- 3 Задайте свойства StackPanel.** При двойном щелчке на элементе StackPanel на панели инструментов был добавлен элемент StackPanel без свойств. По умолчанию он находится в левом верхнем углу сетки, поэтому теперь остается настроить выравнивание и отступы. **Щелкните на теге StackPanel в редакторе XAML**, чтобы выделить тег. Когда тег будет выделен в редакторе кода, список его свойств появляется в окне свойств. Выберите режим вертикального и горизонтального выравнивания Center и установите отступы 20.



← Когда вы щелкаете на элементе в редакторе кода XAML и просматриваете его свойства в окне свойств, код XAML будет обновлен немедленно.

Сейчас элемент StackPanel в коде XAML должен выглядеть так:

```
<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center" Margin="20" />
```

Это означает, что все отступы равны 20. Стоит заметить, что свойства Margin на панели инструментов содержат набор значений «20, 20, 20, 20» — это означает то же самое.

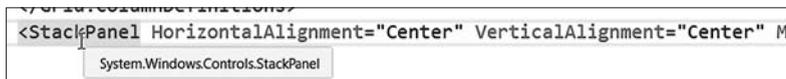
Формирование макета окна приложения Card Picker

Сформируйте макет окна приложения так, чтобы элементы управления располагались слева, а выбранные карты — справа. Элемент **StackPanel** размещается в левой верхней ячейке. Данный элемент является **контейнером**; это означает, что он содержит другие элементы, как и Grid. Но вместо того чтобы выстраивать элементы в ячейках, он выстраивает их по вертикали или по горизонтали. После того как в StackPanel будут размещены элементы Label и Slider, мы добавим элемент ListBox по аналогии с тем, как это делалось в главе 2.

Сконструируйте это!



- 1** Добавьте Label и Slider в StackPanel. StackPanel является контейнером. Когда элемент StackPanel не содержит других элементов, *он не виден в конструкторе*, а это усложняет перетаскивание на него элементов. К счастью, добавить элементы так же просто, как задать его свойства. **Щелкните на элементе StackPanel, чтобы выделить его.**



Пока элемент StackPanel остается выделенным, **сделайте двойной щелчок на элементе Label на панели инструментов**, чтобы поместить новый элемент Label *внутри* StackPanel. Элемент Label отображается в конструкторе, а в редакторе кода XAML появляется тег Label.

Затем раскройте раздел *All WPF Controls* на панели инструментов и **сделайте двойной щелчок на элементе Slider**. В левой верхней ячейке должен появиться элемент StackPanel, который содержит элемент Label над Slider.

- 2** **Задайте свойства элементов Label и Slider.** Теперь, когда элемент StackPanel содержит элементы Label и Slider, остается задать их свойства:

- ★ Щелкните на элементе Label в конструкторе. Раскройте раздел Common в окне свойств и введите текст *How many cards should I pick?*, затем раскройте раздел text и назначьте ему размер шрифта 20px.
- ★ Нажмите клавишу Escape, чтобы снять выделение с Label, **затем щелкните на элементе Slider в конструкторе**, чтобы выделить его. Воспользуйтесь полем Name в верхней части окна свойств, чтобы изменить его имя на numberOfCards.
- ★ Раскройте раздел Layout и сбросьте ширину при помощи кнопки с квадратиком (■).
- ★ Раскройте раздел Common и задайте свойству Maximum значение 15, свойству Minimum — значение 1, свойству AutoToolTipPlacement — значение TopLeft и свойству TickPlacement — значение BottomRight. Затем щелкните на стрелке (▼), чтобы раскрыть раздел Layout с дополнительными свойствами, включая свойство IsSnapToTickEnabled. Задайте ему значение true.
- ★ Сделаем шкалу ползунка более заметной. Раскройте раздел Brush в окне свойств и **щелкните на большом прямоугольнике справа от Foreground** — это позволит вам выбрать основной цвет ползунка. Щелкните на поле R и введите 0, затем также введите 0 в полях G и B. Поле Foreground должно быть черным, и метки под ползунком должны быть черными.

Код XAML должен выглядеть так (если у вас возникнут проблемы с конструктором, просто отредактируйте XAML напрямую):

```
<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center" Margin="20">
  <Label Content="How many cards should I pick?" FontSize="20"/>
  <Slider x:Name="numberOfCards" Minimum="1" Maximum="15" TickPlacement="BottomRight"
    IsSnapToTickEnabled="True" AutoToolTipPlacement="TopLeft" Foreground="Black"/>
</StackPanel>
```

3 Добавьте элемент **Button** в левую нижнюю ячейку. Перетащите элемент **Button** с панели инструментов в левую нижнюю ячейку сетки, задайте его свойства:

- ★ Раскройте раздел **Common** и задайте его свойству **Content** значение `Pick some cards`.
- ★ Раскройте раздел **Text** и задайте размер шрифта `20px`.
- ★ Раскройте раздел **Layout**. Сбросьте его отступы, ширину и высоту. Затем задайте режимы вертикального и горизонтального выравнивания **Center** ( и .

Код XAML элемента **Button** должен выглядеть так:

```
<Button Grid.Row="1" Content="Pick some cards" FontSize="20"
        HorizontalAlignment="Center" VerticalAlignment="Center" />
```

4 Добавьте элемент **ListBox**, который заполняет правую половину окна и занимает две строки. Перетащите элемент **ListBox** в правую верхнюю ячейку и задайте его свойства.

- ★ Используйте поле **Name** в верхней части окна свойств, чтобы задать **ListBox** имя `listOfCards`.
- ★ Раскройте раздел **Text** и задайте размер шрифта `20px`.
- ★ Раскройте раздел **Layout**. Задайте отступы размером `20`, как это было сделано для элемента **StackPanel**. Проследите за тем, чтобы ширина, высота, горизонтальное и вертикальное выравнивание были сброшены.
- ★ Убедитесь в том, что поле **Row** содержит значение `0`, а поле **Column** — значение `1`. Затем **задайте RowSpan значение 2**, чтобы элемент **ListBox** занимал весь столбец и охватывал обе строки:

Row	<input type="text" value="0"/>	<input type="checkbox"/>	RowSpan	<input type="text" value="2"/>	<input checked="" type="checkbox"/>
Column	<input type="text" value="1"/>	<input checked="" type="checkbox"/>	ColumnSpan	<input type="text" value="1"/>	<input type="checkbox"/>

Код XAML элемента **ListBox** должен выглядеть так:

```
<ListBox x:Name="listOfCards" Grid.Column="1" Grid.RowSpan="2"
        FontSize="20" Margin="20,20,20,20"/>
```

Если свойство содержит значение "20" вместо "20, 20, 20, 20", это нормально — фактически это то же самое.

5 **Задайте текст заголовка и размер окна.** Когда вы создаете новое приложение WPF, Visual Studio создает главное окно шириной 450 пикселей и высотой 800 пикселей с заголовком «Main Window». Изменим его размеры по аналогии с тем, как это делалось в игре с поиском пар:

- ★ Щелкните на строке заголовка окна в конструкторе, чтобы выделить окно.
- ★ В разделе **Layout** задайте ширину `300`.
- ★ В разделе **Common** введите текст заголовка `Card Picker`.

Прокрутите редактор XAML и найдите последнюю строку тега **Window**. Она содержит следующие свойства:

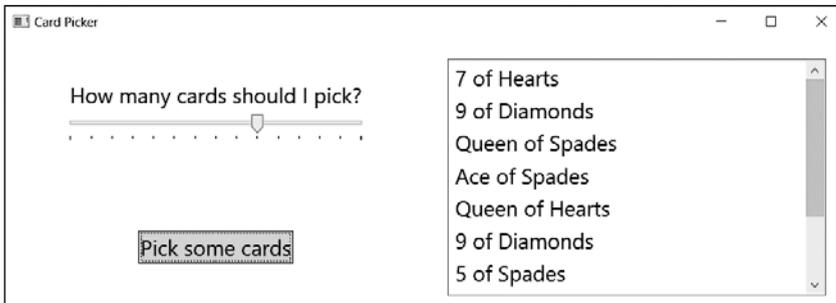
```
Title="Card Picker" Height="300" Width="800"
```

6 Добавьте обработчик события Click в элемент Button. Код программной части — код C# из файла MainWindow.xaml.cs, связанного с кодом XAML, — состоит из одного метода. Сделайте двойной щелчок на кнопке в конструкторе; IDE добавляет метод Button_Click и назначает его обработчиком события Click, как было показано в главе 1. Код нового метода:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    string[] pickedCards = CardPicker.PickSomeCards((int)numberOfCards.Value);
    listOfCards.Items.Clear();
    foreach (string card in pickedCards)
    {
        listOfCards.Items.Add(card);
    }
}
```

Запустите свое приложение. При помощи ползунка выберите количество случайных карт, а затем нажмите кнопку, чтобы заполнить ими список ListBox.

Отличная работа!



Код C#, связанный с окном XAML и содержащий обработчики событий, называется кодом программной части.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Классы содержат методы, а методы состоят из команд, выполняющих действия. В хорошо спроектированных классах используются содержательные имена методов.
- Некоторые методы имеют **возвращаемый тип**, который задается в объявлении метода. Метод с объявлением, начинающимся с ключевого слова `int`, возвращает значение `int`. Пример команды, возвращающей значение `int`: `return 37`.
- Если метод имеет возвращаемый тип, он **должен** содержать команду `return` со значением, соответствующим возвращаемому типу. Таким образом, если в объявлении метода указан строковый возвращаемый тип, этот метод должен содержать команду `return`, которая возвращает строку.
- Как только в методе будет выполнена команда `return`, программа возвращается к команде, из которой был вызван метод.
- Не все методы имеют возвращаемый тип. Метод с объявлением, начинающимся с `public void`, ничего не возвращает. При этом для выхода из метода `void` может использоваться команда `return`: `if (finishedEarly) { return; }`
- Разработчики часто хотят **повторно использовать** один код в разных программах. Классы помогают расширить возможности повторного использования кода.
- Когда вы **выбираете элемент** в редакторе XAML, свойства этого элемента можно изменить в окне свойств.

Прототипы Анны выглядят замечательно...

Анна обнаружила, что кто бы ни преследовал игрока в ее игре — инопланетянин, пират, зомби или злобный клоун, для представления врага можно использовать одни и те же методы ее класса Enemy. Ее игра начинает понемногу обретать форму.

...но что, если противников должно быть несколько?

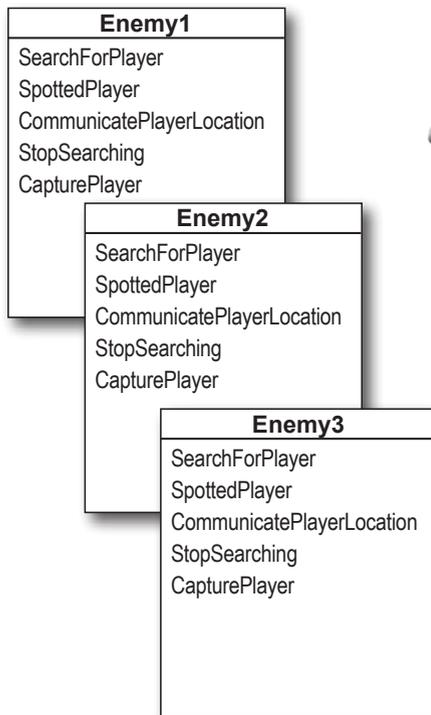
И все идет замечательно... Пока Анна не решает использовать более одного врага, который присутствовал в ее ранних прототипах. Что ей следует сделать, чтобы добавить в игру второго или третьего врага?

Анна *может* скопировать код класса Enemy и вставить его еще в два файла. Тогда ее программа сможет использовать методы для управления сразу тремя разными врагами. Так что формально код используется повторно... разве нет?

Эй, Анна, что скажешь?

Enemy
SearchForPlayer
SpottedPlayer
CommunicatePlayerLocation
StopSearching
CapturePlayer

В ее словах есть смысл. А если ей потребуется уровень, на котором бегают десятки зомби? Создавать десятки одинаковых классов просто непрактично.



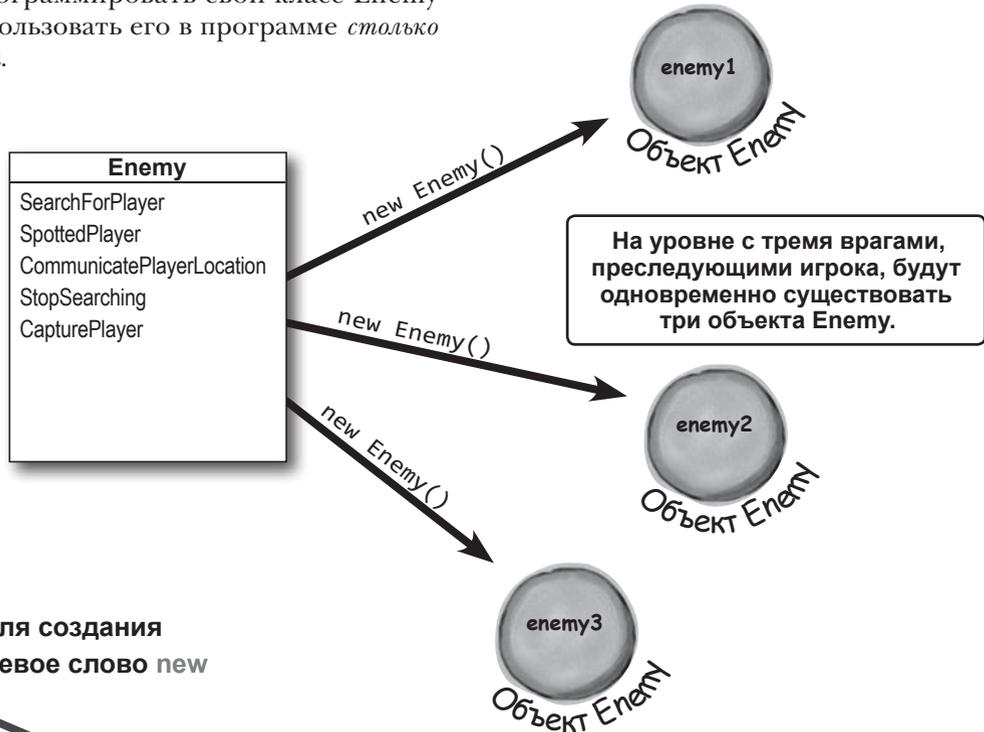
Вы шутите? Использование отдельных идентичных классов для разных врагов — **ужасная мысль**. А если в какой-то момент мне понадобится более трех врагов?

Сопровождение трех копий одного кода сулит массу проблем.

Во многих задачах, которые нам приходится решать, требуется представить **нечто** многими разными способами. В данном случае это враг в видеоигре, но могли бы быть песни в музыкальном проигрывателе или контакты в социальной сети. У всех этих данных имеется одна общая особенность: всем им требуется одинаково работать с некоторыми данными, сколько бы экземпляров этих данных у них ни было. Попробуем найти более удачное решение.

Анна может воспользоваться объектами для решения своей задачи

Объекты в C# предназначены для работы с наборами похожих сущностей. Анна может воспользоваться объектами для того, чтобы запрограммировать свой класс Enemy только один раз и использовать его в программе *столько раз, сколько потребуется*.



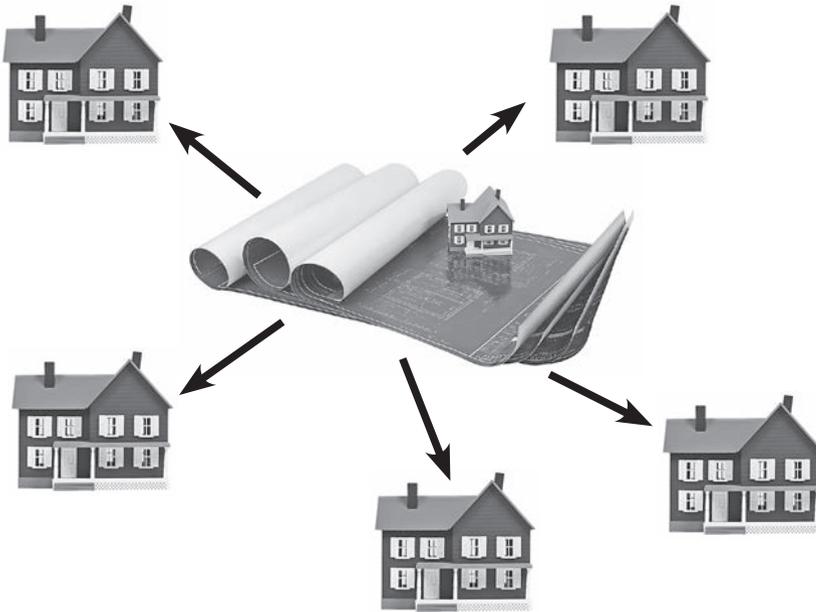
Все, что нужно для создания объекта, — ключевое слово `new` и имя класса.

```
Enemy enemy1 = new Enemy();
enemy1.SearchForPlayer();
if (enemy1.SpottedPlayer()) {
    enemy1.CommunicatePlayerLocation();
} else {
    enemy1.StopSearching();
}
```

Теперь объект может использоваться в программе! Когда вы создаете объект на базе класса, этот объект содержит все методы, определенные в классе.

Класс используется для построения объектов

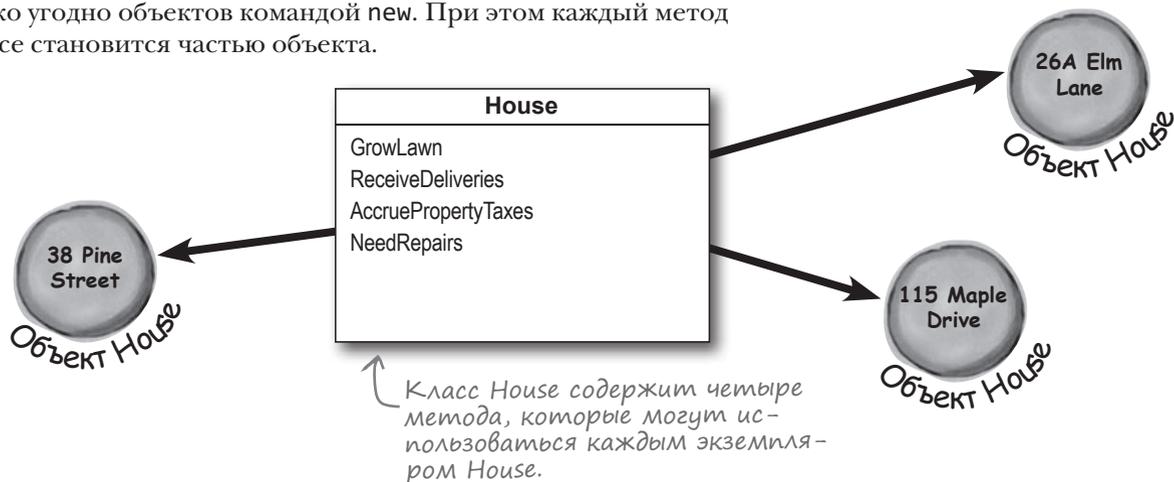
Класс напоминает план для построения объекта. Чтобы построить пять одинаковых домов в пригородной зоне, никто не станет заказывать у архитектора пять комплектов одинаковых планов. Вы просто используете один план для всех пяти домов.



Класс определяет свои компоненты по аналогии с тем, как план определяет компоновку дома. Один план может использоваться для строительства любого количества домов; точно так же на базе одного класса можно создать любое количество объектов.

Объект получает методы от класса

После того как класс будет создан, вы можете создать на его базе сколько угодно объектов командой new. При этом каждый метод в классе становится частью объекта.



Новый объект, созданный на базе класса, называется экземпляром этого класса

Для создания объекта может использоваться **ключевое слово new**. От вас потребуется только переменная, которая будет использоваться для обращения к объекту. Класс указывается как тип для объявления переменной, так что вместо `int` или `bool` следует указать имя класса — например, `House` или `Enemy`.

До создания объекта: так выглядит память вашего компьютера при запуске программы.



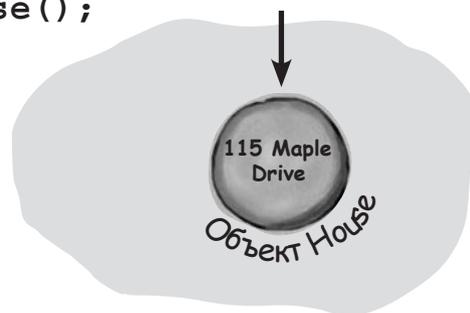
В вашей команде выполняется команда `new`.

```
House mapleDrive115 = new House();
```

↑
Команда `new` создает новый объект `House` и присваивает его переменной с именем `mapleDrive115`.

Эк-зем-пляр, *сущ.*
Копия или отдельное вхождение чего-либо.

После создания объекта: теперь в памяти хранится экземпляр класса `House`.



Ключевое слово `new` выглядит знакомо. Я его уже где-то видел.

Да! Вы уже создавали экземпляры в своем коде.

Вернитесь к программе с поиском пар и найдите следующую строку кода:

```
Random random = new Random();
```

Мы создали экземпляр класса `Random`, а затем вызвали метод `Next`. Теперь просмотрите код класса `CardPicker` и найдите команду `new`. Выходит, мы уже давно пользуемся объектами!

Хорошее решение для Анны (с объектами)

Анна повторно использовала код класса Enemy без хлопот с копированием, которое бы привело к дублированию кода в проекте. Вот как она это сделала.

- 1 Анна создала класс Level, который хранит врагов в массиве Enemy с именем enemies, — подобно тому, как массивы строк использовались для хранения карт и эмодзи с животными.

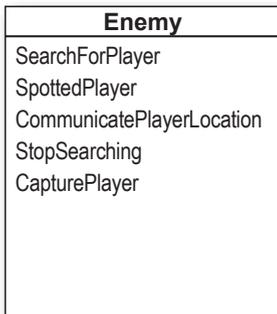
```
public class Level {
    Enemy[] enemyArray = new Enemy[3];
}
```

Используйте имя класса для объявления массива экземпляров этого класса.

ХМ, этот массив находится внутри класса, но за пределами методов. Как вы думаете, что здесь происходит?

Ключевое слово new используется для создания массива объектов Enemy по аналогии с тем, как это делалось со строками.

- 2 Она воспользовалась циклом, в котором выполнялись команды new для создания новых экземпляров класса Enemy для текущего уровня, а созданные экземпляры добавлялись в массив врагов.



```
for (int i = 0; i < 3; i++)
{
    Enemy enemy = new Enemy();
    enemyArray[i] = enemy;
}
```

Объект enemy1 является экземпляром класса Enemy.

Эта команда использует ключевое слово new для создания объекта Enemy.

Эта команда добавляет только что созданный объект Enemy в массив.

- 3 Методы каждого экземпляра Enemy вызываются при каждом обновлении кадра для реализации поведения врагов.



```
foreach (Enemy enemy in enemyArray)
{
    // Код, содержащий вызовы методов Enemy
}
```

Цикл foreach перебирает содержимое массива объектов Enemy.





Минутку! Этой информации **даже отдаленно** не хватит для построения игры Анны.

Верно, не хватает.

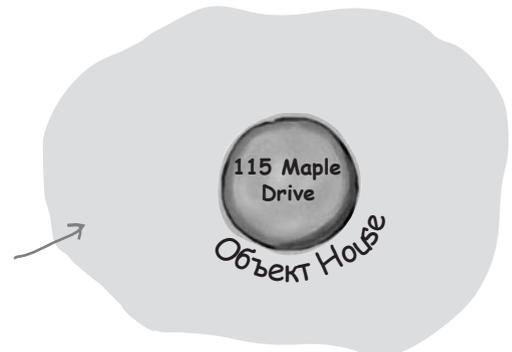
Одни прототипы игр очень просты, другие устроены намного сложнее, но сложные программы *строятся на базе тех же паттернов*, что и простые. Программа Анны является примером того, как бы вы использовали объекты в реальной жизни. И этот принцип применим не только для разработки игр! Какую бы программу вы ни строили, объекты в ней будут использоваться точно так же, как их использовала Анна в своей программе. Пример Анны — всего лишь отправная точка для закрепления этой концепции в вашем мозгу. Мы приведем еще **очень много примеров** в оставшейся части главы — и эта концепция настолько важна, что мы еще вернемся к ней в будущих главах.

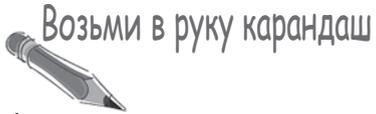
Теория и практика

Раз уж речь зашла о паттернах, существует один паттерн, который мы еще неоднократно увидим в книге. Мы представляем новую концепцию или идею (например, объекты) на нескольких страницах, используя картинки и короткие фрагменты кода для ее пояснения. Это дает вам возможность задержаться и попробовать разобраться в происходящем, не отвлекаясь на то, как заставить работать конкретную программу.

```
House mapleDrive115 = new House();
```

Когда в книге вводится новая концепция (например, объекты), обращайтесь особое внимание на картинки и фрагменты кода.





Итак, теперь вы лучше представляете, как работают объекты. Самое время вернуться к классу CardPicker и поближе познакомиться с классом Random, который в нем используется.

1. Установите курсор внутри любого метода, нажмите клавишу Enter, чтобы начать новую команду, и введите **random**. — как только вы введете точку, Visual Studio открывает окно IntelliSense со списком методов. Каждый метод помечается значком в виде кубика (⊞). Мы заполнили некоторые методы. Допишите отсутствующие методы в диаграмму класса Random.

Random
Equals
GetHashCode
GetType
.....
.....
.....
ToString

2. Напишите код создания нового массива double с именем **randomDoubles**. Добавьте в массив 20 значений double в цикле **for**. Добавляйте только случайные числа с плавающей точкой в интервале от 0.0 (включительно) до 1.0 (не включая). Используйте окно IntelliSense для выбора метода класса Random, который должен использоваться в вашем коде.

```

Random random =
.....
double[] randomDoubles = new double[20];
.....
{
.....
double value =
.....
}
.....

```

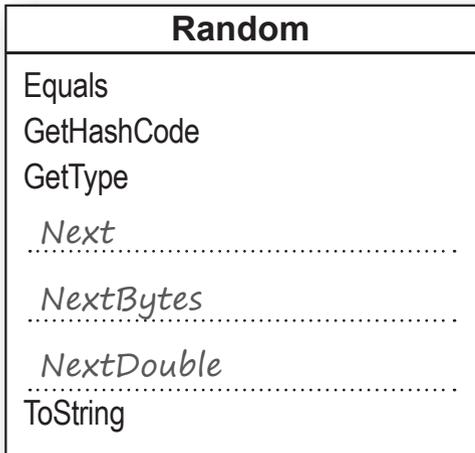
Мы заполнили часть кода, включая фигурные скобки. Ваша задача — дописать эти команды, а затем написать остальной код.



Возьми в руку карандаш Решение

Итак, теперь вы лучше представляете, как работают объекты. Самое время вернуться к классу `CardPicker` и поближе познакомиться с классом `Random`, который в нем используется.

1. Установите курсор внутри любого метода, нажмите клавишу `Enter`, чтобы начать новую команду, и введите `random.` — как только вы введете точку, Visual Studio открывает окно IntelliSense со списком методов. Каждый метод помечается значком в виде кубика (☒). Мы заполнили некоторые методы. Допишите отсутствующие методы в диаграмму класса `Random`.



`random.`

- ☒ Equals
- ☒ GetHashCode
- ☒ GetType
- ☒ **Next** int Random.Next() (+ 2 overloads)
Returns a non-negative random integer.
- ☒ NextBytes
- ☒ NextDouble
- ☒ ToString

Окно IntelliSense, которое появляется в Visual Studio, когда вы вводите «`random.`» в одном из методов `CardPicker`.

Когда вы выбираете `NextDouble` в окне IntelliSense, появляется краткая документация по использованию метода.

```
double Random.NextDouble()
Returns a random floating-point number that is greater than or equal to 0.0, and less than 1.0.
```

2. Напишите код создания нового массива `double` с именем `randomDoubles`. Добавьте в массив 20 значений `double` в цикле `for`. Добавляйте только случайные числа с плавающей точкой в интервале от 0.0 (включительно) до 1.0 (не включая). Используйте окно IntelliSense для выбора метода класса `Random`, который должен использоваться в вашем коде.

```
Random random = new Random();
double[] randomDoubles = new double[20];
for (int i = 0; i < 20; i++)
{
    double value = random.NextDouble();
    randomDoubles[i] = value;
}
```

Этот код очень похож на тот, который использовался в классе `CardPicker`.

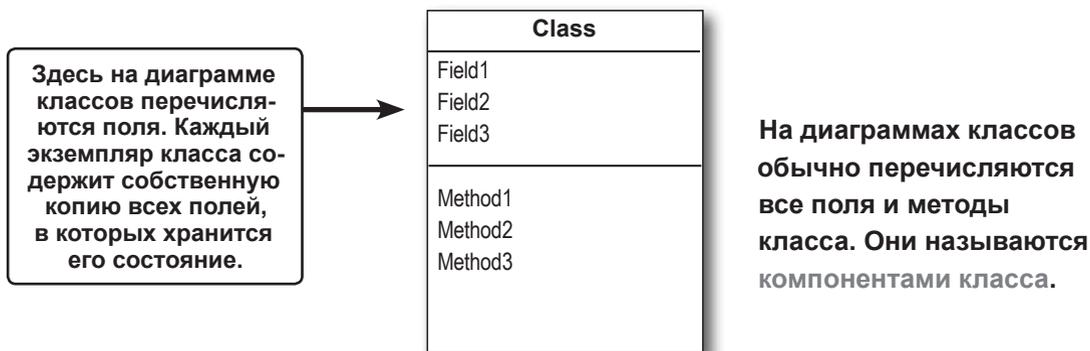
Экземпляры хранят данные в полях

Вы видели, что классы могут содержать как поля, так и методы. Также вы видели, как ключевое слово `static` используется для объявления поля в классе `CardPicker`:

```
static Random random = new Random();
```

Что произойдет, если убрать ключевое слово `static`? Поле становится **полем экземпляра**, и каждый раз, когда вы создаете экземпляр класса, новый экземпляр получает *собственную копию* этого поля.

Чтобы включить поля в диаграмму класса, разделите прямоугольник класса горизонтальной линией. Над линией перечисляются поля, а методы перечисляются под линией.



Методы определяют, что делает объект. Поля содержат информацию, известную объекту.

Когда прототип Анны создал три экземпляра класса `Enemy`, каждый из этих объектов использовался для отслеживания отдельного врага в игре. В каждом экземпляре хранится индивидуальная копия одних и тех же данных: изменение поля экземпляра `enemy2` никак не влияет на экземпляры `enemy1` или `enemy3`.



Поведение объекта определяется его методами, а поля используются для отслеживания его состояния.

Я использовал ключевое слово `new` для создания экземпляра `Random`, но **нигде не создавал новый экземпляр** класса `CardPicker`. Означает ли это, что методы могут вызываться без создания объектов?



Да! Именно для этого в объявлении было использовано ключевое слово `static`.

Взгляните еще раз на начальные строки класса `CardPicker`:

```
class CardPicker
{
    static Random random = new Random();
    public static string PickSomeCards(int numberOfCards)
```

Если вы указали ключевое слово **`static`** при объявлении поля или метода в классе, то для обращения к этому полю или методу не обязательно указывать экземпляр класса. Вы просто вызываете метод с указанием имени класса:

```
CardPicker.PickSomeCards(numberOfCards)
```

Так вызываются статические методы. Если же в объявлении класса `PickSomeCards` отсутствует ключевое слово **`static`**, то для вызова метода придется создать экземпляр `CardPicker`. Если не считать этого различия, статические методы ведут себя точно так же, как методы объектов: они могут получать аргументы, могут возвращать значения и существуют внутри классов.

У статического поля существует **только одна копия, которая совместно используется всеми экземплярами**. Следовательно, если вы создадите несколько экземпляров `CardPicker`, все они будут использовать одно и то же поле `random`. Статическим даже можно объявить целый класс; тогда все поля и методы этого класса **должны** быть статическими. Если вы попытаетесь добавить нестатический метод в статический класс, то ваша программа не построится.

Часто задаваемые вопросы

В: Когда что-то называют «статическим», обычно имеется в виду, что оно остается неизменным. Означает ли это, что нестатические методы могут изменяться, а статические методы — нет? Они ведут себя по-разному?

О: Нет, статические методы ведут себя абсолютно одинаково. Единственное различие заключается в том, что статические методы не связаны с конкретным экземпляром, а нестатические — связаны.

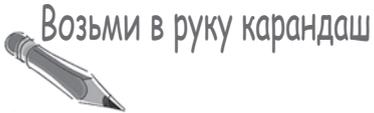
В: Значит, я не смогу использовать класс, пока не создам экземпляр объекта?

О: Вы сможете использовать его статические методы, но если класс содержит нестатические методы, то перед их использованием придется создать экземпляр.

В: Зачем нужны методы, которым необходим экземпляр? Почему бы не объявить все методы статическими?

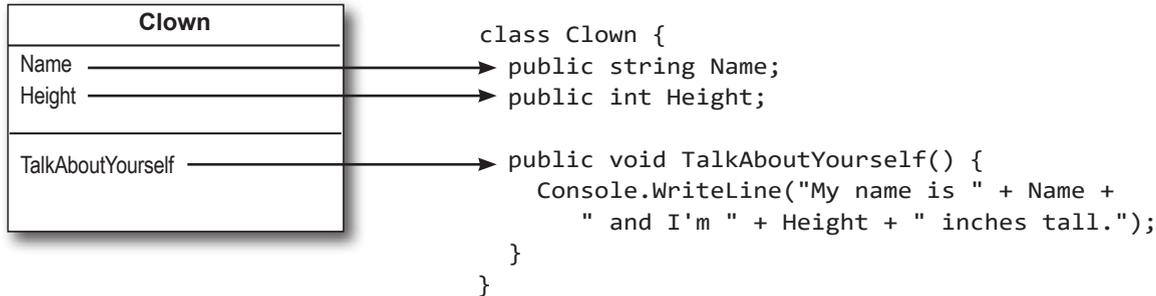
О: Потому, что если ваш объект отслеживает некоторые данные (как экземпляры класса `Enemy`, в которых хранились сведения о разных врагах в игре), методы каждого экземпляра могут использоваться для работы с этими данными. Таким образом, когда игра Анны вызывает метод `StopSearching` для экземпляра `enemy2`, только этот враг перестает искать игрока. Вызов метода никак не влияет на объекты `enemy1` и `enemy3`, они продолжают поиски. Так Анна может создавать прототипы игры с любым количеством врагов, и ее программы смогут отслеживать всех этих врагов одновременно.

Если поле объявлено статическим, его единственная копия совместно используется всеми экземплярами.



Перед вами консольное приложение .NET, которое выводит несколько строк на консоль. Приложение включает класс Clown с двумя полями, Name и Height, а также метод с именем TalkAboutYourself. Вам предлагается прочитать код и записать строки, которые будут выведены на консоль.

Диаграмма класса и код класса Clown:



Далее следует метод Main консольного приложения. Рядом с каждым вызовом метода TalkAboutYourself, который выводит строку на консоль, располагается комментарий. Заполните пропуски в комментариях, чтобы они соответствовали выводимым данным.

```

static void Main(string[] args) {
    Clown oneClown = new Clown();
    oneClown.Name = "Boffo";
    oneClown.Height = 14;
    oneClown.TalkAboutYourself();    // My name is _____ and I'm ____ inches tall."

    Clown anotherClown = new Clown();
    anotherClown.Name = "Biff";
    anotherClown.Height = 16;
    anotherClown.TalkAboutYourself(); // My name is _____ and I'm ____ inches tall."

    Clown clown3 = new Clown();
    clown3.Name = anotherClown.Name;
    clown3.Height = oneClown.Height - 3;
    clown3.TalkAboutYourself();    // My name is _____ and I'm ____ inches tall."

    anotherClown.Height *= 2;
    anotherClown.TalkAboutYourself(); // My name is _____ and I'm ____ inches tall."
}

```

*Оператор *= приказывает C# взять операнд, который стоит слева от оператора, и умножить его на операнд в правой части, после чего эта команда обновляет поле Height.*

Куча

Когда программа создает объект, он существует в части компьютерной памяти, которая называется **кучей**. Когда ваш код создает объект командой `new`, C# немедленно резервирует в куче память, где будут храниться данные этого объекта.

Память при запуске приложения.
Как видите, она пуста.



Когда программа создает новый объект, он добавляется в кучу.



Возьми в руку карандаш

Решение

Ниже приведены результаты, которые будут выводиться программой на консоль. Выделите несколько минут на создание нового консольного приложения .NET, добавьте класс `Clown`, определите в нем метод `Main` и выполните его в отладчике в пошаговом режиме.

```
static void Main(string[] args) {
    Clown oneClown = new Clown();
    oneClown.Name = "Boffo";
    oneClown.Height = 14;
    oneClown.TalkAboutYourself(); // My name is Boffo and I'm 14 inches tall."

    Clown anotherClown = new Clown();
    anotherClown.Name = "Biff";
    anotherClown.Height = 16;
    anotherClown.TalkAboutYourself(); // My name is Biff and I'm 16 inches tall."

    Clown clown3 = new Clown();
    clown3.Name = anotherClown.Name;
    clown3.Height = oneClown.Height - 3;
    clown3.TalkAboutYourself(); // My name is Biff and I'm 11 inches tall."

    anotherClown.Height *= 2;
    anotherClown.TalkAboutYourself(); // My name is Biff and I'm 32 inches tall."
}
```

При выполнении этого метода в отладчике в пошаговом режиме значение поля `Height` должно стать равным 14 после выполнения этой строки.

Эта строка использует поле `Height` старого экземпляра `oneClown` для заполнения поля `Height` нового экземпляра `clown3`.

Что на уме у вашей программы

Присмотримся повнимательнее к программе из упражнения «Возьми в руку карандаш», начиная с первой строки метода Main. В действительности это **две команды**, объединенные в одну:

```
Clown oneClown = new Clown();
```

Эта команда объявляет переменную с именем `oneClown` типа `Clown`.

Эта команда создает новый объект и присваивает его переменной `oneClown`.

А теперь посмотрим, как выглядит куча после выполнения каждой группы команд:

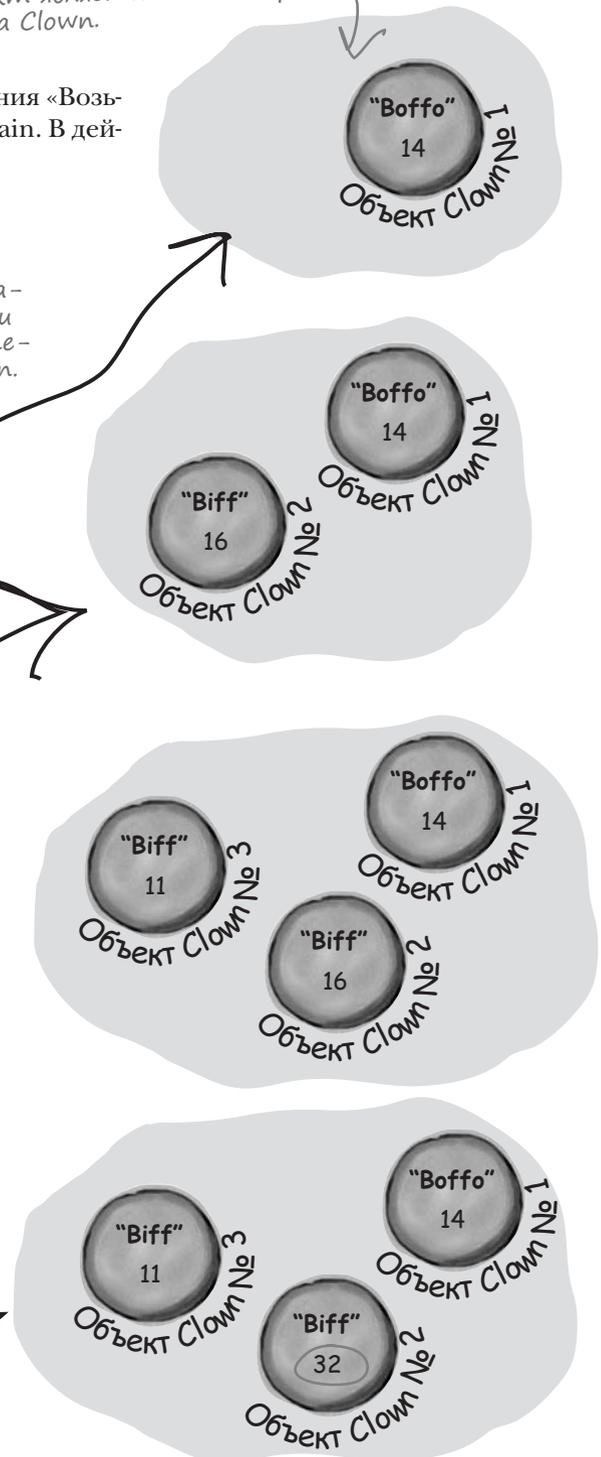
```
// Эти команды создадут экземпляр класса Clown
// и присваивают значения его полям
Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;
oneClown.TalkAboutYourself();
```

```
// Эти команды создадут второй объект Clown
// и заполняют его данными
Clown anotherClown = new Clown();
anotherClown.Name = "Biff";
anotherClown.Height = 16;
anotherClown.TalkAboutYourself();
```

```
// При создании третьего объекта Clown
// мы используем данные двух других экземпляров
// для присваивания значений его полям
Clown clown3 = new Clown();
clown3.Name = anotherClown.Name;
clown3.Height = oneClown.Height - 3;
clown3.TalkAboutYourself();
```

```
// Обратите внимание на отсутствие команды new
// Мы не создаем новый объект, а изменяем
// объект, уже существующий в памяти
anotherClown.Height *= 2;
anotherClown.TalkAboutYourself();
```

Объект является экземпляром класса `Clown`.



Иногда код плохо читается

Даже если вы не осознаете этого, вы постоянно принимаете решения относительно структуры вашего кода. Вы используете один метод для выполнения некоторой операции? Разбиваете его на несколько методов? А нужен ли вообще новый метод? Решения, принимаемые вами относительно методов, делают ваш код более наглядным и доступным или, если вы действуете неосторожно, намного более запутанным.

Ниже приведен компактный блок кода из программы, управляющей машиной для изготовления шоколадных батончиков:

```
int t = m.chkTemp();
if (t > 160) {
    T tb = new T();
    tb.clsTrpV(2);
    ics.Fill();
    ics.Vent();
    m.airsyschk();
}
```

Слишком компактный код может создать проблемы

Присмотритесь к этому коду. Сможете ли вы разобраться, что он делает? Не огорчайтесь, если вам это не удастся — он очень плохо читается! Это объясняется несколькими причинами:

- ★ В программе используются переменные с именами `tb`, `ics` и `m`. Имена выбраны просто ужасно! Мы понятия не имеем, что они делают. И для чего нужен класс `T`?
- ★ Метод `chkTemp` возвращает целое число... но что он делает? По имени можно предположить, что он проверяет температуру... чего?
- ★ Метод `clsTrpV` получает один параметр. Что должен содержать этот параметр? Почему он равен 2? И для чего нужно число 160?



Код C# в промышленном оборудовании?!
Разве C# не предназначен для настольных приложений,
бизнес-систем, веб-сайтов и игр?

C# и .NET применяются везде... буквально везде.

Вы когда-нибудь развлекались с Raspberry PI? Это недорогой одноплатный компьютер, и такие компьютеры можно найти на самых разных устройствах. Благодаря концепции Windows IoT («Интернет вещей») ваш код C# может выполняться на таких компьютерах. Существует бесплатная версия для построения прототипов, так что вы можете начать эксперименты с оборудованием в любой момент.

О приложениях .NET IoT можно больше узнать по адресу <https://dotnet.microsoft.com/apps/iot>.

Руководства к программам обычно не прилагаются

Такие команды обычно не дают никаких подсказок относительно того, почему код делает то, что он делает. В данном случае программист был доволен результатом, потому что ему удалось упаковать весь код в один метод. Но компактность кода сама по себе особой пользы не приносит! Разобьем его на методы, чтобы код лучше читался, и позаботимся о том, чтобы имена, присвоенные классам, были содержательными.

Для начала разберемся, что должен делать код. К счастью, нам известно, что этот код является частью **встроенной системы**, или контроллера, являющегося частью большей электрической или механической системы. И еще нам повезло, что мы располагаем документацией по этому коду, а конкретно руководством, которое использовалось программистами при исходном построении системы.

Руководство для машины по изготовлению шоколадных батончиков General Electronics Type 5

Температура нуги должна проверяться каждые 3 минуты автоматизированной системой. Если температура превышает **160 °С**, начинка слишком горячая, поэтому система должна **выполнить процедуру вентиляции изолированной системы охлаждения (CICS)**:

- Закрыть клапан регулятора на турбине № 2.
- Заполнить изолированную систему охлаждения сплошным потоком воды.
- Спустить воду.
- Запустить автоматизированную проверку присутствия воздуха в системе.

Как определить, что должен делать ваш код? Весь код пишется по какой-то причине. А значит, вам придется определить эту причину самостоятельно! В данном случае нам повезло — можно обратиться к руководству, по которому работал программист.

Сравним код с руководством, в котором говорится, что должен делать код. Добавление комментариев определенно поможет разобраться в происходящем:

```
/* Этот код выполняется каждые 3 минуты для проверки температуры
 * Если температура превышает 160 °С, необходимо запустить систему охлаждения.
 */
int t = m.chkTemp();
if (t > 160) {
    // Получить систему управления для турбин
    T tb = new T();

    // Закрыть клапан регулятора на турбине 2
    tb.clsTrpV(2);

    // Заполнить и включить вентиляцию изолированной
    // системы охлаждения
    ics.Fill();
    ics.Vent();

    // Запустить проверку воздуха в системе
    m.airsyschk();
}
```

Добавление пустых строк в некоторых местах упрощает чтение вашего кода.



МОЗГОВОЙ ШТУРМ

Комментарии в коде — хорошее начало. А вы сможете предложить еще какие-нибудь решения, которые сделают этот код еще более понятным?

Использование содержательных имен классов и методов

Страница из руководства значительно упростила понимание кода. Кроме того, она содержит ряд полезных подсказок, которые позволяют разобраться в логике кода. Посмотрим на первые две строки:

```
/* Этот код выполняется каждые 3 минуты для проверки температуры.
 * Если температура превышает 160С, необходимо запустить систему охлаждения.
 */
int t = m.chkTemp();
if (t > 160) {
```

Добавленный комментарий многое объясняет. Теперь мы знаем, почему условная команда сравнивает переменную `t` с `160°`, — в руководстве говорится, что при любой температуре, превышающей `160°С`, нуга становится слишком горячей. Как выясняется, `m` — класс, управляющий машиной для производства шоколадных батончиков; он содержит статические методы для проверки температуры нуги и проверки системы охлаждения.

Давайте выделим проверку температуры в отдельный метод и выберем для класса и методов имена, которые наглядно поясняют их предназначение. Первые две строки будут выделены в отдельный метод, который возвращает логическое значение: `true`, если нуга слишком горячая, или `false` при нормальной температуре:

```
/// <summary>
/// Если температура нуги превышает 160 °С, она слишком горячая.
/// </summary>
public bool IsNougatTooHot() {
    int temp = CandyBarMaker.CheckNougatTemperature();
    if (temp > 160) {
        return true;
    } else {
        return false;
    }
}
```

После того как классу будет присвоено имя «`CandyBarMaker`», а методу — имя «`CheckNougatTemperature`», код становится более понятным.

А вы заметили, что буква `C` в имени `CandyBarMaker` имеет верхний регистр? Если имена классов всегда начинаются с буквы верхнего регистра, а имена переменных — с букв нижнего регистра, вам будет проще понять, когда вызывается статический метод, а когда вызывается метод экземпляра.

А вы заметили специальные комментарии `///` над методом? Они называются *документирующими комментариями XML*. IDE использует эти комментарии для вывода справки по методам вроде той, которая выводилась, когда вы использовали окно IntelliSense для выбора метода класса `Random`.

Об использовании IDE: документация XML для методов и полей

Visual Studio помогает добавлять документацию XML. Установите курсор в строке над любым методом, введите три символа `/`, и IDE добавит пустой шаблон для документации. Если ваш метод имеет параметры и возвращаемый тип, для них также будут добавлены теги `<param>` и `<returns>`. Попробуйте вернуться к классу `CardPicker` и ввести `///` в строке над методом `PickSomeCards` — IDE добавит пустой шаблон документации XML. Заполните его и убедитесь в том, что информация появляется в IntelliSense.

```
/// <summary>
/// Выбирает несколько карт и возвращает их
/// </summary>
/// <param name="numberOfCards">Количество выбираемых карт.</param>
/// <returns>Массив строк с названиями карт.</returns>
```

Документацию XML также можно создавать и для полей. Попробуйте установить курсор в строке над любым полем и ввести три символа `/` в IDE. Весь текст, который следует за `<summary>`, выводится в окне IntelliSense для этого поля.

Что предлагает делать руководство, если нуга слишком горячая? Оно предлагает выполнить процедуру вентиляции изолированной системы охлаждения (CICS). Создадим другой метод, выберем содержательное имя для класса T (который, как выяснилось, управляет турбиной) и для класса ics (который управляет системой охлаждения и содержит два статических метода для заполнения и вентиляции системы), а заодно дополним все краткой документацией XML:

```

/// <summary>
/// Выполнить процедуру вентиляции изолированной системы охлаждения (CICS).
/// </summary>
public void DoCICSVentProcedure() {
    TurbineController turbines = new TurbineController();
    turbines.CloseTripValve(2);
    IsolationCoolingSystem.Fill();
    IsolationCoolingSystem.Vent();
    Maker.CheckAirSystem();
}

```

Когда ваш метод объявляется с возвращаемым типом `void`, это означает, что он не возвращает значения и команда `return` ему не нужна. Во всех методах, написанных вами в предыдущей главе, использовалось ключевое слово `void`!

Итак, теперь у нас имеются методы `IsNougatTooHot` и `DoCICSVentProcedure` и мы можем переписать исходный невразумительный код в виде нового метода и присвоить ему имя, которое четко выражает, что он делает:

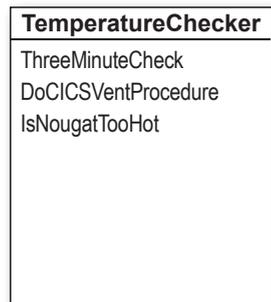
```

/// <summary>
/// Этот код выполняется каждые 3 минуты для проверки температуры.
/// Если температура превышает 160 °C, необходимо запустить систему охлаждения.
/// </summary>
public void ThreeMinuteCheck() {
    if (IsNougatTooHot() == true) {
        DoCICSVentProcedure();
    }
}

```

Код стал намного более понятным! Даже если вы не знаете, что процедура вентиляции CICS должна выполняться, если нуга слишком горячая, стало намного понятнее, что делает этот код.

Мы упаковали новые методы в класс с именем `TemperatureChecker`. Диаграмма класса выглядит так:



Используйте диаграммы классов для планирования

Диаграмма класса — полезный инструмент для проектирования вашего кода ДО ТОГО, как вы начнете писать этот код. Запишите имя класса в верхней части диаграммы. Затем запишите все методы в прямоугольнике в нижней части. Теперь все части класса видны с первого взгляда, а у вас появляется возможность обнаружить проблемы, которые затруднят использование вашего кода или его понимание.



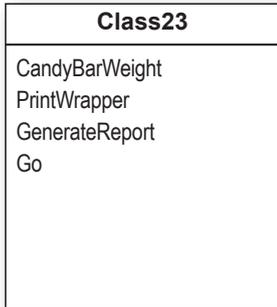
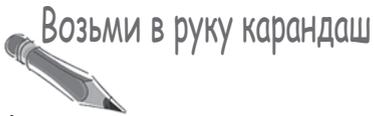
Постойте, сейчас мы сделали нечто действительно интересное! Мы только что внесли множество изменений в блок кода. Теперь он выглядит совершенно иначе и читается намного проще, но **при этом** **делает то же самое.**

Все верно. Когда вы изменяете структуру кода без изменения его поведения, этот процесс называется рефакторингом.

Опытные разработчики пишут предельно простой и понятный код (в том числе и вам, если вы не прикасались к нему в течение долгого времени). Комментарии полезны, но ничто не сравнится с выбором осмысленных имен для методов, классов, переменных и полей.

Чтобы упростить чтение и написание кода, вам следует основательно задуматься над задачей, для которой был написан ваш код. Если выбрать имена методов, которые будут понятны каждому, кто понимает вашу задачу, то ваш код будет намного проще как для анализа, так и для разработки. Как бы мы ни планировали свой код, почти никогда не удастся сделать все правильно с первого раза.

Вот почему *опытные разработчики постоянно проводят рефакторинг своего кода*. Они перемещают свой код в методы и присваивают этим методам осмысленные имена. Они переименовывают переменные. Каждый раз, когда они обнаруживают код, не очевидный на сто процентов, они тратят несколько минут на его рефакторинг. Они знают, что это время не пропадет даром, потому что эта процедура упростит добавление нового кода через час (через день, месяц или год!).



В каждом из этих классов имеется серьезный структурный недостаток. Напишите, что, по вашему мнению, не так с каждым классом и как бы вы решили проблему.

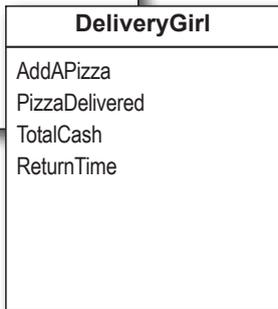
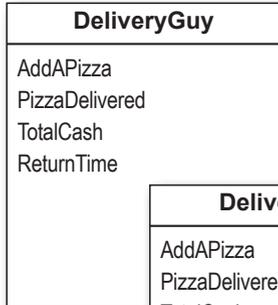
Этот класс является частью системы производства шоколадных батончиков, описанной выше.

.....

.....

.....

.....



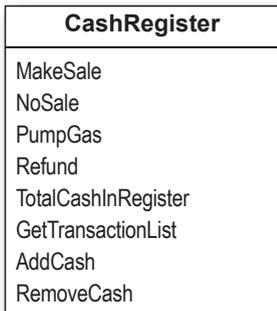
Эти два класса являются частью системы, которая используется пиццерией для отслеживания заказов, переданных в доставку.

.....

.....

.....

.....



Класс CashRegister является частью программы, используемой системой оплаты в круглосуточном магазине при автозаправке.

.....

.....

.....

.....



Возьми в руку карандаш Решение

Ниже показаны предложенные нами исправления. Мы показываем лишь один из возможных способов решения проблемы, тем не менее, существует много других способов исправления архитектуры этих классов в зависимости от их использования.

Этот класс является частью системы производства шоколадных батончиков, описанной выше.

Имя класса не описывает назначение класса. Программист, который видит строку с вызовом `Class23.Go`, понятия не имеет, что делает эта строка. Также методу запуска системы присвоено более содержательное имя — мы выбрали `MakeTheCandy`, но оно вполне может быть другим.

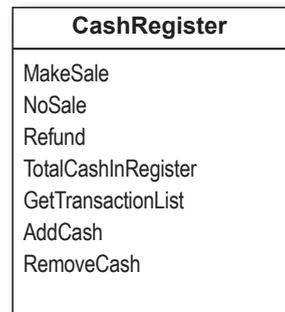
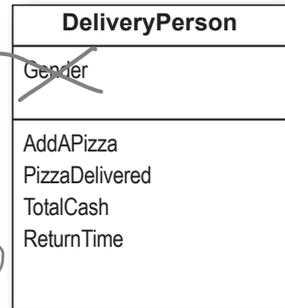
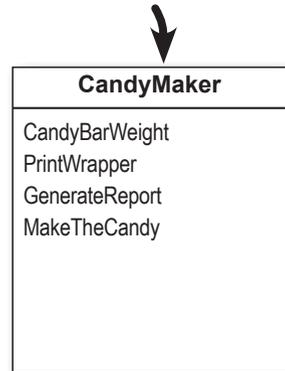
Эти два класса являются частью системы, которая используется пиццерией для отслеживания заказов, переданных в доставку.

Похоже, классы `DeliveryGuy` и `DeliveryGirl` делают одно и то же — они хранят информацию о курьере, который выехал на доставку пиццы. Правильнее было бы заменить их одним классом, в который добавляется поле для хранения пола курьера.

Мы решили НЕ добавлять поле `Gender`, потому что на самом деле пиццерии совершенно незачем хранить данные о поле курьера — следует уважать их личную информацию!

Класс `CashRegister` является частью программы, используемой системой оплаты в круглосуточном магазине при автозаправке.

Все методы класса связаны с кассовыми операциями — про-дажей, получением списка операций, добавлением наличных... кроме одной: заправки бензином (`PrintGas`). Этот метод лучше вынести из класса и включить его в другой класс.



Несколько идей для проектирования понятных классов

Мы собираемся вернуться к написанию кода. Немало кода будет написано в оставшейся части этой главы и МНОГО кода в книге. Это означает, что мы будем создавать множество классов. Приведем несколько советов, о которых стоит помнить, когда вы принимаете решения по их проектированию:

★ **Ваша программа строится для решения определенной задачи.**

Выделите время на обдумывание задачи. Насколько легко она разбивается на части? Как бы вы объяснили эту задачу другим? Все это стоит продумать при проектировании классов.

★ **Какие реальные сущности будут использоваться в программе?**

В программе для отслеживания графиков кормления животных в зоопарке могут присутствовать классы для разных видов корма и типов животных.

★ **Используйте содержательные имена для классов и методов.**

Чтобы другие люди могли получить представление о том, что делают ваши классы и методы, им должно быть достаточно просто взглянуть на их имена.

★ **Ищите сходство между классами.**

Иногда два класса можно объединить в один, если эти классы действительно похожи. Система производства шоколадных батончиков может содержать три или четыре турбины, но только один метод для закрытия клапана регулятора, который получает номер турбины в параметре.



Если вы зашли в тупик при написании кода, не огорчайтесь. На самом деле это даже хорошо!

Сутью написания кода является решение задач, а некоторые задачи оказываются очень хитрыми! Но если помнить некоторые обстоятельства, программирование пойдет намного эффективнее:

- ★ Программисту очень легко столкнуться с синтаксическими проблемами (такими, как пропущенные круглые скобки или кавычки). Одна пропущенная скобка может породить сразу несколько ошибок при построении.
- ★ **Гораздо лучше** смотреть на решение, чем мучиться над задачей. Когда вы испытываете досаду, мозг не желает учиться.
- ★ Весь код в книге был протестирован, и он определенно работает в Visual Studio 2019! Тем не менее при вводе легко допустить ошибку (например, ввести 1 вместо буквы L в нижнем регистре).
- ★ Если ваше решение не строится, попробуйте загрузить его из репозитория GitHub этой книги — в нем содержится работоспособный код для всех упражнений в книге: <https://github.com/head-first-csharp/fourth-edition>.

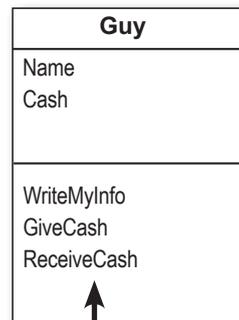
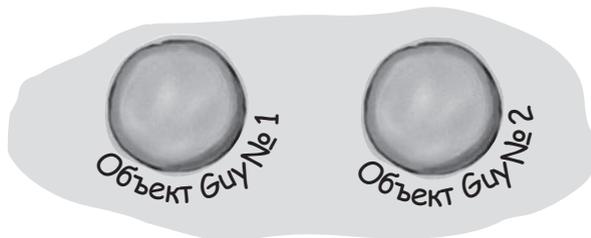
При чтении кода можно получить много полезной информации. Таким образом, если вы столкнулись с проблемой при выполнении упражнений, не бойтесь подсматривать в решение. Это не жульничество!

Классы, парни и деньги

Джо и Боб постоянно одалживают друг другу деньги. Создадим класс для хранения информации о том, сколько денег имеется у каждого из них. Начнем с краткого обзора того, что нам предстоит сделать.

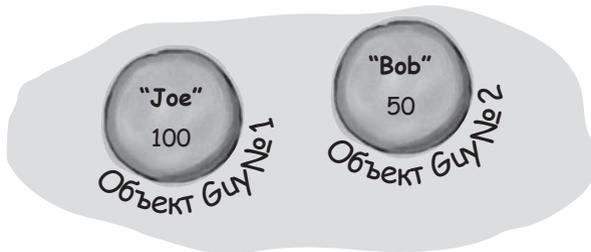
1 Мы создадим два экземпляра класса Guy.

Для хранения экземпляров будут использоваться две переменные Guy с именами joe и bob. После их создания куча будет выглядеть так:



2 Мы присвоим значения полей Cash и Name каждого объекта Guy.

Два объекта представляют двух разных парней с разными именами и разными суммами денег в кармане. У каждого парня имеется поле Name для хранения имени, а также поле Cash для хранения суммы денег.

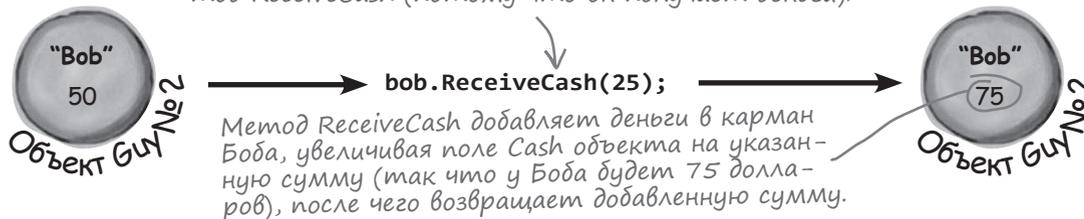


Мы выбрали осмысленные имена методов. Чтобы получить от объекта Guy некоторую сумму, следует вызвать его метод GiveCash, а чтобы дать ему денег — вызвать метод ReceiveCash.

3 Мы добавим методы для передачи и получения денег.

При вызове метода GiveCash парень выдает деньги (а значение его поля Cash уменьшается); метод возвращает выданную сумму. Чтобы парень получил деньги и положил их в карман (увеличил свое поле Cash), вызывается метод ReceiveCash, который возвращает полученную сумму.

Чтобы дать Бобу 25 долларов, мы вызываем его метод ReceiveCash (потому что он получает деньги).



```

class Guy
{
    public string Name;
    public int Cash;
}

```

В полях *Name* и *Cash* хранится имя парня и сумма денег у него в кармане.

```

    /// <summary>
    /// Выводит значения моих полей Name и Cash на консоль.
    /// </summary>
    public void WriteMyInfo()
    {
        Console.WriteLine(Name + " has " + Cash + " bucks.");
    }

    /// <summary>
    /// Выдает часть моих денег, удаляя их из кармана (или выводит на консоль
    /// сообщение о том, что денег недостаточно).
    /// </summary>
    /// <param name="amount">Выдаваемая сумма.</param>
    /// <returns>
    /// Сумма денег, взятая из кармана, или 0, если денег не хватает
    /// (или если сумма недействительна).
    /// </returns>
    public int GiveCash(int amount)
    {
        if (amount <= 0)
        {
            Console.WriteLine(Name + " says: " + amount + " isn't a valid amount");
            return 0;
        }
        if (amount > Cash)
        {
            Console.WriteLine(Name + " says: " +
                "I don't have enough cash to give you " + amount);
            return 0;
        }
        Cash -= amount;
        return amount;
    }

    /// <summary>
    /// Получает деньги, добавляя их в мой карман (или выводит
    /// сообщение на консоль, если сумма недействительна).
    /// </summary>
    /// <param name="amount">Получаемая сумма.</param>
    public void ReceiveCash(int amount)
    {
        if (amount <= 0)
        {
            Console.WriteLine(Name + " says: " + amount + " isn't an amount I'll take");
        }
        else
        {
            Cash += amount;
        }
    }
}

```

Иногда бывает нужно приказать объекту выполнить некоторую операцию — например, вывести описание этого объекта на консоль.

Методы *GiveCash* и *ReceiveCash* и *ReceiveCash* предполагают, что сумма, которую требуется выдать или получить, действительно. Это делается для того, чтобы нельзя было вызвать метод получения денег с отрицательным значением, что привело бы к потере средств.

Сравните комментарии в коде с диаграммой класса и схемами объектов *Guy*. Если что-то покажется непонятным, не жалейте времени и основательно разберитесь в происходящем.

Простой способ инициализации объектов в C#

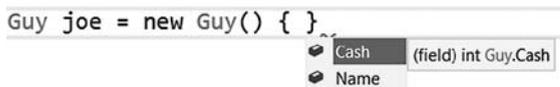
Почти каждый объект, который вы создаете в программе, необходимо каким-то образом инициализировать. Объект `Guy` не является исключением — он бесполезен, пока вы не зададите его поля `Name` и `Cash`. Необходимость в инициализации полей встречается так часто, что в C# для нее существует специальная сокращенная запись, называемая **инициализатором объектов**. Функция IDE IntelliSense поможет вам в работе с ней.

Сейчас мы опробуем ее при создании двух объектов `Guy`. Конечно, вы можете использовать команду `new` и еще две команды для инициализации ее полей:

```
joe = new Guy();
joe.Name = "Joe";
joe.Cash = 50;
```

Вместо этого введите команду `Guy joe = new Guy() {`

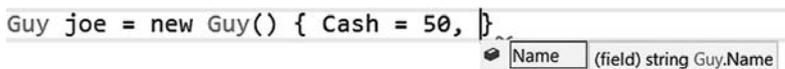
Как только вы введете левую фигурную скобку, IDE открывает окно IntelliSense со списком всех полей, которые вы можете инициализировать:



Выберите поле `Cash`, присвойте значение `50` и добавьте запятую:

```
Guy joe = new Guy() { Cash = 50,
```

Теперь введите пробел — открывается еще одно окно IntelliSense с оставшимся полем:



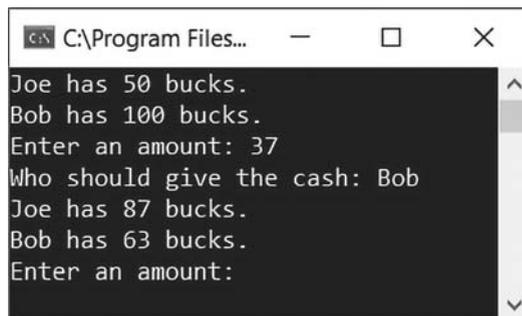
Задайте значение поля `Name` и введите точку с запятой. Теперь у вас есть одна команда, которая инициализирует ваш объект:

```
Guy joe = new Guy() { Cash = 50, Name = "Joe" }; ←
```

Новое объявление делает то же самое, что и три строки в начале страницы, но оно короче и проще читается.

Теперь у вас есть все необходимое для построения консольного приложения, использующего два экземпляра класса `Guy`. Его результаты выглядят примерно так: →

Сначала оно вызывает метод `WriteMyInfo` каждого объекта `Guy`. Затем приложение читает сумму из входного потока и спрашивает, кто должен получить эту сумму. После этого оно сначала вызывает метод `GiveCash` одного объекта `Guy`, а затем метод `ReceiveCash` другого объекта `Guy`. Это продолжается до тех пор, пока пользователь не введет пустую строку.





Упражнение

Ниже приведен метод Main для консольного приложения, которое обеспечивает передачу денег между объектами Guy. Вам предлагается заменить комментарии кодом — прочитайте каждый комментарий и напишите код, который выполняет описанные операции. После этого у вас появится программа, которая работает так, как показано на снимке экрана на предыдущей странице.

```
static void Main(string[] args)
{
    // Создайте новый объект Guy в переменной с именем joe
    // Присвойте его полю Name значение "Joe"
    // Присвойте его полю Cash значение 50

    // Создайте новый объект Guy в переменной с именем bob
    // Присвойте его полю Name значение "Bob"
    // Присвойте его полю Cash значение 100

    while (true)
    {
        // Вызовите методы WriteMyInfo для каждого объекта Guy

        Console.Write("Enter an amount: ");
        string howMuch = Console.ReadLine();
        if (howMuch == "") return;
        // Используйте метод int.TryParse для преобразования строки howMuch в int
        // (как это было сделано ранее в этой главе)
        {
            Console.Write("Who should give the cash: ");
            string whichGuy = Console.ReadLine();
            if (whichGuy == "Joe")
            {
                // Вызовите метод GiveCash объекта joe и сохраните результат
                // Вызовите метод ReceiveCash объекта bob с сохраненным результатом
            }
            else if (whichGuy == "Bob")
            {
                // Вызовите метод GiveCash объекта bob и сохраните результат
                // Вызовите метод GiveCash объекта joe с сохраненным результатом
            }
            else
            {
                Console.WriteLine("Please enter 'Joe' or 'Bob'");
            }
        }
        else
        {
            Console.WriteLine("Please enter an amount (or a blank line to exit).");
        }
    }
}
```

← Замените все комментарии кодом, который выполняет описанные операции.



Упражнение Решение

Ниже приведен метод Main для консольного приложения. В нем используется бесконечный цикл, который запрашивает у пользователя, сколько денег следует передать между объектами Guy. Если пользователь вводит пустую строку, метод выполняет команду return, что приводит к выходу из метода Main и завершению программы.

```
static void Main(string[] args)
{
    Guy joe = new Guy() { Cash = 50, Name = "Joe" };
    Guy bob = new Guy() { Cash = 100, Name = "Bob" };

    while (true)
    {
        joe.WriteMyInfo();
        bob.WriteMyInfo();
        Console.Write("Enter an amount: ");
        string howMuch = Console.ReadLine();
        if (howMuch == "") return;
        if (int.TryParse(howMuch, out int amount))
        {
            Console.Write("Who should give the cash: ");
            string whichGuy = Console.ReadLine();
            if (whichGuy == "Joe")
            {
                int cashGiven = joe.GiveCash(amount);
                bob.ReceiveCash(cashGiven);
            }
            else if (whichGuy == "Bob")
            {
                int cashGiven = bob.GiveCash(amount);
                joe.ReceiveCash(cashGiven);
            }
            else
            {
                Console.WriteLine("Please enter 'Joe' or 'Bob'");
            }
        }
        else
        {
            Console.WriteLine("Please enter an amount (or a blank line to exit).");
        }
    }
}
```

Когда метод Main выполняет эту команду return, программа завершается, потому что консольное приложение прекращает работу при завершении метода Main.

В этом коде один объект Guy отдает деньги из своего кармана, а другой объект Guy получает их.

Не переходите к следующей части упражнения, пока первая часть не заработает и вы не поймете, что в ней происходит. Выделите несколько минут на то, чтобы выполнить программу в пошаговом режиме отладчика, и убедитесь в том, что вы ее действительно поняли.



Упражнение (часть 2)

Итак, у вас имеется работающий класс `Guu`. Попробуем использовать его в азартной игре. Присмотритесь к следующему снимку экрана, попробуйте понять, как работает программа и что она выводит на консоль.

```

Microsoft Visual Studio Debug Console
Welcome to the casino. The odds are 0.75
The player has 100 bucks.
How much do you want to bet: 36
Bad luck, you lose.
The player has 64 bucks.
How much do you want to bet: 27
You win 54
The player has 91 bucks.
How much do you want to bet: 83
Bad luck, you lose.
The player has 8 bucks.
How much do you want to bet: 8
Bad luck, you lose.
The house always wins.
  
```

Порог вероятности

Во время каждого хода игрок делает ставку. При выигрыше он получает удвоенную ставку, а при проигрыше ставка теряется.

Программа выбирает случайное число от 0 до 1. Если число больше порога вероятности, игрок получает удвоенную ставку; в противном случае ставка теряется.

Создайте новое консольное приложение и добавьте тот же класс `Guu`. Затем в методе `Main` объявите три переменные: переменная `Random` с именем `random` инициализируется новым экземпляром класса `Random`; переменная `double` с именем `odds` для хранения порога вероятности инициализируется значением `0.75`; переменная `Guu` с именем `player` инициализируется экземпляром `Guu` с именем "The player" и значением `100`.

Выведите на консоль строку с приветствием и порогом вероятности. Затем запустите следующий цикл:

1. Объект `Guu` выводит сумму денег в своем кармане.
2. Запросите у пользователя размер ставки.
3. Прочитайте строку в строковую переменную с именем `howMuch`.
4. Попытайтесь преобразовать ее в переменную `int` с именем `amount`.
5. Если попытка преобразования завершится удачно, ставка игрока присваивается переменной `int` с именем `pot`. Она умножается на 2, потому что в результате игры игрок либо получает удвоенную ставку, либо теряет свою ставку.
6. Программа выбирает случайное число от 0 до 1.
7. Если число больше `odds`, игрок получает сумму из `pot`.
8. В противном случае игрок теряет ставку.
9. Программа продолжает работать, пока у пользователя остаются деньги.

Возьми в руку карандаш



Дополнительный вопрос: имя `Guu` действительно хорошо подходит для класса? Почему да (или почему нет)?

.....



Упражнение Решение

Ниже приведен рабочий метод Main для игры. Сможете ли вы предложить, как сделать его более интересным? Удастся ли вам придумать, как добавить новых игроков, как поддерживать разные варианты порога вероятности, а может, вы предложите что-то, еще более занятное? Это ваша возможность проявить фантазию!

```
static void Main(string[] args)
{
    double odds = .75;
    Random random = new Random();

    Guy player = new Guy() { Cash = 100, Name = "The player" };

    Console.WriteLine("Welcome to the casino. The odds are " + odds);
    while (player.Cash > 0)
    {
        player.WriteMyInfo();
        Console.Write("How much do you want to bet: ");
        string howMuch = Console.ReadLine();
        if (int.TryParse(howMuch, out int amount))
        {
            int pot = player.GiveCash(amount) * 2;
            if (pot > 0)
            {
                if (random.NextDouble() > odds)
                {
                    int winnings = pot;
                    Console.WriteLine("You win " + winnings);
                    player.ReceiveCash(winnings);
                } else
                {
                    Console.WriteLine("Bad luck, you lose.");
                }
            }
            else
            {
                Console.WriteLine("Please enter a valid number.");
            }
        }
        Console.WriteLine("The house always wins.");
    }
}
```

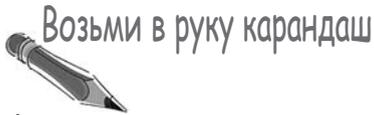
...а также немного потренироваться в написании кода — как говорилось выше, это лучший способ стать хорошим разработчиком.

Ваш код слегка отличается от нашего? Если он работает и выдает правильные результаты, это нормально! Одну и ту же программу можно написать многими разными способами.

↑
...и по мере того как вы будете продвигаться в изучении книги, а ответы к упражнениям будут становиться все длиннее, ваш код будет все сильнее отличаться от нашего. Помните: всегда можно посмотреть в решение, пока вы работаете над упражнением!

 Возьми в руку карандаш Наш ответ на дополнительный вопрос — а ваш ответ был другим?

Когда мы использовали объекты Guy («парень») для представления Джо и Боба, имя было подходящим. Но теперь, когда оно используется для представления игрока, лучше выбрать более содержательное имя класса (такое, как Bettor или Player).



Перед вами консольное приложение .NET, которое выводит три строки на консоль. Постарайтесь определить, какие результаты оно выведет, без использования компьютера. Начните с первой строки метода и отслеживайте значения всех полей объекта в процессе выполнения.

```
class Pizzazz
{
    public int Zippo;

    public void Bamboo(int eek)
    {
        Zippo += eek;
    }
}

class Abracadabra
{
    public int Vavavoom;

    public bool Lala(int floq)
    {
        if (floq < Vavavoom)
        {
            Vavavoom += floq;
            return true;
        }
        return false;
    }
}

class Program
{
    public static void Main(string[] args)
    {
        Pizzazz foxtrot = new Pizzazz() { Zippo = 2 };
        foxtrot.Bamboo(foxtrot.Zippo);
        Pizzazz november = new Pizzazz() { Zippo = 3 };
        Abracadabra tango = new Abracadabra() { Vavavoom = 4 };
        while (tango.Lala(november.Zippo))
        {
            november.Zippo *= -1;
            november.Bamboo(tango.Vavavoom);
            foxtrot.Bamboo(november.Zippo);
            tango.Vavavoom -= foxtrot.Zippo;
        }
        Console.WriteLine("november.Zippo = " + november.Zippo);
        Console.WriteLine("foxtrot.Zippo = " + foxtrot.Zippo);
        Console.WriteLine("tango.Vavavoom = " + tango.Vavavoom);
    }
}
```

Что программа выведет на консоль?

november.Zippo =

foxtrot.Zippo =

tango.Vavavoom =

Чтобы проверить решение, введите программу в Visual Studio и запустите ее. Если ответ оказался ошибочным, выполните программу в пошаговом режиме и проследите за изменениями всех полей объекта.

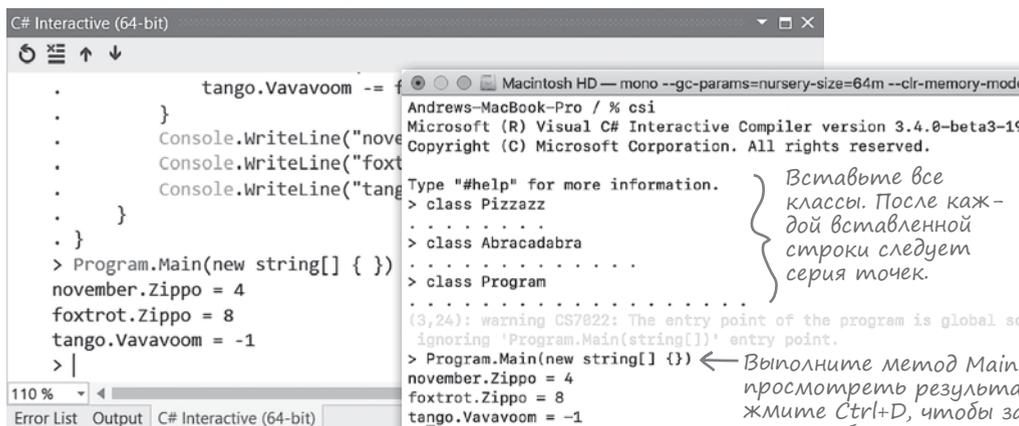
Если вы не хотите вводить весь код вручную, его можно загрузить по адресу <https://github.com/head-first-csharp/fourth-edition>.

Если вы используете Mac, IDE сгенерирует класс с именем MainClass вместо Program. В данном упражнении это ни на что не повлияет.

Используйте интерактивное окно C# для выполнения кода C#

Если вы хотите выполнить фрагмент кода C#, можно обойтись и без создания нового проекта в Visual Studio. Любой код C#, введенный в **интерактивном окне C#**, выполняется немедленно. Окно открывается командой View>>Other Windows>>C# Interactive. Попробуйте открыть его и **вставьте код** из ответа к упражнению. Чтобы выполнить его, введите следующую команду и нажмите Enter: `Program.Main(new string[] { })`.

В параметре "args" передается пустой массив.



Если вы используете Mac, в вашей IDE может не быть интерактивного окна, но вы можете выполнить команду csi в Терминале, чтобы воспользоваться интерактивным компилятором C# dotnet.

Вставьте все классы. После каждой вставленной строки следует серия точек.

Выполните метод Main, чтобы посмотреть результат. Нажмите Ctrl+D, чтобы завершить работу.

Не обращайте внимания на ошибку с упоминанием точки входа.

Также можно запустить интерактивный сеанс C# из командной строки. В системе Windows проведите в меню Пуск поиск по тексту `developer command prompt`, запустите окно командной строки и введите `csi`. В macOS или Linux выполните команду `csharp`, чтобы запустить оболочку Mono C# Shell. В обоих случаях вы можете вставить классы Pizzazz, Abracadabra и Program из предыдущего упражнения прямо в приглашении, после чего выполните команду `Program.Main(new string[] { })` для запуска точки входа вашего консольного приложения.

КЛЮЧЕВЫЕ МОМЕНТЫ

- **Ключевое слово new** используется для создания экземпляров класса. В программе может быть много экземпляров одного класса.
- Каждый **экземпляр** содержит все методы класса и получает собственные копии всех полей.
- При включении в код `new Random()`; создается **экземпляр класса Random**.
- **Ключевое слово static** объявляет поле или метод класса статическим. Для обращения к статическим методам или полям не нужно указывать экземпляр класса.
- Если поле является **статическим**, то существует только одна копия такого поля, которая совместно используется всеми экземплярами. Если ключевое слово **static** включено в объявление класса, все поля и методы такого класса тоже должны быть статическими.
- Если убрать ключевое слово **static** из объявления статического поля, оно становится **полем экземпляра**.
- Поля и методы класса называются его **компонентами**.
- Когда программа создает объект, он существует в специальной части компьютерной памяти, которая называется **кучей**.
- Visual Studio помогает добавлять **документацию XML** к полям и методам и выводит ее в окне IntelliSense.
- **Диаграммы классов** помогают планировать классы и упрощают работу с ними.
- Когда вы изменяете структуру кода без изменения его поведения, это называется **рефакторингом**. Опытные разработчики постоянно проводят рефакторинг своего кода.
- **Инициализаторы объектов** экономят время, делая ваш код более компактным и удобочитаемым.

Данные и ссылки



Чем были бы наши приложения без данных? Задумайтесь на минуту. Без данных наши программы... в общем, трудно представить, что кто-то станет писать код без данных. Вы запрашиваете **информацию** у ваших пользователей; эта информация используется для поиска данных или генерирования новой информации, которая возвращается пользователю. Собственно, практически все, что вы делаете в программировании, требует **работы с данными** в той или иной форме. В этой главе вы изучите все тонкости **типов данных** и **ссылок C#**, поймете, как работать с данными в программах, и даже узнаете кое-что новое об объектах (*представьте, объекты — тоже данные!*).

Оуэну нужна наша помощь!

Оуэн — гейм-мастер (и очень хороший). Он ведет группу, которая еженедельно собирается у него дома для проведения ролевых игр (RPG). И как любой хороший гейм-мастер, он основательно трудится, чтобы сделать времяпрепровождение интересным для игроков.



Повествование, фантазия и механика

Оуэн — особенно хороший рассказчик. За несколько последних лет он создал продуманный фантастический мир для своих друзей, но он не совсем доволен механикой игровой системы.

Поможем ли мы Оуэну улучшить его RPG?



← Характеристики персонажа (сила, выносливость, харизма, интеллект) стали важной механикой во многих ролевых играх. Игроки часто бросают кубики и определяют характеристики своих персонажей по специальным формулам.

На листах персонажей хранятся разные виды данных

Если вы когда-нибудь играли в RPG, то вы уже видели листы персонажей: страницу с подробной информацией, статистикой, историей и вообще любыми другими заметками, которые можно сделать по поводу персонажа. Если вы хотите создать класс, представляющий лист персонажа, какие типы вы бы использовали для его полей?

Character Sheet

ELLIWYNN
Character Name

7
Level

LAWFUL GOOD
Alignment

WIZARD
Character Class



Picture

Strength

Dexterity

Intelligence

Wisdom

Charisma

Spell Saving Throw

Poison Saving Throw

Magic Wand Saving Throw

Arrow Saving Throw

CharacterSheet

CharacterName
Level
PictureFilename
Alignment
CharacterClass
Strength
Dexterity
Intelligence
Wisdom
Charisma
SpellSavingThrow
PoisonSavingThrow
MagicWandSavingThrow
ArrowSavingThrow

ClearSheet
GenerateRandomScores

Поле для портрета персонажа. Если бы вы строили класс C# для представления листа персонажа, изображение можно было бы сохранить в графическом файле.

В RPG, в которую играет Оуэн, спас-броски дают игроку шанс избежать некоторых видов атак, бросая кубики. У этого персонажа имеется спас-бросок от магического жезла, поэтому игрок закрасил этот кружок.

Игроки создают персонажей, бросая кубики для каждой из своих характеристик, и записывают результаты в этих полях.



Взгляните на поля, перечисленные на диаграмме класса CharacterSheet. Какой тип вы бы выбрали для каждого поля?

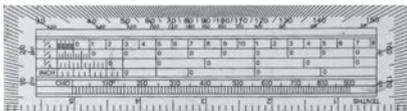
Тип переменной определяет, какие данные в ней могут храниться

В C# встроено много **типов** данных, предназначенных для хранения разных видов информации. Вам уже известны такие распространенные типы, как `int`, `string`, `bool` и `float`. Также есть и другие типы, которые вам еще не попадались; они тоже могут принести пользу.

Некоторые типы, которыми вы будете активно пользоваться.



- ★ **int** позволяет хранить любое целое число от $-2\,147\,483\,648$ до $2\,147\,483\,647$. Целые числа не имеют дробной части.



- ★ **float** позволяет хранить вещественные числа от $\pm 1.5 \times 10^{-45}$ до $\pm 3.4 \times 10^{38}$ с точностью до 8 знаков.

Лучше остроумный дурак,
чем глупый остряк.

- ★ **string** может хранить текст произвольной длины (включая пустые строки `""`).



- ★ **bool** используется для логических значений — `true` или `false`. Он используется для представления всего, что может находиться только в одном из двух состояний: либо одно, либо другое.



- ★ **double** позволяет хранить вещественные числа от $\pm 5.0 \times 10^{-324}$ до $\pm 1.7 \times 10^{308}$ с точностью до 16 знаков. Этот тип очень часто используется при работе со свойствами XAML.



Как вы думаете, почему в C# предусмотрено несколько типов для хранения чисел с дробной частью?

В C# существует несколько типов для хранения целых чисел

В C# для хранения целых чисел существуют и другие типы — не только `int`. На первый взгляд это выглядит немного странно. Зачем создавать столько типов для чисел без дробной части? Для большинства программ в книге не важно, будете ли вы использовать `int` или `long`. Но если вы пишете программу, которая должна хранить многие миллионы целых значений, выбор меньшего типа (например, `byte`) вместо большего типа (такого, как `long`) может сэкономить очень много памяти.

- ★ В типе `byte` может храниться любое **целое** число от 0 до 255.
- ★ В типе `sbyte` может храниться любое **целое** число от -128 до 127.
- ★ В типе `short` может храниться любое **целое** число от -32 768 до 32 767.
- ★ В типе `long` может храниться любое **целое** число от -9 233 372 036 854 775 808 до 9 233 372 036 854 775 807.



↑
Тип `byte` хранит только малые целые числа от 0 до 255.

Если вам нужно хранить большее число, используйте тип `short`, который способен хранить целые числа от -32 768 до 32 767.



↓
Тип `long` также хранит целые числа, но в нем могут поместиться огромные значения.



А вы заметили, что в типе `byte` хранятся только положительные числа, тогда как `sbyte` также подходит для отрицательных чисел? Оба типа позволяют хранить 256 возможных значений. Различие в том, что значения типа `sbyte` (как и `short` и `long`) могут быть отрицательными, вот почему они называются **типами со знаком** («s» в `sbyte` означает «sign», т. е. «знак»!). И если `byte` является версией `sbyte` **без знака**, такие версии существуют и у типов `short`, `int` и `long`; их имена начинаются с «u»:

- ★ В типе `ushort` может храниться любое **положительное целое** число от 0 до 65 535.
- ★ В типе `uint` может храниться любое **положительное целое** число от 0 до 4 294 967 295.
- ★ В типе `ulong` может храниться любое **положительное целое** число от 0 до 18 446 744 073 709 551 615.

Типы для хранения ОГРОМНЫХ и очень маленьких чисел

Иногда точности float оказывается недостаточно. И хотите верьте, хотите нет, но 10^{38} может оказаться недостаточно большим, а 10^{-45} — недостаточно малым. Многие программы, написанные для научных и финансовых вычислений, постоянно сталкиваются с подобными проблемами, поэтому C# предоставляет другие типы с плавающей точкой для работы с очень большими и очень малыми значениями:

- ★ В типе *float* может храниться любое число от $\pm 1.5 \times 10^{-45}$ до $\pm 3.4 \times 10^{38}$ с 6–9 значащими цифрами.
- ★ *double* позволяет хранить **вещественные** числа от $\pm 5.0 \times 10^{-324}$ до $\pm 1.7 \times 10^{308}$ с 15–17 значащими цифрами.
- ★ В типе *decimal* может храниться любое число от $\pm 1.0 \times 10^{-28}$ до $\pm 7.9 \times 10^{28}$ с 28–29 значащими цифрами. Если вашей программе **приходится работать с денежными суммами**, всегда стоит использовать тип *decimal* для хранения числа.

Тип *decimal* обеспечивает существенно более высокую точность (больше значащих цифр), поэтому он лучше подходит для финансовых вычислений.

Числа с плавающей точкой под увеличительным стеклом



Типы *float* и *double* называются типами «с плавающей точкой», потому что точка может перемещаться внутри числа (в отличие от чисел «с фиксированной точкой», у которых дробная часть всегда состоит из постоянного количества цифр). Этот факт — а на самом деле и многие аспекты, относящиеся к числам с плавающей точкой, особенно точность, — выглядит немного **странно**, поэтому стоит немного пояснить происходящее.

«Значащие цифры» представляют точность числа: 1 048 415, 104.8415 и 0.000001048415 содержат 7 значащих цифр. Таким образом, когда мы говорим, что тип *float* может хранить большие числа до 3.4×10^{38} или малые числа до -1.5×10^{-45} , это означает, что он может хранить числа из 8 цифр, за которыми следуют 30 нулей, или числа из 37 нулей, за которыми следуют 8 цифр.

Типы *float* и *double* также могут иметь специальные значения, включая положительный и отрицательный ноль, положительную и отрицательную бесконечность, а также специальное значение **NaN** (не число), которое представляет... в общем, значение, которое вообще не является числом. Также они содержат статические методы и позволяют проверять эти специальные значения. Попробуйте выполнить следующий цикл:

```
for (float f = 10; float.IsFinite(f); f *= f)
{
    Console.WriteLine(f);
}
```

Теперь попробуйте выполнить тот же цикл с *double*:

```
for (double d = 10; double.IsFinite(d); d *= d)
{
    Console.WriteLine(d);
}
```

Если вы уже давно не пользовались экспоненциальной записью, 3.4×10^{38} означает число 34, за которым идут 37 нулей, а -1.5×10^{-45} означает $-00\dots$ (еще 40 нулей)... 0015.

Поговорим о строках

Вы уже писали код, работающий со **строками**. Что же именно представляет собой строка?

В любом приложении .NET строка является объектом. Ее полное имя класса имеет вид `System.String` — иначе говоря, класс обладает именем `String` и принадлежит к пространству имен `System` (как и класс `Random`, который использовался ранее). Используя ключевое слово `C# string`, вы на самом деле работаете с объектами `System.String`. Более того, можно заменить `string` на `System.String` во всем коде, написанном вами до настоящего момента, и он все равно будет работать! (Ключевое слово `string` называется синонимом — с точки зрения кода `C# string` и `System.String` означают одно и то же.)

Также существуют два специальных строковых значения: пустая строка `""` (или строка без символов) и `null`, т. е. строка, которой вообще ничего не присвоено. Мы вернемся к `null` позднее в этой главе.

Строки состоят из символов, а конкретно из символов Юникода (об этом вы узнаете больше в этой книге). Иногда требуется сохранить в программе отдельный символ (например, `Q`, `j` или `$`); в таких случаях используется тип `char`. Литеральные значения `char` всегда заключаются в одинарные кавычки (`'x'`, `'3'`). Также в одинарных кавычках могут содержаться служебные последовательности: `'\n'` — разрыв строки, `'\t'` — табуляция и т. д.). Для записи служебной последовательности в коде `C#` используются два символа, но ваша программа хранит каждую служебную последовательность в памяти как один символ.

Наконец, существует еще более важный тип `object`. Если переменная имеет тип `object`, *ей можно присвоить любое значение*. Ключевое слово `object` также является синонимом — оно эквивалентно `System.Object`.

Возьми в руку карандаш



Мы записали первый ответ за вас.

```
..... int i;
..... long l;
..... float f;
..... double d;
..... decimal m;
..... byte b;
..... char c;
..... string s;
..... bool t;
```

Иногда объявление переменной и присваивание ей значения объединяются в одну команду: `int i = 37`; но вы уже знаете, что инициализировать переменную при объявлении не обязательно. Что произойдет, если использовать переменную без присваивания значения? Давайте проверим! Воспользуйтесь интерактивным окном `C#` (или консолью .NET, если вы работаете на Mac), чтобы объявить переменную и проверить ее значение.

Откройте интерактивное окно `C#` (из меню `View>>Other Windows`) или выполните команду `csi` из терминала Mac. Объявите каждую переменную, после чего введите имя переменной, чтобы просмотреть ее значение по умолчанию. Запишите значение по умолчанию для каждого типа в обозначенном месте.

```
C# Interactive (64-bit)
Type "#help" for more information.
> int i;
> i
0
> |
```

```
Andrews-MacBook-Pro ~ % csi
Microsoft (R) Visual C# Interactive Compiler version 3.4.0-beta3-19521-01 ()
Copyright (C) Microsoft Corporation. All rights reserved.

Type "#help" for more information.
> int i;
> i
0
> |
```

Литерал — значение, записанное непосредственно в вашем коде

Литерал — число, строка или другое фиксированное значение, включенное в ваш код. Вы уже неоднократно пользовались литералами; приведем лишь несколько примеров чисел, строк и других литералов, встречавшихся в книге:

```
int number = 15;
string result = "the answer";
public bool GameOver = false;
Console.WriteLine("Enter the number of cards to pick: ");
if (value == 1) return "Ace";
```

А вы сможете найти все литералы в этих командах из кода, написанного в предыдущей главе? Последняя команда содержит два литерала.

Таким образом, когда вы записываете команду `int i = 5;`, в этой команде 5 является литералом.

Использование суффиксов для определения типа литерала

Когда вы записываете в Unity команды следующего вида:

```
InvokeRepeating("AddABall", 1.5F, 1);
```

возникает вопрос: для чего нужен суффикс F?

А вы заметили, что без F в литералах 1.5F и 0.75F программа строиться не будет? Это связано с тем, что у **литералов есть тип**. Каждому литералу тип назначается автоматически, а в C# существуют специальные правила относительно объединения разных типов. Вы можете самостоятельно увидеть, как работает этот механизм. Добавьте следующую строку в любую программу C#:

```
int wholeNumber = 14.7;
```

При попытке построить программу IDE выдает следующее сообщение об ошибке в окне Error List:

CS0266 Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)

IDE сообщает о том, что у литерала 14.7 имеется тип, — это double. При помощи суффикса можно изменить его тип. Например, попробуйте преобразовать его во float, добавив суффикс F в конце (14.7F), или в decimal добавлением суффикса M (14.M — кстати, буква «M» обозначает «money», т. е. «деньги»). Теперь в сообщении об ошибке говорится, что программе не удастся преобразовать float или decimal. Добавьте D (или полностью опустите суффикс), и ошибка исчезнет.

Возьми в руку карандаш

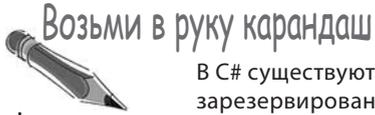


Решение

```
..... int i;
..... long l;
..... float f;
```

```
..... double d;
..... decimal m;
..... byte b;
..... char c;
..... string s;
..... bool t;
```

Если вы использовали командную строку на Mac или в системах Unix, возможно, вы видели, что в качестве значения по умолчанию для char используется '\0' вместо '\0'. Что это значит, мы объясним позднее, когда речь пойдет о Юникоде.



В С# существуют десятки **зарезервированных слов**, называемых **ключевыми словами**. Эти слова, зарезервированные компилятором С#, не могут использоваться в качестве имен переменных. Многие ключевые слова вам уже известны — ниже приведена краткая сводка, которая поможет вам закрепить их в мозгу. Напишите, что, по вашему мнению, делает каждое из этих ключевых слов в С#.

`namespace`

`for`

`class`

`else`

`new`

`using`

`if`

`while`

Если вам непременно хочется использовать зарезервированное ключевое слово в качестве имени переменной, поставьте перед ним @, но ближе подойти к зарезервированному слову компилятор не позволит. То же самое при желании можно делать и с незарезервированными именами.



Возьми в руку карандаш

Решение

В C# существуют десятки **зарезервированных слов**, называемых **ключевыми словами**. Эти слова, зарезервированные компилятором C#, не могут использоваться в качестве имен переменных. Многие ключевые

слова вам уже известны — ниже приведена краткая сводка, которая поможет вам закрепить их в мозгу. Напишите, что, по вашему мнению, делает каждое из этих ключевых слов в C#.

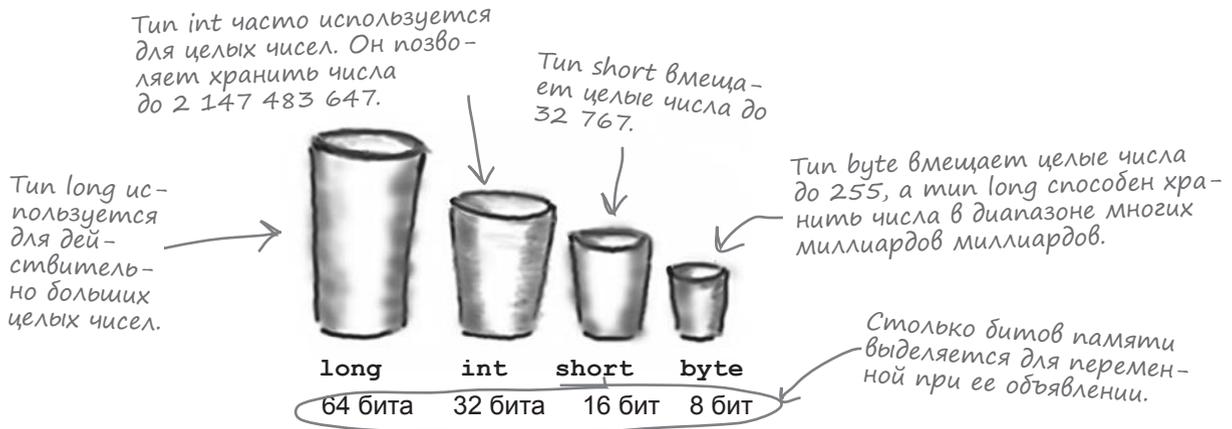
namespace	<i>Все классы и методы программы принадлежат некоторому пространству имен.</i>
	<i>Пространства имен гарантируют, что имена, которые вы используете в своей программе, не будут конфликтовать с именами из .NET Framework или других классов.</i>
for	<i>Определяет цикл, заголовок которого выполняет три команды. Сначала объявляется используемая переменная, следующая команда проверяет переменную по заданному условию. Третья команда делает что-то со значением.</i>
class	<i>Классы содержат методы и поля и используются для создания экземпляров объектов. Поля содержат информацию, известную объекту, а методы определяют поведение объекта.</i>
else	<i>Блок кода, начинающийся с else, должен следовать сразу же за блоком if. Он выполняется, если условие предшествующей команды if ложно.</i>
new	<i>Используется для создания нового экземпляра объекта.</i>
using	<i>Используется для перечисления всех пространств имен, используемых в вашей программе. Команда using позволяет использовать классы из разных частей .NET Framework.</i>
if	<i>Один из способов организации условной логики в программе. Если условие истинно, то выполняется один блок; если ложно — делается что-то другое.</i>
while	<i>Циклы while продолжают работать, пока условие в начале цикла остается истинным.</i>

Переменные как емкости для данных

Не все данные хранятся в куче. Типы значений обычно хранят свои данные в другой части памяти, называемой стеком. Об этом будет подробно рассказано позднее.

Все данные занимают место в памяти. (Помните кучу из предыдущей главы?) В ходе своей работы вам придется думать о том, сколько места вам потребуется для хранения строки или числа в программе. Это одна из причин для использования переменных — они позволяют выделить достаточный объем памяти для хранения ваших данных.

Представьте переменную в виде чашки или другой емкости для хранения данных. В C# используются разные чашки для хранения разных типов данных. Как известно, в кофейнях существуют чашки разных размеров; точно так же существуют разные размеры переменных.



Класс `Convert` может использоваться для анализа битов и байтов

← Преобразуйте!

Вы наверняка слышали, что все программирование сводится к манипуляциям с 0 и 1. В .NET существует статический класс `Convert` для преобразования между разными типами данных. Воспользуемся им для рассмотрения примера, который демонстрирует, как работают биты и байты.

Бит принимает значения 1 или 0. Байт состоит из 8 битов, так что байтовая переменная содержит 8-битное число, т. е. число, которое может быть представлено 8 битами. Как это выглядит? Воспользуемся классом `Convert` для преобразования двоичных чисел в байты:

```
Convert.ToByte("10111", 2) // возвращает 23
Convert.ToByte("11111111", 2); // возвращает 255
```

В первом аргументе `Convert.ToByte` передается преобразуемое число, а во втором — основание системы. Двоичные числа используют основание 2.

Байты могут хранить числа от 0 до 255, потому что они используют 8 бит памяти — 8-разрядное число представляется двоичным числом в диапазоне от 0 до 11111111 (или от 0 до 255 в десятичной системе).

Тип `short` хранит 16-битные значения. Воспользуемся `Convert.ToInt16` для преобразования двоичного значения 11111111111111 (15 единиц) в `short`. Значение `int` является 32-битным, так что мы воспользуемся `Convert.ToInt32` для преобразования 31 единицы в `int`:

```
Convert.ToInt16("11111111111111", 2); // возвращает 32767
Convert.ToInt32("111111111111111111111111111111", 2); // возвращает 2147483647
```

Другие типы тоже могут иметь разные размеры

Числа с дробной частью хранятся не так, как целые числа, и разные типы с плавающей точкой занимают разные объемы памяти. Для большинства чисел с дробной частью можно пользоваться типом **float** — наименьшим типом данных для хранения целых чисел. Если вам потребуется более высокая точность, используйте **double**. Если вы пишете финансовое приложение, в котором должны храниться денежные величины, всегда стоит использовать тип **decimal**.

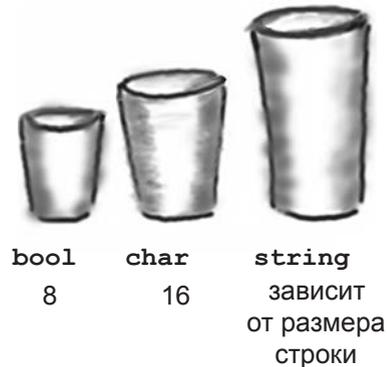
Да, и последнее: никогда не используйте double для денежных сумм — только decimal.



Эти типы предназначены для дробных величин. Большие переменные способны представлять больше знаков в дробной части.

Мы уже говорили о строках, и вы знаете, что компилятор C# умеет работать с **символами** и **нечисловыми типами**. Тип **char** рассчитан на один символ, а строка содержит множество «сцепленных» символов. Для объекта строки не существует фиксированного размера — он расширяется по размерам тех данных, которые вы хотите в нем хранить. Тип данных **bool** используется для хранения значений **true** или **false** вроде тех, которые вы использовали в командах **if**.

В C# также предусмотрены типы для хранения данных, которые не предназначены для числовой информации.



Строки могут быть большими... ОЧЕНЬ большими! В C# для хранения длины строки используется 32-битное число, поэтому максимальная длина строки равна 2^{31} (bkb 2 147 483 648) символов.

Разные типы с плавающей точкой занимают разные объемы памяти: **float** — наименьший, **decimal** — наибольший.

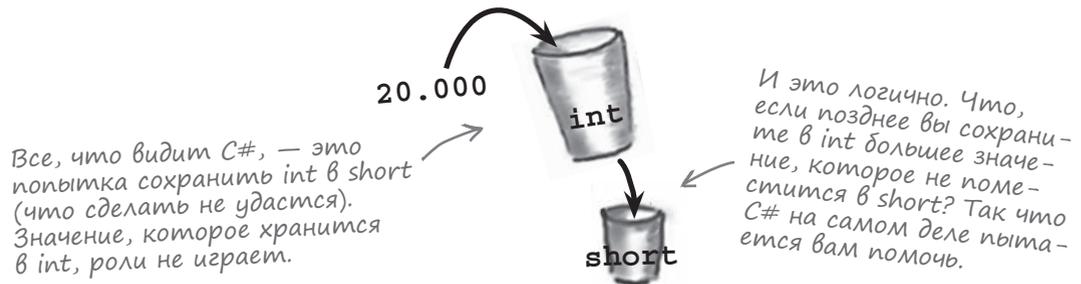
10 литров в 5-литровой банке



Когда вы объявляете переменную с некоторым типом, компилятор C# **выделяет** (или резервирует) всю память, необходимую для хранения максимального значения этого типа. Даже если значение далеко от верхней границы объявленного типа, компилятор видит чашку, в которой хранятся данные, а не находящееся в ней число. Таким образом, следующая попытка не сработает:

```
int leaguesUnderTheSea = 20000;
short smallerLeagues = leaguesUnderTheSea;
```

20 000 поместится в типе short, не проблема. Но так как переменная leaguesUnderTheSea объявлена с типом int, C# воспринимает ее как имеющую размер int и считает, что эта переменная слишком велика для хранения в контейнере short. Компилятор не позволит выполнять подобные преобразования на ходу. Вы должны следить за тем, чтобы для данных, с которыми вы работаете, выбирался подходящий тип.



Возьми в руку карандаш

Три команды в следующем списке не построятся — либо потому, что мы пытаемся втиснуть слишком много данных в маленькую переменную, либо потому, что данные не соответствуют типам переменных. Обведите эти команды и запишите краткое объяснение того, что же с ними не так.

<code>int hours = 24;</code>	<code>string taunt = "your mother";</code>
<code>short y = 78000;</code>	<code>byte days = 365;</code>
<code>bool isDone = yes;</code>	<code>long radius = 3;</code>
<code>short RPM = 33;</code>	<code>char initial = 'S';</code>
<code>int balance = 345667 - 567;</code>	<code>string months = "12";</code>

Приведение типов позволяет копировать значения, которые C# не может автоматически преобразовать к другому типу

Посмотрим, что происходит при попытке присвоить значение `decimal` переменной `int`.

← (Делайте это!)

- 1 Создайте новый проект консольного приложения и добавьте следующий код в метод `Main`:

```
float myFloatValue = 10;
int myIntValue = myFloatValue;
Console.WriteLine("myIntValue is " + myIntValue);
```

Неявное преобразование означает, что C# может автоматически преобразовать значение к другому типу без потери информации.

- 2 Попробуйте построить свою программу. Вы получите ту же ошибку CS0266, которую видели ранее:

✘ CS0266 Cannot implicitly convert type 'float' to 'int'. An explicit conversion exists (are you missing a cast?)

Присмотритесь к последним словам сообщения об ошибке: «(возможно, вы пропустили приведение типа?)». Компилятор C# дает по-настоящему полезную подсказку о том, как можно решить проблему.

- 3 Чтобы устранить ошибку, выполните **приведение** (casting) `decimal` в `int`. Для этого тип, к которому преобразуется значение, заключается в круглые скобки: `(int)`. После того как вы измените строку к приведенному ниже состоянию, программа будет успешно компилироваться и выполняться:

```
int myIntValue = (int) myFloatValue;
```

Здесь значение `decimal` приводится к типу `int`.

Когда значение с плавающей точкой приводится к `int`, оно округляется в нижнюю сторону до ближайшего целого числа.

Что же произошло?

Компилятор C# не позволяет присвоить значение переменной неподходящего типа — даже если значение может поместиться в переменной! Как выясняется, МНОГИЕ ошибки возникают из-за проблем с типами, и **компилятор вам помогает**, подталкивая в нужном направлении. Когда вы используете приведение типов, фактически вы говорите компилятору: да, я знаю, что типы разные, но гарантирую, что в этом конкретном случае данные поместятся в новой переменной.



Возьми в руку карандаш

Решение

Три команды в следующем списке не построятся — либо потому, что мы пытаемся втиснуть слишком много данных в маленькую переменную, либо потому, что данные не соответствуют типам переменных. Обведите эти команды и запишите краткое объяснение того, что же с ними не так.

`short y = 78000;`

Тип `short` может хранить числа от -32 767 до 32 768. Число слишком большое!

`byte days = 365;`

`bool isDone = yes;`

Переменной `bool` можно присваивать только значения `true` и `false`.

В переменной `byte` хранятся только значения от 0 до 255. Для такого значения потребуется `short`.

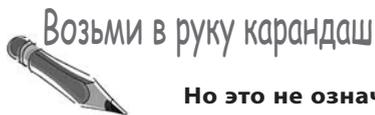
Когда вы выполняете приведение типа со слишком большим значением, C# отсекает его до размеров нового контейнера

Вы уже видели, что значение `decimal` может быть приведено к `int`. Как выясняется, любое число можно привести к *любому* другому числовому типу. Однако это не означает, что *значение* останется неизменным в результате приведения. Допустим, имеется переменная `int`, которой присвоено значение 365. Если вы хотите привести ее к типу `byte` (максимальное значение 255), то вместо ошибки произойдет *циклический возврат*. 256 при приведении к `byte` дает значение 0, 257 преобразуется в 1, 258 преобразуется в 2 и так далее до 365, которое преобразуется в **109**. Когда вы снова доберетесь до 255, преобразование снова «сбрасывается» до 0.

Если вы используете + (или *, / или -) с двумя разными числовыми типами, оператор **автоматически преобразует** меньший тип к большему. Пример:

```
int myInt = 36;
float myFloat = 16.4F;
myFloat = myInt + myFloat;
```

Так как значение `int` может поместиться в `float`, но `float` в `int` не поместится, оператор + преобразует `myInt` к типу `float` перед тем, как прибавлять его к `myFloat`.



Но это не означает, что любой тип можно привести к любому другому типу.

Создайте новый проект консольного приложения и введите эти команды в метод `Main`. Затем постройте программу — она выдает много ошибок. Вычеркните команды, для которых выводятся ошибки. Это поможет вам определить, какие приведения типов возможны, а какие нет!

```
int myInt = 10;
byte myByte = (byte)myInt;
double myDouble = (double)myByte;
bool myBool = (bool)myDouble;
string myString = "false";
```

```
myBool = (bool)myString;
myString = (string)myInt;
myString = myInt.ToString();
myBool = (bool)myByte;
myByte = (byte)myBool;
short myShort = (short)myInt;
char myChar = 'x';
myString = (string)myChar;
long myLong = (long)myInt;
decimal myDecimal = (decimal)myLong;
myString = myString + myInt +
myByte + myDouble + myChar;
```

По ссылке можно найти гораздо больше информации о разных типах значений в C# — поверьте, она заслуживает вашего внимания:
<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/value-types>



Я объединял числа и строки в своих сообщениях еще с момента работы с циклами в главе 2! Выходит, я все это время занимался преобразованиями типов?

Да! Когда вы выполняете конкатенацию, C# преобразует значения.

Когда вы используете оператор + для объединения строки с другим значением, это называется конкатенацией. При конкатенации строки с int, bool, float или другим типом значения это значение автоматически преобразуется. Такие преобразования отличаются от приведения типов, потому что во внутренней реализации для значения вызывается метод ToString... а .NET среди прочего гарантирует, что **каждый объект содержит метод ToString**, который преобразует его в строку (но каждый отдельный класс сам определяет, имеет ли смысл эта строка).

Циклический возврат своими руками!

В «циклическом возврате» при приведениях числовых типов нет ничего загадочного — вы можете легко проделать все самостоятельно. Откройте любое приложение-калькулятор с функцией Mod (вычисление остатка от деления — иногда поддерживается только в инженерном режиме) и вычислите $365 \text{ Mod } 256$.

Возьми в руку карандаш Решение



Произвольный тип не всегда можно привести к любому другому типу. Создайте новый проект консольного приложения и введите эти команды в метод Main. Затем постройте программу — она выдает много ошибок. Вычеркните команды, для которых выводятся ошибки. Это поможет вам определить, какие приведения типов возможны, а какие нет!

```
int myInt = 10;
byte myByte = (byte)myInt;
double myDouble = (double)myByte;
bool myBool = (bool)myDouble;
string myString = "false";
myBool = (bool)myString;
myString = (string)myInt;
myString = myInt.ToString();
myBool = (bool)myByte;
myByte = (byte)myBool;
short myShort = (short)myInt;
char myChar = 'x';
myString = (string)myChar;
long myLong = (long)myInt;
decimal myDecimal = (decimal)
myLong;

myString = myString + myInt +
myByte + myDouble + myChar;
```

C# выполняет некоторые преобразования автоматически

Существуют две важные разновидности преобразований, которые не требуют приведения типов. Первое – автоматическое преобразование, которое выполняется каждый раз, когда вы используете арифметические операторы, как в следующем примере:

```
long l = 139401930;
short s = 516;
double d = l - s;
d = d / 123.456;
Console.WriteLine("The answer is " + d);
```

Оператор - вычитаем short из long, а оператор = преобразует результат в double.

Другой вариант автоматического преобразования типов встречается при использовании оператора + для **конкатенации** строк (что означает обычное присоединение одной строки к концу другой, как это делалось для окон сообщений). Когда вы используете оператор + для конкатенации строки с данными, относящимися к другому типу, оператор автоматически преобразует числа в строки. Ниже приведен пример – попробуйте добавить эти строки в любую программу C#. Первые две строки выполняются нормально, но третья не компилируется:

```
long number = 139401930;
string text = "Player score: " + number;
text = number;
```

Компилятор C# выдает следующую ошибку в третьей строке:

 CS0029 Cannot implicitly convert type 'long' to 'string'

ScoreText.text является строковым полем, так что при использовании оператора + для конкатенации строки значение было присвоено нормально. Но когда вы пытаетесь присвоить ему значение напрямую, C# не имеет возможности автоматически преобразовать значение long в string. Чтобы преобразовать его в строку, вызовите для него метод ToString.

Часто
Задаваемые
Вопросы

В: Вы использовали методы Convert.ToByte, Convert.ToInt32 и Convert.ToInt64 для преобразования строк с двоичными числами в целые числа. А можно ли преобразовать целые числа обратно в двоичные?

О: Да, можно. Класс Convert содержит метод Convert.ToString, который преобразует разные типы значений в строки. Окно IntelliSense показывает, как он работает:

```
Console.WriteLine(Convert.ToString(8675309, 2));
```

▲ 26 of 36 ▼ string Convert.ToString(int value, int toBase)
Converts the value of a 32-bit signed integer to its equivalent string representation in a specified base.
value: The 32-bit signed integer to convert.

Таким образом, Convert.ToString(255, 2) возвращает строку "1111111", а Convert.ToString(8675309, 2) возвращает строку "10000100010111111101101" — поэкспериментируйте, чтобы лучше понять, как работают двоичные числа.

При вызове метода аргументы должны быть совместимы с типами параметров

В предыдущей главе мы воспользовались классом `Random` для выбора случайного числа от 1 до 5 (не включая), при помощи которого выбиралась масть карты:

```
int value = random.Next(1, 5);
```

Попробуйте заменить первый аргумент 1 на 1.0:

```
int value = random.Next(1.0, 5);
```

Литерал `double` передается методу, который рассчитывает получить значение `int`. А значит, вас не удивит, что компилятор откажется строить программу — вместо этого он выдает ошибку:

```
✘ CS1503 Argument 1: cannot convert from 'double' to 'int'
```

Иногда `C#` может выполнить преобразование автоматически. `C#` не знает, как преобразовать `double` в `int` (например, 1.0 в 1), но знает, как преобразовать `int` в `double` (например, 1 в 1.0). Говоря конкретнее:

- ★ Компилятор `C#` знает, как преобразовать целое число в число с плавающей точкой.
- ★ А еще он знает, как преобразовать целый тип в другой целый тип или тип с плавающей точкой в другой тип с плавающей точкой.
- ★ Но эти преобразования он может выполнить только в том случае, если преобразуемый тип имеет одинаковый или меньший размер, чем тип, к которому выполняется преобразование. Таким образом, `C#` может преобразовать `int` в `long` или `float` в `double`, но не сможет преобразовать `long` в `int` или `double` в `float`.

Но `Random.Next` — не единственный метод, который выдает ошибки компилятора, если вы попытаетесь передать переменную, тип которой не соответствует параметру. Это относится ко *всем* методам — *даже тем, которые написаны вами*. Добавьте следующий метод в консольное приложение:

```
public int MyMethod(bool add3) {  
    int value = 12;  
  
    if (add3)  
        value += 3;  
    else  
        value -= 2;  
  
    return value;  
}
```

Попробуйте передать методу `string` или `long` — вы получите ошибку `CS1503`, в которой говорится о невозможности преобразования аргумента в `bool`. Некоторые люди не могут запомнить, чем **параметры отличаются от аргументов**. Давайте разберемся:

Параметр — то, что вы определяете в своем методе. **Аргумент** — то, что вы передаете методу. Методу с параметром `int` можно передать аргумент `byte`.

Когда компилятор выдает сообщение об ошибке «недопустимый аргумент», это означает, что вы попытались при вызове метода передать переменные, типы которых не соответствуют параметрам метода.

Часто задаваемые вопросы

В: В последней команде `if` используется короткое условие `if (add3)`. Это то же самое, что `if (add3 == true)`?

О: Да. Взгляните еще раз на команду `if/else`:

```
if (add3)
    value += 3;
else
    value -= 2;
```

Команда `if` всегда проверяет условие на истинность. Так как переменная `add3` относится к типу `bool`, при выполнении она дает результат `true` или `false`. Следовательно, явно включать проверку `== true` не обязательно.

Также можно проверить условие на ложность, для чего используется оператор `!` (логическое отрицание, или оператор NOT). Запись `if (!add3)` эквивалентна `if (add3 == false)`.

В дальнейших примерах кода при условной проверке логических переменных мы обычно используем запись `if (add3)` или `if (!add3)`, а не используем `==` для явной проверки истинности или ложности переменной.

В: Также в блоках `if` и `else` отсутствуют фигурные скобки. Означает ли это, что их можно не указывать?

О: Да — но только если блок `if` или `else` состоит из одной команды. В данном случае фигурные скобки `{ }` можно опустить, потому что блок `if` состоит из одной команды `{return 45;}`, как и блок `else {return 61;}`. Если позднее вы захотите добавить в такой блок еще одну команду, его придется заключить в фигурные скобки:

```
if (add3)
    value += 3;
else {
    Console.WriteLine("Subtracting 2");
    value -= 2;
}
```

Даже если фигурные скобки можно опустить, будьте осторожны— существует высокий риск написать код, который делает не то, чего вы ожидали. От фигурных скобок вреда никогда не будет, но вам стоит привыкнуть к виду команд `if` с фигурными скобками и без них.

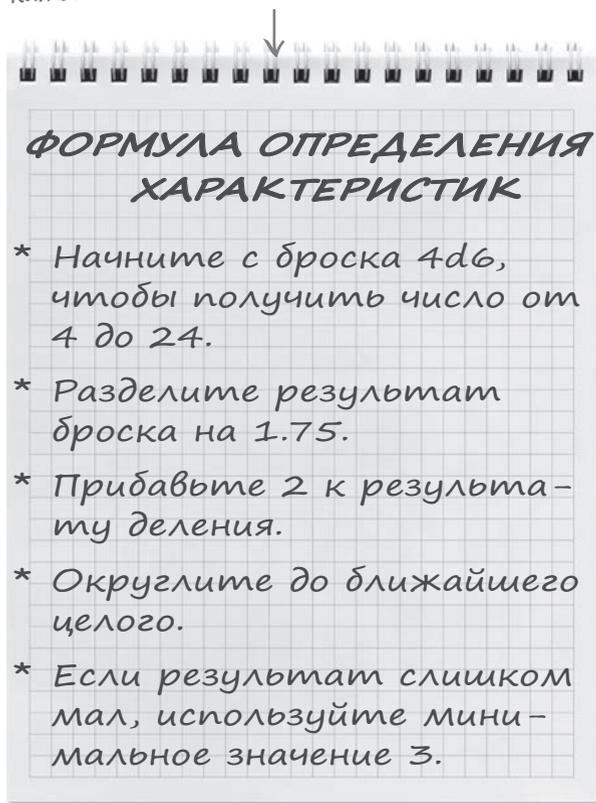
КЛЮЧЕВЫЕ МОМЕНТЫ

- Существуют **типы значений** для переменных, которые могут содержать числа разных размеров. Для самых больших целых чисел стоит использовать тип `long`, а самые малые (до 255) могут объявляться с типом `byte`.
- Каждый тип значения обладает определенным **размером**. Значение большего типа невозможно поместить в меньшую переменную независимо от фактического размера данных.
- Когда вы используете **литеральные значения**, используйте суффикс `F` для обозначения типа `float` (15.6F) и суффикс `M` для типа `decimal` (36.12M).
- **Используйте тип `decimal` для денежных сумм**. Точность формата с плавающей точкой... не идеальна.
- Некоторые типы C# умеет **преобразовывать** автоматически (неявное преобразование): `short` в `int`, `int` в `double`, `float` в `double`.
- Если компилятор не позволяет присвоить переменной значение другого типа, необходимо применить приведение типов. Чтобы **привести** значение к другому типу, укажите целевой тип в круглых скобках перед значением.
- В языке существуют **зарезервированные** ключевые слова, которые не могут использоваться в качестве имен переменных. Эти слова (`for`, `while`, `using`, `new` и т. д.) решают конкретные задачи в языке.
- **Параметр** — то, что вы определяете в методе. **Аргумент** — то, что вы передаете методу.
- При построении кода IDE использует **компилятор C#** для преобразования кода в исполняемую программу.
- Методы статического **класса `Convert`** используются для преобразования значений между разными типами.

Оуэн постоянно старается улучшить свою игру...

Хорошие гейм-мастера стремятся создать у игроков наилучшие впечатления от игры. Группа Оуэна собирается начать новую кампанию с новыми персонажами, и Оуэн думает, что с небольшими изменениями в формуле определения характеристик игра станет более интересной.

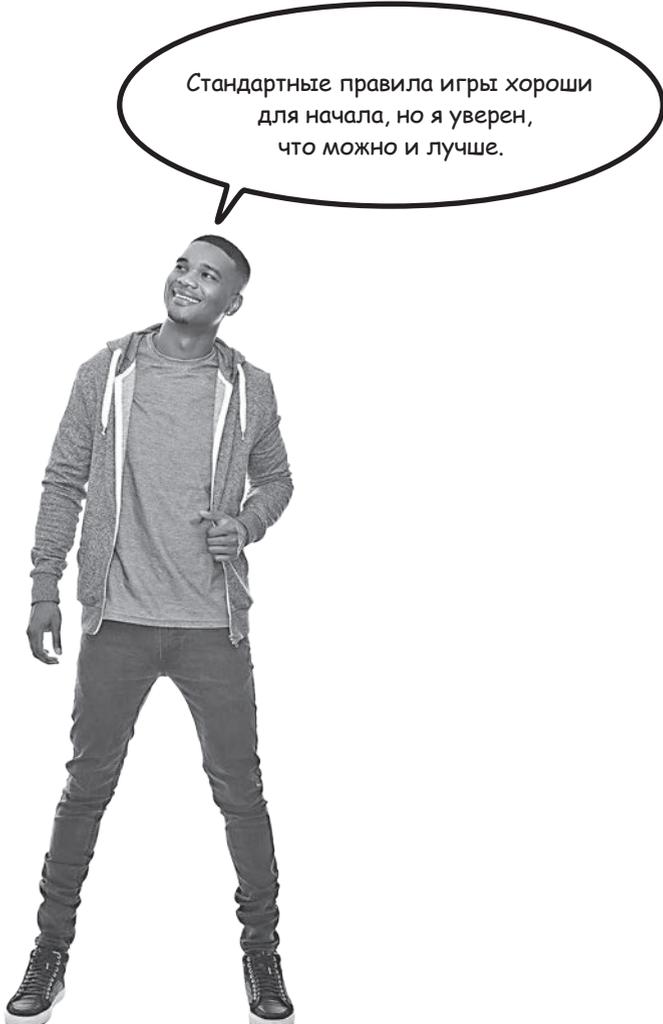
Когда игроки заполняют свои листы персонажей в начале игры, они выполняют следующие действия для вычисления каждой из начальных характеристик своего персонажа:



**ФОРМУЛА ОПРЕДЕЛЕНИЯ
ХАРАКТЕРИСТИК**

- * Начните с броска $4d6$, чтобы получить число от 4 до 24.
- * Разделите результат броска на 1.75.
- * Прибавьте 2 к результату деления.
- * Округлите до ближайшего целого.
- * Если результат слишком мал, используйте минимальное значение 3.

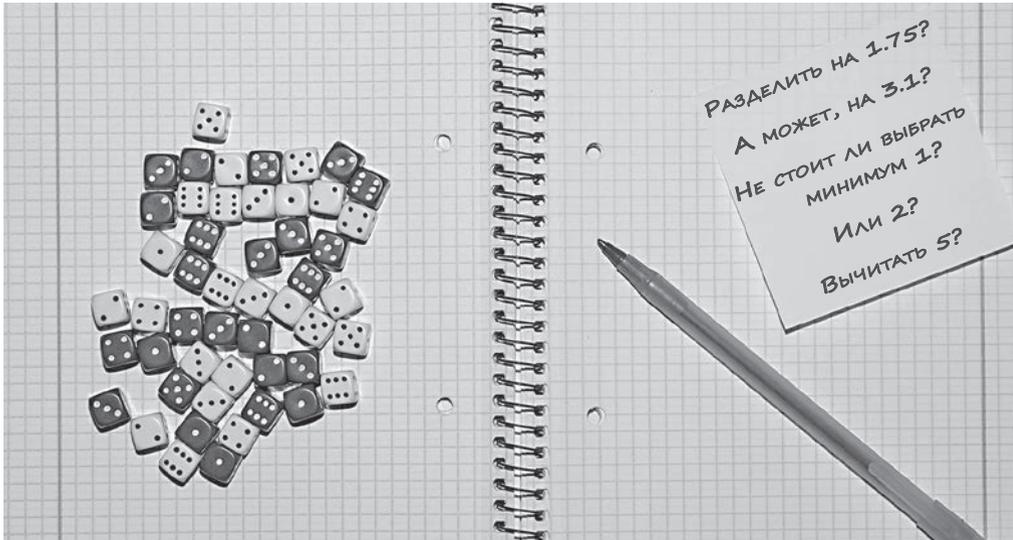
«Бросок $4d6$ » означает, что вы бросаете четыре обычных шестигранных кубика и складываете результаты.



Стандартные правила игры хороши для начала, но я уверен, что можно и лучше.

...но процесс проб и ошибок может занимать много времени

Оуэн экспериментировал с различными вариантами настройки вычисления характеристик. Он уверен, что формула в целом хороша, но ему хотелось бы иметь возможность экспериментировать с числами.



Оуэну нравится общая формула: бросить 4d6, разделить, вычесть, округлить, использовать минимальное значение... но он не уверен в правильности конкретных чисел.



Мне кажется, что 1.75 маловато для деления результата броска... И возможно, к результату лучше прибавить 3, а не 4. Наверняка должен быть более удобный способ проверки всех этих идей!

МОЗГОВОЙ ШТУРМ

Что мы можем сделать, чтобы помочь Оуэну в поиске оптимальной комбинации значений для обновленной формулы характеристик?

Поможем Оуэну в экспериментах с характеристиками

В следующем проекте мы построим консольное приложение .NET Core, при помощи которого Оуэн сможет протестировать свою формулу вычисления характеристик с разными значениями и проверить, как они влияют на результат. Формула получает **четыре входных значения**: *начальный бросок 4d6*; *делитель*, на который делится результат; *приращение*, которое прибавляется к результату деления, и *минимум*, который используется, если результат окажется слишком маленьким.

Оуэн вводит четыре входных значения в приложении, которое будет вычислять характеристики по этим данным. Вероятно, он захочет протестировать набор разных значений, поэтому для удобства приложение будет снова и снова запрашивать входные данные, пока приложение не будет завершено, отслеживать значения, введенные при каждой итерации, и использовать предыдущие значения **по умолчанию** при следующей итерации.

Вот что должен видеть Оуэн при запуске приложения:

Страница из книги гейм-мастера с формулой вычисления характеристик.

The screenshot shows a terminal window with the following text:

```

C:\Users\public\source\repos\AbilityScoreTester\AbilityScoreTester\bin\Debug\netcoreapp3.1\At
Starting 4d6 roll [14]:
  using default value 14
Divide by [1.75]:
  using default value 1.75
Add amount [2]:
  using default value 2
Minimum [3]:
  using default value 3
Calculated ability score: 10
Press Q to quit, any other key to continue
Starting 4d6 roll [14]:
  using default value 14
Divide by [1.75]: 2.15
  using value 2.15
Add amount [2]: 5
  using value 5
Minimum [3]: 2
  using value 2
Calculated ability score: 11
Press Q to quit, any other key to continue
Starting 4d6 roll [14]: 21
  using value 21
Divide by [2.15]:
  using default value 2.15
Add amount [5]:
  using default value 5
Minimum [2]:
  using default value 2
Calculated ability score: 14
Press Q to quit, any other key to continue
    
```

Handwritten annotations in Russian:

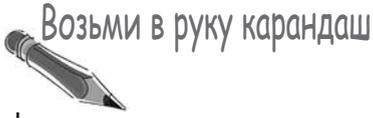
- Next to the first run: "Приложение запрашивает различные значения, используемые для вычисления характеристик. Значения по умолчанию выводятся в квадратных скобках (например, [14] или [1.75]). Оуэн может ввести значение или просто нажать Enter, чтобы подтвердить значение по умолчанию."
- Next to the second run: "Здесь Оуэн опробует новые значения: результат броска делится на 2.15 (вместо 1.75), результат деления увеличивается на 5 (вместо 2), и при вычислении используется минимальное значение 2 (вместо 3). С исходным броском 14 будет получено значение 11."
- Next to the third run: "Теперь Оуэн хочет проверить те же значения при другом исходном броске 4d6, поэтому он вводит 21 на первый запрос и нажимает Enter, чтобы подтвердить значения по умолчанию, сохраненные приложением при предыдущей итерации. На этот раз будет получено значение 14."

Reference to a game manual page:

ФОРМУЛА ОПРЕДЕЛЕНИЯ ХАРАКТЕРИСТИК

- * Начните с броска 4d6, чтобы получить число от 4 до 24.
- * Разделите результат броска на 1.75.
- * Прибавьте 2 к результату деления.
- * Округлите до ближайшего целого.
- * Если результат слишком мал, используйте минимальное значение 3.

Этот проект больше предыдущего консольного приложения, которое вы построили, поэтому мы рассмотрим его за несколько этапов. Сначала мы разберемся в коде вычисления характеристики, затем будет написан остальной код приложения, и наконец, займемся диагностикой ошибок в коде. **Итак, за дело!**



Мы построили класс, который поможет Оуэну в вычислении характеристик. Чтобы использовать его, необходимо задать значения его полей `Starting4D6Roll`, `DivideBy`, `AddAmount` и `Minimum` (или оставить полям значения, заданные при объявлении) и вызвать метод `CalculateAbilityScore`. К сожалению, **в одной строке кода допущена ошибка**. Обведите строку с ошибкой и напишите, что с ней не так.

```
class AbilityScoreCalculator
```

```
{
    public int RollResult = 14;
    public double DivideBy = 1.75;
    public int AddAmount = 2;
    public int Minimum = 3;
    public int Score;
```

Эти поля инициализируются значениями из формулы вычисления характеристик. Приложение использует их при выводе значений по умолчанию.

Удастся ли вам обнаружить проблему, не вводя класс в IDE? Найдете ли вы строку, из-за которой компилятор выдает сообщение об ошибке?

```
public void CalculateAbilityScore()
```

```
{
    // Результат броска делится на значение поля DivideBy
    double divided = RollResult / DivideBy;

    // AddAmount прибавляется к результату деления
    int added = AddAmount += divided;
```

```
    // Если результат слишком мал, использовать значение Minimum
    if (added < Minimum)
    {
        Score = Minimum;
    } else
    {
        Score = added;
    }
}
```

Подсказка: сравните комментарии в коде с формулой вычисления характеристик на странице из книги Оуэна. Какая часть формулы отсутствует в комментариях?

```
}
```

После того как вы **пометите строку кода с проблемой**, запишите, какие проблемы вы в ней обнаружили.

.....

.....

Использование компилятора C# для поиска проблемной строки кода

Создайте проект консольного приложения .NET Core Console App с именем AbilityScoreTester. Затем добавьте класс AbilityScoreCalculator с кодом из упражнения «Возьми в руку карандаш». Если код был введен правильно, вы получите ошибку компилятора C#:

```
AddAmount += divided;
```

```
(field) int AbilityScoreCalculator.AddAmount
```

```
CS0266: Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)
```

```
Show potential fixes (Alt+Enter or Ctrl+.)
```

Ошибка компилятора C# буквально напоминает о том, что вы могли пропустить приведение типа.

Каждый раз, когда компилятор C# выдает сообщение об ошибке, тщательно прочитайте его. Обычно в нем присутствует подсказка, которая поможет обнаружить проблему. В данном случае причина точно обозначена: компилятор не может преобразовать double в int без приведения типа. Переменная divided объявлена с типом double, но C# не позволит добавить ее к полю int (такому, как AddAmount), потому что не знает, как преобразовать ее.

Компилятор дает чрезвычайно ценную подсказку о том, что вы должны выполнить приведение типа double-переменной divided, прежде чем прибавлять ее к int-полю AddAmount.

Добавим приведение типа, чтобы класс AbilityScoreCalculator компилировался...

Теперь мы знаем, в чем заключается суть проблемы, и можем добавить **приведение типа** для исправления проблемной строки кода в AbilityScoreCalculator. Ошибка «Не удастся неявно преобразовать тип» выдается следующей строкой:

```
int added = AddAmount += divided;
```

Ошибка возникает из-за того, что команда **AddAmount += divided** возвращает значение double, которое не может быть присвоено int-переменной added.

Проблему можно решить **приведением divided** к **int**, чтобы при прибавлении к AddAmount было возвращено другое значение int. Замените в этой строке кода divided на (int)divided:

```
int added = AddAmount += (int)divided;
```

← Преобразуем!

Приведение также добавляет отсутствующую часть формулы Оуэна:

**ОКРУГЛИТЕ ДО БЛИЖАЙШЕГО ЦЕЛОГО.*

Когда вы приводите double к int, C# округляет результат — так что, например, (int)19.7431D дает 19. Добавляя это приведение, вы также добавляете пункт формулы, отсутствующий в классе.

...но ошибка все равно осталась!

Работа еще не закончена! Ошибка компилятора исправлена, так что проект успешно строится. Но хотя компилятор C# не протестует, **проблема все еще осталась**. Удается ли вам найти ошибку в следующей строке кода?

↑
Похоже, заполнять ответ во врезке «Возьми в руку карандаш» еще рано!



Упражнение

Завершим построение консольного приложения, использующего класс `AbilityScoreCalculator`. В этом упражнении мы предоставим метод `Main` для консольного приложения. Ваша задача — написать код двух методов: метод `ReadInt` читает ввод от пользователя и преобразует его в `int` вызовом `int.TryParse`, а метод `ReadDouble` делает то же самое, но работает со значениями `double` вместо `int`.

1. Добавьте следующий метод `Main`. Почти весь его код уже знаком вам по предыдущим проектам. Единственным новшеством оказывается вызов метода `Console.ReadKey`:

```
char keyChar = Console.ReadKey(true).KeyChar;
```

Метод `Console.ReadKey` читает одно нажатие клавиши с консоли. При передаче аргумента `true` ввод перехватывается и не выводится на консоль. Сцепленный вызов `.KeyChar` возвращает нажатие клавиши в виде `char`.

Итак, перед вами полный метод `Main` — добавьте его в программу:

```
static void Main(string[] args)
{
    AbilityScoreCalculator calculator = new
    AbilityScoreCalculator();
    while (true)
    {
        calculator.RollResult = ReadInt(calculator.RollResult, "Starting 4d6 roll");
        calculator.DivideBy = ReadDouble(calculator.DivideBy, "Divide by");
        calculator.AddAmount = ReadInt(calculator.AddAmount, "Add amount");
        calculator.Minimum = ReadInt(calculator.Minimum, "Minimum");
        calculator.CalculateAbilityScore();
        Console.WriteLine("Calculated ability score: " + calculator.Score);
        Console.WriteLine("Press Q to quit, any other key to continue");
        char keyChar = Console.ReadKey(true).KeyChar;
        if ((keyChar == 'Q') || (keyChar == 'q')) return;
    }
}
```

Мы будем использовать один экземпляр `AbilityScoreCalculator`. Пользовательский ввод будет обновлять поля экземпляра, чтобы он запомнил значения по умолчанию для следующей итерации цикла `while`.

2. Добавьте метод с именем `ReadInt`. Метод получает два параметра: сообщение для пользователя и значение по умолчанию. Сообщение выводится на консоль, за ним следует значение по умолчанию в квадратных скобках. Затем метод читает строку с консоли и пытается преобразовать ее. Если преобразование проходит успешно, то метод использует это значение; в противном случае используется значение по умолчанию.

```
/// <summary>
/// Выводит сообщение и читает значение int с консоли.
/// </summary>
/// <param name="lastUsedValue">Значение по умолчанию.</param>
/// <param name="prompt">Сообщение, выводимое на консоль.</param>
/// <returns>Прочитанное значение int или значение по умолчанию, если преобразование
/// невозможно.</returns>
static int ReadInt(int lastUsedValue, string prompt)
{
    // Вывести сообщение, за которым следует [значение по умолчанию]:
    // Прочитать строку из ввода и попытаться преобразовать ее вызовом int.TryParse
    // Если преобразование прошло успешно, вывести на консоль строку " using value" + value.
    // В противном случае вывести на консоль строку " using default value" + lastUsedValue
}
```

3. Добавьте метод `ReadDouble`, который полностью повторяет `ReadInt`, но использует `double.TryParse` вместо `int.TryParse`. Метод `double.TryParse` работает так же, как `int.TryParse`, но его переменная `out` должна относиться к типу `double` вместо `int`.



Упражнение Решение

Ниже приведены методы `ReadInt` и `ReadDouble`, которые выводят сообщение со значением по умолчанию, читают строку с консоли, пытаются преобразовать ее в `int` или `double` и выводят на консоль сообщение с преобразованным значением или со значением по умолчанию.

```
static int ReadInt(int lastUsedValue, string prompt)
{
    Console.Write(prompt + " [" + lastUsedValue + "]: ");
    string line = Console.ReadLine();
    if (int.TryParse(line, out int value))
    {
        Console.WriteLine("    using value " + value);
        return value;
    } else
    {
        Console.WriteLine("    using default value " + lastUsedValue);
        return lastUsedValue;
    }
}

static double ReadDouble(double lastUsedValue, string prompt)
{
    Console.Write(prompt + " [" + lastUsedValue + "]: ");
    string line = Console.ReadLine();
    if (double.TryParse(line, out double value)) ←
    {
        Console.WriteLine("    using value " + value);
        return value;
    } else
    {
        Console.WriteLine("    using default value " + lastUsedValue);
        return lastUsedValue;
    }
}
```

Основательно разберитесь в том, как каждая итерация цикла `while` в методе `Main` сохраняет в полях значения, введенные пользователем, а затем использует их как значения по умолчанию для следующей итерации.

Вызов `double.TryParse` работает так же, как версия для `int`, не считая того, что для выходной переменной должен использоваться тип `double`.



Спасибо за ваше приложение!
Мне не терпится опробовать его в деле.

Результаты работы приложения.

```

Starting 4d6 roll [14]: 18
  using value 18
Divide by [1.75]: 2.15
  using value 2.15
Add amount [2]: 5
  using value 5
Minimum [3]:
  using default value 3
Calculated ability score: 13
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
  using default value 18
Divide by [2.15]: 3.5
  using value 3.5
Add amount [13]: 5
  using value 5
Minimum [3]:
  using default value 3
Calculated ability score: 10
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
  using default value 18
Divide by [3.5]:
  using default value 3.5
Add amount [10]: 7
  using value 7
Minimum [3]:
  using default value 3
Calculated ability score: 12
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
  using default value 18
Divide by [3.5]:
  using default value 3.5
Add amount [12]: 4
  using value 4
Minimum [3]:
  using default value 3
Calculated ability score: 9
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
  using default value 18
Divide by [3.5]:
  using default value 3.5
Add amount [9]:
  using default value 9
Minimum [3]:
  using default value 3
Calculated ability score: 14
Press Q to quit, any other key to continue
    
```

Что-то не так. Класс должен запоминать значения, которые я ввел, но он не всегда это делает.

Вот! При первой итерации я ввел приращение 5. Все остальные значения были сохранены правильно, но для приращения приложение выводит значение по умолчанию 10.



Странно. На предыдущей итерации Оуэн ввел приращение 5, но программа выдает значение по умолчанию 10.

И снова: последнее приращение было равно 7, но приложение выводит значение по умолчанию 12. Непонятно.

Откуда вообще взялось число 9? Мы видели его прежде? Можно ли сделать из этого какие-то выводы о причинах ошибки?

Ты прав, Оуэн. В коде ошибка.

Оуэн хочет опробовать разные значения для своей формулы вычисления характеристик, поэтому мы в цикле запрашиваем эти значения снова и снова.

Чтобы Оуэну было проще изменять значения по одному, мы включили в приложение функцию, которая запоминает введенные ранее значения и предлагает их как значения по умолчанию. Данная возможность была реализована хранением класса AbilityScore CostCalculator и обновлением его полей при каждой итерации цикла while.

И все же с приложением что-то не так. Большинство значений запоминается нормально, но для приращения запоминается ошибочное число. При первой итерации Оуэн ввел 5, но в качестве значения по умолчанию предлагается 10. Затем он вводит 7, но получает значение по умолчанию 12. Что происходит?



МОЗГОВОЙ ШТУРМ

Какие шаги мы можем предпринять для выявления ошибки в калькуляторе характеристик?



По следу

В процессе отладки кода вы выполняете детективную работу. В приложении что-то вызывает ошибку, и ваша задача — определить подозреваемых и пройти по их следам. Проведем небольшое расследование в духе Шерлока Холмса и посмотрим, удастся ли нам найти виновника.

Похоже, проблема существует только с приращением, поэтому для начала найдем любую строку с упоминанием поля AddAmount — установите в нем точку прерывания:

```
39 calculator.DivideBy = ReadDouble(calculator.DivideBy, "Divide by");
40 calculator.AddAmount = ReadInt(calculator.AddAmount, "Add amount");
41 calculator.Minimum = ReadInt(calculator.Minimum, "Minimum");
```

А вот еще одна строка в методе AbilityScoreCalculator.CalculateAbilityScore — еще один подозреваемый:

```
20 // Add to the result
21 int added = AddAmount += (int)divided;
```

← Эта команда должна обновлять переменную "added", но не изменять поле AddAmount.

Теперь запустите программу. Когда в методе Main сработает точка прерывания, выберите calculator.AddAmount и добавьте отслеживание (если просто щелкнуть правой кнопкой мыши на AddAmount и выбрать команду Add Watch в меню, то будет добавлено отслеживание для AddAmount, но не calculator.AddAmount). Что-нибудь выглядит странно? Ничего необычного не видно. Значение читается и обновляется вполне нормально. Видимо, проблема вызвана чем-то другим — отключите или удалите эту точку прерывания.

Продолжайте выполнять программу. При достижении точки прерывания AbilityScoreCalculator.CalculateAbilityScore **добавьте отслеживание для AddAmount**. По формуле Оуэна эта строка кода должна прибавлять AddAmount к результату деления. Выполните команду в пошаговом режиме, и...



Стоп, что?! Значение AddAmount изменилось. Но... такого не должно быть — это невозможно! Верно? Как говорил Шерлок Холмс, «если исключить невозможное, тогда то, что останется, будет ответом, каким бы невероятным он ни казался».

Похоже, мы обнаружили источник проблемы. Команда должна привести divided к int и округлить до целого числа, а затем прибавить результат к AddAmount и сохранить результат в added. Но у нее также имеется непредвиденный побочный эффект: она обновляет AddAmount суммой, потому что в команде используется оператор +=, который не только возвращает сумму, но и присваивает ее AddAmount.

Наконец-то мы можем исправить ошибку в приложении Оуэна

Теперь мы знаем, что произошло, и можем **исправить ошибку** — изменение оказывается незначительным. Нужно лишь изменить команду, чтобы в ней использовался оператор + вместо +=:

```
int added = AddAmount + (int)divided;
```

Заменили += на +, чтобы строка кода не обновляла переменную AddAmount. «Элементарно», как говорил Шерлок.



Возьми в руку карандаш
Решение

Итак, причина ошибки найдена, и мы наконец-то можем привести решение.

Мы построили класс, который поможет Оуэну в вычислении характеристик. Чтобы использовать его, необходимо задать значения его полей Starting4D6Roll, DivideBy, AddAmount и Minimum (или оставить полям значения, заданные при объявлении) и вызвать метод CalculateAbilityScore. К сожалению, **в одной строке кода допущена ошибка**. Обведите строку с ошибкой и напишите, что с ней не так.

```
int added = AddAmount += divided;
```

После того как вы **пометите строку кода с проблемой**, запишите, какие проблемы вы в ней обнаружили.

Во-первых, она не компилируется, потому что результат AddAmount += divided имеет тип double и для присваивания его int потребуются приведение типа. Во-вторых, в ней используется += вместо +, из-за чего строка обновляет AddAmount.

Часто задаваемые вопросы

В: Я все еще не до конца понимаю, чем оператор + отличается от оператора +=. Как они работают и в каких случаях нужно использовать тот или иной оператор?

О: Некоторые операторы могут объединяться со знаком =. К их числу относятся += для сложения, -= для вычитания, /= для деления, *= для умножения и %= для вычисления остатка. Операторы, работающие с двумя значениями (такие, как +), называются **бинарными**.

С бинарными операторами можно выполнять так называемое **комбинированное присваивание**. Иначе говоря, вместо:

```
a = a + c;
```

можно использовать запись:

```
a += c;
```

Оператор += призывает C# вычислить a + c, а затем сохранить результат в a.

Это означает то же самое. Если вы предпочитаете техническое объяснение, комбинированное присваивание `x op= y` эквивалентно `x = x op y`.

Операторы, объединяющие бинарный оператор со знаком = (такие, как += или *=), называются комбинированными операторами присваивания.

В: Но как тогда обновляется переменная added?

О: Вся путаница в калькуляторе произошла из-за того, что **оператор присваивания = тоже возвращает значение**. Вы можете использовать команду вида:

```
int q = (a = b + c)
```

Эта команда, как обычно, вычисляет `a = b + c`. Оператор = **возвращает** значение, поэтому **переменная q также будет обновлена результатом**. А значит, команда:

```
int added = AddAmount += divided;
```

эквивалентна следующей:

```
int added = (AddAmount = AddAmount + divided);
```

В результате AddAmount увеличивается на divided, но результат также сохраняется в added.

В: Погодите, что? Оператор присваивания возвращает значение?

О: Да, = возвращает присвоенное значение. Следовательно, в коде

```
int first;
int second = (first = 4);
```

и first, и second в итоге будет присвоено 4. Откройте консольное приложение и проверьте с помощью отладчика. Это действительно работает!

Преобразуйте!



Эй, детка! Хочешь увидеть нечто по-настоящему странное?

o o



Попробуйте включить следующую команду `if/else` в консольное приложение:

```
if (0.1M + 0.2M == 0.3M) Console.WriteLine("They're equal");
else Console.WriteLine("They aren't equal");
```

Под второй командой Console появляется зеленая волнистая черта – это предупреждение об обнаружении **недоступного кода**. Компилятор C# знает, что $0.1 + 0.2$ всегда дает результат 0.3, так что код никогда не достигнет части `else` в команде. Выполните код – он выводит на консоль сообщение `They're equal`.

Теперь **замените литералы float на double** (помните: такие литералы, как 0.1, по умолчанию интерпретируются как double):

```
if (0.1 + 0.2 == 0.3) Console.WriteLine("They're equal");
else Console.WriteLine("They aren't equal");
```

Происходит что-то странное: предупреждение переместилось в первую строку команды `if`. Попробуйте выполнить программу. Этого не может быть! На консоль выводится сообщение `They aren't equal`. Как $0.1 + 0.2$ может быть не равно 0.3?

А теперь еще одно. Замените 0.3 на 0.30000000000000004 (с 15 нулями между 3 и 4). Теперь программа снова выводит `They're equal`. Очевидно, $0.1D + 0.2D$ равно 0.30000000000000004.

←
Что-что?!

Вот, значит, почему для **финансовых вычислений всегда нужно использовать decimal, а не double?**



Точно. Тип decimal обеспечивает куда большую точность, чем double или float, поэтому проблемы 0.30000000000000004 в нем не существует.

Некоторые типы с плавающей точкой – не только в C#, но и в большинстве языков программирования! – могут создавать **редкие** и странные ошибки. Как в результате сложения $0.1 + 0.2$ можно получить 0.30000000000000004?

Оказывается, некоторые числа просто не имеют точного представления в виде `double` – это связано со способом их хранения в двоичных данных (0 и 1 в памяти). Например, `.1D` не *точно* равно `.1`. Попробуйте умножить `.1D * .1D` – вы получите 0.010000000000000002 вместо 0.01. С другой стороны, `.1M * .1M` дает точный ответ. Типы `float` и `double` очень полезны для многих операций (например, для позиционирования `GameObject` в Unity). Если вам нужна большая точность, как, например, в финансовых приложениях, работающих с денежными суммами, – выбирайте `decimal`.

Часто
Задаваемые Вопросы

В: Я все еще плохо понимаю, чем преобразования отличаются от приведения типов. Можно объяснить чуть понятнее?

О: Преобразование — общий, универсальный термин для перевода данных из одного типа в другой. Приведение — куда более конкретная операция, с явными правилами относительно того, какие типы могут быть приведены к другим типам и что делать, если данные значения одного типа не полностью совпадают с типом, к которому осуществляется приведение. Пример такого правила вам уже встречался — когда число с плавающей точкой преобразуется к `int`, оно округляется усечением дробной части. Еще одно правило проявляется при циклическом возврате для целочисленных типов: если число не помещается в целевом типе, оно усекается с использованием оператора вычисления остатка.

В: В формуле Оуэна мы делили два значения, а затем округляли результат до ближайшего целого. Как это согласуется с приведением типов?

О: Допустим, у вас имеется набор значений с плавающей точкой:

```
float f1 = 185.26F;
double d2 = .0000316D;
decimal m3 = 37.26M;
```

и вы хотите привести их к типу `int`, чтобы присвоить их переменным `int i1, i2 и i3`. Мы знаем, что в переменных `int` могут храниться только целые числа, так что программа должна что-то сделать с дробной частью числа.

По этой причине `C#` руководствуется универсальным правилом: дробная часть отбрасывается. `f1` преобразуется в 185, `d2` — в 0, а `m3` — в 37. Впрочем, не верьте нам на слово — напишите собственный код `C#`, который преобразует эти три значения с плавающей точкой в `int`, и посмотрите, что произойдет.

Существует целый веб-сайт, посвященный проблеме 0.30000000000000004! Откройте сайт <https://0.30000000000000004.com>, чтобы увидеть примеры на многих других языках.

Пример `0.1D + 0.2D != 0.3D` является граничным случаем, т. е. проблемой, которая возникает только в конкретной редкой ситуации — например, когда параметр принимает одно из крайних значений (очень большое или очень малое число). Если вы захотите больше узнать об этом, Джон Скит (John Skeet) написал отличную статью о хранении чисел с плавающей точкой в памяти `.NET`. Статья доступна по адресу <https://csharpindepth.com/Articles/FloatingPoint>.

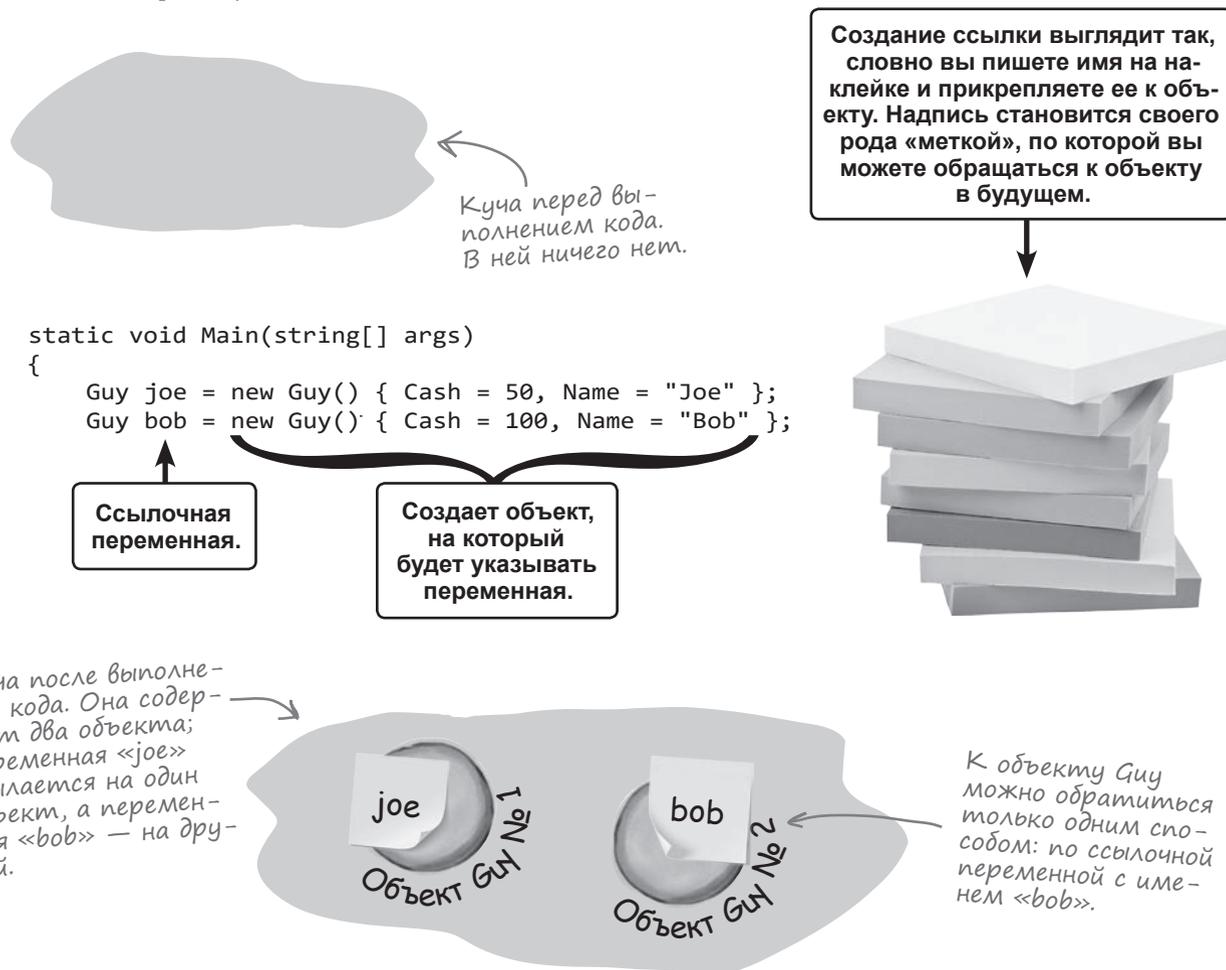
↑
Джон предоставил нам ряд ценнейших технических замечаний по первому изданию книги, которые были очень важны для нас. Спасибо, Джон!

Использование ссылочных переменных для обращения к объектам

Создавая новый объект, вы создаете экземпляр командой `new` — например, `new Guy()` в вашей программе в конце предыдущей главы создает новый объект `Guy` в куче. При этом к объекту все равно нужно как-то обратиться, для чего используются такие переменные, как `joe: Guy joe = new Guy()`. Давайте разберемся в том, что же здесь происходит.

Команда `new` создала экземпляр, но одного создания экземпляра недостаточно. *Понадобится ссылка на объект.* Поэтому мы создали ссылочную переменную, т. е. переменную типа `Guy` с именем (например, `joe`). Таким образом, `joe` содержит ссылку на только что созданный объект `Guy`. Каждый раз, когда вы захотите использовать этот конкретный объект `Guy`, к нему можно обратиться по ссылочной переменной `joe`.

Любая переменная, относящаяся к объектному типу, является ссылочной переменной, т. е. она содержит ссылку на конкретный объект. Просто убедимся в том, что мы одинаково понимаем эти термины, потому что они будут часто использоваться в этой главе. Воспользуемся первыми двумя строками программы «Джо и Боб» из предыдущей главы:

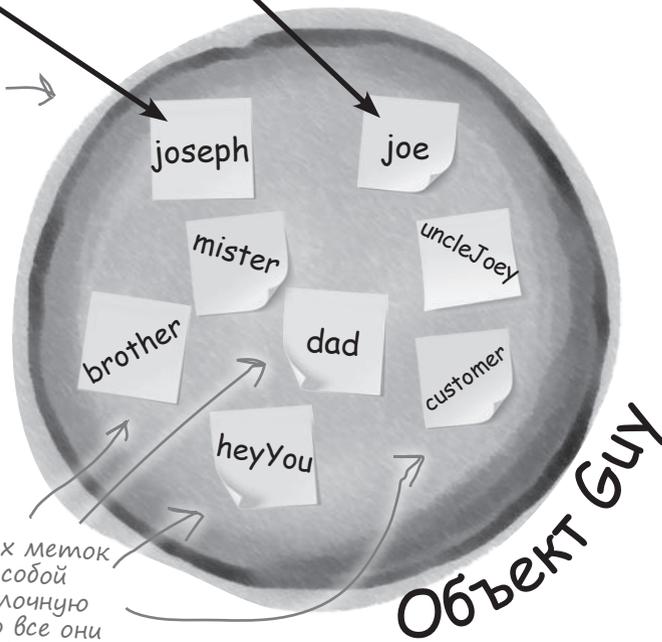


Ссылки напоминают наклейки на ваших объектах

Вероятно, на вашей кухне есть контейнеры для соли и сахара. Если вы случайно меняете местами наклейки, вряд ли еду можно будет есть — хотя надписи изменились, содержимое контейнеров осталось прежним. *Ссылки похожи на эти наклейки.* Наклейки можно перемещать, но набор доступных методов и данных зависит от **объекта**, а не от ссылки — и ссылки можно **копировать** точно так же, как вы копируете значения.

```
Guy joe = new Guy();
Guy joseph = joe;
```

Мы создали объект Guy ключевым словом «new» и скопировали ссылку на него оператором «=».



Каждая из этих меток представляет собой отдельную ссылочную переменную, но все они указывают на ОДИН И ТОТ ЖЕ объект Guy.

Ссылка представляет собой своего рода метку, которая используется в вашем коде для обращений к конкретному объекту. Она используется для обращения к полям и вызова методов того объекта, на который она указывает.

В этом конкретном случае существует множество разных ссылок на объект Guy, потому что многие разные методы используют его для разных целей. Каждой ссылке присваивается отдельное имя, которое имеет смысл в этом контексте.

Вот почему может быть очень полезно иметь *несколько ссылок, указывающих на один экземпляр*. Следовательно, в программу можно включить команду `Guy dad = joe`, а затем вызвать метод `dad.GiveCash()`. Если вы хотите написать код, работающий с объектом, вам понадобится ссылка на этот объект. Без ссылки обращения к объекту невозможны.

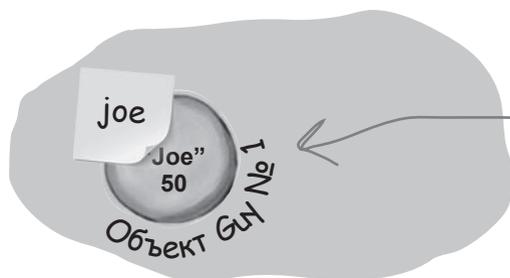
Если ни одной ссылки не осталось, объект уничтожается сборщиком мусора

Если с объекта была снята последняя наклейка, программа не сможет обратиться к объекту. Это означает, что С# может пометить объект для **сборки мусора**. После этого С# избавляется от любых объектов, на которые не существует ни одной ссылки, и освобождает память, которую занимали эти объекты.

1 Код, создающий объект.

На всякий случай вспомним, о чем говорилось ранее: когда вы используете команду `new`, вы тем самым приказываете С# создать объект. Когда вы берете ссылочную переменную (например, `joe`) и присваиваете ей созданный объект, все выглядит так, словно вы прикрепляете к нему новую наклейку.

```
Guy joe = new Guy() { Cash = 50, Name = "Joe" };
```

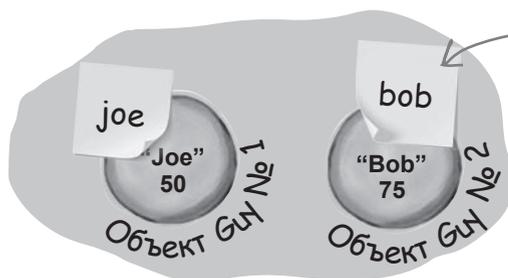


Для создания этого объекта `Guy` использовался инициализатор объекта. Поле `Name` содержит строку "Joe", полю `Cash` присвоено значение `int 50`, а ссылка на объект сохраняется в переменной с именем «joe».

2 Теперь создадим второй объект.

После этого мы имеем два объекта `Guy` и две ссылочные переменные: одна переменная (`joe`) для первого объекта `Guy` и другая переменная (`bob`) для второго.

```
Guy bob = new Guy() { Cash = 100, Name = "Bob" };
```



Мы создали другой объект `Guy` и переменную с именем «bob», которая указывает на него. Переменные похожи на наклейки — это простые метки, которые можно «прикрепить» к любому объекту.

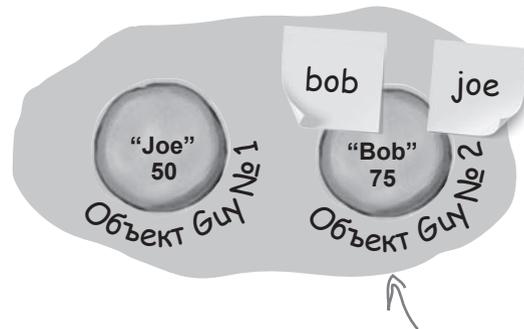
3 Возьмем ссылку на первый объект `Guy` и переведем ее на второй объект `Guy`.

Внимательно присмотримся к тому, что происходит при создании нового объекта `Guy`. Мы берем переменную и используем оператор присваивания `=`, чтобы задать ей новое значение — в данном случае ссылку, которую возвращает команда `new`. Присваивание работает, потому что **ссылки можно копировать точно так же, как вы копируете значения.**

Попробуем скопировать это значение:

```
joe = bob;
```

Команда сообщает C#, что переменная `joe` должна указывать на тот же объект, что и `bob`. Теперь переменные `joe` и `bob` указывают на **один объект.**



После того как CLR (см. далее в интервью «Откровенно о сборке мусора»!) удалит последнюю ссылку на объект, он помечается для сборки мусора.

4 Ссылок на первый объект `Guy` больше не осталось... поэтому он уничтожается сборщиком мусора.

Теперь, когда переменная `joe` указывает на тот же объект, что и `bob`, на объект `Guy`, на который она указывала ранее, не осталось ни одной ссылки. Что же происходит? C# помечает объект для сборки мусора и *со временем* уничтожает его. Бах — и его нет!

CLR отслеживает все ссылки на каждый объект и при исчезновении последней ссылки помечает его для уничтожения. Но возможно, у CLR сейчас есть более неотложные дела, поэтому объект может просуществовать еще несколько миллисекунд — и даже более!



Чтобы объект оставался в куче, должны существовать ссылки на него. Через какое-то время после исчезновения последней ссылки исчезает и сам объект.

```
public partial class Dog {
    public void GetPet() {
        Console.WriteLine("Woof!");
    }
}
```

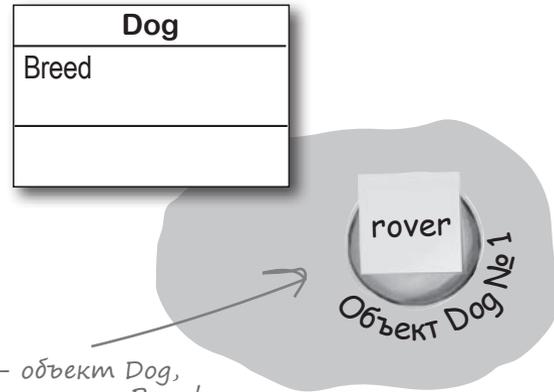
Множественные ссылки и их побочные эффекты

При перемещении ссылочных переменных необходимо действовать осторожно. Во многих случаях создается впечатление, что переменная просто начинает указывать на другой объект. Тем не менее при этом может быть удалена последняя ссылка на другой объект. Это не всегда плохо, но может быть и не тем, чего вы ожидали. Взгляните:

1 `Dog rover = new Dog();`
`rover.Breed = "Greyhound";`

Objects: 1

References: 1

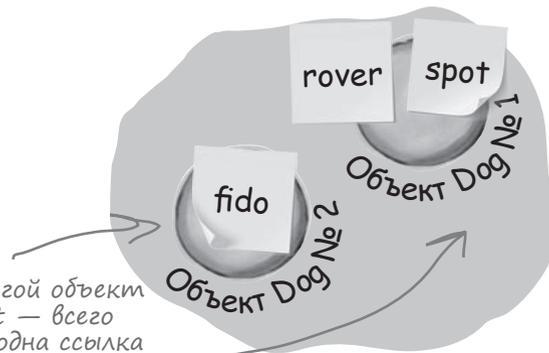


rover — объект Dog, у которого поле Breed содержит Greyhound.

2 `Dog fido = new Dog();`
`fido.Breed = "Beagle";`
`Dog spot = rover;`

Objects: 2

References: 3

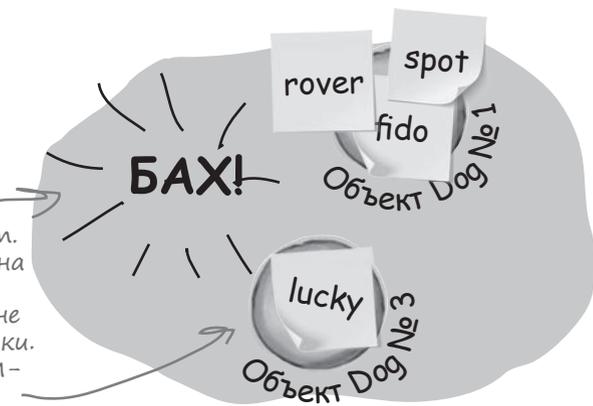


fido — другой объект Dog, а spot — всего лишь еще одна ссылка на первый объект.

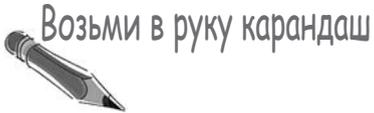
3 `Dog lucky = new Dog();`
`lucky.Breed = "Dachshund";`
`fido = rover;`

Objects: 2

References: 4



lucky — третий объект. fido теперь указывает на объект № 1. Таким образом, на объект № 2 не остается ни одной ссылки. С точки зрения программы работа закончена.



Теперь ваша очередь. Ниже приведен один длинный блок кода. Определите, сколько объектов и ссылок существует на каждой стадии. Справа нарисуйте схему с представлением объектов и ссылок в куче.

```
1 Dog rover = new Dog();
  rover.Breed = "Greyhound";
  Dog rinTinTin = new Dog();
  Dog fido = new Dog();
  Dog greta = fido;
```

Objects: _____

References: _____

```
2 Dog spot = new Dog();
  spot.Breed = "Dachshund";
  spot = rover;
```

Objects: _____

References: _____

```
3 Dog lucky = new Dog();
  lucky.Breed = "Beagle";
  Dog charlie = fido;
  fido = rover;
```

Objects: _____

References: _____

```
4 rinTinTin = lucky;
  Dog laverne = new Dog();
  laverne.Breed = "pug";
```

Objects: _____

References: _____

```
5 charlie = laverne;
  lucky = rinTinTin;
```

Objects: _____

References: _____



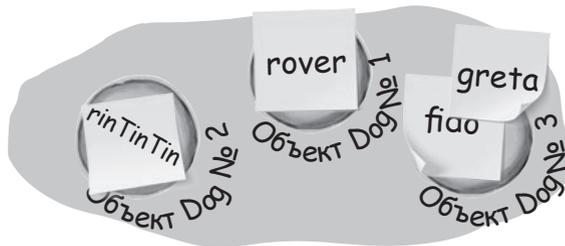
Возьми в руку карандаш Решение

```
1 Dog rover = new Dog();
  rover.Breed = "Greyhound";
  Dog rinTinTin = new Dog();
  Dog fido = new Dog();
  Dog greta = fido;
```

Objects: 3

References: 4

Создается один новый объект Dog, но spot содержит единственную ссылку на него. Когда spot присваивается ссылка rover, этот объект пропадает.



```
2 Dog spot = new Dog();
  spot.Breed = "Dachshund";
  spot = rover;
```

Objects: 3

References: 5

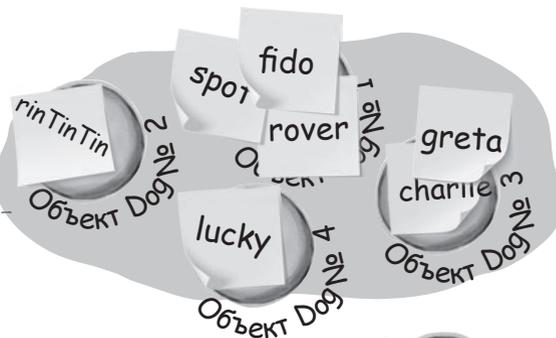


```
3 Dog lucky = new Dog();
  lucky.Breed = "Beagle";
  Dog charlie = fido;
  fido = rover;
```

Objects: 4

References: 7

charlie была присвоена ссылка fido, когда ссылка fido еще указывала на объект № 3. После этого ссылка fido была переведена на объект № 1.



```
4 rinTinTin = lucky;
  Dog laverne = new Dog();
  laverne.Breed = "pug";
```

Objects: 4

References: 8

Объект Dog № 2 потерял свою последнюю ссылку, поэтому он уничтожается.



Когда rinTinTin переходит на объект lucky, старый объект rinTinTin исчез.

```
5 charlie = laverne;
  lucky = rinTinTin;
```

Objects: 4

References: 8

На этом этапе ссылки перемещаются, но новые ссылки не создаются. Присваивание lucky ссылке tinTin не делает ничего, потому что обе ссылки уже указывают на один и тот же объект.





Откровенно о сборке мусора

Интервью недели:
.NET CLR

Head First: Хорошо, мы понимаем, что вы решаете для нас довольно важную задачу. Можете чуть подробнее рассказать, чем же вы занимаетесь?

CLR (Common Language Runtime): В общем-то все просто: я выполняю ваш код. Каждый раз, когда вы запускаете приложение .NET, я обеспечиваю его работу.

Head First: Что вы имеете в виду — «обеспечиваю работу»?

CLR: Я беру на себя всю низкоуровневую «черную работу», становясь своего рода «посредником» между вашей программой и компьютером, на котором она выполняется. Когда речь заходит о создании экземпляров или сборке мусора, именно я выполняю все эти операции.

Head First: И как же именно это происходит?

CLR: Когда вы запускаете программу в Windows, Linux, macOS или другой операционной системе, ОС загружает код на машинном языке из двоичного файла.

Head First: Извините, вынужден перебить. А вы можете немного отступить и рассказать, что такое машинный язык?

CLR: Конечно. Программа, написанная на машинном языке, состоит из кода, выполняемого непосредственно процессором, — и читать его намного сложнее, чем код C#.

Head First: Если процессор выполняет реальный машинный код, то что делает ОС?

CLR: ОС следит за тем, чтобы каждой программе был выделен отдельный процесс, чтобы она соблюдала правила безопасности системы, а также предоставляет различные API.

Head First: Для наших читателей, которые не знают, что такое API?..

CLR: API, или интерфейс прикладного программирования, — это набор методов, представляемых ОС, библиотекой или программой. API ОС позволяют выполнять такие операции, как работа с файловой системой или взаимодействие с оборудованием. Но порой они достаточно сложны в использовании (особенно API управления памятью) и изменяются в зависимости от ОС.

Head First: Хорошо, возвращаемся к теме. Вы упомянули о двоичном файле. Что это такое?

CLR: Двоичный файл (обычно) создается **компилятором** — программой, задача которой сводится к преобразованию высокоуровневого языка в низкоуровневый код — например, машинный. Двоичные файлы Windows обычно завершаются суффиксом *.exe* или *.dll*.

Head First: Что-то здесь не так. Вы сказали: «низкоуровневый код — например, машинный». Означает ли это, что существуют и другие разновидности низкоуровневого кода?

CLR: Точно. Я не использую тот же машинный язык, что и процессор. Когда вы строите ваш код C#, Visual Studio приказывает компилятору C# генерировать код **на языке CIL** (Common Intermediate Language). Именно его я и выполняю. Код C# преобразуется в код CIL, а я читаю и выполняю этот код.

Head First: Вы упомянули об управлении памятью. Какое место в ней занимает сборка мусора?

CLR: Да! Среди прочего, я реализую очень, очень полезную возможность — я слежу за тем, когда ваша программа завершает работу с некоторыми объектами. И когда объект становится ненужным, я уничтожаю его, чтобы освободить занимаемую им память. Когда-то программистам приходилось делать это самостоятельно, но благодаря мне вам теперь не придется беспокоиться об этом. Может, вы об этом и не знали, но я сильно упрощаю вашу задачу по изучению C#.

Head First: Вы упоминали двоичные файлы Windows. А если я запускаю программы .NET на Mac или в Linux? В этих ОС вы делаете то же самое?

CLR: Если вы используете macOS или Linux или запускаете Mono в Windows, то формально вы используете не меня, а моего родственника Mono Runtime. Он реализует тот же стандарт ECMA CLI (Common Language Infrastructure), что и я. Таким образом, когда дело доходит до того, о чем я говорил ранее, мы оба делаем одно и то же.



Упражнение

Создайте программу с классом `Elephant`. Создайте два экземпляра `Elephant`, после чего поменяйте местами ссылки, которые указывают на них, — так, чтобы ни один из экземпляров не был уничтожен в ходе сборки мусора. Ниже показано, как должен выглядеть результат при выполнении программы.

Мы построим новое консольное приложение, которое содержит класс с именем `Elephant`.

Пример вывода программы:

Press 1 for Lloyd, 2 for Lucinda, 3 to swap

You pressed 1

Calling `lloyd.WhoAmI()`

My name is Lloyd.

My ears are 40 inches tall.

You pressed 2

Calling `lucinda.WhoAmI()`

My name is Lucinda.

My ears are 33 inches tall.

You pressed 3

References have been swapped

You pressed 1

Calling `lloyd.WhoAmI()`

My name is Lucinda.

My ears are 33 inches tall.

You pressed 2

Calling `lucinda.WhoAmI()`

My name is Lloyd.

My ears are 40 inches tall.

You pressed 3

References have been swapped

You pressed 1

Calling `lloyd.WhoAmI()`

My name is Lloyd.

My ears are 40 inches tall.

You pressed 2

Calling `lucinda.WhoAmI()`

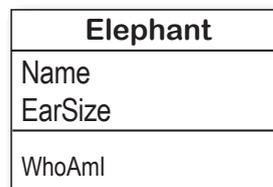
My name is Lucinda.

My ears are 33 inches tall.

Класс `Elephant` содержит метод `WhoAmI`, который выводит на консоль две строки со значениями полей `Name` и `EarSize`.

Перестановка ссылок заставляет переменную `lloyd` вызывать метод объекта `Lucinda`, и наоборот.

Диаграмма класса `Elephant`, который вам предстоит создать.



Повторная перестановка возвращает ситуацию к исходному состоянию на момент запуска программы.

CLR уничтожает в ходе сборки мусора любой объект, на который не осталось ни одной ссылки. Поэтому вот вам подсказка для этого упражнения: если вы хотите перелить чашку кофе в другую чашку, наполненную чаем, вам понадобится третья чашка, в которую можно перелить чай...



Ваша задача — создать консольное приложение .NET Core с классом Elephant, структура которого соответствует диаграмме класса. Поля и методы класса должны выводить показанный результат.

1 Создайте новое консольное приложение .NET Core и добавьте класс Elephant.

Добавьте в проект класс Elephant. Взгляните на диаграмму класса Elephant — вам понадобится поле int с именем EarSize и поле string с именем Name. Добавьте их и убедитесь в том, что оба поля объявлены открытыми (public). Затем добавьте метод с именем WhoAmI, который выводит на консоль две строки со значениями полей Name и EarSize. Чтобы понять, как именно должен выглядеть результат, просмотрите вывод примера.

2 Создайте два экземпляра Elephant и ссылку.

Воспользуйтесь инициализаторами объектов для создания двух объектов Elephant:

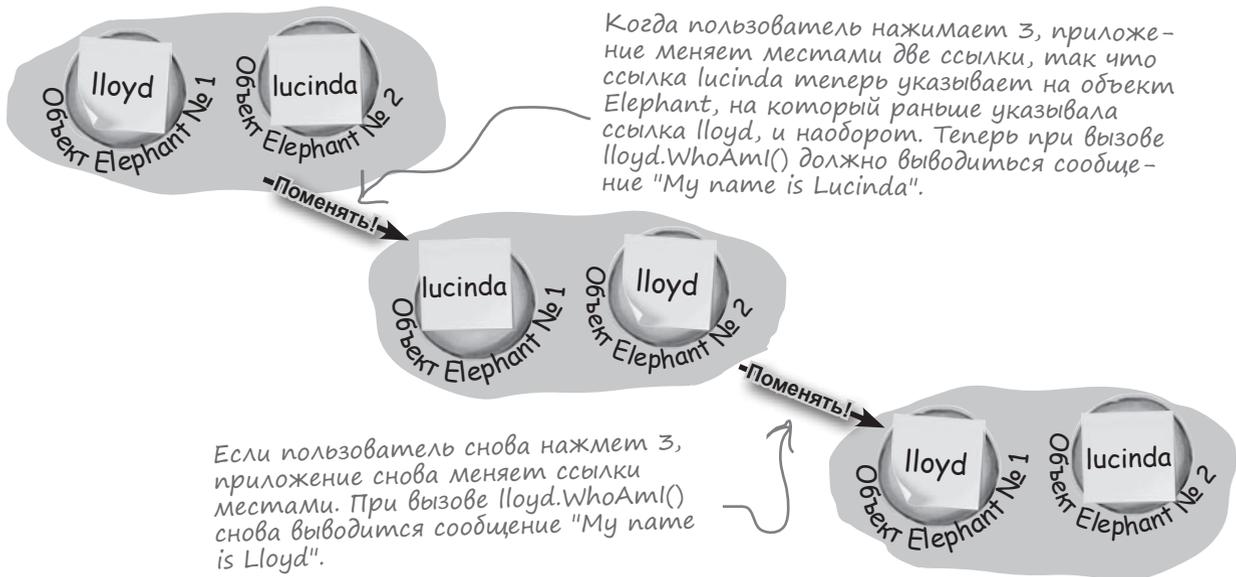
```
Elephant lucinda = new Elephant() { Name = "Lucinda", EarSize = 33 };
Elephant lloyd = new Elephant() { Name = "Lloyd", EarSize = 40 };
```

3 Вызовите их методы WhoAmI.

Когда пользователь нажимает 1, вызовите lloyd.WhoAmI. Когда пользователь нажимает 2, вызовите lucinda.WhoAmI. Убедитесь в том, что результат совпадает с приведенным в книге.

4 А теперь самое интересное: поменяйте местами ссылки.

Самая интересная часть упражнения: когда пользователь нажимает 3, приложение должно вызвать метод, который *меняет местами две ссылки*. Этот метод должны написать вы. После того как ссылки поменяются местами, при нажатии 1 на консоль должно выводиться сообщение объекта lucinda, а при нажатии 2 — сообщение объекта lloyd. При повторной перестановке ссылок все должно вернуться к нормальному состоянию.





Упражнение

Создайте программу с классом Elephant. Создайте два экземпляра Elephant, после чего поменяйте местами ссылки, которые указывают на них, — так, чтобы ни один из экземпляров не был уничтожен в ходе сборки мусора.

Класс Elephant:

```
class Elephant
{
    public int EarSize;
    public string Name;
    public void WhoAmI()
    {
        Console.WriteLine("My name is " + Name + ".");
        Console.WriteLine("My ears are " + EarSize + " inches tall.");
    }
}
```

Elephant
Name
EarSize
WhoAmI

А это метод Main класса Program:

```
static void Main(string[] args)
{
    Elephant lucinda = new Elephant() { Name = "Lucinda", EarSize = 33 };
    Elephant lloyd = new Elephant() { Name = "Lloyd", EarSize = 40 };

    Console.WriteLine("Press 1 for Lloyd, 2 for Lucinda, 3 to swap");
    while (true)
    {
        char input = Console.ReadKey(true).KeyChar;
        Console.WriteLine("You pressed " + input);
        if (input == '1')
        {
            Console.WriteLine("Calling lloyd.WhoAmI()");
            lloyd.WhoAmI();
        } else if (input == '2')
        {
            Console.WriteLine("Calling lucinda.WhoAmI()");
            lucinda.WhoAmI();
        } else if (input == '3')
        {
            Elephant holder;
            holder = lloyd;
            lloyd = lucinda;
            lucinda = holder;
            Console.WriteLine("References have been swapped");
        }
        else return;
        Console.WriteLine();
    }
}
```

Если просто присвоить lloyd ссылке lucinda, то в программе не останется ни одной ссылки на объект Lloyd и объект будет потерян. Вот почему необходима дополнительная переменная (мы назвали ее «holder») для хранения ссылки на объект Lloyd, пока эта ссылка не будет присвоена lucinda.

При объявлении переменной holder команда new не используется, потому что создавать дополнительный экземпляр Elephant не нужно.

Две ссылки — ДВЕ переменные, по которым можно изменять данные одного объекта

Помимо потери последней ссылки на объект, наличие нескольких ссылок может привести к непреднамеренному изменению объекта. Иначе говоря, одна ссылка на объект может *изменить* объект, тогда как другая ссылка на этот объект *понятия не имеет* об этих изменениях. Посмотрим, как это работает.

Добавьте еще один блок «else if» в метод Main. Сможете ли вы предположить, что произойдет при его выполнении?

```
else if (input == '3')
{
    Elephant holder;
    holder = lloyd;
    lloyd = lucinda;
    lucinda = holder;
    Console.WriteLine("References have been swapped");
}
else if (input == '4')
{
    lloyd = lucinda;
    lloyd.EarSize = 4321;
    lloyd.WhoAmI();
}
else
{
    return;
}
```

После этой команды переменные lloyd и lucinda указывают на ОДИН объект Elephant.

Эта команда присваивает EarSize значение 4321 у того объекта, на который указывает ссылка, хранящаяся в переменной lloyd.

А теперь запустите свою программу. Вот что вы увидите:

```
You pressed 4
My name is Lucinda
My ears are 4321 inches tall.

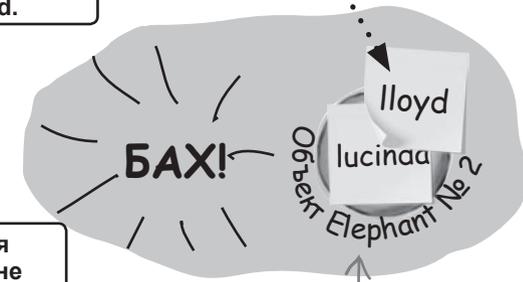
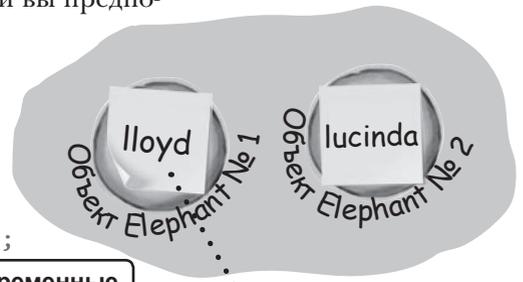
You pressed 1
Calling lloyd.WhoAmI()
My name is Lucinda
My ears are 4321 inches tall.

You pressed 2
Calling lucinda.WhoAmI()
My name is Lucinda
My ears are 4321 inches tall.
```

Программа ведет себя нормально... пока вы не нажмете 4. После этого при нажатии 1 или 2 будет выводиться один и тот же результат, а при нажатии 3 для перестановки ссылок ничего происходить не будет.

После нажатия 4 и выполнения нового кода переменные lloyd и lucinda содержат одну и ту же ссылку на второй объект Elephant. При нажатии 1 для вызова lloyd.WhoAmI выводится точно такое же сообщение, как при нажатии 2 для вызова lucinda.WhoAmI. Перестановка ни на что не влияет, потому что вы меняете местами две одинаковые ссылки.

Сделайте это!



Когда вы меняете местами эти две наклейки, ничего не изменится, потому что они прикреплены к одному объекту.

А поскольку ссылка lloyd уже не указывает на первый объект Elephant, она уничтожается сборщиком мусора... и вернуть ее уже не удастся!

Объекты используют ссылки для взаимодействия друг с другом

До сих пор формы взаимодействовали с объектами, используя ссылочные переменные для вызова своих методов и проверки их полей. Объекты также могут вызывать методы других объектов по ссылкам. Собственно, форма не может сделать ничего такого, чего бы не могли сделать ваши объекты, потому что **форма тоже является объектом**. Когда объекты взаимодействуют друг с другом, они могут использовать полезное ключевое слово `this`. Каждый раз, когда объект использует ключевое слово `this`, он ссылается на самого себя, т. е. эта ссылка указывает на тот объект, в котором она используется. Чтобы вы лучше поняли, что происходит, изменим класс `Elephant`, чтобы экземпляры могли вызывать методы друг друга.

Elephant
Name EarSize
WhoAml HearMessage SpeakTo

1 Добавьте в класс `Elephant` метод для прослушивания сообщений.

Добавим новый метод в класс `Elephant`. Первый параметр метода содержит сообщение от другого объекта `Elephant`. Во втором параметре передается объект `Elephant`, отправивший сообщение:

```
public void HearMessage(string message, Elephant whoSaidIt) {
    Console.WriteLine(Name + " heard a message");
    Console.WriteLine(whoSaidIt.Name + " said this: " + message);
}
```

(Делайте это!)

Вызов метода выглядит примерно так:

```
lloyd.HearMessage("Hi", lucinda);
```

Мы вызываем метод `HearMessage` по ссылке `lloyd` и передаем ему два параметра: строку `"Hi"` и ссылку на объект `Lucinda`. Метод использует параметр `whoSaidIt` для обращения к полю `Name` переданного объекта `Elephant`.

2 Добавьте в класс `Elephant` метод для отправки сообщения.

Теперь в класс `Elephant` необходимо включить метод `SpeakTo`. В этом методе используется специальное ключевое слово `this`. Эта ссылка позволяет объекту **получить ссылку на самого себя**.

```
public void SpeakTo(Elephant whoToTalkTo, string message) {
    whoToTalkTo.HearMessage(message, this);
}
```

Повнимательнее присмотримся к тому, что здесь происходит.

При вызове метода `SpeakTo` объекта `Lucinda`:

```
lucinda.SpeakTo(lloyd, "Hi, Lloyd!");
```

будет вызван метод `HearMessage` объекта `Lloyd`:

```
whoToTalkTo.HearMessage("Hi, Lloyd!", this);
```

Lucinda использует ссылку `whoToTalkTo` (которая сейчас содержит ссылку на `Lloyd`) для вызова `HearMessage`.

`this` заменяется ссылкой на объект `Lucinda`.

```
[ссылка на Lloyd].HearMessage("Hi, Lloyd!", [ссылка на Lucinda]);
```

Метод `SpeakTo` класса `Elephant` использует ключевое слово `<<this>>` для отправки ссылки на текущий объект другому объекту `Elephant`.

3 Вызовите новые методы.

Добавьте еще один блок `else if` в метод `Main`, чтобы объект `Lucinda` отправлял сообщение объекту `Lloyd`:

```
else if (input == '4')
{
    lloyd = lucinda;
    lloyd.EarSize = 4321;
    lloyd.WhoAmI();
}
else if (input == '5')
{
    lucinda.SpeakTo(lloyd, "Hi, Lloyd!");
}
else
{
    return;
}
```

При помощи
ключевого слова
this объект
получает ссылку
на самого себя.

Запустите программу и нажмите 5. Результат должен выглядеть так:

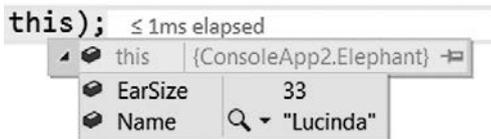
```
You pressed 5
Lloyd heard a message
Lucinda said this: Hi, Lloyd!
```

4 Используйте отладчик и разберитесь, что здесь происходит.

Установите точку прерывания в команде, которая только что была добавлена в метод `Main`:



1. Запустите программу и нажмите 5.
2. Когда выполнение достигает точки прерывания, используйте команду `Debug>>Step Into (F11)` для пошагового выполнения с заходом в метод `SpeakTo`.
3. Добавьте отслеживание для `Name`, чтобы увидеть, в каком объекте `Elephant` находится управление. В настоящее время оно находится в объекте `Lucinda`, что вполне логично, так как метод `Main` вызвал `Lucinda.SpeakTo`.
4. Наведите указатель мыши на ключевое слово **this** в конце строки и раскройте его. Оно содержит ссылку на объект `Lucinda`:



Наведите указатель мыши на переменную `whoToTalkTo` и раскройте ее — она содержит ссылку на объект `Lloyd`.

5. Метод `SpeakTo` содержит всего одну команду — вызов `whoTalkTo.HearMessage`. Зайдите в метод.
6. Управление должно находиться внутри метода `HearMessage`. Снова проверьте отслеживание — теперь поле `Name` содержит значение «Lloyd» — объект `Lucinda` вызвал метод `HearMessage` объекта `Lloyd`.
7. Наведите указатель мыши на переменную `whoSaidIt` и раскройте ее. Она содержит ссылку на объект `Lucinda`.

Завершите пошаговое выполнение кода. Поразмышляйте немного над тем, что здесь происходит.

Строки и массивы отличаются от других типов данных, встречавшихся в этой главе, потому что только они не имеют жестко ограниченного фиксированного размера (подумайте над этим).

Массивы содержат группы значений

Если вам приходится хранить большой объем однотипных данных, таких как списки цен или клички собак из некоторого набора, для этого можно воспользоваться **массивом**. Массив занимает особое место среди типов данных, потому что он содержит **группу переменных**, которая рассматривается как один объект. Массив предоставляет средства для хранения и изменения нескольких фрагментов данных без отслеживания каждой переменной по отдельности. При создании массив объявляется как обычная переменная, с именем и типом, — не считая того, что **за типом следуют квадратные скобки**.

```
bool[] myArray;
```

Для создания массива используется ключевое слово `new`. Создайте массив из 15 элементов `bool`:

```
myArray = new bool[15];
```

Чтобы задать значение отдельного элемента массива, используйте квадратные скобки. Следующая команда присваивает пятому элементу `myArray` значение `true`, для чего в квадратных скобках указывается **индекс 4**. Индекс соответствует пятому элементу, потому что первый элемент обозначается `myArray[0]`, второй — `myArray[1]` и т. д.:

```
myArray[4] = false;
```

Элементы массива используются как обычные переменные

Для создания массива используется ключевое слово `new`, потому что массив является объектом, — таким образом, переменная массива является разновидностью ссылочной переменной. В C# индексы массивов **начинаются с 0**, т. е. первому элементу всегда соответствует индекс 0.

Чтобы использовать массив, сначала необходимо **объявить ссылочную переменную**, которая указывает на массив. Затем следует **создать объект массива** командой `new` и указать желательный размер массива. После этого можно **задать элементы** массива. Ниже приведен пример кода, в котором объявляется и заполняется массив, и показано, что происходит в куче при его выполнении. Первому элементу массива соответствует **индекс 0**.

```
// Объявление нового массива decimal
// с 7 элементами
decimal[] prices = new decimal[7];
prices[0] = 12.37M;
prices[1] = 6_193.70M;

// Значение элемента с индексом 2
// не задано, он сохраняет
// значение по умолчанию 0
prices[3] = 1193.60M;
prices[4] = 58_000_000_000M;
prices[5] = 72.19M;
prices[6] = 74.8M;
```

Переменная `prices` является **ссылочной**, как и любая другая ссылка на объект. Объект, на который она указывает, представляет собой массив значений `decimal`, причем все эти значения хранятся в одном смежном блоке кучи.



Массивы могут содержать ссылочные переменные

Вы можете создать массив ссылок на объекты — по аналогии с тем, как вы создаете массив чисел или строк. Массив не интересуется, переменные какого типа в нем хранятся; это ваше дело. Таким образом, вы можете создать массив с элементами `int` или массив объектов `Duck` — ни малейших проблем не будет.

Следующий код создает массив из семи элементов `Dog`. Строка, инициализирующая массив, только создает ссылочные переменные. Так как программа содержит только две строки `new Dog()`, создаются только два экземпляра класса `Dog`.

```
// Объявление переменной для массива
// ссылок на объекты Dog
Dog[] dogs = new Dog[7];

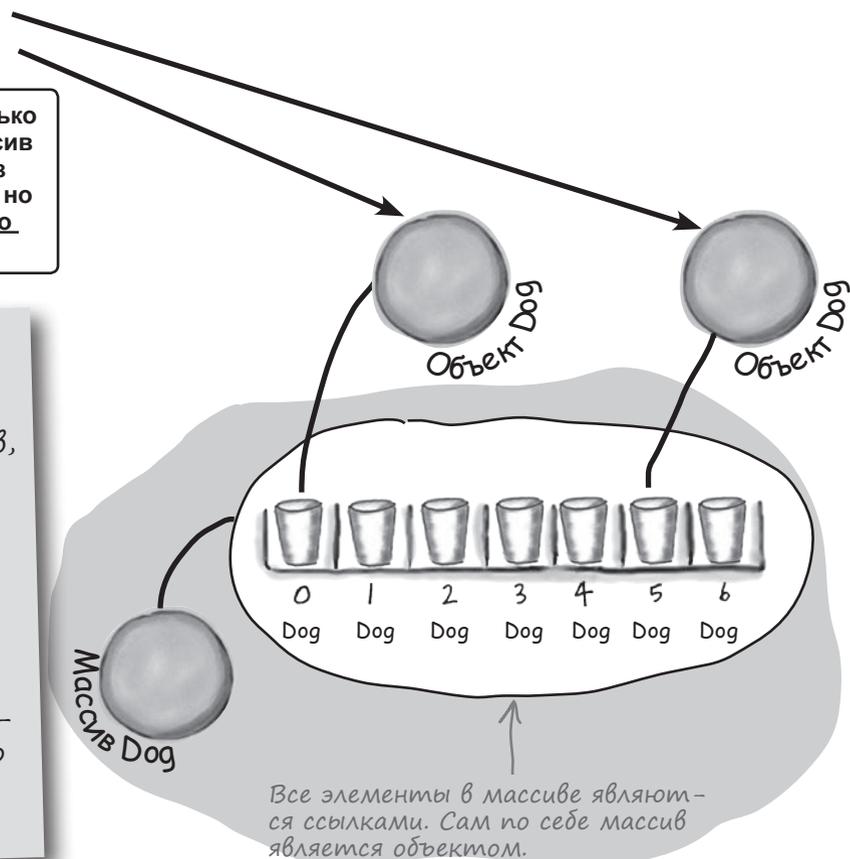
// Создаем два экземпляра Dog и помещаем
// их в элементы с индексами 0 и 5
dogs[5] = new Dog();
dogs[0] = new Dog();
```

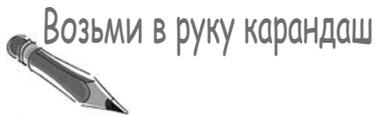
Первая строка кода создала только массив, но не экземпляры. Массив представляет собой список из семи ссылок на объекты `Dog` — но при этом были созданы только два объекта `Dog`.

Длина массива

Вы можете узнать, сколько элементов содержит массив, при помощи его свойства `Length`. Таким образом, если имеется массив с именем `prices`, то для получения его длины используется выражение `prices.Length`. Если массив содержит семь элементов, вы получите значение 7 — это означает, что элементы пронумерованы от 0 до 6.

Когда вы присваиваете или читаете элемент из массива, число в квадратных скобках называется индексом. Первому элементу массива соответствует индекс 0.





Перед вами массив объектов Elephant и цикл, который перебирает их и находит элемент с наибольшим значением EarSize. Каким будет значение biggestEars.EarSize **после** каждой итерации цикла for?

```
private static void Main(string[] args)
```

```
{
```

```
    Elephant[] elephants = new Elephant[7];
    elephants[0] = new Elephant() { Name = "Lloyd", EarSize = 40 };
    elephants[1] = new Elephant() { Name = "Lucinda", EarSize = 33 };
    elephants[2] = new Elephant() { Name = "Larry", EarSize = 42 };
    elephants[3] = new Elephant() { Name = "Lucille", EarSize = 32 };
    elephants[4] = new Elephant() { Name = "Lars", EarSize = 44 };
    elephants[5] = new Elephant() { Name = "Linda", EarSize = 37 };
    elephants[6] = new Elephant() { Name = "Humphrey", EarSize = 45 };
```

Создаем массив с семью ссылками на Elephant.

Массивы начинаются с индекса 0, так что первый объект Elephant в массиве обозначается elephants[0].

```
    Elephant biggestEars = elephants[0];
```

Итерация № 1 biggestEars.EarSize = _____

```
    for (int i = 1; i < elephants.Length; i++)
```

```
    {
```

```
        Console.WriteLine("Iteration #" + i);
```

Итерация № 2 biggestEars.EarSize = _____

```
        if (elephants[i].EarSize > biggestEars.EarSize)
```

```
        {
```

```
            biggestEars = elephants[i];
```

Итерация № 3 biggestEars.EarSize = _____

```
        }
```

Переменной biggestEars присваивается ссылка на объект, на который указывает elephants[i].

Итерация № 4 biggestEars.EarSize = _____

```
        Console.WriteLine(biggestEars.EarSize.ToString());
```

```
    }
```

Итерация № 5 biggestEars.EarSize = _____

```
}
```

Будьте внимательны — цикл начинается со второго элемента массива (с индексом 1) и выполняется шесть раз, пока «i» не достигнет длины массива.

Итерация № 6 biggestEars.EarSize = _____

null означает, что ссылка не указывает ни на что

При работе с объектами часто используется другое важное ключевое слово. Когда вы создаете новую ссылку, но ничего ей не присваиваете, у этой ссылки все равно есть значение. В исходном состоянии ссылка содержит `null`; это означает, что **она не указывает ни на какой объект**. Для начала рассмотрим к происходящему повнимательнее:

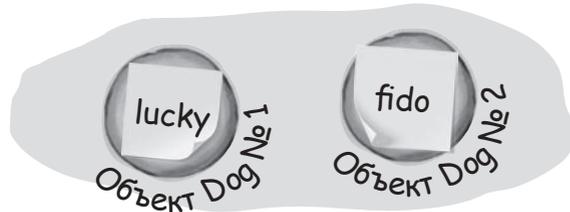
Значение по умолчанию для любой ссылочной переменной равно `null`. Так как `fido` еще не присвоено никакого значения, переменная содержит `null`.

```
Dog fido;
Dog lucky = new Dog();
```



Переменной `fido` присваивается ссылка на другой объект, так что она уже не равна `null`.

```
fido = new Dog();
```



Когда `lucky` присваивается `null`, переменная уже не указывает на свой объект, поэтому тот помечается для сборки мусора.

```
lucky = null;
```



А `null` действительно пригодится мне в программе?



Да. Ключевое слово `null` может быть очень полезным.

Существует несколько типичных вариантов использования `null` в программах. Чаще всего `null` используется при проверке того, что ссылка указывает на объект:

```
if (lloyd == null) {
```

Эта проверка вернет `true`, если ссылка `lloyd` содержит `null`.

Другой вариант использования ключевого слова `null` встречается тогда, когда вы *хотите*, чтобы объект был уничтожен уборщиком мусора. Если у вас есть ссылка на объект и вы завершили работу с объектом, присваивание ссылке `null` немедленно помечает его для сборки мусора (если только где-то не существует другая ссылка).



Возьми в руку карандаш

Решение

Перед вами массив объектов Elephant и цикл, который перебирает их и находит элемент с наибольшим значением EarSize. Каким будет значение biggestEars.EarSize **после** каждой итерации цикла for?

Цикл for начинается со второго объекта Elephant и сравнивает его с объектом Elephant, на который указывает biggestEars. Если у текущего объекта значение EarSize больше, то ссылка biggestEars переводится на этот объект Elephant. Далее цикл переходит к следующему элементу, затем к следующему... в итоге к концу цикла biggestEars будет указывать на объект Elephant с наибольшим значением EarSize.

```
private static void Main(string[] args)
```

```
{
```

```
    Elephant[] elephants = new Elephant[7];
```

```
    elephants[0] = new Elephant() { Name = "Lloyd", EarSize = 40 };
```

```
    elephants[1] = new Elephant() { Name = "Lucinda", EarSize = 33 };
```

```
    elephants[2] = new Elephant() { Name = "Larry", EarSize = 42 };
```

```
    elephants[3] = new Elephant() { Name = "Lucille", EarSize = 32 };
```

```
    elephants[4] = new Elephant() { Name = "Lars", EarSize = 44 };
```

```
    elephants[5] = new Elephant() { Name = "Linda", EarSize = 37 };
```

```
    elephants[6] = new Elephant() { Name = "Humphrey", EarSize = 45 };
```

Итерация № 1 biggestEars.EarSize = 40

```
    Elephant biggestEars = elephants[0];
```

```
    for (int i = 1; i < elephants.Length; i++)
```

```
    {
```

```
        Console.WriteLine("Iteration #" + i);
```

Итерация № 2 biggestEars.EarSize = 42

```
        if (elephants[i].EarSize > biggestEars.EarSize)
```

```
        {
```

```
            biggestEars = elephants[i];
```

Итерация № 3 biggestEars.EarSize = 42

```
        }
```

Итерация № 4 biggestEars.EarSize = 44

```
        Console.WriteLine(biggestEars.EarSize.ToString());
```

```
    }
```

Итерация № 5 biggestEars.EarSize = 44

```
}
```

Ссылка biggestEars указывает на объект Elephant с наибольшим значением EarSize, встретившимся до настоящего момента. Воспользуйтесь отладчиком для проверки! Установите точку прерывания и наблюдайте за состоянием biggestEars.EarSize.

Итерация № 6 biggestEars.EarSize = 45

А вы помните, что цикл начинается со второго элемента массива? Как вы думаете, почему?

Часть Задаваемые Вопросы

В: Я все еще не до конца понимаю, как работают ссылки.

О: Ссылки предоставляют механизм использования всех методов и полей объекта. Если вы создаете ссылку на объект Dog, то в дальнейшем вы сможете пользоваться этой ссылкой для обращения к любым методам, определенным для объекта Dog. Если класс Dog содержит (нестатические) методы с именами Bark и Fetch, вы можете создать ссылку с именем spot, а затем использовать ее для вызова методов spot.Bark() и spot.Fetch(). Также по ссылке можно изменять информацию, хранимую в полях объекта (таким образом, поле Breed можно изменить конструкцией spot.Breed).

В: Не означает ли это, что каждый раз, когда я изменяю значение по ссылке, оно также изменяется для всех остальных ссылок, указывающих на этот объект?

О: Да. Если переменная rover содержит ссылку на тот же объект, что и spot, то после изменения rover.Breed на "beagle" при обращении к spot.Breed также будет получено значение "beagle".

В: Напомните еще раз — что делает this?

О: this — специальная переменная, которая может использоваться только внутри объекта. Внутри класса this может использоваться для обращения к любому полю или методу этого конкретного экземпляра. Ключевое слово this особенно полезно при работе с классом, методы которого обращаются с вызовами к другим классам. Объект может использовать его для передачи ссылки на самого себя другому объекту. Таким образом, если spot вызывает один из методов rover и передает this в параметре, объект rover получает ссылку на объект spot.

В коде создаваемого вами объекта можно использовать специальную переменную this, которая содержит ссылку на этот объект.

В: Вы часто упоминаете сборку мусора, но кто на самом деле ею занимается?

О: Каждое приложение .NET выполняется внутри среды CLR (или Mono Runtime, если вы запускаете свои приложения в macOS, Linux или используете Mono в Windows). CLR делает достаточно много полезного, но есть две *особенно важные* операции, которые интересуют нас в данный момент. Во-первых, CLR выполняет ваш код, а конкретно вывод, генерируемый компилятором C#. Во-вторых, CLR управляет памятью, используемой программой. Это означает, что CLR отслеживает все объекты, определяет, когда последняя ссылка на объект исчезает, и освобождает память, занимаемую объектом. Команда .NET в Microsoft и команда Mono в Xamarin (много лет это была отдельная компания, но сейчас она является частью Microsoft) провели огромную работу, чтобы все работало быстро и эффективно.

В: Я не совсем понял, что там происходит с разными типами, в которых хранятся значения разного размера. Можете объяснить еще раз?

О: Конечно. Переменные связывают с вашим числом определенный размер независимо от того, насколько велико само значение. Таким образом, если у вас имеется переменная и ей назначен тип long, то даже для небольшого числа (допустим, 5) CLR зарезервирует достаточно памяти для хранения максимально возможного значения. И если задуматься, это очень удобно. В конце концов, переменные так называются именно потому, что они постоянно изменяются.

CLR считает, что вы знаете, что делаете, и не будете назначать переменной тип больше необходимого. Даже если число сейчас может быть большим, существует вероятность того, что после каких-то математических вычислений оно изменится. CLR выделяет память, достаточную для хранения самого большого значения этого типа.



Разработка игр... и не только

Настольные игры

У настольных игр богатая история — и как выясняется, настольные игры давно влияли на развитие видеоигр, по крайней мере со времен появления первых коммерческих ролевых игр.

- Первое издание «Dungeons & Dragons» (D&D) было выпущено в 1974 году. И с этого же года игры, в названиях которых встречались слова «dnd» и «dungeon», стали появляться на университетских мейнфреймах.
- Мы использовали класс `Random` для генерирования чисел. Идея использования случайных чисел в играх появилась очень давно — например, в настольных играх традиционно использовались кубики, карты и другие источники случайности.
- В предыдущей главе было показано, что бумажный прототип может стать важным первым шагом при проектировании видеоигры. Бумажные прототипы похожи на настольные игры. Собственно, бумажный прототип часто можно превратить в настольную игру и использовать ее для тестирования некоторых игровых механик.
- Настольные игры, особенно карточные и классические абстрактные, могут стать хорошим учебным пособием для понимания более общей концепции игровых механик. Раздача карт, тасование колоды, броски кубиков, правила перемещения фигур на поле, использование таймера (песочных часов), правила кооперативной игры — все это примеры механик.
- К механикам игры «Go Fish» относится раздача карт, требование карты у другого игрока, произнесение фразы «Go Fish» при отсутствии требуемой карты, определение победителя и т. д. Выделите минуту на чтение правил: https://askwiki.ru/wiki/Go_Fish#The_game.

Если вы никогда не играли в Go Fish, ознакомьтесь с правилами. Они будут использоваться позднее в этой книге!



Даже если вы не пишете код для видеоигр, из настольных игр можно многое узнать.

Многие программы зависят от **случайных чисел**. Например, мы уже использовали класс `Random` для создания случайных чисел в некоторых приложениях. У большинства читателей нет практического опыта использования случайных чисел... кроме игр. Бросание кубиков, тасование карт, подбрасывание монеты — все это отличные примеры **генераторов случайных чисел**. Класс `Random` выполняет функции генератора случайных чисел в .NET; он используется во многих наших программах, и ваш опыт применения случайных чисел в настольных играх поможет вам лучше понять, что он делает.

Тест-драйв со случайными числами



Класс .NET Random еще не раз встретится вам в этой книге. Чтобы лучше освоить его, просто необходимо провести тест-драйв: сесть за руль и сделать пару пробных кругов. Запустите Visual Studio и следуйте за нами — обязательно выполните свой код несколько раз, потому что вы будете каждый раз получать разные случайные числа.



- 1 **Создайте новое консольное приложение** — весь дальнейший код пойдет в метод Main. Начните с создания нового экземпляра Random, генерирования случайного значения int и вывода его на консоль:

```
Random random = new Random();
int randomInt = random.Next();
Console.WriteLine(randomInt);
```

Укажите максимальное значение для генерируемых чисел. Числа должны генерироваться в диапазоне от 0 до максимума (не включая). При максимуме 10 генерируются числа от 0 до 9:

```
int zeroToNine = random.Next(10);
Console.WriteLine(zeroToNine);
```

- 2 Теперь **смоделируйте бросание кубика**. При минимуме 1 и максимуме 7 будут генерироваться случайные числа от 1 до 6:

```
int dieRoll = random.Next(1, 7);
Console.WriteLine(dieRoll);
```

- 3 **Метод NextDouble** генерирует случайные значения double. Наведите указатель мыши на имя метода, чтобы на экране появилась подсказка, — метод генерирует числа с плавающей точкой от 0.0 до 1.0:

```
double randomDouble = random.NextDouble();
```

```
double Random.NextDouble()
Returns a random floating-point number that is greater than or equal to 0.0, and less than 1.0.
```

Чтобы генерировать случайные числа в более широком диапазоне, следует умножить double на соответствующее число. Например, если вам нужны случайные значения double от 1 до 100, умножьте случайное значение double на 100:

```
Console.WriteLine(randomDouble * 100);
```

Для преобразования случайных чисел double в другие типы используйте механизм **приведения типов**. Попробуйте выполнить этот код многократно — вы заметите незначительные различия в значениях float и double.

```
Console.WriteLine((float)randomDouble * 100F);
Console.WriteLine((decimal)randomDouble * 100M);
```

- 4 Используйте максимальное значение 2 для **моделирования подбрасывания монеты**. В результате будет генерироваться одно из двух случайных чисел: 0 или 1. Специальный класс Convert содержит статический метод ToBoolean, который преобразует такой результат в логическое значение:

```
int zeroOrOne = random.Next(2);
bool coinFlip = Convert.ToBoolean(zeroOrOne);
Console.WriteLine(coinFlip);
```



Как бы вы использовали класс Random для выбора случайной строки из массива строк?

Добро пожаловать в забегаловку эконом-класса «У неторопливого Джо»!

«У Неторопливого Джо» подают сэндвичи. У него есть мясо, гора хлеба и больше приправ, чем вы можете себе представить. Но вот меню у него нет! Сможете ли вы построить программу, которая генерирует *случайное* новое меню на каждый день? Да, вы определенно можете это сделать... при помощи **нового приложения WPF**, массивов и пары полезных приемов.

Сделайте это!

1 Добавьте в проект новый класс MenuItem с набором полей.

Взгляните на диаграмму класса. Он содержит четыре поля: экземпляр Random и три массива для хранения различных составляющих сэндвича. Поля-массивы используют **инициализаторы коллекций**: вы определяете элементы массива, заключая их в фигурные скобки.

```
class MenuItem
{
    public Random Randomizer = new Random();

    public string[] Proteins = { "Roast beef", "Salami", "Turkey",
        "Ham", "Pastrami", "Tofu" };
    public string[] Condiments = { "yellow mustard", "brown mustard",
        "honey mustard", "mayo", "relish", "french dressing" };
    public string[] Breads = { "rye", "white", "wheat", "pumpernickel", "a roll" };

    public string Description = "";
    public string Price;
}
```

MenuItem
Randomizer
Proteins
Condiments
Breads
Description
Price
Generate

2 Добавьте метод GenerateMenuItem в класс MenuItem.

Этот метод использует уже хорошо знакомый вам метод Random.Next для выбора случайных элементов из массивов в полях Proteins, Condiments и Breads и их конкатенации в строку:

```
public void Generate()
{
    string randomProtein = Proteins[Randomizer.Next(Proteins.Length)];
    string randomCondiment = Condiments[Randomizer.Next(Condiments.Length)];
    string randomBread = Breads[Randomizer.Next(Breads.Length)];
    Description = randomProtein + " with " + randomCondiment + " on " + randomBread;

    decimal bucks = Randomizer.Next(2, 5);
    decimal cents = Randomizer.Next(1, 98);
    decimal price = bucks + (cents * .01M);
    Price = price.ToString("c");
}
```

Этот метод вычисляет случайную цену в диапазоне от 2.01 до 5.97 преобразованием двух случайных переменных int в decimal. Внимательно присмотритесь к последней строке — она возвращает price.ToString("c"). Параметр метода ToString определяет формат. В данном случае формат "c" приказывает ToString отформатировать значение с локальной денежной единицей: в США будет выводиться знак \$, в Великобритании — £, в Европе — € и т. д.

Версия этого проекта для Mac доступна в приложении «Visual Studio для пользователей Mac».

3 Создайте XAML для формирования макета окна.

Наше приложение выводит случайные пункты меню в два столбца: широкий столбец предназначен для названия блюда, а узкий — для цены. С каждой ячейкой в сетке связан элемент `TextBlock`, у которого свойству `FontSize` задано значение `18px`, кроме последней строки, которая содержит один элемент `TextBlock` с выравниванием по правому краю, занимающий оба столбца. Заголовок окна имеет высоту `350` и ширину `550`. Сетка снабжена отступом размером `20`.

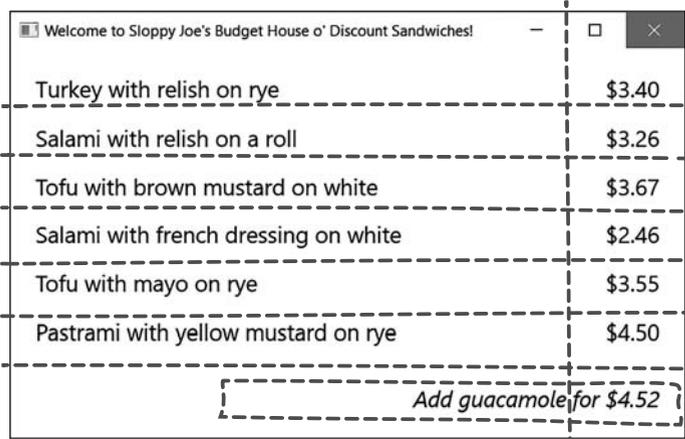
В этом примере мы расширим разметку XAML, которая была представлена в двух последних проектах WPF. Постройте макет в конструкторе, введите код вручную или совместите эти два способа.

Сетке назначаются отступы 20, чтобы вокруг меню было немного свободного места.

```
<Grid Margin="20">
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="5*" />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
```

Сетка состоит из 7 строк равной высоты

Сетка состоит из двух столбцов с шириной 5 и 1**



Нижний элемент TextBlock охватывает оба столбца

Присвойте элементам `TextBlock` в левом столбце имена `item1`, `item2` и т. д., а элементам `TextBlock` в правом столбце — имена `price1`, `price2` и т. д. Присвойте нижнему элементу `TextBlock` имя `guacamole`.

```
<TextBlock x:Name="item1" FontSize="18px" />
<TextBlock x:Name="price1" FontSize="18px" HorizontalAlignment="Right" Grid.Column="1"/>
<TextBlock x:Name="item2" FontSize="18px" Grid.Row="1"/>
<TextBlock x:Name="price2" FontSize="18px" HorizontalAlignment="Right"
  Grid.Row="1" Grid.Column="1"/>
<TextBlock x:Name="item3" FontSize="18px" Grid.Row="2" />
<TextBlock x:Name="price3" FontSize="18px" HorizontalAlignment="Right" Grid.Row="2"
  Grid.Column="1"/>
<TextBlock x:Name="item4" FontSize="18px" Grid.Row="3" />
<TextBlock x:Name="price4" FontSize="18px" HorizontalAlignment="Right" Grid.Row="3"
  Grid.Column="1"/>
<TextBlock x:Name="item5" FontSize="18px" Grid.Row="4" />
<TextBlock x:Name="price5" FontSize="18px" HorizontalAlignment="Right" Grid.Row="4"
  Grid.Column="1"/>
<TextBlock x:Name="item6" FontSize="18px" Grid.Row="5" />
<TextBlock x:Name="price6" FontSize="18px" HorizontalAlignment="Right" Grid.Row="5"
  Grid.Column="1"/>
<TextBlock x:Name="guacamole" FontSize="18px" FontStyle="Italic" Grid.Row="6"
  Grid.ColumnSpan="2" HorizontalAlignment="Right" VerticalAlignment="Bottom"/>
```

```
</Grid>
```

4 Добавьте код программной части для окна XAML.

Меню генерируется методом `MakeTheMenu`, который вызывается вашим окном сразу же после вызова `InitializeComponent`. Он использует массив классов `MenuItem` для генерирования каждого пункта меню. Первые три элемента должны быть нормальными, следующие два вида сэндвичей должны подаваться на нестандартных видах хлебной основы, а последний пункт является специальным блюдом с собственным набором ингредиентов.

```
public MainWindow()
{
    InitializeComponent();
    MakeTheMenu();
}

private void MakeTheMenu()
{
    MenuItem[] menuItem = new MenuItem[5];
    string guacamolePrice;

    for (int i = 0; i < 5; i++)
    {
        menuItem[i] = new MenuItem();
        if (i >= 3)
        {
            menuItem[i].Breads = new string[] {
                "plain bagel", "onion bagel", "pumpernickel bagel", "everything bagel"
            };
            menuItem[i].Generate();
        }

        item1.Text = menuItem[0].Description;
        price1.Text = menuItem[0].Price;
        item2.Text = menuItem[1].Description;
        price2.Text = menuItem[1].Price;
        item3.Text = menuItem[2].Description;
        price3.Text = menuItem[2].Price;
        item4.Text = menuItem[3].Description;
        price4.Text = menuItem[3].Price;
        item5.Text = menuItem[4].Description;
        price5.Text = menuItem[4].Price;

        MenuItem specialMenuItem = new MenuItem()
        {
            Proteins = new string[] { "Organic ham", "Mushroom patty", "Mortadella" },
            Breads = new string[] { "a gluten free roll", "a wrap", "pita" },
            Condiments = new string[] { "dijon mustard", "miso dressing", "au jus" }
        };
        specialMenuItem.Generate();

        item6.Text = specialMenuItem.Description;
        price6.Text = specialMenuItem.Price;

        MenuItem guacamoleMenuItem = new MenuItem();
        guacamoleMenuItem.Generate();
        guacamolePrice = guacamoleMenuItem.Price;

        guacamole.Text = "Add guacamole for " + guacamoleMenuItem.Price;
    }
}
```

Использует
«new string[]»
для объявления
типа инициа-
лизируемо-
го массива.
Включать это
уточнение в
поля MenuItem
не нужно, по-
тому что
у них уже
есть тип.

Внимательно разберитесь, что здесь происходит. Пункты меню 4 и 5 (индексы 3 и 4) получают объект `MenuItem`, инициализируемый с помощью инициализатора объекта по аналогии с тем, как это делалось в примере с Джо и Бобом. Инициализация объекта присваивает полю `Breads` новый массив строк. Этот массив строк использует инициализатор коллекции с четырьмя строками, описывающими разные типы хлебной основы. А вы заметили, что инициализатор коллекции включает тип массива (`new string[]`)? Вы не включали его при определении полей. При желании `new string[]` можно добавить к инициализаторам коллекций в полях `MenuItem`, но этого можно не делать. Эти определения не обязательны, потому что поля содержат определения типа в своих объявлениях.

Не забудьте вызвать метод `Generate`, в противном случае поля `MenuItem` останутся пустыми, и страница будет большей частью пустой.

Последний пункт меню предназначен для специального «сэндвича дня», приготовленного из ингредиентов класса «люкс», поэтому он получает собственный объект `MenuItem`, у которого все три поля строковых массивов инициализируются с помощью инициализаторов объектов.

Отдельный пункт меню, предназначенный для создания новой цены на гуакамоле.



Как это работает...

Метод `Randomizer.Next(7)` выдает случайное значение `int`, меньшее 7. `Breads.Length` возвращает количество элементов в массиве `Breads`. Таким образом, `Randomizer.Next(Breads.Length)` дает случайное число, большее или равное нулю, но меньшее количества элементов в массиве `Breads`.

`Breads[Randomizer.Next(Breads.Length)]`

`Breads` содержит массив строк. Массив содержит пять элементов с индексами от 0 до 4. Таким образом, значение `Breads[0]` равно "rye", а значение `Breads[3]` равно «a roll».

Я обедаю только «У Неторопливого Джо»!



Если ваш компьютер достаточно производительен, возможно, ваша программа не столкнется с этой проблемой. Но если запустить ее на более медленном компьютере, вы наверняка увидите ее.

5 Запустите программу и просмотрите сгенерированное меню.

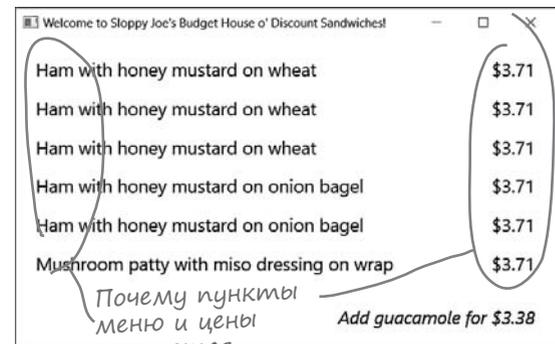
Э-э-э... Что-то не так. Все цены в новом меню одинаковы, и пункты меню выглядят странно — первые три позиции совпадают, потом следующие две, и начинаются они одинаково. Что происходит?

Оказывается, класс `.NET Random` на самом деле является генератором псевдослучайных чисел; это означает, что он по математической формуле генерирует последовательность чисел, удовлетворяющих некоторым статистическим критериям случайности. Такие числа достаточно хороши для любых приложений, которые строим мы с вами (только не используйте их в системах безопасности, зависящих от действительно случайных чисел!). Вот почему метод называется `Next` — он выдает следующее число в последовательности. Формула начинается со специального значения, которое называется «затравкой», — это значение используется для вычисления следующего значения в серии. Когда вы создаете новый экземпляр `Random`, системные часы используются для получения «затравки» формулы, но вы можете задать собственное значение. Попробуйте ввести в интерактивном окне `C#` вызов `new Random(12345).Next()`; несколько раз. Тем самым вы приказываете создать новый экземпляр `Random` с одним начальным значением (12345), поэтому метод `Next` будет каждый раз давать одно и то же «случайное» число.

Когда вы видите, что несколько разных экземпляров `Random` дают одно и то же значение, это объясняется тем, что они инициализировались практически в один момент времени, показания системных часов не изменились, поэтому все они использовали одно начальное значение. Как решить проблему? Используйте один экземпляр `Random` и объявите поле `Randomizer` статическим, чтобы все команды `MenuItem` совместно использовали один экземпляр `Random`:

```
public static Random Randomizer = new Random();
```

Снова запустите программу — на этот раз меню станет случайным.



КЛЮЧЕВЫЕ МОМЕНТЫ

- Ключевое слово `new` **возвращает ссылку на объект**, которую можно сохранить в ссылочной переменной.
- **На один объект могут указывать несколько ссылок.** Объект можно изменить по одной ссылке, а затем обратиться к результатам изменения по другой ссылке.
- Чтобы объект оставался в куче, на него должны существовать **ссылки**. Как только последняя ссылка на объект исчезает, объект будет уничтожен сборщиком мусора, а используемая им память будет освобождена.
- Программы .NET выполняются в среде **CLR** (Common Language Runtime) — «прослойке» между ОС и вашей программой. Компилятор C# переводит ваш код на язык CIL (Common Intermediate Language), который выполняется CLR.
- **Ключевое слово** `this` позволяет объекту получить ссылку на самого себя.
- **Массивы** представляют собой объекты, содержащие несколько значений. В массивах могут храниться как значения, так и ссылки.
- Чтобы объявить **переменную-массив**, поставьте квадратные скобки после типа в объявлении переменной (например, `bool[] trueFalseValues` или `Dog[] kennel`).
- Для создания нового массива используется ключевое слово `new`, с указанием длины массива в квадратных скобках (например, `new bool[15]` или `new Dog[3]`).
- Метод `Length` массива используется для получения его длины (например, `kennel.Length`).
- Чтобы обратиться к отдельному элементу массива, укажите его **индекс** в квадратных скобках (например, `bool[3]` или `Dog[0]`). Индексы массивов начинаются с 0.
- `null` означает ссылку, которая не указывает ни на какой объект. Ключевое слово `null` позволяет проверить, равна ли ссылка `null`, или очистить ссылочную переменную, чтобы объект был помечен для сборки мусора.
- Используйте **инициализаторы коллекций** для инициализации массивов: при присваивании ссылки на массив указывается ключевое слово `new`, за которым следует тип массива со списком значений, разделенных запятыми, в фигурных скобках (например, `new int[] { 8, 6, 7, 5, 3, 0, 9 }`). При инициализации переменной или поля в той команде, в которой они были объявлены, тип массива указывать не обязательно.
- Методу `ToString` объекта или значения можно передать **параметр с форматом**. Если вы вызываете метод `ToString` для числового типа, передача значения формата "c" форматирует значение с локальной денежной единицей.
- Класс .NET `Random` представляет генератор псевдослучайных чисел, инициализируемый показаниями системных часов. Используйте один экземпляр `Random` для того, чтобы несколько экземпляров генератора с одинаковым исходным значением не генерировали одинаковые последовательности чисел.

Лабораторный курс Unity № 2

Написание кода C# для Unity

Unity – не только мощный кроссплатформенный движок и редактор для построения 2D- и 3D-игр и моделирования. Также это **отличный способ потренироваться в написании кода C#**.

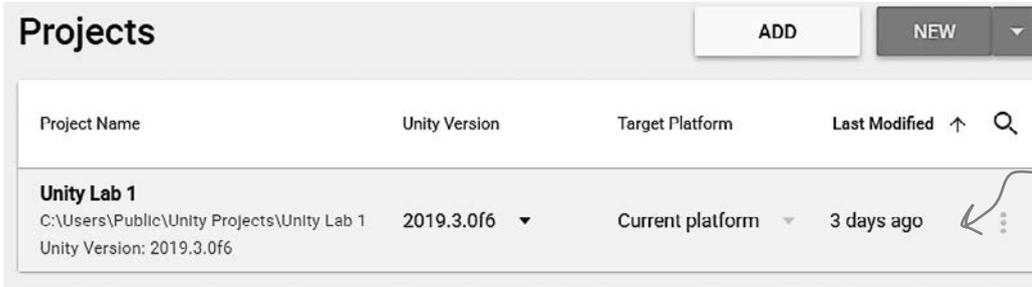
В предыдущей лабораторной работе вы научились ориентироваться в Unity и в трехмерном пространстве сцены, а также начали создавать и исследовать объекты `GameObject`. Пришло время написать код для управления объектами `GameObject`. В прошлой лабораторной работе мы ставили перед собой одну цель: научить вас ориентироваться в редакторе Unity (заодно эта глава позволит вам легко вспомнить материал, если вы что-то забудете).

В этой лабораторной работе мы начнем писать код для управления объектами `GameObject`. Мы напишем код C# для исследования концепций, которые будут использоваться в других лабораторных работах Unity, начиная с метода для вращения бильярдного шара, созданного в предыдущей лабораторной работе Unity. Также мы начнем пользоваться отладчиком Visual Studio с Unity для диагностики проблем в ваших играх.

Сценарии C# добавляют поведение к объектам GameObject

Итак, вы знаете, как добавить объекты GameObject в сцену. Теперь необходимо как-то заставить его... в общем, что-нибудь делать. Unity использует сценарии C# для определения всего поведения в игре.

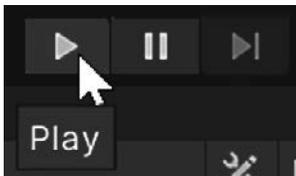
В этой лабораторной работе будут представлены инструменты, используемые при работе с C# и Unity. Мы построим простую «игру», которая на самом деле ограничивается чисто визуальным эффектом: бильярдный шар, летающий по сцене. Зайдите на Unity Hub и откройте проект, созданный в первой лабораторной работе Unity.



Эта лабораторная работа продолжается с того момента, на котором закончилась первая, поэтому зайдите на Unity Hub и откройте проект, созданный в предыдущей лабораторной работе.

В этой лабораторной работе будет сделано следующее:

- 1 Присоединение сценария C# к GameObject.** С объектом GameObject типа Sphere будет связан компонент Script. При его добавлении Unity создаст класс за вас. Мы изменим этот класс, чтобы он управлял поведением бильярдного шара.
- 2 Использование Visual Studio для редактирования сценария.** Помните, как мы включали использование Visual Studio в качестве редактора сценариев в настройках редактора Unity? Это означает, что по двойному щелчку на сценарии в редакторе Unity этот редактор откроется в Visual Studio.
- 3 Воспроизведение игры в Unity.** В верхней части экрана расположена кнопка Play. Нажатие этой кнопки запускает выполнение всех сценариев, связанных с объектами GameObject в сцене. Мы воспользуемся этой кнопкой для выполнения сценария, добавленного к сфере.



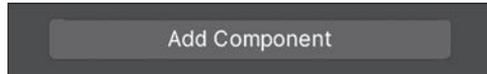
Кнопка Play не сохраняет вашу игру! Помните, что сохраняться нужно пораньше и почаще. У многих разработчиков вырабатывается привычка сохранять сцену при каждом запуске игры.

- 4 Совместное использование Unity и Visual Studio для отладки сценария.** Вы уже убедились в полезности отладчика Visual Studio, когда мы занимались поиском ошибок в коде C#. Unity идеально интегрируется с Visual Studio, так что вы можете расставлять точки прерывания, использовать окно Locals и пользоваться другими знакомыми средствами отладчика Visual Studio во время выполнения игры.

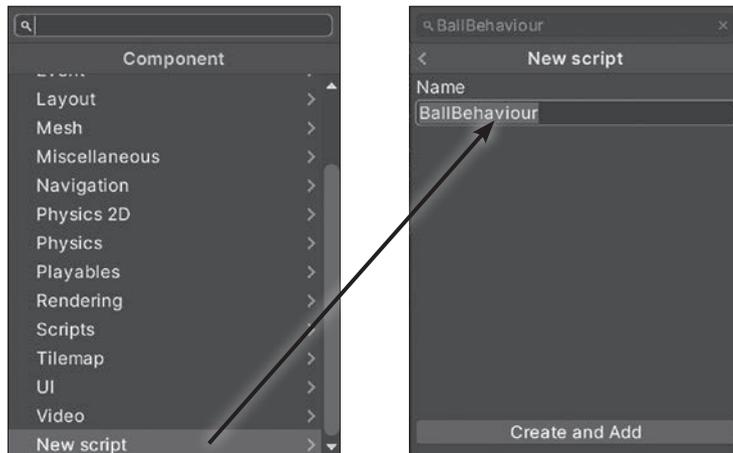
Добавление сценария C# к объекту GameObject

Unity – нечто большее, чем просто замечательная платформа для построения 2D- и 3D-игр. Многие разработчики используют Unity для художественной работы, визуализации данных, расширенной реальности и других целей. Платформа Unity особенно ценна для вас как начинающего разработчика C#, потому что вы можете написать код для управления всем, что вы видите в игре Unity. Все это делает Unity **замечательным инструментом для изучения и исследования C#**.

Переходим к использованию C# в Unity. Убедитесь в том, что в сцене выбран объект GameObject Sphere, и **щелкните на кнопке «Add Component»** в нижней части окна Inspector.



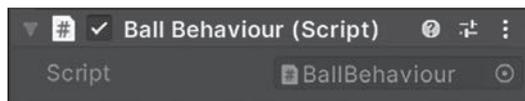
Unity открывает окно с перечнем различных видов компонентов, которые можно добавить к объекту, — и их действительно много. Выберите вариант **New script**, чтобы добавить новый сценарий C# к объекту GameObject Sphere. Вам будет предложено ввести имя. **Присвойте сценарию имя BallBehaviour**.



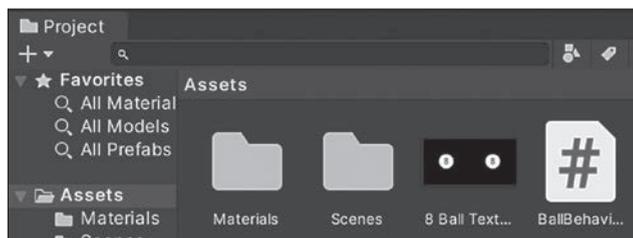
В окне Project отображается иерархическое представление проекта. Ваш проект Unity состоит из файлов: аудио/видео/графики, файлов данных, сценариев C#, текстур и т. д. В Unity эти файлы называются ресурсами. В тот момент, когда вы сделали щелчок правой кнопкой для импортирования текстуры, в окне Project отображалась папка Assets, поэтому Unity добавляет текстуру в эту папку.



Щелкните на кнопке «Create and Add», чтобы добавить сценарий. В окне Inspector появляется компонент с именем *Ball Behaviour (Script)*.



Сценарий C# также появляется в окне Project.



А вы заметили, что сразу же после перетаскивания текстуры бильярдного шара на сферу в окне Project появилась папка с именем Materials?

Написание кода C# для поворота сферы

В первой лабораторной работе мы приказали Unity использовать Visual Studio в качестве внешнего редактора сценариев. **Сделайте двойной щелчок на новом сценарии C#**. При этом *Unity открывает сценарий в Visual Studio*. Сценарий C# содержит класс с именем BallBehaviour, содержащий два пустых метода, Start и Update:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BallBehaviour : MonoBehaviour
{
    // Start вызывается перед первым обновлением кадра
    void Start()
    {

    }

    // Update вызывается один раз на кадр
    void Update()
    {

    }
}
```

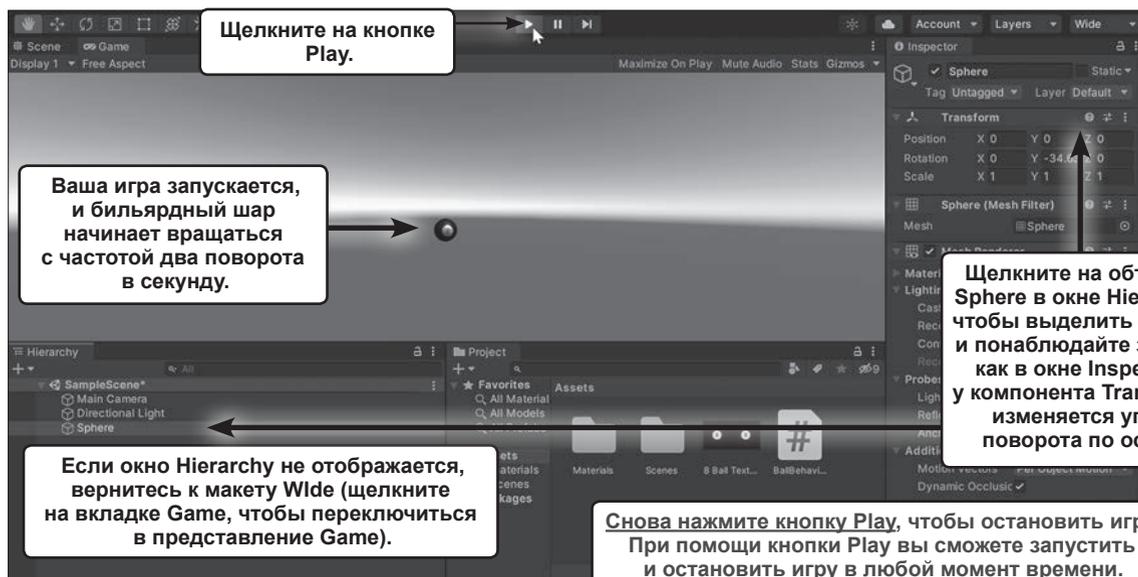
Вы открыли сценарий C# в Visual Studio, щелкнув на нем в окне Hierarchy. В этом окне выводится список всех объектов GameObject в текущей сцене. При создании вашего проекта Unity добавляет сцену SampleScene с камерой и источником света. Далее вы добавили сферу, поэтому в окне Hierarchy будут отображаться все эти объекты.

Если Unity не запустит Visual Studio для открытия сценария C#, вернитесь к началу лабораторной работы №1 и убедитесь в том, что вы выполнили действия по настройке предпочтений в категории External Tools.

А вот как выглядит строка кода для поворота сферы. **Добавьте ее** в метод Update:

```
transform.Rotate(Vector3.up, 180 * Time.deltaTime);
```

Теперь **вернитесь к редактору Unity** и щелкните на кнопке Play на панели инструментов, чтобы запустить игру. 





Код под увеличительным стеклом

```
using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;
```

О пространствах имен было рассказано в главе 2. При создании файла со сценарием C# Unity добавляет в начало файла строки using, чтобы иметь возможность использовать код из UnityEngine и других часто используемых пространств имен.

```
public class BallBehaviour : MonoBehaviour  
{  
    // Start вызывается перед первым обновлением кадра  
    void Start()  
    {  
    }  
}
```

Кадр — одна из фундаментальных концепций анимации. Unity выводит один статический кадр, после чего очень быстро выводит следующий, и человеческий глаз воспринимает изменения в кадрах как движение. Unity вызывает метод Update для каждого объекта GameObject до того, как тот сможет двигаться, вращаться или вносить другие необходимые изменения. На более мощном компьютере частота кадров (FPS) будет выше, чем на медленном.

```
// Update вызывается один раз на кадр  
void Update()  
{  
    transform.Rotate(Vector3.up, 180 * Time.deltaTime);  
}  
}
```

Метод transform.Rotate заставляет объект GameObject вращаться. В первом параметре передается ось, вокруг которой осуществляется поворот. В данном случае наш код использует значение Vector3.up, при котором поворот выполняется вокруг оси Y. Второй параметр задает величину поворота в градусах.

На разных компьютерах игра воспроизводится с разной частотой кадров. Если она работает с частотой 60 FPS, один поворот должен занимать 60 кадров. Если же игра работает с частотой 120 FPS, один поворот должен занимать 240 кадров. Частота кадров в вашей игре даже может динамически изменяться, если в игре выполняется более или менее сложный код.

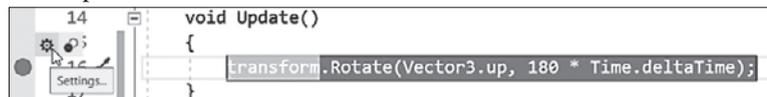
В такой ситуации пригодится значение Time.deltaTime. Каждый раз, когда ядро Unity вызывает метод GameObject Update (один раз на кадр), полю Time.deltaTime присваивается доля секунды, прошедшая с момента вывода последнего кадра. Так как мы хотим, чтобы бильярдный шар совершал полный поворот каждые 2 секунды (т. е. поворот на 180 градусов в секунду), для этого достаточно умножить 180 на Time.deltaTime, чтобы поворот был выполнен ровно на такой угол, который необходим для текущего кадра.

Внутри метода Update умножение любого значения на Time.deltaTime преобразует его в долю этого значения в секунду.

Поле Time.deltaTime является статическим, и как было показано в главе 3, для его использования экземпляра класса Time не нужен.

Добавление точки прерывания и отладка игры

Займемся отладкой игры Unity. Сначала остановите игру, если она все еще работает (повторным нажатием кнопки Play). Переключитесь на Visual Studio и **установите точку прерывания** в строке, добавленной в метод Update.



Теперь найдите в верхней части окна Visual Studio кнопку запуска отладчика:

- ★ В Windows эта кнопка выглядит так: — или выберите команду меню Debug>>Start Debugging (F5).
- ★ В macOS эта кнопка выглядит так: — или выберите команду Run>>Start Debugging (⌘↵).

Щелкните на этой кнопке, чтобы **запустить отладчик**. Снова переключитесь в редактор Unity. Если вы впервые отлаживаете этот проект, редактор Unity открывает диалоговое окно с тремя кнопками:



Выберите кнопку «Enable debugging for this section» (или, если вы хотите, чтобы это окно больше не появлялось, выберите «Enable debugging for all projects»). Visual Studio *присоединяется к Unity*; это означает, что отладчик Visual Studio теперь может использоваться для отладки вашей игры.

Нажмите кнопку Play в Unity, чтобы запустить игру. Так как среда Visual Studio присоединена к Unity, она *немедленно прерывает игру* в добавленной вами точке прерывания (как и в любой другой точке прерывания, которую бы вы могли установить).

← Поздравляем, вы занимаетесь отладкой игры!

Использование счетчика попаданий для пропуска кадров

Иногда бывает полезно дать вашей игре некоторое время поработать, прежде чем она может быть остановлена точкой прерывания. Допустим, вы хотите, чтобы ваша игра сгенерировала и переместила врагов, прежде чем в ней сможет сработать точка прерывания. Прикажем точке прерывания срабатывать через каждые 500 кадров. Для этого добавим к точке прерывания **условие счетчика попаданий**:

- ★ В Windows щелкните правой кнопкой мыши на маркере точки прерывания (●) слева от строки, выберите в контекстном меню **Conditions**, после чего выберите в раскрывающихся списках варианты *Hit Count* и *Is a multiple of*:



- ★ В macOS щелкните правой кнопкой мыши на маркере точки прерывания (⊙), выберите в меню команду **Edit breakpoint...**, после чего выберите в раскрывающемся списке вариант *When hit count is a multiple of* и введите 500 в соответствующем поле:

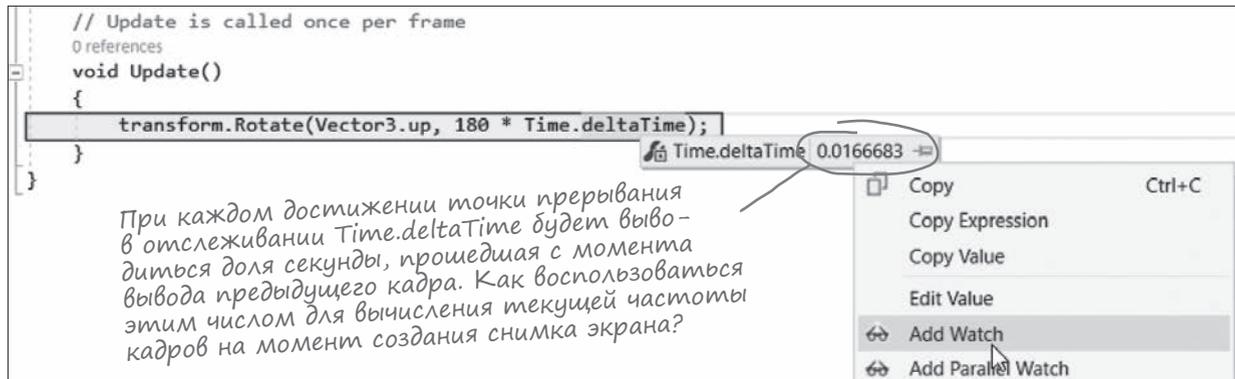


Теперь точка прерывания будет приостанавливать программу через 500 срабатываний метода Update, или каждые 500 кадров. Так как ваша игра работает с частотой кадров 60 FPS, это означает, что при нажатии Continue игра отработает чуть более 8 секунд перед повторным прерыванием. **Нажмите Continue, снова переключитесь в Unity** и наблюдайте за вращением шара перед срабатыванием точки прерывания.

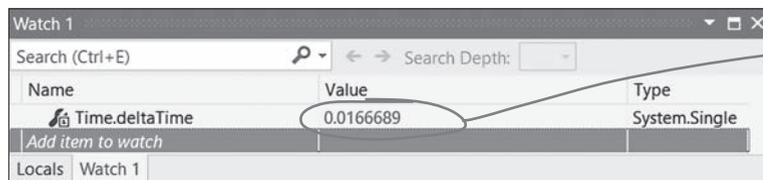
Использование отладчика для понимания Time.deltaTime

Поле Time.deltaTime будет использоваться во многих наших проектах Unity. Воспользуемся точкой прерывания и отладчиком и постараемся понять, что же именно происходит с этим значением.

Пока ваша игра приостановлена на точке прерывания в Visual Studio, **наведите указатель мыши на Time.deltaTime**, чтобы увидеть долю секунды, прошедшую с момента вывода предыдущего кадра. Затем **добавьте отслеживание для Time.deltaTime** – выберите Time.deltaTime и команду Add Watch из меню, открываемого щелчком правой кнопки мыши.



Продолжите отладку (F5 в Windows, ⌘↵ в macOS), чтобы возобновить игру. Шар снова начнет вращаться, и через 500 кадров точка прерывания сработает снова. Вы можете продолжить выполнение игры по 500 кадров. Обращайте внимание на окно Watch при каждом прерывании.



Нажмите кнопку Continue, чтобы получить новое значение Time.deltaTime, затем еще одно. Чтобы вычислить приблизительную частоту кадров, следует разделить 1 на Time.deltaTime.

Остановите отладку (Shift+F5 в Windows, ⌘↵ в macOS), чтобы остановить выполнение программы. Затем **снова запустите отладку**. Так как ваша игра продолжает работать, точка прерывания продолжит работать при повторном присоединении Visual Studio к Unity. Завершив отладку, снова отключите точку прерывания, чтобы она оставалась, но не вызывала прерывания при переходе. Еще раз **остановите отладку**, чтобы отключиться от Unity.

Вернитесь в Unity и **остановите игру** – и сохраните ее, потому что кнопка Play не сохраняет игру автоматически.

Кнопка Play в Unity запускает и останавливает игру. Среда Visual Studio остается присоединенной к Unity, несмотря на то что игра остановлена.



Снова включите отладку игры и наведите указатель мыши на «Vector3.up», чтобы просмотреть значение. Выводится значение (0.0, 1.0, 0.0). Как вы думаете, что оно означает?

Добавление цилиндра для обозначения оси Y

Ваша сфера вращается вокруг оси Y в самом центре сцены. Добавим очень высокий и очень узкий цилиндр, чтобы наглядно представить эту ось. **Создайте цилиндр** командой `3D Object >> Cylinder` из меню GameObject. Убедитесь в том, что цилиндр выбран в окне Hierarchy, затем обратитесь к окну Inspector и убедитесь в том, что он был создан в позиции (0, 0, 0), а если нет, воспользуйтесь контекстным меню (☰) и верните его в эту точку.

Сделаем цилиндр высоким и узким. Выберите инструмент Scale на панели инструментов; либо щелкните на нем (☞), либо нажмите клавишу R. У цилиндра появляется манипулятор Scale:



Манипулятор Scale очень похож на манипулятор Move, не считая того, что оси заканчиваются кубиками вместо конусов. Ваш цилиндр располагается на сфере — возможно, вы видите небольшой кусочек сферы, простирающийся из середины цилиндра. Когда вы уменьшите ширину цилиндра, изменяя его масштаб по осям X и Z, сфера снова будет видна.

Щелкните и перетащите зеленый куб, чтобы растянуть цилиндр по оси Y. Затем щелкните на красном кубе и перетащите его, чтобы сузить цилиндр по оси X; сделайте то же самое с синим кубом, чтобы сузить его по оси Z. Следите за панелью Transform в окне Inspector во время масштабирования цилиндра — масштаб по оси Y должен увеличиться, а масштабы по осям X и Z должны стать намного меньше.

Transform			
Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 0.2175312	Y 6.331524	Z 0.2783782

Щелкните на метке X в строке Scale на панели Transform, перетащите ее вверх-вниз. Проследите за тем, чтобы перетаскивалась надпись X слева от текстового поля с числом. Когда вы щелкаете на метке, она окрашивается в синий цвет, а вокруг значения X появляется синий прямоугольник. При перетаскивании указателя мыши вверх-вниз число в поле увеличивается и уменьшается, а представление Scene обновляется при изменении масштаба. Будьте внимательны при перетаскивании — масштаб может быть как положительным, так и отрицательным.

Теперь **выделите число в поле X и введите .1** — цилиндр становится очень узким. Нажмите клавишу Tab и введите 20, затем снова нажмите Tab, введите .1 и нажмите Enter.

Transform			
Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 0.1	Y 20	Z 0.1

Теперь через сферу проходит очень длинный цилиндр, обозначающий ось Y, где Y=0.



Добавление полей для угла поворота и скорости

В главе 3 вы узнали, что классы C# могут содержать **поля** для хранения данных, которые могут использоваться методами. Изменим наш код так, чтобы в нем использовались поля. Добавьте следующие четыре строки под объявлением класса, **сразу же за первой фигурной скобкой** {:

```
public class BallBehaviour : MonoBehaviour
{
    public float XRotation = 0;
    public float YRotation = 1;
    public float ZRotation = 0;
    public float DegreesPerSecond = 180;
}
```

Такие же поля, как те, которые добавлялись в проекты в главах 3 и 4. Фактически это переменные, в которых хранятся значения — при каждом вызове Update одно и то же поле используется снова и снова.

Каждое из полей XRotation, YRotation и ZRotation содержит значение от 0 до 1. Набор этих чисел создает **вектор**, определяющий направление поворота:

```
new Vector3(XRotation, YRotation, ZRotation)
```

Поле DegreesPerSecond содержит угол поворота в градусах в секунду, который следует умножить на Time.deltaTime, как и прежде. **Измените метод Update, чтобы в нем использовались поля.** Новый код создает переменную Vector3 с именем axis и передает ее методу transform.Rotate:

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.Rotate(axis, DegreesPerSecond * Time.deltaTime);
}
```

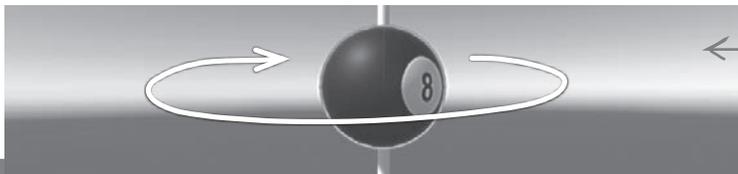
Выделите объект Sphere в окне Hierarchy. Теперь поля отображаются в компоненте Script. При отображении полей компонента Script после начальных прописных букв добавляются пробелы, чтобы они лучше читались.



Когда вы добавляете открытые поля в класс в сценарии Unity, компонент Script выводит текстовые элементы, в которых можно редактировать эти поля. Если вы измените их, пока игра не работает, обновленные значения будут сохранены со сценой. Также значения можно редактировать во время выполнения игры, но они вернутся к исходным значениям после остановки игры.

Снова запустите игру. **Пока она выполняется**, выделите объект Sphere в окне Hierarchy и измените значение поля Degrees per second на 360 или 90 — шар начинает вращаться вдвое быстрее (или вдвое медленнее). **Остановите игру** — поле возвращается к значению 180.

Когда игра остановится, в редакторе Unity введите в поле X Rotation значение 1, а в поле Y Rotation — значение 0. Запустите игру; шар будет вращаться в другом направлении. Щелкните на надписи X Rotation и перетащите указатель мыши вверх-вниз, чтобы изменить значение во время выполнения игры. Как только число станет отрицательным, шар начнет вращаться в обратном направлении. Вернитесь к положительному значению, и шар вернется к прежнему направлению вращения.



Если вы воспользуетесь редактором Unity, чтобы присвоить полю X Rotation значение 1, а затем запустите игру, шар будет вращаться по часовой стрелке по оси Y.

Debug.DrawRay и 3D-векторы

Вектор характеризуется **длиной** и **направлением**. Если вы уже сталкивались с векторами в учебном курсе геометрии, вероятно, вы видели диаграммы вроде следующей:

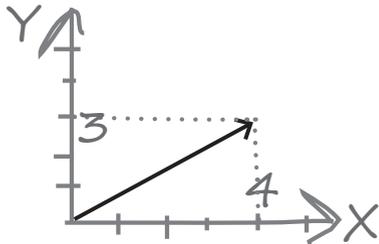


Диаграмма двумерного вектора. Вектор представляется двумя числами: координатой конечной точки по оси X (4) и по оси Y (3). Обычно эти числа записываются в виде (4, 3).

Но даже те из нас, кто проходил векторы на занятиях, не всегда *интуитивно* понимают, как они работают, особенно в трехмерном пространстве. Это еще одна область, в которой C# и Unity могут использоваться для обучения и аналитики.

Использование Unity для наглядного представления векторов в трехмерном пространстве

Мы добавим в игру код, который поможет вам разобраться в том, как работают 3D-векторы. Для начала взгляните на первую строку метода Update:

```
Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
```

Что эта строка сообщает о векторе?

- ★ У него есть тип: **Vector3**. Каждое объявление переменной начинается с типа. Вместо string, int или bool вектор объявляется с типом Vector3. Этот тип используется в Unity для 3D-векторов.
- ★ У него есть имя переменной: **axis**.
- ★ Для создания **Vector3** используется ключевое слово **new**. При этом поля XRotation, YRotation и ZRotation используются для создания векторов с указанными параметрами.

Как же выглядит 3D-вектор? Не будем гадать — проще воспользоваться одним из полезных отладочных инструментов Unity для рисования вектора. **Добавьте новую строку кода в конец метода Update:**

```
void Update()  
{  
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);  
    transform.Rotate(axis, DegreesPerSecond * Time.deltaTime);  
    Debug.DrawRay(Vector3.zero, axis, Color.yellow);  
}
```

Debug.DrawRay — специальный метод, предоставляемый Unity для отладки игр. Метод рисует **луч** — вектор, проходящий от одной точки к другой; в параметрах он получает начальную точку, конечную точку и цвет. Но есть одна загвоздка: луч появляется только в представлении Scene. Методы класса Debug в Unity были спроектированы так, чтобы они не препятствовали выполнению игры. Как правило, они влияют только на взаимодействие вашей игры с редактором Unity.

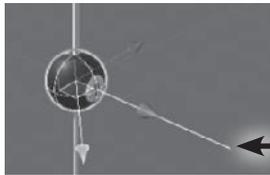
Запуск игры для отображения луча в представлении Scene

Теперь снова запустите игру. В представлении Game вы не увидите ничего нового, потому что Debug.DrawRay — отладочный инструмент, никак не влияющий на игровой процесс. Используйте вкладку Scene, чтобы переключиться в представление Scene. Возможно, вам также придется переключиться на макет Wide, выбрав вариант Wide в раскрывающемся списке Layout.



Итак, вы снова вернулись к знакомому представлению Scene. Чтобы получить более полное представление о том, как работают 3D-векторы, выполните следующие действия:

- ★ Используйте окно Inspector для изменения полей сценария BallBehaviour. Введите в поле X Rotation значение 0, в поле Y Rotation — значение 0 и в поле **Z Rotation** — значение 3. В сцене должен появиться желтый луч, проходящий прямо по оси Z, а шар должен вращаться вокруг него (помните: луч виден только в представлении Scene).



Вектор (0, 0, 3) распространяется на 3 единицы по оси Z. Присмотритесь внимательнее к сетке в редакторе Unity — длина вектора составляет ровно 3 единицы. Попробуйте щелкнуть на надписи Z Rotation и перетаскивать указатель мыши вверх-вниз в окне Inspector. При перетаскивании луч будет увеличиваться или уменьшаться. Если значение Z вектора становится отрицательным, шар начинает вращаться в другом направлении.

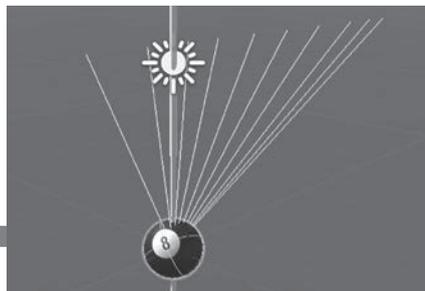
- ★ Верните в поле Z Rotation значение 3. Поэкспериментируйте с перетаскиванием значений X Rotation и Y Rotation и посмотрите, что при этом происходит с лучом. Не забывайте сбрасывать компонент Transform каждый раз, когда вы их изменяете.
- ★ Воспользуйтесь инструментом Hand и манипулятором Scene, чтобы получить более удобное представление. Щелкните на конусе X манипулятора Scene, чтобы выбрать вид справа. Продолжайте щелкать на конусах манипулятора Scene, пока не получите вид спереди. При этом легко запутаться — в таком случае сбросьте макет **Wide**, чтобы вернуться к знакомому представлению.

Добавление продолжительности вывода луча

При вызове метода Debug.DrawRay можно добавить четвертый аргумент со значением времени, в течение которого луч должен оставаться на экране. Добавьте аргумент `.5f`, чтобы каждый луч оставался на экране полсекунды:

```
Debug.DrawRay(Vector3.zero, axis, Color.yellow, .5f);
```

Снова запустите игру и переключитесь в представление Scene. Теперь при перетаскивании чисел вверх-вниз на экране будет оставаться след из лучей. Эффект выглядит весьма интересно, но что еще важнее, он является хорошим средством визуализации 3D-векторов.



След, остающийся за лучом, может создать интуитивное представление о том, как работают 3D-векторы.

Поворот шара вокруг точки сцены

Наш код вызывает метод `transform.Rotate`, чтобы шар вращался относительно его центра. **Выделите объект Sphere в окне Hierarchy и введите значение 5 в поле X Position** компонента Transform. Затем воспользуйтесь контекстным меню (☰) компонента Script BallBehaviour для сброса полей. Снова запустите игру — теперь шар будет находиться в позиции (5, 0, 0) и будет вращаться относительно своей оси Y.



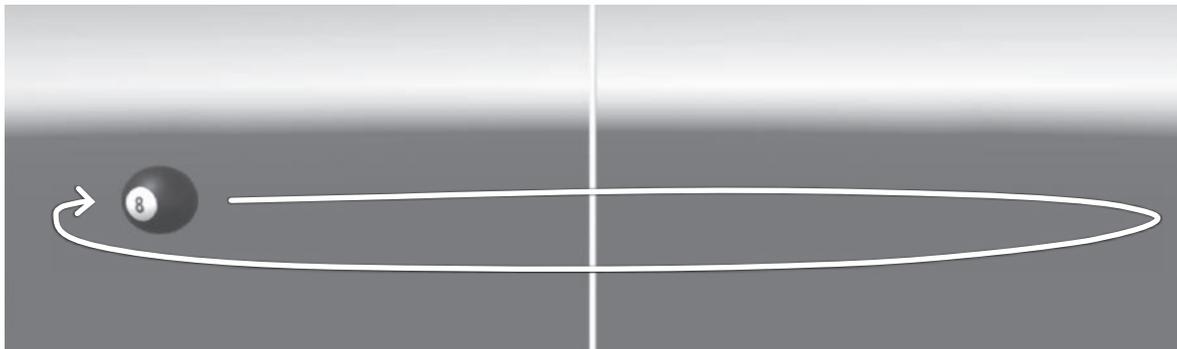
Изменим метод Update, чтобы в нем использовалась другая разновидность вращения. Заставим шар вращаться относительно центральной точки сцены — точки с координатами (0, 0, 0), — используя **метод transform.RotateAround**, который поворачивает объект GameObject относительно заданной точки сцены. (Он отличается от метода `transform.Rotate`, который использовался ранее и поворачивал объект GameObject вокруг своего центра.) В первом параметре передается точка, относительно которой осуществляется вращение. Мы будем передавать в этом параметре `Vector3.zero`, который является сокращением для записи `new Vector3(0, 0, 0)`.

Новый метод Update выглядит так:

```
void Update()  
{  
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);  
    transform.RotateAround(Vector3.zero, axis, DegreesPerSecond * Time.deltaTime);  
    Debug.DrawRay(Vector3.zero, axis, Color.yellow, .5f);  
}
```

↩ Новый метод Update поворачивает шар вокруг точки (0, 0, 0) сцены.

Теперь запустите свой код. В новой версии шар описывает большие круги относительно центральной точки:

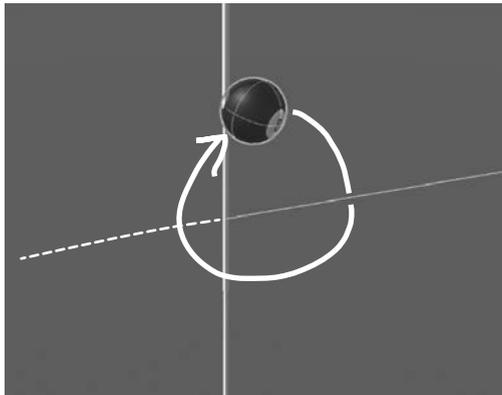


Эксперименты с поворотами и векторами в Unity

Мы будем работать с 3D-объектами и сценами в остальных лабораторных работах Unity в этой книге. Даже те из нас, кто проводит много времени за 3D-играми, не имеют абсолютно четкого представления о том, как работают векторы и 3D-объекты и как выполняются перемещения и повороты в трехмерном пространстве. К счастью, Unity является отличным инструментом для **исследования того, как работают 3D-объекты**. Давайте немного поэкспериментируем.

Пока ваш код работает, попробуйте изменить параметры, чтобы поэкспериментировать с поворотами:

- ★ **Вернитесь к представлению Scene**, чтобы на экране появился желтый луч, который отображается методом `Debug.DrawRay` в методе `BallBehaviour.Update`.
- ★ **Выделите объект Sphere** в окне Hierarchy. Перечень компонентов должен появиться в окне Inspector.
- ★ Введите значение 10 в полях **X Rotation**, **Y Rotation** и **Z Rotation** компонента Script, чтобы вектор отображался в виде длинного луча. Используйте инструмент Hand(Q) для поворота представления Scene, пока луч не будет хорошо виден.
- ★ Используйте контекстное меню компонента Transform (☰) для сброса компонента **Transform**. Так как центр сферы находится в начале координат сцены (0, 0, 0), сфера будет поворачиваться относительно своего центра.
- ★ Введите в поле **X Position** компонента Transform значение 2. Шар должен вращаться относительно вектора. Вы увидите, что шар, пролетающий рядом с цилиндром оси Y, отбрасывает на него тень.



← Пока игра работает, введите в полях X, Y и Z Rotation компонента Script BallBehaviour значение 10, сбросьте состояние компонента Transform сферы и введите в его поле X Position значение 2 — как только это будет сделано, сфера начнет вращаться вокруг луча.

Попробуйте **повторить три последних шага** для разных значений X, Y и Z, каждый раз сбрасывая компонент Transform, чтобы начать с фиксированной точки. Затем попробуйте пощелкать на надписях полей Rotation и перетащить их вверх-вниз — это поможет вам лучше понять, как работают повороты.

Unity — отличный инструмент для исследования работы 3D-объектов, позволяющий изменять свойства объектов GameObject в реальном времени.

Проявите фантазию!

Ваша очередь для **самостоятельных экспериментов с C# и Unity**. Вы уже видели, как C# интегрируется с объектами GameObject в Unity. Выделите немного времени и поэкспериментируйте с различными инструментами и методами Unity, о которых вы узнали в первых двух лабораторных работах. Приведем несколько идей:

- ★ Добавьте в сцену кубы, цилиндры или капсулы. Присоедините к ним новые сценарии (проследите за тем, чтобы каждому сценарию было присвоено уникальное имя!) и заставьте их вращаться в разных направлениях.
- ★ Попробуйте разместить вращающиеся объекты GameObject в разных позициях сцены. Удастся ли вам создать интересные визуальные эффекты из нескольких вращающихся объектов GameObject?
- ★ Добавьте к сцене источник света. Что произойдет, если использовать метод `transform.RotateAround` для поворота нового источника света по различным осям?
- ★ Небольшая задача из области программирования: попробуйте использовать оператор `+=` для добавления значения к одному из полей сценария `BallBehaviour`. Не забудьте умножить значение на `Time.deltaTime`. Попробуйте добавить команду `if`, которая сбрасывает поле до 0, если его значение стало слишком большим.

↑
Прежде чем запускать код, определите, что он сделает. Работает ли код так, как положено? Прогнозирование поведения нового кода — отличный прием для повышения квалификации C#.

Поэкспериментируйте с только что изученными приемами и инструментами. Это отличный способ использования Unity и Visual Studio в целях обучения и исследования.

КЛЮЧЕВЫЕ МОМЕНТЫ

- **Манипулятор Scene** всегда отображает ориентацию камеры.
- К любому объекту GameObject можно присоединить любой объект GameObject. Метод Update сценария вызывается один раз для каждого кадра.
- Метод **transform.Rotate** заставляет объект GameObject повернуться на заданный угол в градусах по заданной оси.
- В методе Update умножение любого значения на **Time.deltaTime** преобразует это значение в расчете на секунду.
- Отладчик Visual Studio можно **присоединить** к Unity, чтобы проводить отладку игры во время ее выполнения. Отладчик остается присоединенным к Unity даже в том случае, если игра не выполняется.
- При добавлении **условия счетчика попаданий** точка прерывания будет срабатывать после выполнения соответствующей команды некоторое количество раз.
- **Поле** представляет собой переменную, которая существует внутри класса за пределами его методов. Значения полей сохраняются между вызовами метода.
- Если вы включите открытые поля в класс в сценарии Unity, компонент Script отображает **текстовые элементы для изменения этих полей**. Между прописными буквами вставляются пробелы, чтобы имена лучше читались.
- 3D-векторы могут создаваться вызовом `new Vector3`. (Ключевое слово `new` рассматривалось в главе 3.)
- Метод **Debug.DrawRay** рисует вектор в представлении Scene (но не в представлении Game). Векторы могут использоваться не только для отладки, но и как учебный инструмент.
- Метод **transform.RotateAround** поворачивает объект GameObject вокруг заданной точки сцены.

Умейте хранить секреты



Вам когда-нибудь хотелось, чтобы посторонние не лезли в ваши личные дела? Вот и вашим объектам этого иногда хочется. И если вы не желаете, чтобы чужие люди читали ваш дневник или просматривали банковские выписки, хорошие объекты не позволяют другим объектам копаться в их полях. В этой главе вы узнаете о мощи инкапсуляции — приеме программирования, который делает ваш код более гибким. Такой код трудно использовать некорректно. Данные вашего объекта объявляются приватными, и к ним добавляются свойства, защищающие обращения к этим данным.

Поможем Оуэну реализовать броски на повреждения

Оуэну настолько понравился калькулятор характеристик, что он захотел создать другие программы C#, которые он мог бы использовать в своих играх. В игре, которую он ведет в настоящее время, при любой атаке мечом бросаются кубики и по формуле вычисляются повреждения от атаки. Оуэн записал **формулу вычисления повреждений** от удара мечом в своем блокноте гейм-мастера.

Ниже приведен класс **SwordDamage**, который выполняет вычисления. Внимательно прочитайте код — вам предстоит написать приложение, в котором он будет использоваться.

* ЧТОБЫ ОПРЕДЕЛИТЬ КОЛИЧЕСТВО ПОВРЕЖДЕНИЙ (НР) ОТ АТАКИ МЕЧОМ, БРОСЬТЕ 3D6 (ТРИ ШЕСТИГРАННЫХ КУБИКА) И ДОБАВЬТЕ «БАЗОВЫЕ ПОВРЕЖДЕНИЯ» 3 НР.

* ЕСЛИ ПРИ АТАКЕ ИСПОЛЬЗУЕТСЯ ОГНЕННЫЙ МЕЧ, АТАКА ПРИЧИНЯЕТ ДОПОЛНИТЕЛЬНЫЕ 2 НР ПОВРЕЖДЕНИЯ.

* НА НЕКОТОРЫЕ МЕЧИ НАЛОЖЕНЫ МАГИЧЕСКИЕ ЗАКЛЯТИЯ. ДЛЯ ВОЛШЕБНЫХ МЕЧЕЙ БРОСОК 3D6 УМНОЖАЕТСЯ НА 1.75 И ОКРУГЛЯЕТСЯ В НИЖНЮЮ СТОРОНУ, ПОСЛЕ ЧЕГО К НЕМУ ПРИБАВЛЯЮТСЯ БАЗОВЫЕ ПОВРЕЖДЕНИЯ И ПОВРЕЖДЕНИЯ ОТ ОГНЯ.

Описание формулы вычисления повреждений из блокнота гейм-мастера Оуэна.

```
class SwordDamage
{
    public const int BASE_DAMAGE = 3;
    public const int FLAME_DAMAGE = 2;

    public int Roll;
    public decimal MagicMultiplier = 1M;
    public int FlamingDamage = 0;
    public int Damage;

    public void CalculateDamage()
    {
        Damage = (int)(Roll * MagicMultiplier) + BASE_DAMAGE + FlamingDamage;
    }

    public void SetMagic(bool isMagic)
    {
        if (isMagic)
        {
            MagicMultiplier = 1.75M;
        }
        else
        {
            MagicMultiplier = 1M;
        }
        CalculateDamage();
    }

    public void SetFlaming(bool isFlaming)
    {
        CalculateDamage();
        if (isFlaming)
        {
            Damage += FLAME_DAMAGE;
        }
    }
}
```

Еще одна полезная возможность C#. Так как базовые повреждения и повреждения от огня в программе не будут изменяться, их можно объявить как константы с ключевым словом **const**. Константы в целом похожи на переменные, но их значение никогда не изменяется. Если вы напишете код, который пытается изменить константу, компилятор выдаст сообщение об ошибке.

Формула для вычисления повреждений. Выделите немного времени и прочитайте код, чтобы понять, как он реализует эту формулу.

Теперь я могу проводить меньше времени за вычислениями и больше времени посвящать тому, чтобы сделать игру более интересной для игроков.

Так как огненные мечи наносят еще больше повреждений в дополнение к броску, метод **SetFlaming** вычисляет повреждения и прибавляет к ним значение **FLAME_DAMAGE**.





Создание консольного приложения для вычисления повреждений

Построим консольное приложение, которым Оуэн будет пользоваться для работы с классом `SwordDamage`. Приложение выводит на консоль сообщение, которое предлагает указать, является ли меч волшебным и/или огненным, а затем выполняет вычисления. Пример вывода приложения:

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 0
Rolled 11 for 14 HP
```

↖ Бросок 11 для неволшебного и неогненного меча наносит повреждения в размере $11+3=14$ HP.

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 0
Rolled 15 for 18 HP
```

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 1
Rolled 11 for 22 HP
```

↖ Бросок 11 для волшебного меча наносит повреждения в размере $(11 \times 1.75 = 19) + 3 = 22$ HP.

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 1
Rolled 8 for 17 HP
```

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 2
Rolled 10 for 15 HP
```

↖ Бросок 17 для волшебного огненного меча наносит повреждения в размере $(17 \times 1.75=29) + 3 + 2 = 34$ HP.

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 3
Rolled 17 for 34 HP
```

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: q
Press any key to continue...
```



Упражнение

Нарисуйте диаграмму класса `SwordDamage`. Затем создайте **новое консольное приложение** и добавьте класс `SwordDamage`. В процессе ввода внимательно проследите за тем, как работают методы `SetMagic` и `SetFlaming` и чем они отличаются друг от друга. Когда вы будете уверены в том, что понимаете их логику, можно переходить к построению метода `Main`. Он будет действовать по следующей схеме:

1. Создать новый экземпляр класса `SwordDiagram`, а также новый экземпляр `Random`.
2. Вывести на консоль запрос и определить нажатие клавиши. Вызвать метод `Console.ReadKey(false)`, чтобы введенная пользователем клавиша была выведена на консоль. Если нажатая клавиша отлична от 0, 1, 2 или 3, выполнить команду `return` для завершения программы.
3. Смоделировать бросок 3d6. Для этого вызвать `random.Next(1, 7)` три раза, сложить результаты и присвоить значение полю `Roll`.
4. Если пользователь нажал 1 или 3, вызвать `SetMagic(true)`; в противном случае вызвать `SetMagic(false)`. Команда `if` для этого не нужна: `key == '1'` возвращает `true`, так что вы можете использовать `||` для проверки нажатой клавиши прямо внутри аргумента.
5. Если пользователь нажал 2 или 3, вызвать `SetFlaming(true)`; в противном случае вызвать `SetMagic(false)`. И снова это можно сделать в одной команде с использованием `==` и `||`.
6. Вывести результаты на консоль. Внимательно просмотрите результат и используйте `\n` для вставки разрывов строк там, где они нужны.



Упражнение Решение

Консольное приложение создает новый экземпляр класса `SwordDamage`, который мы предоставили (и экземпляр `Random` для моделирования бросков 3d6), и выдает результат, соответствующий приведенному примеру.

```
public static void Main(string[] args)
{
    Random random = new Random();
    SwordDamage swordDamage = new SwordDamage();
    while (true)
    {
        Console.WriteLine("0 for no magic/flaming, 1 for magic, 2 for flaming, " +
            "3 for both, anything else to quit: ");
        char key = Console.ReadKey().KeyChar;
        if (key != '0' && key != '1' && key != '2' && key != '3') return;
        int roll = random.Next(1, 7) + random.Next(1, 7) + random.Next(1, 7);
        swordDamage.Roll = roll;
        swordDamage.SetMagic(key == '1' || key == '3');
        swordDamage.SetFlaming(key == '2' || key == '3');
        Console.WriteLine("\nRolled " + roll + " for " + swordDamage.Damage + " HP\n");
    }
}
```

SwordDamage

Roll
MagicMultiplier
FlamingDamage
Damage

CalculateDamage
SetMagic
SetFlaming



Превосходно! Но я хотел спросить...
Нельзя ли создать более наглядное приложение?

Да! Мы можем построить приложение WPF, которое использует тот же класс.

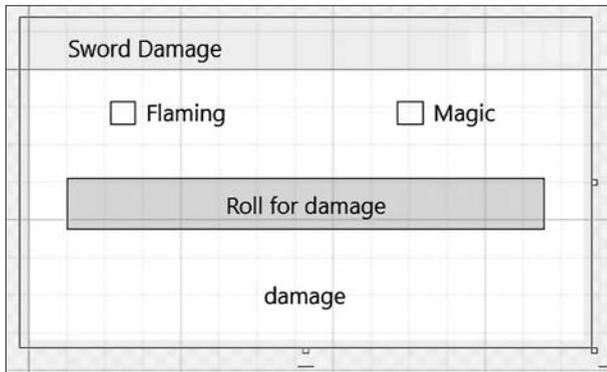
Попробуем **повторно использовать** класс `SwordDamage` в приложении WPF. Первая проблема, с которой мы сталкиваемся, — создание интуитивного повторного использования. Меч может быть волшебным, огненным, и тем и другим или ни тем ни другим; нужно как-то реализовать все эти варианты в графическом интерфейсе, а вариантов много. Можно создать набор переключателей или раскрывающийся список с четырьмя вариантами — по аналогии с четырьмя вариантами, которые предоставляет консольное приложение. Тем не менее мы решили, что более наглядным будет решение с флажками.

В WPF класс флажка `CheckBox` использует свойство `Content` для вывода надписи справа от элемента подобно тому, как `Button` использует свойство `Content` для выводимого текста. Имеются методы `SetMagic` и `SetFlaming`, и мы можем использовать события `Checked` и `Unchecked` элемента `CheckBox` для задания методов, которые должны вызываться при установке или снятии флажка пользователем.

Версия этого проекта для Mac доступна в приложении «Visual Studio для пользователей Mac».

Разработка XAML для WPF-версии калькулятора повреждений

Создайте новое приложение WPF, задайте текст заголовка главного окна **Sword Damage**, выберите высоту **175** и ширину **300**. Включите в сетку три строки и два столбца. В верхней строке должны находиться два элемента **CheckBox** с надписями **Flaming** и **Magic**, в средней — элемент **Button** с надписью «Roll for Damage», занимающий оба столбца, и в нижней — элемент **TextBlock**, также занимающий оба столбца.



Сделайте это!

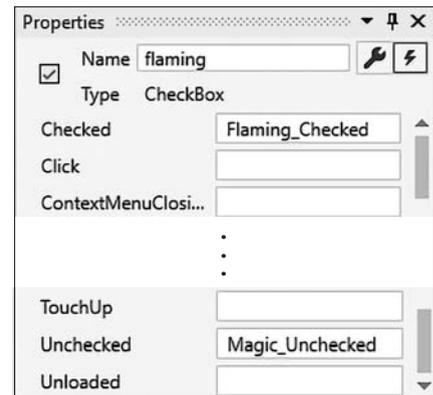
Выделите элемент **CheckBox**, затем воспользуйтесь кнопкой **Events** в окне **Properties** для отображения событий. После ввода имени элемента в верхней части окна вы можете сделать двойной щелчок на элементах **Checked** и **Unchecked** — IDE добавит их автоматически и использует имена элементов для генерирования имен методов — обработчиков событий.

Ниже приведен код XAML — конечно, для построения формы можно воспользоваться конструктором, но при желании вы также можете ввести XAML вручную.

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
```

Присвойте элементам **CheckBox** имена **magic** и **flaming**, а элементу **TextBlock** — имя **damage**. Убедитесь в том, что имена правильно указаны в свойствах **x:Name** в коде XAML.

```
<CheckBox x:Name="flaming" Content="Flaming"
  HorizontalAlignment="Center" VerticalAlignment="Center"
  Checked="Flaming_Checked" Unchecked="Flaming_Unchecked"/>
<CheckBox x:Name="magic" Content="Magic" Grid.Column="1"
  HorizontalAlignment="Center" VerticalAlignment="Center"
  Checked="Magic_Checked" Unchecked="Magic_Unchecked" />
<Button Grid.Row="1" Grid.ColumnSpan="2" Margin="20,10"
  Content="Roll for damage" Click="Button_Click"/>
<TextBlock x:Name="damage" Grid.Row="2" Grid.ColumnSpan="2" Text="damage"
  VerticalAlignment="Center" HorizontalAlignment="Center"/>
</Grid>
```



Обработчики событий **Checked** и **Unchecked** вызываются при установке или снятии флажков.

Этот текст будет заменен выводом («Rolled 17 for 34HP»).

Код программной части для WPF-калькулятора повреждений

Добавьте этот код программной части в свое приложение WPF. Он создает экземпляры `SwordDamage` и `Random`, а также включает флажки и кнопку в вычисление повреждений:

```
public partial class MainWindow : Window
{
```

```
    Random random = new Random();
    SwordDamage swordDamage = new SwordDamage();
```

```
    public MainWindow()
```

```
    {
        InitializeComponent();
        swordDamage.SetMagic(false);
        swordDamage.SetFlaming(false);
        RollDice();
    }
```

```
    public void RollDice()
```

```
    {
        swordDamage.Roll = random.Next(1, 7) + random.Next(1, 7) + random.Next(1, 7);
        DisplayDamage();
    }
```

```
    void DisplayDamage()
```

```
    {
        damage.Text = "Rolled " + swordDamage.Roll + " for " + swordDamage.Damage + " HP";
    }
```

```
    private void Button_Click(object sender, RoutedEventArgs e)
```

```
    {
        RollDice();
    }
```

```
    private void Flaming_Checked(object sender, RoutedEventArgs e)
```

```
    {
        swordDamage.SetFlaming(true);
        DisplayDamage();
    }
```

```
    private void Flaming_Unchecked(object sender, RoutedEventArgs e)
```

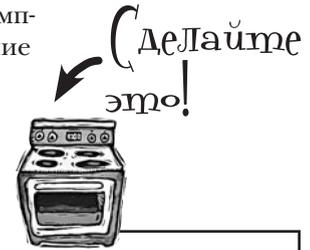
```
    {
        swordDamage.SetFlaming(false);
        DisplayDamage();
    }
```

```
    private void Magic_Checked(object sender, RoutedEventArgs e)
```

```
    {
        swordDamage.SetMagic(true);
        DisplayDamage();
    }
```

```
    private void Magic_Unchecked(object sender, RoutedEventArgs e)
```

```
    {
        swordDamage.SetMagic(false);
        DisplayDamage();
    }
}
```



Готовый код

Вы уже видели, что код конкретной программы можно написать множеством разных способов. Возможно, во многих проектах этой книги можно было бы найти другой — не менее эффективный — способ решения проблемы. Но для калькулятора повреждений Оуэна введите код точно в таком виде, в каком он приведен здесь, потому что (внимание, спойлер!) мы **намеренно включили в него несколько ошибок**.

Очень внимательно прочитайте этот код. Сможете ли вы найти какие-то ошибки до его выполнения?

Разговор за столом (или, может, дискуссия о кубиках?)

Наступает ночь игры! Вся группа Оуэна в сборе, и он собирается пустить в ход свой новейший калькулятор повреждений. Посмотрим, что происходит.

Хорошо, коллеги,
у нас новое правило. Приготовьтесь: вы будете в шоке
от невероятных чудес **современной технологии**.

Джейден: Оуэн, о чем ты говоришь?

Оуэн: Я говорю о новом приложении, которое вычисляет повреждения от ударов мечом... автоматически.

Мэтью: Потому что бросать кубики очень, очень утомительно.

Джейден: Да ладно, зачем столько сарказма. Давайте попробуем.

Оуэн: Спасибо, Джейден. И момент самый подходящий, потому что Бриттани только что ударила корову-оборотня своим огненным волшебным мечом. Давай, Бриттани, попробуй.

Бриттани: Хорошо. Мы только что запустили приложение, я установила флажок Magic. Похоже, здесь какой-то неправильный бросок, давайте я снова нажму кнопку, и...

Джейден: Нет, что-то не так. Ты выбросила 14, но приложение все равно выдает всего 3 HP. Нажми еще раз. Выпало 11, и снова 3 HP. Пробуем еще раз. 9, 10, 5 — и каждый раз 3 HP. Оуэн, в чем дело?

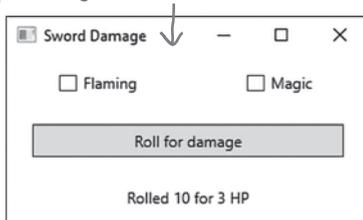
Бриттани: Вообще-то приложение в какой-то степени работает. Если смоделировать бросок, а потом пару раз установить флажки, со временем оно начинает выдавать правильный ответ. Смотрите, я выбросила 10 с повреждениями 22 HP.

Джейден: Ты права. Причем делать все следует в строго определенном порядке. Сначала нажать кнопку броска, потом установить нужные флажки и обязательно установить флажок Flaming дважды.

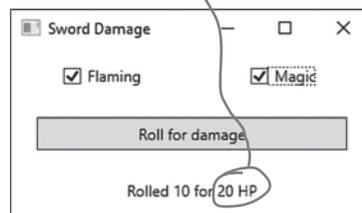
Оуэн: Верно. И если проделать все именно в таком порядке, программа работает. Но если где-то нарушить этот порядок, она ломается. Ладно, можно и так.

Мэтью: Или... Может, просто делать все по-старому, с настоящими кубиками?

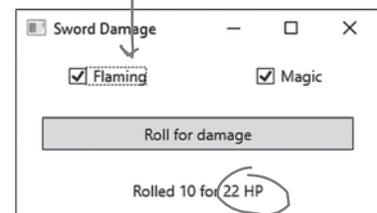
Бриттани и Джейден правы. Программа работает, но только если делать все в строго определенном порядке. Вот как приложение выглядит при запуске.



Попробуем вычислить повреждения для удара огненным волшебным мечом — сначала установим флажок Flaming, а потом флажок Magic. Стоп — неправильный результат.



Но если щелкнуть на флажке Flaming дважды, число будет правильным.





Попробуем исправить ошибку

Когда вы запускаете программу, что она делает в первую очередь? Повнимательнее присмотримся к этому методу в самом начале класса MainWindow с кодом программной части окна:

```
public partial class MainWindow : Window
{
    Random random = new Random();
    SwordDamage swordDamage = new SwordDamage();

    public MainWindow()
    {
        InitializeComponent();
        swordDamage.SetMagic(false);
        swordDamage.SetFlaming(false);
        RollDice();
    }
}
```

Этот метод называется **конструктором**. Он вызывается при создании экземпляра класса MainWindow и может использоваться для инициализации экземпляра. **Конструктор не имеет возвращаемого типа, а его имя совпадает с именем класса.**

Если у класса имеется конструктор, то при создании нового экземпляра этого класса конструктор станет первым выполняемым кодом. Когда приложение запускается и создает экземпляр MainWindow, сначала оно инициализирует поля (включая создание объекта SwordDamage), а затем вызывает конструктор. Таким образом, программа вызывает RollDice непосредственно перед отображением окна, и проблема встречается при каждом нажатии кнопки броска — следовательно, не исключено, что решение проблемы может быть реализовано в методе RollDice. **Внесите следующие изменения в метод RollDice:**

```
public void RollDice()
{
    swordDamage.Roll = random.Next(1, 7) + random.Next(1, 7) + random.Next(1, 7);
    swordDamage.SetFlaming(flaming.IsChecked.Value);
    swordDamage.SetMagic(magic.IsChecked.Value);
    DisplayDamage();
}
```

Исправьте!

Вызов IsChecked.Value для флажка возвращает true, если флажок установлен, или false для снятого флажка.

Протестируйте свой код. Запустите программу и несколько раз щелкните на кнопке. Пока все хорошо — числа выглядят нормально. Теперь **установите флажок Magic** и щелкните на кнопке еще несколько раз. Отлично, исправление сработало! Осталось протестировать последнюю мелочь. Установите флажок Flaming, щелкните на кнопке и... **Ой!** Снова не работает. Когда вы щелкаете на кнопке, множитель 1.75 для волшебного меча применяется, но дополнительные 3 HP для огненных мечей не прибавляются. Для получения правильного числа все равно нужно установить и снять флажок Flaming. Программа все еще не работает.



Мы выдвинули предположение и **быстро** написали код, но он не решил проблему, потому что мы **не размышляли** над тем, что же было причиной ошибки.

Всегда думайте над тем, из-за чего произошла ошибка, прежде чем пытаться исправить ее.

Когда в вашем коде что-то идет не так, очень соблазнительно с ходу взяться за дело и немедленно начать писать еще больше кода для исправления. Может показаться, что вы быстро реагируете на проблемы, но в таких ситуациях слишком легко добавить новую порцию ошибочного кода. Всегда надежнее не торопиться и разобраться, что же действительно вызвало ошибку, вместо того чтобы просто попытаться слепить решение на скорую руку.

Использование Debug.WriteLine для вывода диагностической информации

В нескольких последних главах мы пользовались отладчиком для выявления ошибок, но это не единственный инструмент, применяемый разработчиками для поиска ошибок в коде. Когда профессиональные разработчики занимаются диагностикой ошибок, одной из самых частых мер становится **добавление команд диагностического вывода**. Именно это мы сделаем для выявления ошибки.

Строчковая интерполяция

Вы уже использовали оператор `+` для конкатенации строк. Это весьма мощный инструмент — вы можете взять любое значение (отличное от `null`), и оно будет безопасно преобразовано в строку (обычно вызовом метода `ToString`). Проблема в том, что конкатенация нередко усложняет чтение кода.

К счастью, C# предоставляет отличное средство для упрощения конкатенации строк. Этот механизм называется **строчковой интерполяцией**, и чтобы воспользоваться им, достаточно поставить знак `$` перед строкой. Чтобы включить в строку переменную, поле или сложное выражение (даже вызов метода!), заключите их в фигурные скобки. Если в строке должны присутствовать литеральные фигурные скобки, удвойте их: `{{ }}`.

Откройте окно Output в Visual Studio командой Output из меню View (Ctrl+O W). Любой текст, который выводится вызовом `Console.WriteLine` из приложения WPF, отображается в этом окне. Используйте `Console.WriteLine` только для вывода текста, который должен быть виден вашим пользователям. Если же строки выводятся исключительно для отладочных целей, следует использовать **Debug.WriteLine**. Класс `Debug` находится в пространстве имен `System.Diagnostics`, поэтому начните с добавления строки `using` в начало файла класса `SwordDamage`:

```
using System.Diagnostics;
```

Затем добавьте команду `Debug.WriteLine` в конец метода `CalculateDamage`:

```
public void CalculateDamage()
{
    Damage = (int)(Roll * MagicMultiplier) + BASE_DAMAGE + FlamingDamage;
    Debug.WriteLine($"CalculateDamage finished: {Damage} (roll: {Roll})");
}
```

Включите еще одну команду `Debug.WriteLine` в конец метода `SetMagic` и еще одну в конец метода `SetFlaming`. Эти команды должны быть идентичны добавленной в `CalculateDamage`, кроме того, что они выводят «SetMagic» и «SetFlaming» вместо «CalculateDamage»:

```
public void SetMagic(bool isMagic)
{
    // остальные команды метода SetMagic без изменений
    Debug.WriteLine($"SetMagic finished: {Damage} (roll: {Roll})");
}

public void SetFlaming(bool isFlaming)
{
    // остальные команды метода SetFlaming без изменений
    Debug.WriteLine($"SetFlaming finished: {Damage} (roll: {Roll})");
}
```

Теперь ваша программа будет выводить полезную диагностическую информацию в окне Output.

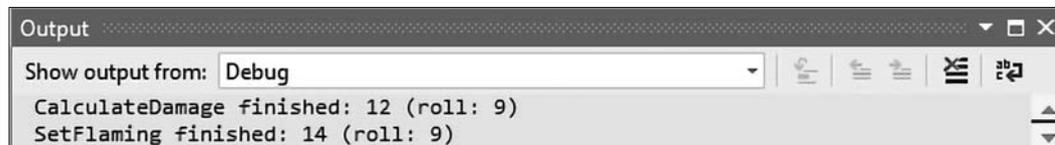
Эту ошибку можно выявить и без установки точек прерывания. Разработчики поступают так очень часто... а значит, вам тоже стоит этому научиться!



По следу

Воспользуемся **окном Output** для отладки приложения. Запустите программу и наблюдайте за окном Output. Во время загрузки вы увидите ряд сообщений, в которых говорится, что CLR загрузила различные DLL (это нормально, не обращайте внимания).

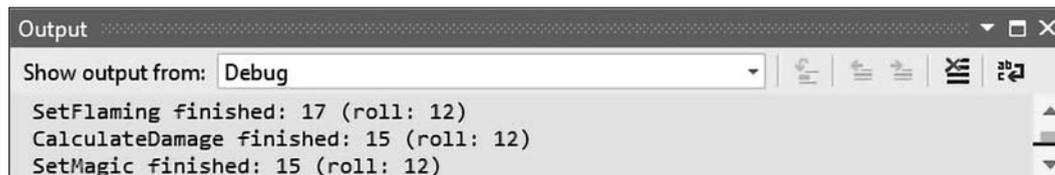
Когда на экране появится главное окно, нажмите кнопку Clear All (☒), чтобы очистить окно Output. Установите флажок Flaming. Следующий снимок экрана был сделан для результата броска 9:



14 — правильный ответ; 9 + базовое повреждение 3 + 2 для огненного меча. Пока все нормально.

Из окна Output видно, что произошло: метод SetFlaming сначала вызвал метод CalculateDamage, который вычислил результат 12. Затем было добавлено значение FLAME_DAMAGE, что дало результат 14, и наконец, была выполнена добавленная вами команда Debug.WriteLine.

Нажмите кнопку, чтобы повторить бросок. Программа должна вывести еще три строки в окне Output:



На кубиках выпало 12, вычисленный результат должен быть равен 17. Что же отладочный вывод говорит о произошедшем?

Сначала был вызван метод SetFlaming, который присвоил Damage значение 17, — и это правильно: 12 + 3 (базовые повреждения) + 2 (огненные повреждения).

Но затем программа вызвала метод CalculateDamage, который **заменяет значение в поле Damage** и вернул значение 15.

Проблема в том, что метод **SetFlaming был вызван до CalculateDamage**, и несмотря на то что огненные повреждения были добавлены правильно, последующий вызов CalculateDamage отменил их. Итак, настоящая причина того, что программа не работает, заключается в том, что поля и методы класса SwordDamage **должны использоваться в строго определенном порядке**:

1. Присвоить полю Roll результат броска 3д6.
2. Вызвать метод SetMagic.
3. Вызвать метод SetFlaming.
4. Не вызывать метод CalculateDamage, потому что SetFlaming делает это за вас.

Вот почему консольное приложение работало, а приложение WPF не работает. Консольное приложение использовало класс SwordDamage конкретным способом, при котором он работает. Приложение WPF вызвало методы в неправильном порядке и поэтому получило неправильные результаты.

Ага! Теперь мы действительно знаем, почему программа не работает.

Debug.WriteLine — один из основных (и самых полезных!) отладочных инструментов в инструментарии разработчика. Иногда самый быстрый способ выявления ошибок в коде заключается в стратегическом размещении команд Debug.WriteLine для получения важной информации, которая поможет в решении проблемы.



Значит, методы просто должны вызываться в определенном порядке. И в чем проблема? Меняем порядок их вызова, и программа начинает работать.

Люди не всегда используют ваши классы в точности так, как вы ожидаете. И чаще всего этими классами будете пользоваться вы сами! Сегодня вы пишете класс, который будет использоваться завтра или в следующем месяце. К счастью, C# предоставляет мощный механизм, который повышает вероятность того, что программа всегда будет работать правильно, даже если пользователи делают что-то такое, что вам в голову не приходило. Этот механизм называется **инкапсуляцией**, и он чрезвычайно полезен при работе с объектами. Целью инкапсуляции является ограничение доступа к «внутренностям» ваших классов, чтобы все поля и методы классов были **безопасными в использовании, а возможности их некорректного использования были сведены к минимуму**. Инкапсуляция позволяет создавать классы, при работе с которыми намного труднее совершить ошибку, — и это **отличный способ предотвращения ошибок** вроде той, которую мы обнаружили в калькуляторе повреждений.

Часть Задаваемые Вопросы

В: Чем отличаются методы `Console.WriteLine` и `Debug.WriteLine`?

О: Класс `Console` используется консольными приложениями для получения входных данных и передачи результатов пользователю. Он использует три **стандартных потока**, предоставляемых операционной системой: стандартный ввод (`stdin`), стандартный вывод (`stdout`) и стандартный поток ошибок (`stderr`). Стандартный ввод — текст, который передается программе, а стандартный вывод — текст, который программа выводит. (Если вы когда-либо использовали перенаправление ввода/вывода в командной оболочке или командной строке с использованием `<`, `>`, `|`, `<<`, `>>` либо `||`, считайте, что вы уже использовали `stdin` и `stdout`). Класс `Debug` принадлежит пространству имен `System.Diagnostics`, что наводит на мысль о его применении: он упрощает диагностику, выявление и исправление ошибок. `Debug.WriteLine` направляет свой вывод **слушателям трассировки** — специальным классам, которые отслеживают диагностический вывод ваших программ и направляют его на консоль, в файлы журналов или диагностические программы, собирающие данные для анализа.



Позднее в этой главе **конструкторы** будут рассмотрены гораздо подробнее.

А пока просто считайте, что конструктор — это специальный метод, используемый для инициализации объектов.

В: Могу ли я использовать конструкторы в своем коде?

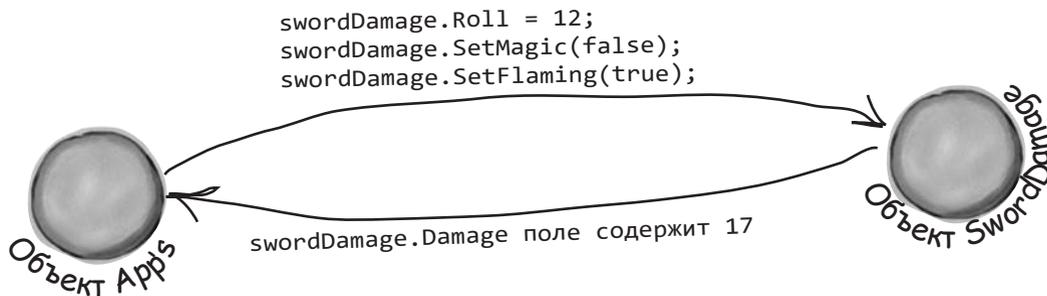
О: Разумеется. Конструктор — метод, который вызывается CLR при создании нового экземпляра объекта. Это самый обычный метод, в нем нет ничего мистического или специального. Вы можете добавить конструктор в любой класс, объявив метод **без возвращаемого типа** (без `void`, `int` или другого типа в начале), имя которого **совпадает с именем класса**. Каждый раз, когда среда CLR встречает такой метод в классе, она распознает его как конструктор и вызывает каждый раз при создании нового объекта и размещении его в куче.

Возможность некорректного использования объектов

Приложение Оуэна столкнулось с проблемами, потому что мы поспешно решили, что метод `CalculateDamage` должен вычислять величину повреждений. Оказалось, что **вызывать этот метод напрямую небезопасно**, потому что он заменил значение `Damage` и стер результаты уже выполненных вычислений. Вместо этого нужно было поручить методу `SetFlaming` вызвать `CalculateDamage` за нас, но **даже этого было недостаточно**, потому что мы также должны были позаботиться о том, чтобы метод `SetMagic` всегда вызывался первым. Итак, хотя класс `SwordDamage` работает с технической точки зрения, при непредвиденных обращениях к нему возникают проблемы.

Как должен был использоваться класс `SwordDamage`

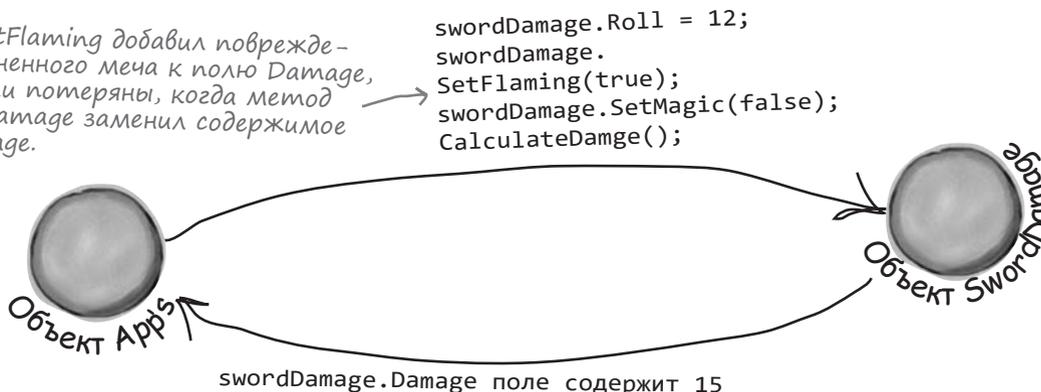
Класс `SwordDamage` предоставил приложению удобный метод для вычисления общих повреждений от меча. Все, что для этого нужно, — смоделировать бросок кубиков, затем вызвать метод `SetMagic` и, наконец, вызвать метод `SetFlaming`. Если все делается именно в таком порядке, то поле `Damage` будет обновлено с учетом вычисленных повреждений. Тем не менее в приложении все происходило не так.



Как использовался класс `SwordDamage`

Вместо этого приложение задавало значение поля `Roll`, затем вызвало `SetFlaming`, что привело к увеличению повреждений в поле `Damage`. Далее был вызван метод `SetMagic`, и наконец, вызов метода `CalculateDamage` сбросил содержимое поля `Damage` с потерей дополнительных повреждений от огня.

Метод `SetFlaming` добавил повреждения для огненного меча к полю `Damage`, но они были потеряны, когда метод `CalculateDamage` заменил содержимое поля `Damage`.



Инкапсуляция подразумевает ограничение доступа к части данных класса

Проблемы некорректного использования объектов можно избежать: для этого нужно позаботиться о том, чтобы класс можно было использовать только строго определенным способом. С# помогает вам в этом, позволяя объявить часть полей приватными (ключевое слово `private`). До настоящего момента вы видели только открытые поля (`public`). Если поле объекта объявлено открытым, то любой другой объект может прочитать или изменить это поле. Если же объявить поле приватным, то к этому полю можно будет обратиться только из этого объекта (или из другого экземпляра *того же класса*).

```
class SwordDamage
{
    public const int BASE_DAMAGE = 3;
    public const int FLAME_DAMAGE = 2;

    public int Roll;
    private decimal magicMultiplier = 1M;
    private int flamingDamage = 0;
    public int Damage;

    private void CalculateDamage()
    {
        ...
    }
}
```

Объявляя метод `CalculateDamage` *приватным*, мы запрещаем приложению случайно вызывать его со сбросом поля `Damage`. Изменение полей, задействованных в вычислении, и перевод их на объявление `private` не позволяет приложению вмешиваться в ход вычислений. Когда вы объявляете некоторые данные приватными, а затем пишете код для использования этих данных, это называется *инкапсуляцией*. Когда класс защищает свои данные и предоставляет компоненты, простые в использовании и снижающие риск злоупотреблений, мы называем такой класс *хорошо инкапсулированным*.



Если возникли сомнения — объявляйте приватным.

Не до конца понимаете, как решить, какие поля и методы стоит объявить приватными? Для начала объявите приватными все поля и методы и преобразуйте их в открытые только тогда, когда это необходимо. **В данном случае** лень работает в вашу пользу. Если опустить объявление `private` или `public`, С# решит, что поле или метод являются приватными.

Если вы хотите сделать поле приватным, достаточно воспользоваться ключевым словом `private` при его объявлении. Тем самым вы сообщаете С#, что у экземпляра `SwordDamage` операции записи и чтения полей `magicMultiplier` и `flamingDamage` могут выполняться только методами экземпляра `SwordDamage`. Для других объектов они будут недоступны.

А вы заметили, что мы также изменили имена приватных полей, чтобы они начинались со строчных букв?

ин-кап-су-ли-ро-ван-ный, прил. снабженный защитным покрытием или мембраной.

Применение инкапсуляции для управления доступом к методам и полям класса

Если вы объявите все поля и методы своего класса открытыми, любой другой класс сможет обратиться к ним. Все нюансы того, что знает и делает ваш класс, становятся открытой книгой для любого другого класса в программе... а вы только что видели, как это может привести к совершенно непредвиденным отклонениям в поведении вашей программы.

Вот почему ключевые слова `public` и `private` называются модификаторами доступа: они изменяют уровень доступа к компонентам класса. Инкапсуляция позволяет вам управлять тем, что может использоваться снаружи, а что должно оставаться скрытым в рамках вашего класса. Давайте посмотрим, как это работает.

- 1 Супершпион Херб Джонс – объект, представляющий **секретного агента в шпионской игре, действие которой происходит в 1960-е годы**. Он защищает свободу, жизнь и стремление к счастью в качестве агента под прикрытием в СССР. Объект является экземпляром класса `SecretAgent`.



RealName: "Herb Jones"
Alias: "Dash Martin"
Password: "the crow flies at midnight"

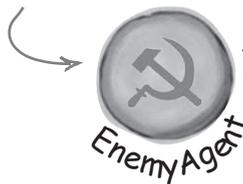
SecretAgent
Alias RealName Password
AgentGreeting

- 2 Агент Джонс разработал план, который поможет ему избежать столкновения с объектом вражеского агента. Он добавил метод `AgentGreeting`, который получает в параметре пароль. Если пароль задан неверно, он раскрывает только свой псевдоним – Dash Martin.

EnemyAgent
Borscht Vodka
ContactComrades OverthrowCapitalists

- 3 Вроде бы абсолютно надежный способ защиты личности агента, не так ли? Если объект агента, обращающийся с вызовом, не предоставит правильный пароль, имя агента остается надежно защищенным.

Этот экземпляр `EnemyAgent` пытается обнаружить сверх-секретную личность нашего героического агента.



AgentGreeting("the jeep is parked outside")

Вражеский агент указал неправильный пароль в приветствии.

"Dash Martin"

Враг узнает только псевдоним агента. Идеально!.. Правда?



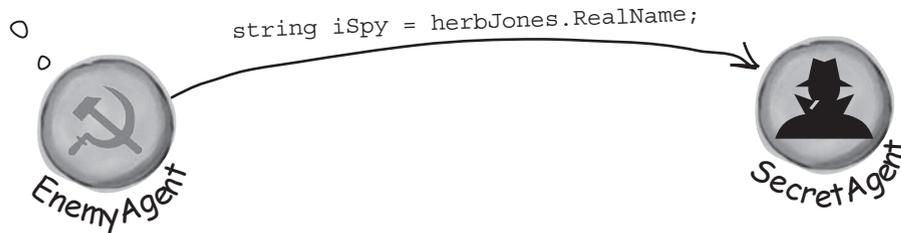
Но ДЕЙСТВИТЕЛЬНО ЛИ поле `RealName` надежно защищено?

Если враг не знает пароль объекта `SecretAgent`, то настоящие имена агентов вроде бы в безопасности. Но если данные хранятся в открытых полях, никакой пользы от такой «защиты» не будет:

```
public string RealName;
public string Password;
```

Объявление полей открытыми означает, что к ним может обратиться (и даже изменить их!) любой другой объект.

Ага! Он оставил поле открытым? Тогда зачем все эти хлопоты с подбором пароля для метода `AgentGreeting`? Я могу просто прочитать имя напрямую.



Что может сделать агент Джонс? Он может объявить поля **приватными**, чтобы держать свою личность в секрете от вражеских агентов. Если поле `realName` будет объявлено приватным, обратиться к нему можно будет только одним способом: **вызвать методы, которые имеют доступ к приватным частям класса**.

Объект `EnemyAgent` не может обратиться к приватным полям, потому что он является экземпляром другого класса.

Достаточно заменить `public` на `private`, и поле становится скрытым от любого объекта, который не является экземпляром того же класса. Объявление полей и методов приватными гарантирует, что внешний код не сможет изменить используемые данные в самый неподходящий момент. Мы переименовали поля, чтобы *они начинались со строчных букв*; это упрощает чтение кода.

```
private string realName;
private string password;
```

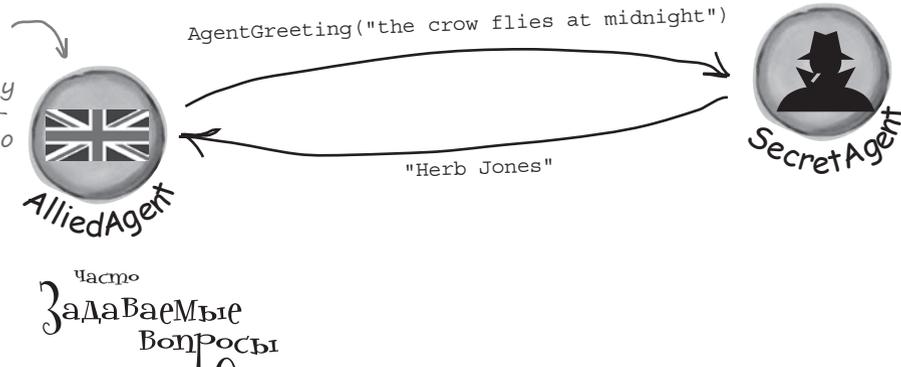


Объявление приватных методов и полей в калькуляторе повреждений предотвращает ошибки, связанные с попытками прямого обращения к ним. **Но проблема все еще осталась!** Если метод `SetMagic` будет *вызван до* метода `SetFlaming`, вы получите неправильный результат. Может ли ключевое слово `private` предотвратить эту неприятность?

К приватным полям и методам могут обращаться только экземпляры того же класса

У объекта есть только одна возможность обратиться к данным, хранящимся в приватных полях другого класса: он должен воспользоваться открытыми полями и методами, которые возвращают данные. Агентам EnemyAgent и AlliedAgent придется использовать метод AgentGreeting, но дружественные агенты, которые также являются экземплярами SecretAgent, все увидят... потому что **любой класс может просматривать приватные поля всех остальных экземпляров того же класса.**

Класс AlliedAgent представляет агента из союзной страны, которому разрешено знать истинную личность секретного агента. Но экземпляры AlliedAgent все равно не имеют доступа к приватным полям объекта SecretAgent. Они видны только другому объекту SecretAgent.



В: Зачем создавать в объекте поля, которые другой объект не может прочитать или записать?

О: Иногда в классе приходится хранить информацию, которая необходима для его работы, но должна оставаться скрытой от других объектов, — вы уже видели пример такого рода. В предыдущей главе было показано, что класс Random использует специальные затравки для инициализации генератора псевдослучайных чисел. Оказывается, во внутренней реализации каждый экземпляр класса Random содержит массив из нескольких десятков чисел. Их использование гарантирует, что вызов Next всегда возвращает случайное число. Однако этот массив является приватным — при создании экземпляра Random вы не сможете обратиться к этому массиву. Если бы это было возможно, то вы могли бы разместить в своем объекте идентичный массив, и объект стал бы генерировать неслучайные значения. Следовательно, затравки должны быть полностью инкапсулированы.

В: Хорошо, к приватным данным нужно обращаться через открытые методы. Что произойдет, если класс с приватным полем не дает мне возможности обратиться к этим данным, но моему объекту они очень нужны?

О: Тогда вы не сможете к ним обратиться. Когда вы пишете класс, всегда позаботьтесь о том, чтобы у других объектов была возможность добраться до нужных им данных. Приватные поля — очень важная часть инкапсуляции, но это лишь часть истории. Написание хорошо инкапсулированного класса означает, что вы предоставляете другим людям средства для разумных и простых обращений к нужным данным, но ограждаете от них критические данные, от которых зависит ваш класс.

В: Я только что заметил, что команда Generate method в IDE использует ключевое слово private. Почему?

О: Потому, что это самое безопасное, что может сделать IDE. Приватными объявляются не только методы, созданные командой Generate method; если вы сделаете двойной щелчок на элементе, чтобы добавить обработчик событий, IDE и для него создаст приватный метод. Дело в том, что **объявление приватного поля или метода предотвращает ошибки** вроде тех, которые были продемонстрированы в калькуляторе повреждений. Вы всегда можете переобъявить поля и методы класса открытыми позднее, если данные должны быть доступны для другого класса.

Чтобы один объект мог получить доступ к приватному полю другого объекта другого класса, он должен воспользоваться открытыми методами, которые возвращают эти данные.



Упражнение

Чтобы немного потренироваться в использовании ключевого слова `private`, мы создадим маленькую игру «Больше-меньше». Игра начинается со ставки 10 долларов и выбирает случайное число от 1 до 10. Игрок пытается угадать, будет следующее число больше или меньше текущего. Если игрок угадал правильно, он выигрывает доллар, в противном случае он теряет доллар. Затем следующее число становится текущим, и игра продолжается.

Создайте для игры **новое консольное приложение**. Метод `Main` выглядит так:

```
public static void Main(string[] args)
{
    Console.WriteLine("Welcome to HiLo.");
    Console.WriteLine($"Guess numbers between 1 and {HiLoGame.MAXIMUM}.");
    HiLoGame.Hint();
    while (HiLoGame.GetPot() > 0)
    {
        Console.WriteLine("Press h for higher, l for lower, ? to buy a hint,");
        Console.WriteLine($"or any other key to quit with {HiLoGame.GetPot()}.");
        char key = Console.ReadKey(true).KeyChar;
        if (key == 'h') HiLoGame.Guess(true);
        else if (key == 'l') HiLoGame.Guess(false);
        else if (key == '?') HiLoGame.Hint();
        else return;
    }
    Console.WriteLine("The pot is empty. Bye!");
}
```

Не забывайте: подсмотреть в решение — не значит жульничать!

Теперь добавьте **статический класс** с именем `HiLoGame` и включите в него **следующие компоненты**. Так как класс является статическим, все его компоненты тоже должны быть статическими. Не забудьте включить ключевое слово `public` или `private` в объявление каждого поля и метода:

1. Целочисленная константа **MAXIMUM**, по умолчанию равная 10. Не забывайте, что ключевое слово `static` не может использоваться с константами.
2. Экземпляр `Random` с именем `random`.
3. Целочисленное поле с именем `currentNumber`, инициализируемое первым случайным числом, которое должен отгадать игрок.
4. Целочисленное поле с именем `pot`, в котором хранится размер ставки. **Объявите это поле приватным.** ←

Мы объявляем `pot` приватным, потому что не хотим, чтобы другие классы могли добавлять деньги, однако методу `Main` все равно нужно иметь возможность вывести размер ставки на консоль. Внимательно присмотритесь к коду метода `Main` — сможете ли вы понять, как сделать значение `pot` доступным для метода `Main`, не давая ему возможности присвоить значение этого поля?

5. **Метод** с именем `Guess`, получающий параметр `bool` с именем `higher`. Метод должен делать следующее (чтобы понять, как он вызывается, присмотритесь к методу `Main`):
 - Он выбирает следующее случайное число, которое должен угадать игрок.
 - Если игрок выбрал «больше», а следующее число \geq текущему **ИЛИ** если игрок выбрал «меньше», а следующее число \leq текущему, **выведите** на консоль сообщение «You guessed right!» и увеличьте ставку.
 - В противном случае **выведите** на консоль сообщение «Bad luck, you guessed wrong» и уменьшите ставку.
 - **Замените** текущее число выбранным в начале метода, а затем **выведите** на консоль сообщение «The current number is» и число.
6. Метод с именем `Hint`, который находит половину максимума, а затем выводит на консоль сообщение «The number is at least {half}» или «The number is at most {half}» и уменьшает ставку.



Упражнение
Решение

Ниже приведен остальной код игры «Больше-меньше». Игра начинается со ставки 10 долларов и выбирает случайное число от 1 до 10. Игрок пытается угадать, будет ли следующее число больше или меньше текущего. Если игрок угадал правильно, он выигрывает доллар, в противном случае он теряет доллар. Затем следующее число становится текущим, и игра продолжается.

Код класса HiLoGame:

```
static class HiLoGame
{
    public const int MAXIMUM = 10;
    private static Random random = new Random();
    private static int currentNumber = random.Next(1, MAXIMUM + 1);
    private static int pot = 10;

    public static int GetPot() { return pot; }

    public static void Guess(bool higher)
    {
        int nextNumber = random.Next(1, MAXIMUM + 1);
        if ((higher && nextNumber >= currentNumber) ||
            (!higher && nextNumber <= currentNumber))
        {
            Console.WriteLine("You guessed right!");
            pot++;
        }
        else
        {
            Console.WriteLine("Bad luck, you guessed wrong.");
            pot--;
        }
        currentNumber = nextNumber;
        Console.WriteLine($"The current number is {currentNumber}");
    }

    public static void Hint()
    {
        int half = MAXIMUM / 2;
        if (currentNumber >= half)
            Console.WriteLine($"The number is at least {half}");
        else Console.WriteLine($"The number is at most {half}");
        pot--;
    }
}
```

При попытке добавить ключевое слово `static` к константе вы получите ошибку компилятора, потому что все константы являются статическими. Попробуйте добавить его в своем классе — вы сможете обратиться к нему из другого класса, как к любому другому статическому полю.

Поле `pot` объявлено приватным, но метод `Main` может использовать метод `GetPot` для получения его значения без возможности изменять его.

Это хороший пример инкапсуляции. Вы защитили поле `pot`, объявив его приватным. Оно может изменяться вызовами методов `Guess` и `Hint`, а метод `GetPot` предоставляет доступ только для чтения.

↑
Важный момент. Выделите несколько минут и хорошенько разберитесь в том, как работает этот механизм.

Метод `Hint` должен быть открытым, потому что он вызывается из `Main`. А вы заметили, что в команду `if/else` не были включены фигурные скобки? В секции `if` или `else`, состоящей из одной строки, фигурные скобки не нужны.

Дополнительный вопрос: открытое поле `random` можно заменить новым экземпляром `Random`, инициализированным другой затравкой. Тогда новый экземпляр `Random` можно будет использовать с той же затравкой, чтобы узнать числа заранее!

```
HiLoGame.random = new Random(1);
Random seededRandom = new Random(1);
Console.Write("The first 20 numbers will be: ");
for (int i = 0; i < 10; i++)
    Console.Write($"{seededRandom.Next(1, HiLoGame.MAXIMUM + 1)}, ");
```

Все экземпляры `Random`, инициализированные одной затравкой, генерируют одну последовательность псевдослучайных чисел.



Что-то здесь не так. Если я объявлю поле приватным, используя его в другом классе, то **моя программа просто не будет компилироваться**. Но если заменить «private» на «public», моя программа снова построится! Получается, что добавление «private» может только нарушить работу моей программы.

Тогда для чего мне **объявлять поле приватным?**

Потому что иногда бывает нужно, чтобы класс скрывал информацию от остального кода программы.

Многим разработчикам идея инкапсуляции кажется странной, потому что идея сокрытия полей, свойств или методов одного класса от другого класса выглядит немного противоречиво. Есть очень веские причины, по которым вы должны тщательно продумать, какая информация должна раскрываться для остальной программы.

Инкапсуляция означает, что один класс скрывает информацию от другого. Соккрытие информации способствует предотвращению ошибок в программах.



Будьте
осторожны!

Инкапсуляция — не то же самое, что безопасность. Приватные поля не защищены.

Если вы строите игру про шпионов, инкапсуляция поможет избежать ошибок. Но если вы строите **программу для настоящих шпионов**, инкапсуляция не обеспечивает защиты данных.

Например, вернемся к игре «Больше-меньше». Установите точку прерывания в первой строке метода `Main`, добавьте отслеживание для `HiLoGame.random` и проведите отладку программы. Развернув раздел **Non-Public Members**, вы увидите внутренние аспекты класса `Random`, включая массив `_seedArray`, используемый для генерирования псевдослучайных чисел.

Не только IDE видит приватные компоненты вашего класса. В .NET существует механизм **отражения** (*reflection*), который позволяет писать код для обращения к объектам в памяти и просмотра их содержимого — даже приватных полей. Рассмотрим краткий пример его использования. Создайте **новое консольное приложение** и добавьте класс с именем `HasASecret`:

```
class HasASecret
{
    // Класс содержит поле secret. Обеспечит ли ключевое слово private его защиту?
    private string secret = "xyzyzy";
}
```

Классы отражения принадлежат пространству имен `System.Reflection`, поэтому в файл с методом `Main` следует добавить команду `using`:

```
using System.Reflection;
```

Ниже приведен главный класс с методом `Main`, который создает новый экземпляр `HasASecret`, а затем использует отражение для чтения поля `secret`. Он вызывает `GetType` — метод, который можно вызвать для любого объекта, чтобы получить информацию о его типе:

```
class MainClass
{
    public static void Main(string[] args)
    {
        HasASecret keeper = new HasASecret();

        // При снятии комментария с команды Console.WriteLine происходит ошибка компиляции:
        // поле 'HasASecret.secret' недоступно из-за его уровня защиты.
        // Console.WriteLine(keeper.secret);

        // Но для получения значения поля secret все еще можно воспользоваться отражением
        FieldInfo[] fields = keeper.GetType().GetFields(
            BindingFlags.NonPublic | BindingFlags.Instance);

        // Этот цикл foreach выводит на консоль строку "xyzyzy"
        foreach (FieldInfo field in fields)
        {
            Console.WriteLine(field.GetValue(keeper));
        }
    }
}
```

У каждого объекта имеется метод `GetType`, который возвращает объект `Type`. Метод `Type.GetFields` возвращает массив объектов `FieldInfo`, по одному для каждого поля. Каждый объект `FieldInfo` содержит информацию о своем поле. Если вызвать метод `GetValue` с экземпляром этого объекта, он вернет значение, хранящееся в поле, — даже если это поле объявлено приватным.

Для чего нужна инкапсуляция? Представьте объект в виде «черного ящика»...

Иногда программисты говорят об объектах как о «черных ящиках», и это вполне неплохая точка зрения. Когда мы говорим, что нечто является «черным ящиком», мы имеем в виду, что мы видим его поведение, но не можем получить информацию о том, как он устроен.

При вызове метода объекта вас не интересует, как работает этот метод, — по крайней мере пока. Важно лишь то, что метод получает входные данные, которые вы ему передаете, и делает то, что ему положено делать.



Когда разработчики говорят о «черном ящике», они имеют в виду нечто, скрывающее свои внутренние механизмы; чтобы пользоваться ими, не нужно знать, как они работают. Если «черный ящик» выполняет всего одну операцию и ему не нужно передавать параметры, он становится программным аналогом черной коробочки с единственной управляющей кнопкой.

К ящику *можно* добавить другие элементы управления, например окно, в котором можно увидеть, что происходит внутри, или рукоятки и рычаги для управления внутренней реализацией. Но если они не делают ничего, что было бы необходимо для вашей системы, то никакой пользы от них не будет и они только создадут лишние проблемы.

С инкапсуляцией ваши классы...

- ★ **Easier to use.** Вы уже знаете, что поля используются классами для отслеживания их состояния. Многие классы используют методы для поддержания полей в актуальном состоянии — методы, которые никогда не будут вызываться другими классами. Достаточно часто встречаются классы с полями, методами и свойствами, которые никогда не будут вызываться другими классами. Если вы объявите эти компоненты приватными, то они не появятся в окне IntelliSense позднее, когда вы будете работать с этим классом. IDE не будет загромождаться лишними компонентами, а это упростит использование класса.
- ★ **Less prone to bugs.** Ошибка в программе Оуэна произошла из-за того, что приложение вызывало метод напрямую, вместо того чтобы доверить его вызов другим методам класса. Если бы этот метод был объявлен приватным, то этой ошибки можно было бы избежать.
- ★ **Flexible.** Нередко вам приходится возвращаться к программе, написанной уже давно, и добавлять в нее новую функциональность. Если ваши классы хорошо инкапсулированы, то вы будете точно знать, как пользоваться ими и расширять позднее.



Как построение плохо инкапсулированного класса может усложнить изменение вашей программы в будущем?

Несколько идей для инкапсуляции классов

★ Все поля и методы вашего класса объявлены открытыми?

Если ваш класс не содержит ничего, кроме открытых полей и методов, вероятно, вам стоит дополнительно подумать об инкапсуляции.

★ Подумайте о возможностях некорректного использования полей и методов класса.

С какими проблемами вы можете столкнуться, если поля не инициализированы или методы вызываются некорректно?

★ Какие поля требуют дополнительной обработки или вычислений при присваивании им значений?

Это главные кандидаты для инкапсуляции. Если кто-то позднее напишет метод, который изменит значение в одном из таких полей, это может создать проблемы для той работы, которую пытается выполнить ваша программа.

Мы использовали константы для базовых и огненных повреждений. Объявление их открытыми не создаст проблем, потому что константы не могут изменяться.

Но так как они не используются другим классом, возможно, их можно объявить приватными.



★ Объявляйте поля и методы открытыми только при необходимости.

Если у вас нет веских причин объявлять что-то открытым, не делайте этого — объявление всех полей программы открытыми может сильно усложнить вашу жизнь. Впрочем, не старайтесь объявлять все поля и методы приватными. Если вы заранее подумаете над тем, какие поля должны быть открытыми, а какие — нет, вы сэкономите себе много времени в будущем.

Но хорошо инкапсулированный класс делает **то же самое**,
что и плохо инкапсулированный!



Точно! Различия в том, что хорошо инкапсулированный класс строится так, чтобы предотвратить ошибки и упростить его использование.

Хорошо инкапсулированный класс можно легко превратить в плохо инкапсулированный: проведите поиск и замените каждое вхождение `private` на `public`.

Здесь проявляется одна из особенностей ключевого слова `private`: обычно вы можете взять любую программу, провести поиск с заменой, и программа будет компилироваться и работать точно так же, как прежде! И это одна из причин, почему некоторым программистам бывает трудно понять суть инкапсуляции, когда они сталкиваются с ней впервые.

Когда вы возвращаетесь к коду, к которому давно не прикасались, легко забыть, как он должен использоваться. Инкапсуляция способна очень сильно упростить вам жизнь!

До настоящего момента в книге речь шла о том, как заставить программы **что-то делать**, т. е. проявлять некоторое поведение. Инкапсуляция работает немного иначе. Она не влияет на поведение вашей программы, скорее она относится к «шахматной» стороне программирования: вы скрываете некоторую информацию в своих классах в процессе их проектирования и построения и тем самым формируете стратегию взаимодействия с ними. Чем лучше стратегия, тем более **гибкой и простой** в сопровождении будет ваша программа и тем большего количества ошибок вы избежите.

↑
И как и в шахматах,
существует почти
неограниченное число
возможных стратеги-
гий инкапсуляции!

**Если вы хорошо
инкапсулируете свои
классы сегодня, это
сильно упростит
их повторное
использование завтра.**

КЛЮЧЕВЫЕ МОМЕНТЫ

- Всегда думайте о причинах возникновения ошибки, прежде чем пытаться исправлять ее. Не жалейте времени и постарайтесь действительно хорошо понять, что же происходит в программе.
- Добавление команд вывода может быть эффективным средством отладки. Используйте метод **Debug.WriteLine** для добавления команд вывода диагностической информации.
- **Конструктор** — метод, вызываемый CLR при создании нового экземпляра объекта.
- **Строковая интерполяция** упрощает чтение операций конкатенации. Чтобы использовать ее, поставьте \$ в начало строки и заключите выводимые значения в фигурные скобки {}.
- Класс System.Console направляет свой вывод в **стандартные потоки**, обеспечивающие ввод и вывод в консольных приложениях.
- Класс System.Diagnostics.Debug передает свои результаты **слушателям трассировки** (специальным классам, выполняющим конкретные действия с диагностическим выводом). Один из них направляет вывод в окно IDE Output (Windows) или Application Output (macOS).
- Люди не всегда используют ваши классы точно так, как вы ожидаете. Механизм **инкапсуляции** обеспечивает гибкость компонентов класса и затрудняет их некорректное использование.
- Инкапсуляция обычно подразумевает использование ключевого слова private для закрытого хранения некоторых полей и методов класса, чтобы они не могли некорректно использоваться другими классами.
- Когда класс защищает свои данные и предоставляет поля и методы, безопасные в использовании и затрудняющие их некорректное использование, такой класс называется **хорошо инкапсулированным**.



Хорошо, мы знаем, что в коде приложения для расчета повреждений есть проблемы. И что с этим делать?

SwordDamage
Roll
MagicMultiplier
FlamingDamage
Damage
CalculateDamage
SetMagic
SetFlaming

← Помните, как мы использовали метод Debug.WriteLine ранее в этой главе, чтобы найти ошибку в приложении? Выяснилось, что класс SwordDamage нормально работает только тогда, когда его методы вызываются в строго определенном порядке. Вся эта глава посвящена инкапсуляции, поэтому можно ожидать, что инкапсуляция будет использована для решения этой проблемы. Но... как именно?

Воспользуемся инкапсуляцией для улучшения класса `SwordDamage`

Мы рассмотрели некоторые идеи для инкапсуляции классов. Посмотрим, удастся ли нам применить их в классе `SwordDamage`, чтобы предотвратить возможную путаницу и злоупотребления со стороны приложения, в которое был включен этот класс.

Все компоненты класса `SwordDamage` объявлены открытыми?

Да, это так. Четыре поля (`Roll`, `MagicNumber`, `FlamingDamage` и `Damage`) объявлены открытыми, как и три метода (`CalculateDamage`, `SetMagic` и `SetFlaming`). Стоит подумать об инкапсуляции.

Поля или методы используются некорректно?

Безусловно. В первой версии калькулятора был вызван метод `CalculateDamage` там, где его вызов следовало бы поручить методу `SetFlaming`. Даже наша попытка исправить ошибку завершилась неудачей, потому что методы использовались некорректно из-за того, что они вызывались в неправильном порядке.

Требуются ли вычисления после присваивания значения поля?

Несомненно. После присваивания поля `Roll` экземпляр должен рассчитать повреждения немедленно.

Какие же поля и методы должны быть открытыми?

Отличный вопрос. Выделите несколько минут и поразмышляйте над ответом. Мы еще вернемся к нему в конце главы.

Объявление компонентов класса приватными может предотвратить ошибки, связанные с вызовом открытых методов из других классов или обновлением его открытых полей непредвиденными способами.



Поразмыслите над этими вопросами, а затем еще раз взгляните на класс `SwordDamage`. Что бы вы сделали, чтобы исправить проблемы в классе `SwordDamage`?

Инкапсуляция обеспечивает безопасность данных

Вы уже узнали, как ключевое слово `private` защищает компоненты класса от прямых обращений и как предотвратить ошибки, вызванные непредвиденными вызовами методов или обновлениями полей, — подобно тому, как метод `GetPot` в игре «Больше-меньше» предоставлял доступ только для чтения к приватному полю `pot` и это поле могло изменяться только методами `Guess` или `Hint`. Следующий класс работает по тому же принципу.

Инкапсуляция в классе

Построим класс `PaintballGun` для видеоигры — симулятора пейнтбольной арены. Игрок может подобрать магазин с шариками и перезарядить его в любой момент, поэтому мы хотим, чтобы класс отслеживал общее количество шариков у игрока и текущее число заряженных шариков. Мы добавим метод, который проверяет, не пуст ли маркер и не нужно ли его перезарядить. Также необходимо отслеживать размер магазина. Каждый раз, когда игрок получает новые боеприпасы, маркер должен автоматически перезарядить полный магазин. Чтобы обеспечить гарантированную перезарядку, мы определим метод для назначения количества шариков, из которого будет вызываться метод `Reload`.

```
class PaintballGun
```

```
{
    public const int MAGAZINE_SIZE = 16;
    private int balls = 0;
    private int ballsLoaded = 0;

    public int GetBallsLoaded() { return ballsLoaded; }
    public bool IsEmpty() { return ballsLoaded == 0; }
    public int GetBalls() { return balls; }
    public void SetBalls(int numberOfBalls)
    {
        if (numberOfBalls > 0)
            balls = numberOfBalls;
        Reload();
    }
    public void Reload()
    {
        if (balls > MAGAZINE_SIZE)
            ballsLoaded = MAGAZINE_SIZE;
        else
            ballsLoaded = balls;
    }
    public bool Shoot()
    {
        if (ballsLoaded == 0) return false;
        ballsLoaded--;
        balls--;
        return true;
    }
}
```

Константа будет объявлена открытой, потому что она будет использоваться методом `Main`.

Когда игре потребуется вывести количество оставшихся шариков и количество заряженных шариков в пользовательском интерфейсе, она может вызвать методы `GetBalls` и `GetBallsLoaded`.

Игра должна иметь возможность задать количество шариков. Метод `SetBalls` защищает поле `balls`, разрешая игре задать только положительное число. Затем он вызывает `Reload`, чтобы автоматически перезарядить маркер.

Перезарядить маркер можно только одним способом: вызвать метод `Reload`, который заряжает полный магазин или заряжает оставшиеся шарики, если их не набирается на полный магазин. Тем самым предотвращается рассинхронизация полей `balls` и `ballsLoaded`.

Метод `Shoot` возвращает `true` и уменьшает поле `balls`, если маркер заряжен, или `false` в противном случае.

Упрощает ли метод `IsEmpty` чтение этого кода? Или он избыточен? Правильного или неправильного ответа не существует — можно привести аргументы в пользу как одной, так и другой позиции.

Консольное приложение для тестирования класса PaintballGun

Сделайте это!

Опробуем новый класс PaintballGun. Создайте **новое консольное приложение** и добавьте в него класс PaintballGun. Ниже приведен метод Main — в нем используется цикл для вызова различных методов класса:

```
static void Main(string[] args)
{
    PaintballGun gun = new PaintballGun();
    while (true)
    {
        Console.WriteLine($"{gun.GetBalls()} balls, {gun.GetBallsLoaded()} loaded");
        if (gun.IsEmpty()) Console.WriteLine("WARNING: You're out of ammo");
        Console.WriteLine("Space to shoot, r to reload, + to add ammo, q to quit");
        char key = Console.ReadKey(true).KeyChar;
        if (key == ' ') Console.WriteLine($"Shooting returned {gun.Shoot()}");
        else if (key == 'r') gun.Reload();
        else if (key == '+') gun.SetBalls(gun.GetBalls() + PaintballGun.MAGAZINE_SIZE);
        else if (key == 'q') return;
    }
}
```

Консольное приложение с циклом, в котором тестируется экземпляр класса, должно быть вам уже хорошо знакомо. Убедитесь в том, что вы можете прочитать код и понимаете, как он работает.

Наш класс хорошо инкапсулирован, но...

Класс работает, и он достаточно хорошо инкапсулирован. Поле **balls защищено**: оно не может содержать отрицательное количество шариков и синхронизируется с полем ballsLoaded. Методы Reload и Shoot работают так, как ожидалось, и нет никаких *очевидных* возможностей некорректно использовать этот класс.

Но давайте присмотримся к следующей строке метода Main:

```
else if (key == '+') gun.SetBalls(gun.GetBalls() + PaintballGun.MAGAZINE_SIZE);
```

Будем откровенны: такой синтаксис менее удобен, чем синтаксис работы с полями. Если бы мы работали с полем, то можно было бы воспользоваться оператором += для увеличения его на размер магазина. Инкапсуляция полезна, но мы не хотим, чтобы с классом было неудобно или трудно работать.

Можно ли как-то обеспечить защиту поля balls, но при этом пользоваться удобным синтаксисом +=?

Регистр символов в именах частных и открытых полей

Мы использовали схему «верблюжий регистр» (camelCase) для частных полей и схему «регистр Pascal» (PascalCase) для открытых полей. Схема PascalCase означает, что каждое слово в имени переменной начинается с буквы верхнего регистра. Схема camelCase похожа на PascalCase, но в ней имя начинается с буквы нижнего регистра. Она называется «верблюжьим регистром», потому что буквы верхнего регистра напоминают горбы верблюда.

Использование разных схем для открытых и частных полей — соглашение, которое соблюдают многие программисты. Последовательное применение регистра в именах полей, свойств, переменных и методов упростит чтение вашего кода.

Свойства упрощают инкапсуляцию

До настоящего момента вы узнали о двух разновидностях компонентов класса: методах и полях. Также существует третья разновидность компонентов класса, используемая для инкапсуляции: **свойства**. Свойство — компонент класса, который *выглядит как поле* при использовании, но *ведет себя как метод*.

Свойство объявляется как поле, с типом и именем, но вместо символа ; за объявлением следует пара фигурных скобок. В скобках определяются **методы доступа** — методы для чтения или присваивания значения свойству. Существуют две разновидности методов доступа:

- ★ **get-метод** возвращает значение свойства. Он начинается с ключевого слова `get`, за которым следует метод в фигурных скобках. Метод должен возвращать значение, соответствующее типу из объявления свойства.
- ★ **set-метод** задает новое значение свойства. Он начинается с ключевого слова `set`, за которым следует метод в фигурных скобках. Внутри метода ключевое слово `value` представляет переменную, доступную только для чтения, которая содержит присваиваемое значение.

Свойство очень часто читает или задает **резервное поле** — так мы называем приватное поле, инкапсулируемое ограничением доступа к нему через свойство.

Замена методов `GetBalls` и `SetBalls` свойством

← Замените!

Ниже приведены методы `GetBalls` и `SetBalls` из класса `PaintballGun`:

```
public int GetBalls() { return balls; }

public void SetBalls(int numberOfBalls)
{
    if (numberOfBalls > 0)
        balls = numberOfBalls;
    Reload();
}
```

Заменяем их свойством. Удалите оба метода и добавьте свойство `Balls`.

```
public int Balls
{
    get { return balls; }

    set
    {
        if (value > 0)
            balls = value;
        Reload();
    }
}
```

← Это **объявление**. Из него следует, что свойство объявляется с именем `Balls` и типом `int`.

← **Get-метод** идентичен методу `GetBalls`, который он заменяет.

← **Set-метод** почти идентичен методу `SetBalls`. Существует только одно различие: в нем используется ключевое слово `value` там, где в `SetBalls` использовался параметр. Ключевое слово `value` всегда содержит значение, присваиваемое `set`-методом.

↑ Старый метод `SetBalls` получал параметр `int` с именем `numberOfBalls`, в котором передавалось новое значение резервного поля. `Set`-метод использует ключевое слово `<value>` везде, где метод `SetBalls` использовал `numberOfBalls`.

Изменение метода Main для использования свойства Balls

Теперь, когда вы заменили методы GetBalls и SetBalls одним свойством с именем Balls, ваш код строиться не будет. Необходимо обновить метод Main, чтобы в нем использовалось свойство Balls вместо старых методов.

Метод GetBalls вызывался в команде Console.WriteLine:

```
Console.WriteLine($"{gun.GetBalls()} balls, {gun.GetBallsLoaded()} loaded");
```

Проблема решается **заменой** GetBalls() на Balls — когда это будет сделано, команда будет работать так же, как прежде. Теперь найдите другую точку, в которой использовались методы GetBalls и SetBalls:

```
else if (key == '+') gun.SetBalls(gun.GetBalls() + PaintballGun.MAGAZINE_SIZE);
```

Это та самая строка кода, такая некрасивая и громоздкая. Свойства очень полезны, потому что они работают как методы, но используются как поля. Теперь используем свойство Balls как поле — **замените эту строку** командой, в которой оператор += используется так же, как если бы свойство Balls было полем:

```
else if (key == '+') gun.Balls += PaintballGun.MAGAZINE_SIZE;
```

Обновленный метод Main выглядит так:

```
static void Main(string[] args)
{
    PaintballGun gun = new PaintballGun();
    while (true)
    {
        Console.WriteLine($"{gun.Balls} balls, {gun.GetBallsLoaded()} loaded");
        if (gun.IsEmpty()) Console.WriteLine("WARNING: You're out of ammo");
        Console.WriteLine("Space to shoot, r to reload, + to add ammo, q to quit");
        char key = Console.ReadKey(true).KeyChar;
        if (key == ' ') Console.WriteLine($"Shooting returned {gun.Shoot()}");
        else if (key == 'r') gun.Reload();
        else if (key == '+') gun.Balls += PaintballGun.MAGAZINE_SIZE;
        else if (key == 'q') return;
    }
}
```

Если бы свойство Balls было полем, то вы бы именно так использовали оператор += для его обновления. Свойства используются точно так же.

Отладка класса PaintballGun поможет понять, как работает свойство

Отладчик поможет лучше понять, как работает новое свойство Ball:

- ★ Установите точку прерывания в фигурных скобках get-метода (return balls;).
- ★ Установите другую точку прерывания в первой строке set-метода (if (value > 0)).
- ★ Установите точку прерывания в начале метода Main и начните отладку. Начните выполнение в пошаговом режиме.
- ★ При выполнении команды Console.WriteLine сработает точка прерывания в get-методе.
- ★ Продолжайте выполнение в пошаговом режиме с обходом методов. При выполнении метода += сработает точка прерывания в set-методе. Добавьте отслеживание для резервного поля balls и ключевого слова value.

Автоматически реализуемые свойства упрощают ваш код Добавьте!

Чрезвычайно распространенный сценарий использования свойств – создание резервного поля и определение get- и set-методов доступа. Создадим новое свойство BallsLoaded, которое использует **существующее поле ballsLoaded** в качестве резервного.

```
private int ballsLoaded = 0;

public int BallsLoaded {
    get { return ballsLoaded; }
    set { ballsLoaded = value; }
}
```

Это свойство использует приватное резервное поле. Его get-метод возвращает значение, содержащееся в поле, а set-метод обновляет его.

Теперь удалите метод **GetBallsLoaded** и измените метод Main, чтобы в нем использовалось свойство:

```
Console.WriteLine($"{gun.Balls} balls, {gun.BallsLoaded} loaded");
```

Снова запустите программу. Она должна работать точно так же, как и прежде.

Использование фрагмента кода для создания автоматически реализуемого свойства

Автоматически реализуемое свойство, иногда называемое **автоматическим свойством**, представляет собой свойство с get-методом, который возвращает значение резервного поля, и set-методом, который его обновляет. Иначе говоря, оно работает точно так же, как только что созданное свойство BallsLoaded. Впрочем, существует одно важное различие: при создании автоматического свойства **резервное поле не определяется**. Вместо этого компилятор C# создает резервное поле за вас, и любые операции с ним могут выполняться только через get- и set-методы.

Visual Studio предоставляет очень полезный инструмент для создания автоматических свойств: **фрагмент кода** – маленький блок кода, который IDE автоматически вставляет в вашу программу. Воспользуемся фрагментом кода для создания автоматического свойства BallsLoaded.

1 Удалите свойство BallsLoaded и резервное поле. Удалите добавленное вами свойство BallsLoaded, потому что мы заменим его автоматически реализуемым свойством. Затем удалите резервное поле ballsLoaded (private int ballsLoaded = 0;), потому что при создании автоматического свойства компилятор C# генерирует скрытое резервное поле за вас.

2 Прикажите IDE вставить фрагмент кода для свойства. Установите курсор в позицию, где находилось поле, введите **prop** и дважды нажмите клавишу **Tab**, чтобы вставить фрагмент. IDE добавит в ваш код следующую строку:

```
public int MyProperty { get; set; }
```

Фрагмент кода представляет собой шаблон, части которого можно редактировать, – фрагмент позволяет изменить тип и имя свойства. Нажмите клавишу **Tab**, чтобы переключиться на имя свойства, **измените имя на BallsLoaded** и нажмите **Enter**, чтобы завершить фрагмент кода:

```
public int BallsLoaded { get; set; }
```

Объявлять резервное поле для автоматического свойства не нужно, потому что компилятор C# создаст его автоматически.

3 Внесите исправления в остальном коде класса. Так как вы удалили поле ballsLoaded, класс PaintballGun снова не компилируется. Проблема легко решается – поле **ballsLoaded** встречается в коде пять раз (один раз в методе IsEmpty и по два раза в методах Reload и Shoot). Замените эти имена на **BallsLoaded** – программа снова работает.

Использование приватного set-метода для создания свойств, доступных только для чтения

Взгляните еще раз на только что созданное автоматическое свойство:

```
public int BallsLoaded { get; set; }
```

Оно безусловно становится хорошей заменой для свойства с get- и set-методами, которые просто обновляют резервное поле. Автоматическое свойство намного лучше читается и содержит меньше кода, чем поле ballsLoaded и метод GetBallsLoaded. Все стало лучше, верно?

Возникает только одна проблема: *мы нарушили инкапсуляцию*. Вся схема приватного поля с открытым методом создавалась для того, чтобы количество заряженных шариков было доступно только для чтения. Метод Main может легко задать свойство BallsLoaded. Мы объявили поле приватным и создали открытый метод для чтения значения, чтобы его можно было изменить только внутри класса PaintballGun.

Объявление set-метода BallsLoader приватным

К счастью, класс PaintballGun можно снова сделать хорошо инкапсулированным. Для этого достаточно поставить модификатор доступа перед ключевым словом get или set.

Чтобы свойство стало **доступным только для чтения** (т. е. его значение нельзя будет задать из другого класса), используйте в его set-методе модификатор private. Собственно, для обычных свойств set-метод можно вообще опустить, но это не относится к автоматическим свойствам, которые *должны* иметь set-метод, без которого код не будет компилироваться.

Объявим set-метод приватным:

```
public int BallsLoaded { get; private set; }
```

Чтобы сделать автоматическое свойство доступным только для чтения, объявите set-метод приватным.

Теперь поле BallsLoaded **доступно только для чтения**. Его можно прочитать откуда угодно, но обновляться оно может только из класса PaintballGun. Класс PaintballGun снова хорошо инкапсулирован.

Часть Задаваемые Вопросы

В: Мы заменили методы свойства. Существуют ли какие-то различия между тем, как работают методы и get/set-методы свойств?

О: Нет. Get- и set-методы доступа просто являются особой разновидностью методов — для других объектов они ничем не отличаются от полей и вызываются при каждом «присваивании» значения поля. Get-методы всегда возвращают значение, тип которого совпадает с типом поля. Set-метод работает точно так же, как работал бы метод с одним параметром value, тип которого соответствует типу поля.

В: Значит, свойство может содержать ЛЮБЫЕ команды?

О: Абсолютно. Все, что можно сделать в методе, можно сделать и в свойстве — в него даже можно включить сложную логику, которая делает все, что может делать обычный метод. Свойство может вызывать другие методы, обращаться к другим полям, даже создавать экземпляры объектов. Помните, что get/set-методы вызываются только при обращении к свойству, так что они должны включать лишь те команды, которые относятся к чтению/записи свойств.

В: Зачем включать сложную логику в get- или set-метод? Ведь это всего лишь еще один способ изменения полей?

О: Потому что иногда при каждом присваивании поля приходится выполнять некоторые вычисления или операции. Вспомните проблему Оуэна — она возникла из-за того, что приложение не вызвало методы SwordDamage в правильном порядке после присваивания значения Roll. Если заменить все методы свойствами, можно гарантировать, что set-методы будут правильно вычислять повреждения. (Собственно, именно это будет сделано в конце главы!)

А если потребуется изменить размер магазина?

В данный момент класс PaintballGun использует const для определения размера магазина:

```
public const int MAGAZINE_SIZE = 16;
```

← Заменить!

А если вы захотите, чтобы игра задавала размер магазина при создании экземпляра маркера? **Заменяем константу свойством.**

- 1 Удалите константу MAGAZINE_SIZE и замените ее свойством, доступным только для чтения.

```
public int MagazineSize { get; private set; }
```

- 2 Измените метод Reload, чтобы в нем использовалось новое свойство.

```
if (balls > MagazineSize)
    BallsLoaded = MagazineSize;
```

- 3 Исправьте в методе Main строку, в которой добавляются боеприпасы.

```
else if (key == '+') gun.Balls += gun.MagazineSize;
```

Но тут возникает проблема... как инициализировать MagazineSize?

Константе MAGAZINE_SIZE присваивалось значение 16. Теперь константа заменена автоматическим свойством, и при желании мы могли бы инициализировать ее значением 16, как и поле, — для этого достаточно **добавить в конец объявления команду присваивания:**

```
public int MagazineSize { get; private set; } = 16;
```

А если вы хотите, чтобы игра позволяла указать количество шариков в магазине? Вероятно, большинство экземпляров маркеров будет создаваться в заряженном виде, но на уровнях повышенной сложности некоторые маркеры могут создаваться незаряженными, чтобы игрок вынужден был перезарядить их перед стрельбой. *Как это сделать?*

Часто
Задаваемые
Вопросы

В: Можно еще раз объяснить, что делает конструктор?

О: Конструктор — метод, который вызывается при создании нового экземпляра класса. Он всегда объявляется как метод **без возвращаемого типа**, имя которого **совпадает с именем класса**. Чтобы понять, как работает конструктор, **создайте новое консольное приложение** и добавьте класс ConstructorTest с конструктором и открытым полем с именем **i**:

```
public class ConstructorTest
{
    public int i = 1;

    public ConstructorTest()
    {
        Console.WriteLine($"i is {i}");
    }
}
```

Затем **добавьте в метод Main команду new:**
new ConstructorTest();

Чтобы по-настоящему понять, как работают конструкторы, воспользуемся отладчиком.

Добавьте три точки прерывания:

- В объявлении поля (i = 1).
- В первой строке конструктора.
- На фигурной скобке } за последней строкой метода Main.

Отладчик сначала прерывается в объявлении поля, затем в конструкторе и, наконец, в конце метода Main. Как видите, ничего загадочного в этом нет — CLR сначала инициализирует поля, затем выполняет конструктор и, наконец, продолжает выполнение с той точки, где была остановка после команды new.

Использование конструктора с параметрами для инициализации свойств

Ранее в этой главе уже было показано, что объект можно инициализировать в конструкторе — специальном методе, который вызывается при создании экземпляра. Конструкторы ничем не отличаются от других методов, а следовательно, они могут получать параметры. Конструкторы с параметрами используются для инициализации свойств.

Конструктор, только что созданный нами в разделе «Часто задаваемые вопросы», выглядит так: `public ConstructorText()`. Это **конструктор без параметров**, и его объявление, как и объявление любого другого метода без параметров, завершается круглыми скобками `()`. Добавим в класс `PaintballGun` **конструктор без параметров**. Этот конструктор выглядит так:

Добавьте конструктор в класс — создайте метод, не имеющий возвращаемого типа, с таким же именем, как у класса.

Конструктор получает три параметра: `int` с именем `balls`, `int` с именем `magazineSize` и `bool` с именем `loaded`.

```
public PaintballGun(int balls, int magazineSize, bool loaded)
{
    this.balls = balls;
    MagazineSize = magazineSize;
    if (!loaded) Reload();
}
```

Конструктор выполняется сразу же после создания нового экземпляра, поэтому мы включаем в тело метода код инициализации количества шариков и размера магазина и перезарядки маркера в случае необходимости. Обратите внимание на ключевое слово `this` в первой строке. Как вы думаете, для чего оно нужно?

Ой-ой, кажется, у нас проблема. Как только вы добавляете конструктор, IDE сообщает об ошибке в методе `Main`:

CS7036 There is no argument given that corresponds to the required formal parameter 'bullets' of 'MachineGun.MachineGun(int, int, bool)'

Как вы думаете, что нужно сделать для исправления этой ошибки?



Будьте осторожны!

Если имя параметра совпадает с именем поля, параметр **замещает** поле.

Имя параметра конструктора `balls` совпадает с именем поля `balls`. При совпадении имен параметр обладает более высоким приоритетом в теле конструктора. Этот механизм называется **замещением**: когда имя параметра или переменной в методе совпадает с именем поля, при использовании в методе это имя будет обозначать переменную или параметр, но не поле. Именно этим объясняется необходимость ключевого слова `this` в конструкторе `PaintballGun`:

```
this.balls = balls;
```

Когда мы используем имя `balls`, оно обозначает параметр. Мы хотим задать значение поля, и так как оно обладает таким же именем, для обращения к нему необходимо использовать `this.balls`.

Кстати, это относится не только к конструкторам, но и к **любым** методам.

Передача аргументов при использовании ключевого слова "new"

Когда вы добавили конструктор, IDE сообщила, что метод Main содержит ошибку в команде new (PaintballGun gun = new PaintballGun()). Ошибка выглядит так:

```
CS7036 There is no argument given that corresponds to the required formal parameter 'bullets' of 'MachineGun.MachineGun(int, int, bool)'
```

Прочитайте текст ошибки — в нем точно описана суть проблемы. Ваш конструктор использует аргументы, поэтому ему должны передаваться параметры. Начните с повторного ввода команды new, и IDE точно скажет, что необходимо добавить:

```
MachineGun gun = new MachineGun()  
MachineGun(int bullets, int magazineSize, bool loaded)
```

Мы используем new для создания экземпляров классов. До сих пор у всех наших классов были конструкторы без параметров, поэтому передавать аргументы было не нужно.

Теперь у нас имеется конструктор с параметрами, и, как и любой метод с параметрами, он требует передачи аргументов, типы которых соответствуют типам параметров.

Изменим метод Main так, чтобы он **передавал параметры конструктору PaintballGun**.

- 1 **Добавьте метод ReadInt, написанный в главе 4 для калькулятора характеристик Оуэна.**

← **Изменить!**

Аргументы конструктора необходимо откуда-то получить. У нас уже имеется идеально подходящий метод, который запрашивает у пользователя значения int, поэтому его стоит использовать повторно.

- 2 **Добавьте код для чтения данных из консольного ввода.**

После добавления метода ReadInt из главы 4 мы используем его для получения двух значений int. Включите следующие четыре строки кода в начало метода Main:

```
int numberOfBalls = ReadInt(20, "Number of balls");  
int magazineSize = ReadInt(16, "Magazine size");  
  
Console.WriteLine($"Loaded [false]: ");  
bool.TryParse(Console.ReadLine(), out bool isLoading);
```

← Если метод TryParse не может разобрать содержимое строки, он оставляет isLoading значение по умолчанию (false для типа bool).

- 3 **Обновите команду new и добавьте аргументы.**

Значения хранятся в переменных, тип которых соответствует типу параметров конструктора, и мы можем обновить команду new, чтобы они передавались конструктору в аргументах:

```
PaintballGun gun = new PaintballGun(numberOfBalls, magazineSize, isLoading);
```

- 4 **Запустите программу.**

Запустите программу. Она запрашивает количество шариков, размер магазина, а также исходное состояние маркера (заряжен или нет). Затем программа создает новый экземпляр PaintballGun, передавая его конструктору аргументы, соответствующие выбранным значениям.



У бассейна

Ваша **задача** — выловить кусочки кода из бассейна и расставить их в коде.

Один фрагмент **может** использоваться многократно, использовать все фрагменты не обязательно. Ваша **цель** — создать классы, которые успешно компилируются и работают и выдают результат, совпадающий с приведенным примером.

Программа задает серию вопросов со случайными операциями сложения и умножения и проверяет ответы. Вот как она выглядит во время игры:

```
8 + 5 = 13
Right!
4 * 6 = 24
Right!
4 * 9 = 37
Wrong! Try again.
4 * 9 = 36
Right!
9 * 8 = 72
Right!
6 + 5 = 12
Wrong! Try again.
6 + 5 = 9
Wrong! Try again.
6 + 5 = 11
Right!
8 * 4 = 32
Right!
8 + 6 = Вуе
Thanks for playing!
```

Игра генерирует случайные вопросы с операциями сложения и умножения.

Если пользователь дал неправильный ответ, программа снова и снова задает вопрос, пока не будет получен правильный ответ.

Игра завершается при вводе любого ответа, который не является числом.

```
class Q {
    public Q(bool add) {
        if (add) ____ = "+";
        else ____ = "*";
        N1 = ____ . ____;
        N2 = ____ . ____;
    }

    public ____ Random R = new Random();
    public ____ N1 { get; ____ set; }
    public ____ Op { get; ____ set; }
    public ____ N2 { get; ____ set; }

    public ____ Check(int ____)
    {
        if (____ == "+") return (a ____ N1 + N2);
        else return (a ____ ____ * ____);
    }
}

class Program {
    public static void Main(string[] args) {
        Q ____ = ____ Q(____.R.____ == 1);
        while (true) {
            Console.WriteLine($"{q.____} {q.____} {q.____} = ");
            if (!int.TryParse(Console.ReadLine(), out int i)) {
                Console.WriteLine("Thanks for playing!");
                ____;
            }
            if (____.____(____)) {
                Console.WriteLine("Right!");
                ____ = ____ Q(____.R.____ == 1);
            }
            else Console.WriteLine("Wrong! Try again.");
        }
    }
}
```

Внимание:
каждый фрагмент может использоваться многократно!

Next()	a	Q			
Next(1, 10)	b	add	add		
Next(2)	c	Main	int		
Next(1, 9)	i	Op	args	class	if
Check	j	Random	bool	void	else
	k	R	string	int	new
	q	N1	double	public	return
	r	N2	float	private	while
	s	out		static	for
					foreach
					+
					*
					-
					*=
					==
					+=

Мы слегка завысили сложность этого упражнения!
Не забывайте: посмотреть в решение — не значит жульничать!

дальше ▶

задача сложная, но она вам по силам!



У бассейна. Решение

```
class Q {
    public Q(bool add) {
        if (add) Op = "+";
        else Op = "*";
        N1 = R.Next(1, 10);
        N2 = R.Next(1, 10);
    }

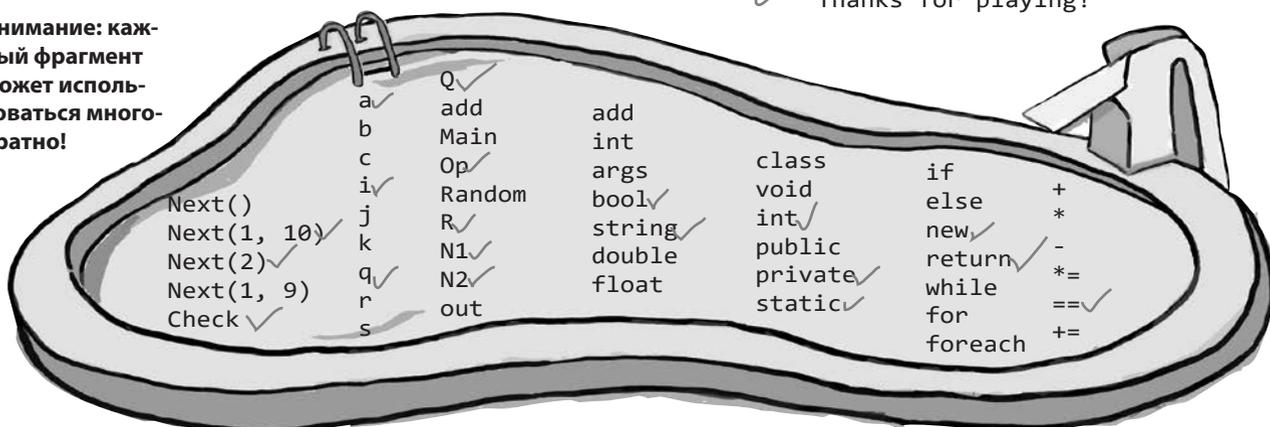
    public static Random R = new Random();
    public int N1 { get; private set; }
    public string Op { get; private set; }
    public int N2 { get; private set; }

    public bool Check(int a)
    {
        if (Op == "+") return (a == N1 + N2);
        else return (a == N1 * N2);
    }
}

class Program {
    public static void Main(string[] args) {
        Q q = new Q(Q.R.Next(2) == 1);
        while (true) {
            Console.WriteLine($"{q.N1} {q.Op} {q.N2} = ");
            if (!int.TryParse(Console.ReadLine(), out int i)) {
                Console.WriteLine("Thanks for playing!");
                return;
            }
            if (q.Check(i)) {
                Console.WriteLine("Right!");
                q = new Q(Q.R.Next(2) == 1);
            }
            else Console.WriteLine("Wrong! Try again.");
        }
    }
}
```

Мы поместили «галочкой» все фрагменты, использованные в решении. ✓

Внимание: каждый фрагмент может использоваться многократно!



Ваша **задача** — выловить кусочки кода из бассейна и расставить их в коде. Один фрагмент **может** использоваться многократно, использовать все фрагменты не обязательно. Ваша **цель** — создать классы, которые успешно компилируются и работают и выдают результат, совпадающий с приведенным примером.

Программа задает серию вопросов со случайными операциями сложения и умножения и проверяет ответы. Вот как она выглядит во время игры:

8 + 5 = 13

Right!

4 * 6 = 24

Right!

4 * 9 = 37

Wrong! Try again.

4 * 9 = 36

Right!

9 * 8 = 72

Right!

6 + 5 = 12

Wrong! Try again.

6 + 5 = 9

Wrong! Try again.

6 + 5 = 11

Right!

8 * 4 = 32

Right!

8 + 6 = **Вуе**

Thanks for playing!

Игра генерирует случайные вопросы с операциями сложения и умножения.

Если пользователь дал неправильный ответ, программа снова и снова задает вопрос, пока не будет получен правильный ответ.

Игра завершается при вводе любого ответа, который не является числом.

Несколько полезных фактов о методах и свойствах

★ Каждый метод класса имеет уникальную сигнатуру.

Первая строка метода, которая содержит модификатор доступа, возвращаемое значение, имя и параметры, называется **сигнатурой** метода. Свойства тоже обладают сигнатурами — они состоят из модификатора доступа, типа и имени.

★ Свойства могут инициализироваться в инициализаторе объекта.

Вы уже использовали инициализаторы объектов:

```
Guy joe = new Guy() { Cash = 50, Name = "Joe" };
```

Свойства также могут указываться в инициализаторе объекта. В таком случае сначала выполняется конструктор, а затем задаются значения свойств. В инициализаторе объекта могут инициализироваться только открытые поля и свойства.

★ Конструктор есть у каждого класса, даже если вы не добавили его самостоятельно.

Конструктор необходим CLR для создания экземпляра объекта — он задействован во внутренних механизмах работы .NET. Таким образом, если вы не добавили конструктор в свой класс, компилятор C# автоматически добавит конструктор без параметров.

★ Чтобы предотвратить создание экземпляров класса из других классов, добавьте приватный конструктор.

В некоторых ситуациях процесс создания объектов должен тщательно контролироваться. Один из способов заключается в том, чтобы объявить конструктор приватным — тогда он может вызываться только из этого класса. Попробуйте сами:

```
class NoNew {
    private NoNew() { Console.WriteLine("I'm alive!"); }
    public static NoNew CreateInstance() { return new NoNew(); }
}
```

Добавьте класс NoNew в консольное приложение. Если вы попытаетесь включить команду `new NoNew();` в метод Main, компилятор C# сообщает об ошибке (*'NoNew.NoNew()' недоступен из-за уровня защиты*), но метод **NoNew.CreateInstance** без малейших проблем создает новый экземпляр.

Самое время поговорить об эстетике видеоигр. Если задуматься, инкапсуляция не дает возможности сделать что-то такое, чего вы не могли сделать до этого. Те же самые программы можно написать без свойств, конструкторов и частных методов — но они будут выглядеть иначе. Дело в том, что работа по программированию не всегда направлена на то, чтобы ваш код делал что-то новое. Часто программа должна делать то же самое, но другим способом. Помните об этом, когда будете читать об эстетике. Она не влияет на поведение вашей программы, она отражается на восприятии игрового процесса игроком.



Разработка игр... и не только

Эстетика

Что вы чувствовали, когда в последний раз проводили время за игрой? Вам было интересно? Вы чувствовали волнение, приток адреналина? Возникало ли у вас ощущение открытия или достижения? Вы конкурировали с другими игроками или сотрудничали с ними? Был ли в игре увлекательный сюжет? Вам было весело? Грустно? Игры вызывают у нас эмоциональный отклик, и именно эта идея лежит в основе эстетики.

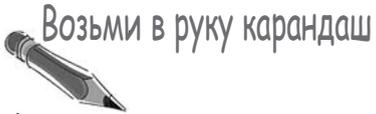
Вас удивляют разговоры о чувствах в видеоиграх? Напрасно — эмоции и чувства всегда играли важную роль в проектировании игр, и у самых успешных игр всегда присутствовал важный эстетический аспект. Вспомните то чувство глубокого удовлетворения, когда вы роняете длинную «четверку» в Тетрисе и она убирает четыре ряда блоков. Или то волнение в Pac-Man, когда вы подбираете энергетическую таблетку и поворачиваетесь к преследующим вас по пятам призракам.

- Очевидно, что на эстетику игры может влиять **художественный стиль и визуальное оформление, музыка и звук**, сюжет, но эстетика — нечто большее, чем художественные элементы игры. Эстетика может формироваться **структурированием** игры.
- Эстетика присуща не только видеоиграм, она также присутствует **в настольных играх**. Poker знаменит своими переживаниями от взлетов и падений, от удачно провернутого блефа. Даже у такой простой игры, как Go Fish!, есть своя эстетика: «перетягивание каната» в процессе того, как игроки выясняют, какие карты на руке у соперников; нарастающие эмоции, когда игрок выкладывает на стол очередную «книгу»; радость от того, что вам пришла самая нужная карта; облегчение, с которым вы произносите «Go Fish!», когда противник требует у вас отсутствующую карту.
- Иногда говорят о «геймплее», но при обсуждении эстетики стоит выражаться точнее.
- Игра **создает испытания** для игрока. Она предоставляет ему препятствия, которые нужно преодолевать, чтобы почувствовать себя победителем.
- Игровое **повествование** вовлекает игрока в драматическую сюжетную линию.
- **Тактильные ощущения** игры — игровой ритм, утробные звуки, с которыми вы подбираете энергетические таблетки, раскатистый звук и размывка ускоряющегося автомобиля — все это работает на то, чтобы вы получили удовольствие.
- Кооперативные и многопользовательские игры создают чувство **принадлежности к сообществу**.
- **Игра с элементами фантастики** не только переносит игрока в другой мир, но и позволяет ему быть совершенно другим человеком (или вообще не человеком!).
- **Игры с элементами самовыражения** помогают игроку лучше разобраться в себе, становятся способом самопознания.

Хотите верить, хотите нет, но идеи, лежащие в основе эстетики, помогут вам сделать **более общие выводы о разработке**, применимые не только к играм, но и к любым программам и приложениям. Не торопитесь и дайте этим идеям закрепиться у вас в мозгу — мы вернемся к ним в следующей главе.



Некоторые разработчики скептически относятся к обсуждениям эстетики — они считают, что важна только механика игры. Небольшой мысленный эксперимент покажет, насколько важной может быть эстетика. Допустим, имеются две игры с идентичной механикой. Между ними есть только одно крошечное различие: в одной игре вы пинаете булыжники, чтобы предотвратить лавину и спасти деревню. В другой игре вы пинаете щенков и котят просто потому, что вы ужасный человек. Даже если все остальные аспекты игры идентичны, это будут две совершенно разные игры. Такова сила эстетики!



В этом коде есть проблемы. Предполагается, что он управляет работой простого автомата по продаже жевательной резинки: вы бросаете монетку, автомат выдает жвачку. Мы обнаружили в программе четыре проблемы, из-за которых возникают ошибки. Напишите, что, по-вашему, не так в строках, на которые указывают стрелки.

```
class GumballMachine {
    private int gumballs;
```

```
    private int price;
    public int Price
    {
        get
        {
            return price;
        }
    }
```

```
    public GumballMachine(int gumballs, int price)
    {
        gumballs = this.gumballs;
        price = Price;
    }
```

```
    public string DispenseOneGumball(
        int price, int coinsInserted)
    {
        // Проверка резервного поля price
        if (this.coinsInserted >= price) {
            gumballs -= 1;
            return "Here's your gumball";
        } else {
            return "Insert more coins";
        }
    }
}
```

Возьми в руку карандаш



Решение

В этом коде есть проблемы. Мы обнаружили в программе четыре проблемы, из-за которых возникают ошибки. Ниже приведено их описание.

Имя `price`, начинающееся с буквы нижнего регистра, относится к параметру конструктора, а не к полю. Эта строка присваивает ПАРАМЕТРУ значение, возвращенное `get`-методом `Price`, но значение `Price` еще не задано, так что ничего полезного этот вызов не сделает. Если поменять местами операнды (`Price = price`), команда будет работать.

Ключевое слово «`this`» применяется не там, где нужно. Выражение `this.gumballs` относится к свойству, а имя `gumballs` относится к параметру.

Этот параметр замещает приватное поле с именем `price`, а в комментарии говорится, что метод должен проверять значение резервного поля `price`.

```
public GumballMachine(int gumballs, int price)
{
    gumballs = this.gumballs;
    price = Price;
}

public string DispenseOneGumball(int price, int coinsInserted)
```

Ключевое слово `this` используется с параметром, где ему не место. Оно должно использоваться с `price`, потому что это поле замещается параметром.

```
// Проверка резервного поля price
if (this.coinsInserted >= price) {
    gumballs -= 1;
    return "Here's your gumball";
} else {
    return "Insert more coins";
}
```

А теперь самое время выделить еще несколько минут и очень внимательно проанализировать этот код. Это распространенные ошибки, которые встречаются у многих новичков при работе с объектами. Если вы научитесь их избегать, программирование станет намного более приятным.

Часто задаваемые вопросы

В: Если конструктор является методом, то почему у него нет возвращаемого типа?

О: Конструктор не имеет возвращаемого типа, потому что каждый конструктор всегда возвращает `void`. Было бы избыточно заставлять разработчика вводить `void` в начале каждого конструктора.

В: Может ли свойство иметь `get`-метод без `set`-метода?

О: Да! Создавая свойство, для которого определен только `get`-метод, вы определяете свойство, доступное только для чтения. Например, класс `SecretAgent` может содержать открытое свойство, доступное только для чтения, с резервным полем:

```
string spyNumber = "007";
public string SpyNumber {
    get { return spyNumber; }
}
```

В: И наверняка свойство может иметь `set`-метод без `get`-метода?

О: Да, если только свойство не является автоматическим. В таком случае компилятор сообщает об ошибке («Автоматически реализуемые свойства должны иметь `get`-методы доступа»). Если вы создаете свойство с `set`-методом, но без `get`-метода, то это свойство будет доступно только для записи. Класс `SecretAgent` может использовать его для создания свойства, значения которого другие экземпляры смогут присвоить, но не смогут узнать:

```
public string DeadDrop {
    set {
        StoreSecret(value);
    }
}
```

Оба варианта — `set`-метод без `get`-метода или наоборот — могут быть чрезвычайно полезными для целей инкапсуляции.



Упражнение

Версия этого проекта для Mac доступна в приложении «Visual Studio для пользователей Mac».

Примените то, что вы узнали выше об инкапсуляции, в калькуляторе повреждений Оуэна. Измените класс `SwordDamage`, чтобы заменить поля свойствами, и добавьте конструктор. Когда это будет сделано, обновите консольное приложение, чтобы в нем использовалась новая версия класса. Наконец, исправьте приложение WPF. (Чтобы вам было проще, создайте новое консольное приложение для первых двух частей и новое приложение WPF для третьей части).

Часть 1: Измените `SwordDamage`, чтобы класс был хорошо инкапсулированным

1. Удалите поле `Roll` и замените его свойством с именем `Roll` и резервным полем с именем `roll`. `Get`-метод возвращает значение резервного поля. `Set`-метод обновляет резервное поле, а затем вызывает метод `CalculateDamage`.
2. Удалите метод `SetFlaming` и замените его свойством с именем `Flaming` и резервным полем с именем `flaming`. Новое свойство должно работать так же, как `Roll`, — `get`-метод возвращает значение резервного поля, `set`-метод обновляет его и вызывает метод `CalculateDamage`.
3. Удалите метод `SetMagic` и замените его свойством с именем `Magic` и резервным полем с именем `magic`, которое работает точно так же, как свойства `Flaming` и `Roll`.
4. Создайте автоматически реализуемое свойство с именем `Damage`. Свойство должно иметь открытый `get`-метод и приватный `set`-метод.
5. Удалите поля `MagicMultiplier` и `FlamingDamage`. Измените метод `CalculateDamage` так, чтобы он проверял значения свойств `Roll`, `Magic` и `Flaming` и выполнял все вычисления внутри метода.
6. Добавьте конструктор, который получает исходный результат броска в параметре. Так как метод `CalculateDamage` вызывается только из `set`-методов свойств и конструктора, вызывать его из другого класса не нужно. Объявите метод приватным.
7. Добавьте документацию XML во все открытые компоненты класса.

Часть 2: Измените консольное приложение, чтобы в нем использовался хорошо инкапсулированный класс `SwordDamage`

1. Создайте статический метод с именем `RollDice`, который возвращает результат броска `3d6`. Экземпляр `Random` должен храниться в статическом поле вместо переменной, чтобы он мог использоваться как методом `Main`, так и методом `RollDice`.
2. Вызовите новый метод `RollDice`, чтобы задать значения свойства `Roll` и аргумента конструктора `SwordDamage`.
3. Измените код с вызовами `SetMagic` и `SetFlaming`, чтобы вместо методов в нем использовались свойства `Magic` и `Flaming`.

Часть 3: Измените приложение WPF, чтобы в нем использовался хорошо инкапсулированный класс `SwordDamage`

1. Скопируйте код из части 1 в новое приложение WPF. Скопируйте код XAML из проекта, приведенного ранее в этой главе.
2. В коде программной части объявите поле `MainWindow.swordDamage` (и создайте его экземпляр в конструкторе):

```
SwordDamage swordDamage;
```
3. В конструкторе `MainWindow` присвойте полю `swordDamage` новый экземпляр `SwordDamage`, инициализированный случайным броском `3d6`. Затем вызовите метод `CalculateDamage`.
4. Методы `RollDice` и `Button_Click` работают точно так же, как было показано ранее в этой главе.
5. Измените метод `DisplayDamage`, чтобы в нем использовалась строковая интерполяция. Метод должен выводить ту же строку.
6. Измените обработчики события `Checked` и `Unchecked`, чтобы оба флажка использовали свойства `Magic` и `Flaming` вместо старых методов `SetMagic` и `SetFlaming`, а затем вызовите `DisplayDamage`.

Протестируйте весь код. Воспользуйтесь отладчиком или командой вывода `Debug.WriteLine` и убедитесь в том, что он **ДЕЙСТВИТЕЛЬНО** работает.



Упражнение
Решение

Теперь у Оуэна появился новый класс для вычисления повреждений. Пользоваться им намного проще, а риск ошибок снижается. Каждое свойство вычисляет повреждения заново, поэтому порядок вызова роли не играет. Ниже приведен код хорошо инкапсулированного класса `SwordDamage`.

```
class SwordDamage
{
    private const int BASE_DAMAGE = 3;
    private const int FLAME_DAMAGE = 2;

    /// <summary>
    /// Contains the calculated damage.
    /// </summary>
    public int Damage { get; private set; }
    private int roll;

    /// <summary>
    /// Sets or gets the 3d6 roll.
    /// </summary>
    public int Roll
    {
        get { return roll; }
        set
        {
            roll = value;
            CalculateDamage();
        }
    }

    private bool magic;

    /// <summary>
    /// True, если меч волшебный; false в противном случае.
    /// </summary>
    public bool Magic
    {
        get { return magic; }
        set
        {
            magic = value;
            CalculateDamage();
        }
    }

    private bool flaming;

    /// <summary>
    /// True, если меч огненный; false в противном случае.
    /// </summary>
    public bool Flaming
    {
        get { return flaming; }
        set
        {
            flaming = value;
            CalculateDamage();
        }
    }
}
```

← Так как эти константы не будут использоваться другими классами, будет разумно объявить их приватными.

← Приватный `set`-метод доступа свойства `Damage` делает его доступным только для чтения, поэтому оно не может быть перезаписано другим классом.

Свойство `Roll` с приватным резервным полем. `Set`-метод доступа вызывает метод `CalculateDamage`, который автоматически обновляет свойство `Damage`.

← Свойства `Magic` и `Flaming` работают так же, как свойство `Roll`. Все они вызывают `CalculateDamage`, так что при присваивании значения любому из них автоматически обновляется свойство `Damage`.



```

/// <summary>
/// Вычисляет повреждения в зависимости от текущих значений свойств.
/// </summary>
private void CalculateDamage()
{
    decimal magicMultiplier = 1M;
    if (Magic) magicMultiplier = 1.75M;

    Damage = BASE_DAMAGE;
    Damage = (int)(Roll * magicMultiplier) + BASE_DAMAGE;
    if (Flaming) Damage += FLAME_DAMAGE;
}

/// <summary>
/// Конструктор вычисляет повреждения для значений Magic и Flaming по умолчанию
/// и начального броска 3d6.
/// </summary>
/// <param name="startingRoll">Начальный бросок 3d6</param>
public SwordDamage(int startingRoll)
{
    roll = startingRoll;
    CalculateDamage();
}
}

```

← Все вычисления инкапсулируются внутри метода `CalculateDamage`. Все зависит только от `get`-методов доступа для свойств `Roll`, `Magic` и `Flaming`.

← Конструктор задает значение зернового поля для свойства `Roll`, после чего вызывает `CalculateDamage`, чтобы обеспечить правильность значения `Damage`.

Код метода Main консольного приложения:

```

class Program
{
    static Random random = new Random();

    static void Main(string[] args)
    {
        SwordDamage swordDamage = new SwordDamage(RollDice());
        while (true)
        {
            Console.WriteLine("0 for no magic/flaming, 1 for magic, 2 for flaming, " +
                "3 for both, anything else to quit: ");
            char key = Console.ReadKey().KeyChar;
            if (key != '0' && key != '1' && key != '2' && key != '3') return;
            swordDamage.Roll = RollDice();
            swordDamage.Magic = (key == '1' || key == '3');
            swordDamage.Flaming = (key == '2' || key == '3');
            Console.WriteLine($"{random.Rolled} {swordDamage.Roll} for {swordDamage.Damage} HP\n");
        }
    }

    private static int RollDice()
    {
        return random.Next(1, 7) + random.Next(1, 7) + random.Next(1, 7);
    }
}

```

← Будет разумно выделить бросок 3d6 в отдельный метод, потому что он вызывается из двух разных точек `Main`. Если вы воспользуетесь для его создания командой «Generate Method», IDE объявит его приватным автоматически.



Упражнение Решение

Ниже приведен код программной части для настольного приложения WPF. Код XAML остается таким же, как прежде.

Мы не предлагали вам переместить бросок 3d6 в отдельный метод. Как вы думаете, добавление метода RollDice (как в консольном приложении) упростит чтение этого кода? Или этот метод лишний? Нельзя сказать, что один ответ однозначно лучше или хуже другого. Попробуйте оба варианта и решите, какой из них вам лучше подходит.

```
public partial class MainWindow : Window
{
    Random random = new Random();
    SwordDamage swordDamage;

    public MainWindow()
    {
        InitializeComponent();
        swordDamage = new SwordDamage(random.Next(1, 7) + random.Next(1, 7)
            + random.Next(1, 7));

        DisplayDamage();
    }

    public void RollDice()
    {
        swordDamage.Roll = random.Next(1, 7) + random.Next(1, 7) + random.Next(1, 7);
        DisplayDamage();
    }

    void DisplayDamage()
    {
        damage.Text = $"Rolled {swordDamage.Roll} for {swordDamage.Damage} HP";
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        RollDice();
    }

    private void Flaming_Checked(object sender, RoutedEventArgs e)
    {
        swordDamage.Flaming = true;
        DisplayDamage();
    }

    private void Flaming_Unchecked(object sender, RoutedEventArgs e)
    {
        swordDamage.Flaming = false;
        DisplayDamage();
    }

    private void Magic_Checked(object sender, RoutedEventArgs e)
    {
        swordDamage.Magic = true;
        DisplayDamage();
    }

    private void Magic_Unchecked(object sender, RoutedEventArgs e)
    {
        swordDamage.Magic = false;
        DisplayDamage();
    }
}
```

Принятие решения о том, стоит ли выделить одну строку дублирующегося кода в отдельный метод, — хороший пример эстетики в коде. Красота в глазах смотрящего.

КЛЮЧЕВЫЕ МОМЕНТЫ

- **Инкапсуляция** повышает безопасность кода, предотвращая возможность некорректных изменений классов или иных злоупотреблений компонентами классов.
- Поля, требующие дополнительной обработки или вычислений, становятся **основными кандидатами** для инкапсуляции.
- Подумайте, как можно **некорректно использовать** поля и методы ваших классов. Объявляйте поля и методы открытыми только при необходимости.
- Использование единой схемы регистра символов в именах полей, свойств, переменных и методов упрощает чтение кода. Многие разработчики используют схему «верблюжьего регистра» (camelCase) для приватных полей или схему «регистра Pascal» (PascalCase) для открытых полей.
- **Свойство** является компонентом класса, который выглядит как поле при использовании, но при выполнении работает как метод.
- Определение **get-метода** состоит из ключевого слова get и метода, возвращающего значение свойства.
- Определение **set-метода** состоит из ключевого слова set и метода, присваивающего значение свойства. Внутри метода ключевое слово value определяет переменную, доступную только для чтения, которая содержит присваиваемое значение.
- Свойства часто читают или присваивают значение **резервного поля**, т. е. приватного поля, которое инкапсулируется за счет ограничения доступа к нему через свойство.
- **Автоматически реализуемое свойство**, иногда называемое автоматическим свойством, имеет get-метод, возвращающий значение резервного поля, и set-метод для его обновления.
- Используйте **фрагменты кода** в Visual Studio для создания автоматически реализуемых свойств; для этого введите «rgr» и дважды нажмите клавишу Tab.
- Используйте ключевое слово private для ограничения доступа к get- или set-методу. Свойство, доступное только для чтения, имеет приватный set-метод.
- При создании объекта CLR сначала **задает** значения всех полей, инициализируемых при объявлении, **выполняет** конструктор и возвращается к команде new, создавшей объект.
- Используйте **конструктор с параметрами** для инициализации свойств. Аргументы, передаваемые конструктору, задаются при использовании ключевого слова new.
- Параметр, имя которого совпадает с именем поля, **замещает** это поле. Используйте ключевое слово this для обращения к полю.
- Если вы не включите конструктор в свой класс, компилятор C# автоматически добавляет **конструктор без параметров**.
- Чтобы запретить создание экземпляров из других классов, добавьте **приватный конструктор**.

6 Наследование

Генеалогическое древо объектов

Входя в крутой поворот, я вдруг понял, что унаследовал свой велосипед от ДвухКолесного, но забыл про метод Тормоза()... В итоге двадцать шесть швов и лишение прогулок на целый месяц.



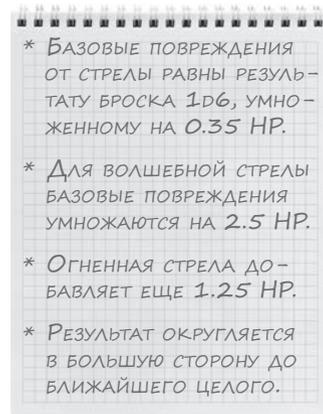
Иногда люди ХОТЯТ быть похожими на своих родителей.

Вы встречали объект, который действует почти так, как нужно? Думали ли вы о том, что при изменении всего нескольких элементов класс стал бы идеальным? **Наследование** позволяет расширять существующие классы, чтобы новый класс получал все поведение существующего, сохраняя при этом **гибкость** для внесения изменений, чтобы его можно было адаптировать под любые конкретные требования. Наследование является одним из самых мощных инструментов С#: в частности, оно помогает **избегать дублирования кода**, более адекватно **моделировать реальный мир** и в конечном итоге **упрощает сопровождение** и **снижает риск ошибок**.

Вычисление повреждений для ДРУГИХ видов оружия

← (Делайте это!)

Обновленный калькулятор повреждений от меча стал настоящим хитом! Теперь Оуэн хочет иметь калькуляторы для всех видов оружия. Начнем с вычисления повреждений для стрелы, которая использует бросок 1d6. **Создадим новый класс ArrowDamage**, который будет вычислять повреждения от стрелы по формуле из блокнота гейм-мастера Оуэна. →



Большая часть кода ArrowDamage будет *идентична коду* класса SwordDamage. Чтобы начать построение нового приложения, выполните следующие действия:

- 1 **Создайте новый проект консольного приложения .NET.** Так как вычисления должны выполняться как для мечей, так и для стрел, **добавьте класс SwordDamage** в проект.
- 2 **Создайте класс ArrowDamage, который является точной копией SwordDamage.** Создайте новый класс с именем ArrowDamage, затем **скопируйте** весь код из SwordDamage и **вставьте его** в новый класс ArrowDamage. Затем замените имя конструктора на ArrowDamage, чтобы программа нормально строилась.
- 3 **Проведите рефакторинг констант.** Формула повреждений от стрелы использует другие значения для базовых и огненных повреждений, поэтому переименуйте константу BASE_DAMAGE в BASE_MULTIPLIER и обновите значения констант. Мы считаем, что эти константы упрощают чтение кода, поэтому также добавьте константу MAGIC_MULTIPLIER:

```
private const decimal BASE_MULTIPLIER = 0.35M;
private const decimal MAGIC_MULTIPLIER = 2.5M;
private const decimal FLAME_DAMAGE = 1.25M;
```

Вы согласны с тем, что с этими константами код лучше читается? А впрочем, если не согласны — это нормально!

ArrowDamage
Roll
Magic
Flaming
Damage

- 4 **Измените метод CalculateDamage.** Чтобы новый класс ArrowDamage заработал, осталось сделать последний шаг: обновить метод CalculateDamage, чтобы он выполнял правильные вычисления:

```
private void CalculateDamage()
{
    decimal baseDamage = Roll * BASE_MULTIPLIER;
    if (Magic) baseDamage *= MAGIC_MULTIPLIER;
    if (Flaming) Damage = (int)Math.Ceiling(baseDamage + FLAME_DAMAGE);
    else Damage = (int) Math.Ceiling(baseDamage);
}
```

Метод Math.Ceiling может использоваться для округления значений в большую сторону. Тип при этом сохраняется, так что значение нужно будет привести к типу int.



Код для выполнения некоторой операции можно написать разными способами. Сможете ли вы предложить другой способ написания кода, вычисляющего повреждения от стрелы?

Команды switch для выбора из нескольких кандидатов

Обновим консольное приложение, чтобы оно запрашивало у пользователя, для какого оружия должны вычисляться повреждения — для меча или для стрелы. Программа предлагает нажать клавишу и использует статический метод `Char.ToUpper` для преобразования к верхнему регистру:

```
Console.WriteLine("\nS for sword, A for arrow, anything else to quit: ");
weaponKey = Char.ToUpper(Console.ReadKey().KeyChar);
```

Метод `Char.ToUpper` преобразует 's' и 'a' в 'S' и 'A'.

Теоретически для этой цели *можно* воспользоваться набором команд `if/else`:

```
if (weaponKey == 'S') { /* вычисление повреждений от меча */ }
else if (weaponKey == 'A') { /* вычисление повреждений от стрелы */ }
else return;
```

Так мы обрабатывали вводимые данные до настоящего момента. Сравнение одной переменной со многими разными значениями — довольно распространенный паттерн, который встречается снова и снова. Он встречается настолько часто, что `C#` содержит специальную разновидность команды, предназначенную *конкретно для этой ситуации*. Команда `switch` позволяет сравнить одну переменную со многими значениями в компактном простом синтаксисе. Следующая команда `switch` делает абсолютно то же самое, что делают приведенные выше команды `if/else`:

```
switch (weaponKey)
```

Сначала идет ключевое слово `switch`, за которым следует то, что должно сравниваться с разными возможными значениями.

```
{
```

Тело команды `switch` представляет собой набор секций, в которых проверяемое значение сравнивается с конкретными вариантами. Каждая проверка начинается с ключевого слова `case`, конкретного проверяемого значения и двоеточия и завершается командой `break`.

```
case 'S':
```

```
/* вычислить повреждения от меча */
```

```
break;
```

```
case 'A':
```

```
/* вычислить повреждения от стрелы */
```

```
break;
```

```
default:
```

```
return;
```

```
}
```

Ключевое слово `default` можно сравнить с последней командой `else` в конце серии команд `if/else`. Эта секция выполняется в том случае, если ни один из предыдущих вариантов не подошел.



Упражнение

Обновите метод `Main` так, чтобы в нем использовалась команда `switch` для выбора типа оружия. Для начала скопируйте методы `Main` и `RollDice` из ответа к упражнению в конце предыдущей главы.

1. Создайте экземпляр `ArrowDamage` в начале метода, сразу же после создания экземпляра `SwordDamage`.
2. Измените метод `RollDice`, чтобы он получал параметр `int` с именем `numberOfRolls`. Таким образом, при вызове `RollDice(3)` будет моделироваться бросок 3d6 (т. е. программа вызывает `random.Next(1,7)` три раза и суммирует результаты), а при вызове `RollDice(1)` будет моделироваться бросок 1d6.
3. Добавьте две строки кода так, как они приведены выше, прочитайте ввод методом `Console.ReadKey`, преобразуйте символ к верхнему регистру `Char.ToUpper` и сохраните его в `weaponKey`.
4. Добавьте команду `switch`. Она должна полностью совпадать с командой `switch`, приведенной выше, кроме того что каждый из комментариев `/* комментарий */` необходимо заменить кодом, который вычисляет повреждения и выводит строку выходных данных на консоль.



Упражнение
Решение

Мы только что познакомили вас с совершенно новым синтаксисом C# — командой `switch` — и предложили использовать его в программе. Группа разработки C# в Microsoft постоянно совершенствует язык, и интеграция новых элементов языка в код **относится к числу важных навыков C#**.

```
class Program
{
    static Random random = new Random();

    static void Main(string[] args)
    {
        SwordDamage swordDamage = new SwordDamage(RollDice(3));
        ArrowDamage arrowDamage = new ArrowDamage(RollDice(1));

        while (true)
        {
            Console.Write("0 for no magic/flaming, 1 for magic, 2 for flaming, " +
                "3 for both, anything else to quit: ");
            char key = Console.ReadKey().KeyChar;
            if (key != '0' && key != '1' && key != '2' && key != '3') return;

            Console.Write("\nS for sword, A for arrow, anything else to quit: ");
            char weaponKey = Char.ToUpper(Console.ReadKey().KeyChar);

            switch (weaponKey)
            {
                case 'S':
                    swordDamage.Roll = RollDice(3);
                    swordDamage.Magic = (key == '1' || key == '3');
                    swordDamage.Flaming = (key == '2' || key == '3');
                    Console.WriteLine(
                        $"{"\nRolled {swordDamage.Roll} for {swordDamage.Damage} HP\n");
                    break;
                case 'A':
                    arrowDamage.Roll = RollDice(1);
                    arrowDamage.Magic = (key == '1' || key == '3');
                    arrowDamage.Flaming = (key == '2' || key == '3');
                    Console.WriteLine(
                        $"{"\nRolled {arrowDamage.Roll} for {arrowDamage.Damage} HP\n");
                    break;
                default:
                    return;
            }
        }

        private static int RollDice(int numberOfRolls)
        {
            int total = 0;
            for (int i = 0; i < numberOfRolls; i++) total += random.Next(1, 7);
            return total;
        }
    }
}
```

Создание экземпляра только что созданного класса ArrowDamage.



Этот блок кода почти идентичен программе из последней главы. Просто на этот раз он не используется в блоке `if/else`, а был преобразован в секцию `case` команды `switch` (кроме того, в нем передается аргумент при вызове `RollDice`).



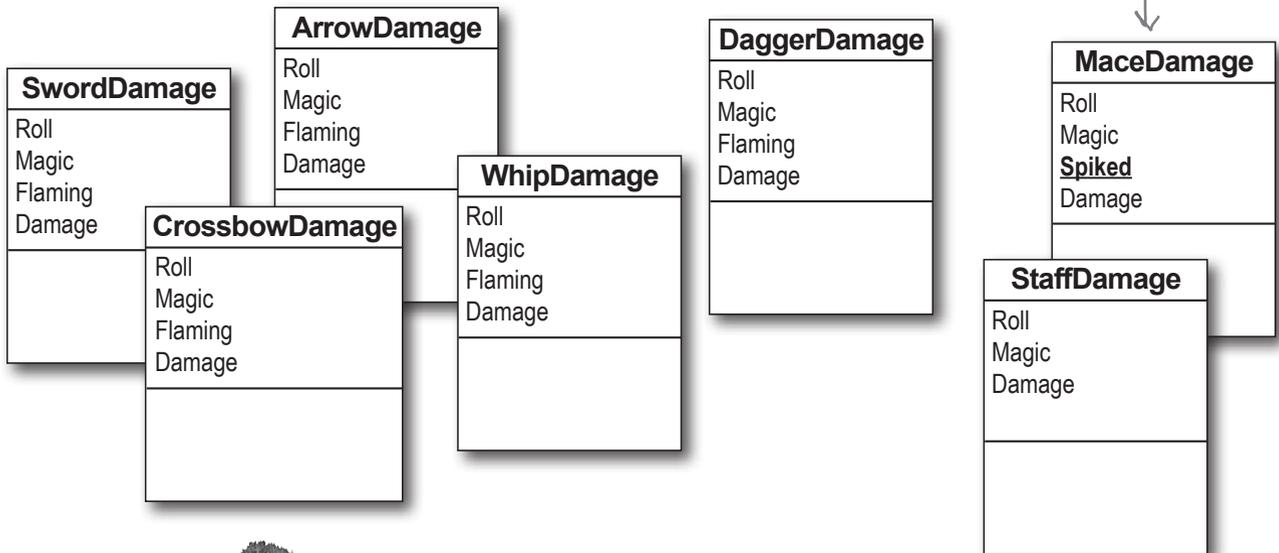
Код, использующий экземпляр `ArrowDamage` для вычисления повреждений, очень похож на код `SwordDamage`. Более того, они почти идентичны. Можно ли как-то сократить дублирование кода и упростить чтение программы?

Попробуйте сами! Установите точку прерывания в команде `switch (weaponKey)`, после чего воспользуйтесь отладчиком для пошагового выполнения команды `switch`. Это поможет вам лучше разобраться в происходящем. Затем попробуйте удалить одну из строк `break` — выполнение продолжится в следующей секции `case` (это называется сквозной передачей управления).

И еще... Можно ли вычислять повреждения от кинжала? От булавы? И шеста? И...

Мы создали два класса для вычисления повреждений от стрелы и меча. Но что, если в игре есть еще три вида оружия? Или четыре? Или двенадцать? А если вам придется заниматься сопровождением этого кода и позднее потребуется вносить новые изменения? Что, если **одно и то же изменение** нужно будет вносить в пяти или шести **тесно связанных** классах? Что, если изменения продолжатся? Ошибки становятся фактически неизбежными — очень легко обновить пять классов, но забыть о шестом.

А если некоторые классы имеют много общего, но не совсем идентичны? Что, если булава может быть шипастой или обычной, но огненной быть не может? Или если шест не может обладать ни одной из этих характеристик?



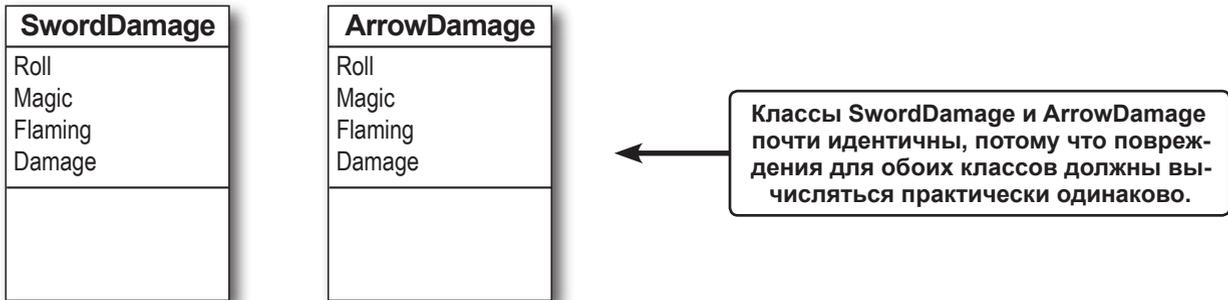
Ого, мне приходится писать один и тот же код снова и снова. Я РАБОТАЮ КРАЙНЕ НЕЭФФЕКТИВНО. Должен быть более эффективный способ.

Верно! Повторение одного кода в нескольких классах неэффективно и ненадежно.

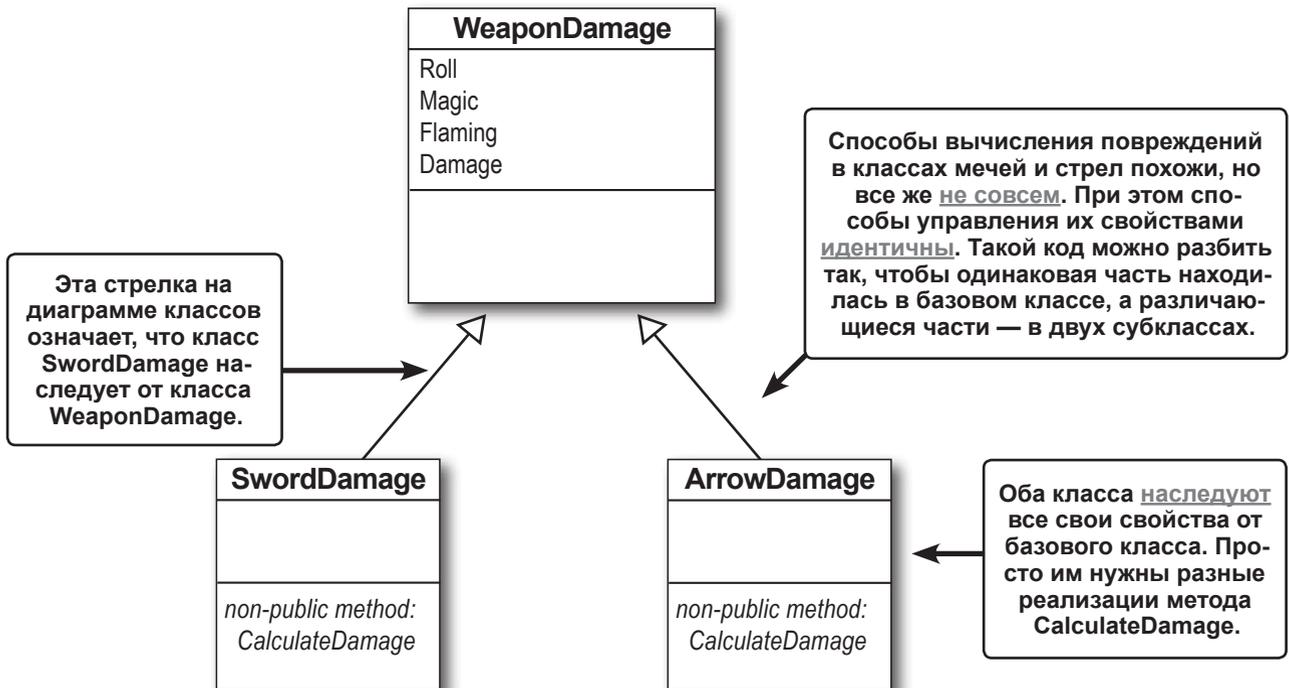
К счастью, C# предоставляет более эффективный способ построения классов, которые связаны друг с другом и обладают общим поведением: наследование.

Если в ваших классах используется наследование, код достаточно написать только один раз

То, что ваши классы `SwordDamage` и `ArrowDamage` содержат много совпадающего кода, — не случайность. При написании программ `C#` вы часто создаете классы, представляющие реальные сущности, и эти сущности обычно связаны друг с другом. Ваши классы содержат **похожий код**, потому что сущности, которые они представляют в реальном мире (два сходных вычисления в ролевой игре), обладают похожим поведением.

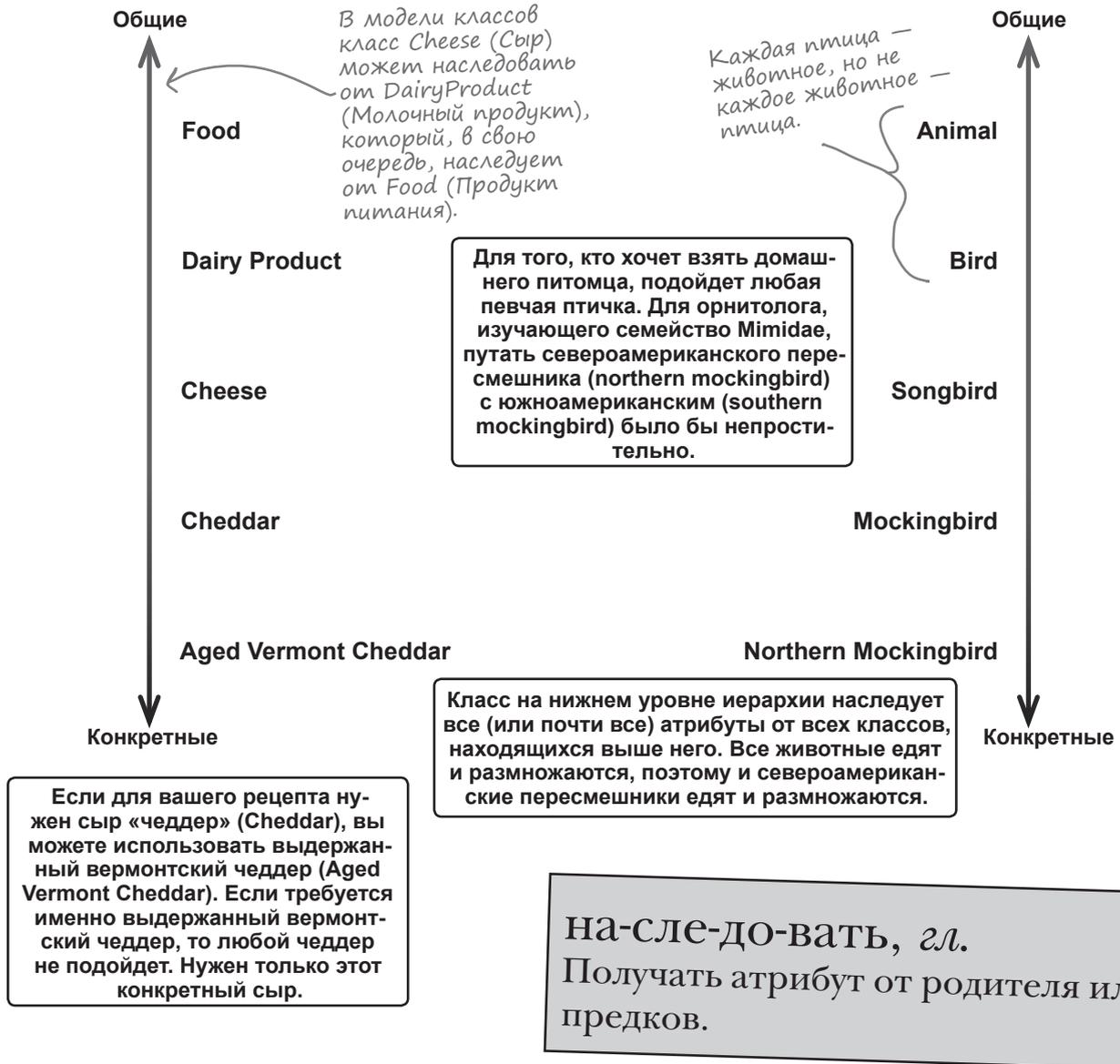


Классы `SwordDamage` и `ArrowDamage` почти идентичны, потому что в обоих случаях вычисления производятся по похожей схеме.



Постройте модель классов: начните с общего и переходите к конкретике

Когда вы строите набор классов, представляющих разные сущности (особенно сущности из реального мира), результатом вашей работы станет **модель классов**. Сущности из реального мира часто образуют **иерархию**, которая идет от общего к конкретному, и в ваших программах создаются **иерархии классов**, которые делают то же самое. В модели классов классы, находящиеся на низких уровнях иерархии, **наследуют** от классов более высоких уровней.



Как бы вы спроектировали симулятор зоопарка?

Львы, тигры и медведи... голова идет кругом! А также гиппопотамы, волки и даже собаки. Вам поручено спроектировать приложение, которое моделирует зоопарк. (Только не увлекайтесь — мы не будем строить код, а ограничимся классами, представляющими животных. А вы уже наверняка подумали, как это будет делаться в Unity!)

Вы получили список некоторых животных, которые будут задействованы в программе (но не всех!). Мы знаем, что каждое животное будет представлено объектом, а объекты будут перемещаться в рамках модели и делать то, что запрограммировано делать каждое животное.

Но что еще важнее, программа должна быть простой в сопровождении для других программистов; это означает, что они должны иметь возможность добавить свои классы позднее, если потребуется включить в симулятор новый вид животных.

Начнем с построения модели классов для животных, которые уже известны.

Каким же будет первый шаг? Прежде чем говорить о **конкретных** животных, необходимо вычислить то **общее**, что есть у каждого животного, — абстрактные характеристики, которыми обладают **все** животные. Затем на основании этих характеристик будет сформирован базовый класс, от которого могут наследовать все классы животных.

Термины «родитель», «суперкласс» и «базовый класс» часто используются как синонимы. Кроме того, термины «расширять» и «наследовать от» означают одно и то же. Термины «дочерний класс» и «субкласс» тоже являются синонимами, но «субкласс» также используется в глаголе «субклассировать».



Некоторые разработчики называют «базовым классом» класс, находящийся на вершине иерархии наследования, но... не на САМОЙ вершине, потому что каждый класс в конечном итоге наследует от Object или субкласса Object.

1 Постарайтесь выявить характеристики, общие для всех животных.



Симулятор зоопарка включает сторожевую собаку, которая охраняет территорию парка и охраняет животных.

2

Построение базового класса для общих характеристик всех животных.

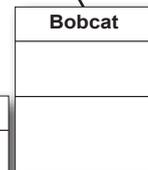
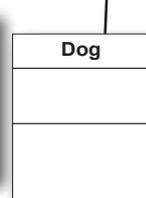
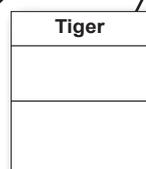
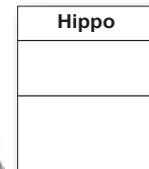
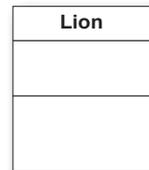
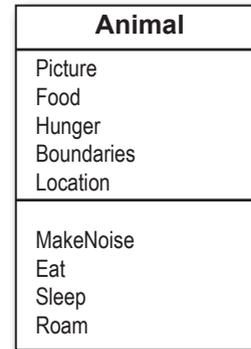
Поля, свойства и методы базового класса дадут всем животным возможность наследовать общее состояние и поведение. Логично, что этот класс должен называться `Animal` (Животное).

Так как дублирующийся код сложно редактировать и еще сложнее читать, выберем методы и поля для базового класса `Animal`, которые будут написаны только один раз и которые будут унаследованы всеми производными классами. Начнем с полей общего доступа:

- ★ `Picture`: картинка, которую можно поместить в `PictureBox`.
- ★ `Food`: тип пищи. Пока у этого поля только два значения: `meat` (мясо) и `grass` (травы).
- ★ `Hunger`: переменная типа `int`, показывающая, насколько животное хочет есть. Она меняется в зависимости от количества выданного корма.
- ★ `Boundaries`: ссылка на класс, в котором хранится информация о высоте, длине и расположении вольера.
- ★ `Location`: координаты X и Y, описывающие местоположение животного.

Кроме того, в классе `Animal` присутствуют четыре метода, которые могут быть унаследованы:

- ★ `MakeNoise()`: метод, позволяющий издавать звуки.
- ★ `Eat()`: поведение при получении предпочитаемого корма.
- ★ `Sleep()`: метод, заставляющий животное спать.
- ★ `Roam()`: метод, учитывающий перемещения по вольеру.



Можно было сделать и другой выбор. Например, написать класс `ZooOccipant`, учитывающий расходы на содержание животных, или класс `Attraction`, показывающий привлекательность для посетителей. Но мы остановились на классе `Animal`. Вы согласны?



У разных животных разное поведение

Львы рычат, собаки лают, а бегемоты, насколько мы знаем, вообще не издают конкретных звуков. Каждый класс, производный от `Animal`, унаследует метод `MakeNoise()`, но коды этих методов будут различаться. Когда производный класс меняет поведение унаследованного метода, говорят о перекрытии (`override`).

Даже если свойство или метод принадлежат базовому классу `Animal`, это не значит, что все subclasses должны использовать их одинаково... и вообще использовать хоть как-то!

3 Определите, что каждое животное делает не так, как базовый класс `Animal`, или не делает вообще.

Каждое животное должно есть, но собака может питаться маленькими кусочками мяса, а гиппопотам — огромными охапками сена. Как будет выглядеть код для такого поведения? И собака, и гиппопотам переопределяют метод `Eat`. У гиппопотама переопределенная версия будет поглощать, скажем, 10 килограммов сена при каждом вызове. С другой стороны, у собаки переопределенная реализация `Eat` будет уменьшать запас продуктов в зоопарке на одну 400-граммовую банку собачьих консервов.

Итак, если у вас имеется subclass, наследующий от базового класса, он должен наследовать все поведение базового класса... Но вы можете изменить его в subclasse, чтобы методы не работали абсолютно одинаково. Собственно, в этом и заключается суть переопределения.



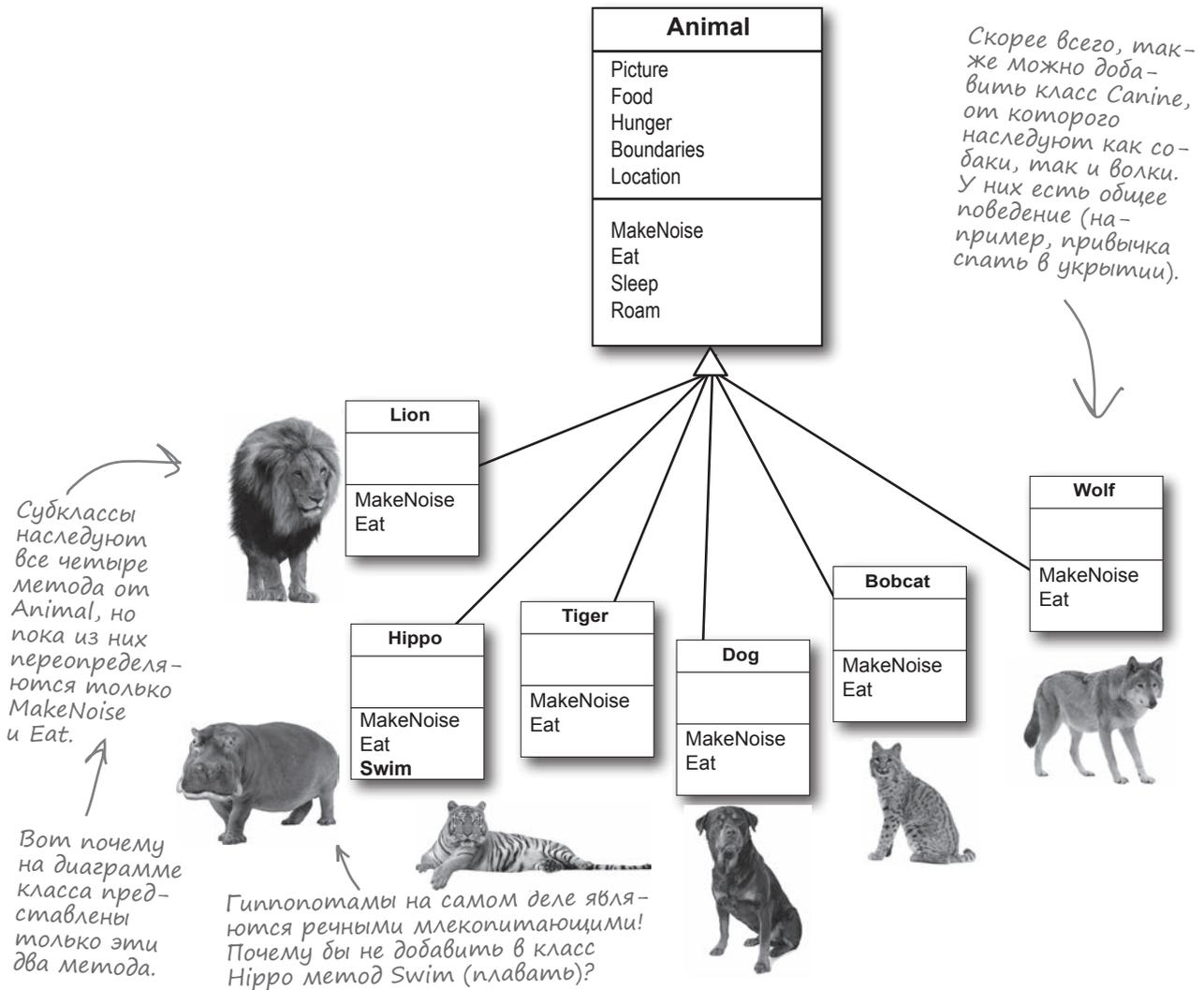
Animal
Picture
Food
Hunger
Boundaries
Location
MakeNoise
Eat
Sleep
Roam

МОЗГОВОЙ ШТУРМ

Мы уже знаем, что некоторые животные переопределяют методы `MakeNoise` и `Eat`. Какие животные будут переопределять методы `Sleep` (Спать) или `Roam` (Бродить)? И будут ли?

4 Определите, какие классы имеют много общего.

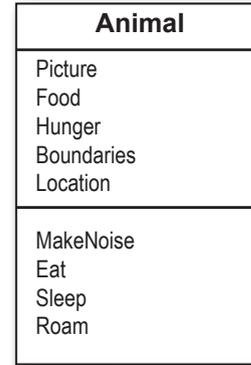
У собак и волков много общего, вы не находите? Оба вида относятся к семейству собачьих, и можно довольно уверенно утверждать, что в их поведении тоже найдется немало сходства. Вероятно, они едят похожую еду и спят приблизительно в одном режиме. А как насчет рысей, тигров и львов? Оказывается, все три вида животных перемещаются в своей среде обитания практически одинаково. Можно с большой уверенностью утверждать, что в иерархию классов можно включить общий класс *Feline*, который существует между *Animal* и этими тремя разновидностями кошачьих; дополнительный класс помогает предотвратить дублирование кода.



5 Завершите свою иерархию классов.

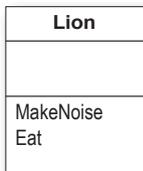
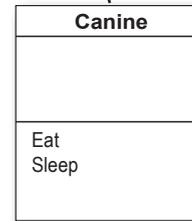
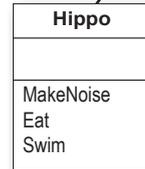
Теперь вы знаете, как упорядочить классы животных, и мы можем добавить классы Feline и Canine.

Когда вы создаете свои классы так, что на вершине иерархии находится базовый класс, а под ним более конкретные subclasses, а у этих subclasses будут свои subclasses, наследующие от них, результат ваших усилий называется **иерархией классов**. И дело даже не в устранении дублирующегося кода, хотя безусловно, это огромное преимущество разумной иерархии. Одно из преимуществ — код становится намного более простым для понимания и сопровождения. Когда вы просматриваете код симулятора зоопарка и видите метод или свойство, определенные в классе Feline, вы сразу же видите, что эта функциональность является общей для всех кошачьих. Ваша иерархия становится картой, по которой можно ориентироваться в логике программы.

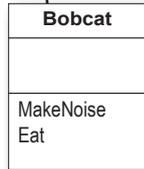
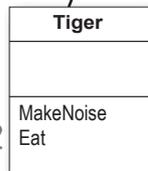


Волки и собаки едят одинаково, поэтому мы переместили их общий метод Eat в класс Canine.

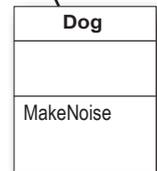
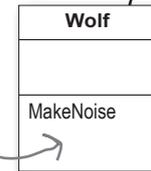
Так как Feline переопределяет Roam, все классы, наследующие от Feline, также получают обновленную реализацию Roam (вместо реализации из Animal).



Три разновидности кошачьих бродят одинаково, поэтому они совместно используют унаследованный метод Roam. При этом каждый вид по-разному ест и издает разные звуки, так что все они переопределяют методы Eat и MakeNoise, наследуемые от Animal.



Объекты Wolf и Dog обладают похожим поведением сна и еды, но издают разные звуки.



Каждый субкласс расширяет свой базовый класс

Ваши возможности не ограничиваются методами, наследуемыми субклассом от базового класса... но вы это уже знаете! В конце концов, вы уже создавали собственные классы. При изменении класса, в результате которого он наследует компоненты (вскоре вы увидите, как это делается в коде C#!), вы берете уже построенный класс и *расширяете* его, добавляя в него все поля, свойства и методы из базового класса. Таким образом, если вы захотите добавить метод Fetch в класс Dog, это вполне нормально. Метод будет существовать только в классе Dog, а в классы Wolf, Canine, Animal, Hippo или любые другие классы он не попадет.

СОЗДАЕТ НОВЫЙ ЭКЗЕМПЛЯР DOG

```
Dog spot = new Dog();
```

ВЫЗЫВАЕТ ВЕРСИЮ ИЗ КЛАССА DOG

```
spot.MakeNoise();
```

ВЫЗЫВАЕТ ВЕРСИЮ ИЗ КЛАССА ANIMAL

```
spot.Roam();
```

ВЫЗЫВАЕТ ВЕРСИЮ ИЗ КЛАССА CANINE

```
spot.Eat();
```

ВЫЗЫВАЕТ ВЕРСИЮ ИЗ КЛАССА CANINE

```
spot.Sleep();
```

ВЫЗЫВАЕТ ВЕРСИЮ ИЗ КЛАССА DOG

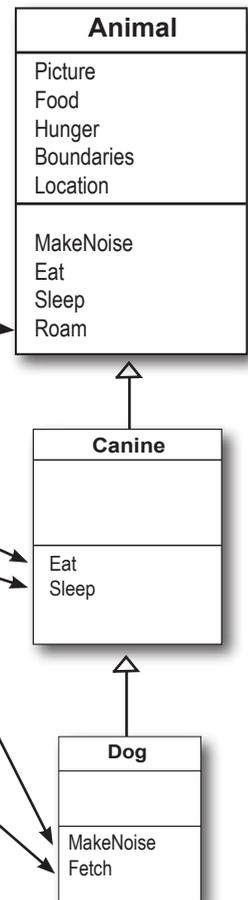
```
spot.Fetch();
```

C# всегда вызывает самый конкретный метод

Если вы вызовете метод Roam для объекта Dog, то вызван может быть только один метод — реализация из класса Animal. А если вызвать MakeNoise? Какая из реализаций будет вызвана?

На самом деле это определяется достаточно легко. Метод класса Dog определяет, как собаки издают звуки. Если метод находится в классе Canine, он определяет поведение, общее для всех представителей семейства собачьих. Если метод находится в Animal, он описывает поведение настолько общее, что оно применимо к любому животному. Итак, если вы берете объект Dog и вызываете для него MakeNoise, сначала C# ищет в классе Dog поведение, относящееся непосредственно к собакам. Если в Dog нет метода MakeNoise, то проверяется метод Canine, а после него будет проверен метод Animal.

и-е-рар-хи-я, суц.
структура или система классификации, в которой группы или отдельные предметы ранжируются по вертикали относительно друг друга.



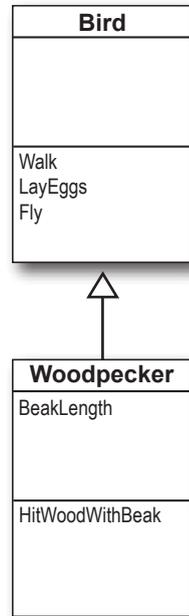
В любом месте, где может использоваться базовый класс, вместо него можно использовать один из subclasses

Одна из самых полезных возможностей наследования — **расширение** класса. Если ваш метод получает объект `Bird` (Птица), то вместо него можно передать экземпляр `Woodpecker` (Дятел). Методу известно только то, что это птица. Он не знает конкретной разновидности птицы, и поэтому может выполнять только операции, общие для всех птиц: ходить, откладывать яйца и т. д., но не долбить дерево клювом, потому что это поведение присуще только дятлам, — метод не знает, что экземпляр представляет именно дятла, а знает лишь то, что это более общий класс `Bird`. **Для него доступны только поля, свойства и методы, являющиеся частью известного ему класса.**

Посмотрим, как это работает в коде. Перед вами метод, получающий ссылку на `Bird`:

```
public void IncubateEggs(Bird bird)
{
    bird.Walk(incubatorEntrance);
    Egg[] eggs = bird.LayEggs();
    AddEggsToHeatingArea(eggs);
    bird.Walk(incubatorExit);
}
```

Даже если передать `IncubateEggs` объект `Woodpecker`, ссылка все равно интерпретируется как ссылка на `Bird`, поэтому использоваться могут только компоненты класса `Bird`.



Например, методу `IncubateEggs` можно передать ссылку на `Woodpecker`, потому что дятел является частным случаем птицы, именно поэтому класс `Woodpecker` наследует от `Bird`:

```
public void GetWoodpeckerEggs()
{
    Woodpecker woody = new Woodpecker();
    IncubateEggs(woody);
    woody.HitWoodWithBeak();
}
```

Это должно быть понятно на интуитивном уровне. Если кто-то потребует у вас птицу и вы предложите ему дятла, то никаких претензий быть не должно. Но если у вас требуют дятла, а вы выдадите голубя, вас не поймут.

Суперкласс может заменяться subclassом, но не наоборот — subclass не может заменяться суперклассом. `Woodpecker` можно передать методу, получающему ссылку на `Bird`, но не наоборот:

```
public void GetWoodpeckerEggs_Take_Two()
{
    Woodpecker woody = new Woodpecker();
    woody.HitWoodWithBeak();

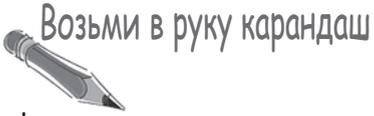
    // Эта строка копирует ссылку Woodpecker в переменную Bird
    Bird birdReference = woody;
    IncubateEggs(birdReference);

    // В СЛЕДУЮЩЕЙ СТРОКЕ ПРОИСХОДИТ ОШИБКА КОМПИЛЯЦИИ!!!
    Woodpecker secondWoodyReference = birdReference;

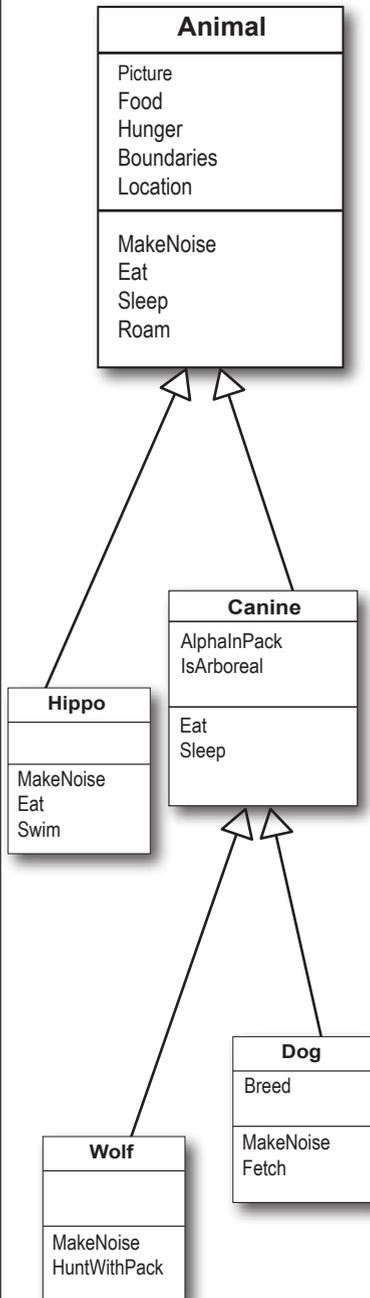
    secondWoodyReference.HitWoodWithBeak();
}
```

`woody` можно присвоить переменной `Bird`, потому что дятел является частным случаем птицы...

...но `birdReference` невозможно присвоить обратно переменной `Woodpecker`, потому что не каждая птица является дятлом! Этим объясняется ошибка в этой строке.



Приведенный ниже код взят из программы, использующей модель классов с классами Animal, Hippo, Canine, Wolf и Dog. Зачеркните все команды, которые не будут компилироваться, и кратко опишите суть проблемы для каждого случая.



```

Canine canis = new Dog();
Wolf charon = new Canine();
charon.IsArboreal = false;
Hippo bailey = new Hippo();
bailey.Roam();
bailey.Sleep();
bailey.Swim();
bailey.Eat();

```

```

Dog fido = canis;
Animal visitorPet = fido;
Animal harvey = bailey;
harvey.Roam();
harvey.Swim();
harvey.Sleep();
harvey.Eat();

```

```

Hippo brutus = harvey;
brutus.Roam();
brutus.Sleep();
brutus.Swim();
brutus.Eat();

```

```

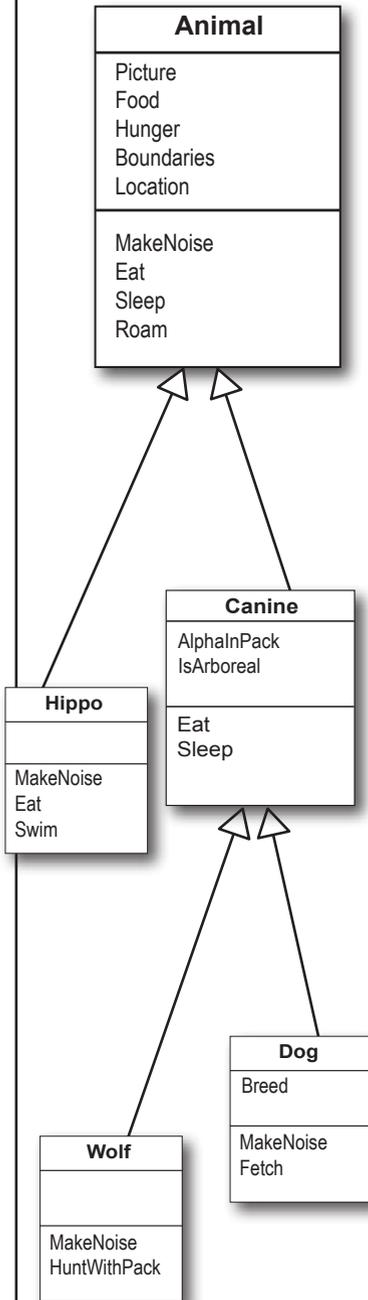
Canine london = new Wolf();
Wolf egypt = london;
egypt.HuntWithPack();
egypt.HuntWithPack();
egypt.AlphaInPack = false;
Dog rex = london;
rex.Fetch();

```

Возьми в руку карандаш
Решение



Шесть из следующих команд не будут компилироваться, потому что они конфликтуют с моделью классов. Вы можете убедиться в этом самостоятельно! Постройте свою версию модели классов с пустыми методами, введите код и прочитайте сообщения об ошибках компилятора.



```

Canine canis = new Dog();
Wolf charon = new Canine();
charon.IsArboreal = false;
Hippo bailey = new Hippo();
bailey.Roam();
bailey.Sleep();
bailey.Swim();
bailey.Eat();
    
```

Wolf является субклассом Canine, так что объект Canine нельзя присвоить переменной Wolf. Взгляните на это так: волк является частным случаем семейства собачьих, но не каждый представитель этого семейства является волком.

```

Dog fido = canis;
Animal visitorPet = fido;
Animal harvey = bailey;
harvey.Roam();
harvey.Swim();
harvey.Sleep();
harvey.Eat();
    
```

Хотя переменная canis является ссылкой на объект Dog, переменная относится к типу Canine, так что присвоить ее Dog нельзя.

harvey — ссылка на объект Hippo, но переменная harvey относится к типу Animal, так что она не может использоваться для вызова метода Hippo.Swim.

```

Hippo brutus = harvey;
brutus.Roam();
brutus.Sleep();
brutus.Swim();
brutus.Eat();
    
```

Не работает по той же причине, по которой не работает Dog fido = canis; harvey может указывать на объект Hippo, но переменная относится к типу Animal, а переменную Animal нельзя присвоить переменной Hippo.

```

Canine london = new Wolf();
Wolf egypt = london;
egypt.HuntWithPack();
egypt.HuntWithPack();
egypt.AlphaInPack = false;
Dog rex = london;
rex.Fetch();
    
```

Та же проблема! Wolf можно присвоить Canine, но Canine нельзя присвоить Wolf...

...и разумеется, Wolf нельзя присвоить Dog.



Все это здорово, конечно... теоретически.
Но как это поможет в моем приложении-
калькуляторе?

МОЗГОВОЙ ШТУРМ

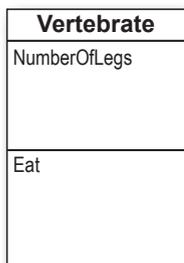
Оуэн задал очень хороший вопрос. Вернитесь к приложению, которое вы построили для Оуэна, — калькулятору повреждений от меча и стрелы. Как бы вы использовали наследование и субклассы для улучшения кода? (Внимание, спойлер: мы займемся этим позднее, читайте дальше.)

КЛЮЧЕВЫЕ МОМЕНТЫ

- Команда `switch` позволяет сравнить одну переменную с несколькими возможными вариантами. Код блока выполняется при совпадении значения. Если ни один вариант не подходит, выполняется блок `default`.
- **Наследование** используется для построения классов, связанных друг с другом и обладающих общим поведением. На диаграммах классов наследование обозначается линиями со стрелками.
- Если имеются два класса, которые являются **конкретными** частными случаями чего-то более **общего**, их можно определить как наследующие от одного базового класса. В таком случае каждый из классов становится **субклассом** общего **базового класса**.
- Набор классов, представляющих различные сущности, называется **моделью классов**. Модель может включать классы, образующие **иерархию** субклассов и базового класса.
- Термины **«родитель»**, **«суперкласс»** и **«базовый класс»** часто используются как синонимы. Кроме того, термины **«расширять»** и **«наследовать от»** означают одно и то же.
- Термины **«дочерний класс»** и **«субкласс»** также являются синонимами. Мы говорим, что субкласс **расширяет** свой базовый класс.
- Когда субкласс изменяет поведение одного из унаследованных методов, мы говорим, что он **переопределяет** метод.
- `C#` всегда вызывает **наиболее конкретную версию метода**. Если метод базового класса использует метод или свойство, переопределенные в субклассе, то будет вызвана переопределенная версия из субкласса.
- Всегда используйте **ссылку на субкласс** вместо базового класса. Если метод получает параметр `Animal`, а `Dog` расширяет `Animal`, методу можно передать аргумент `Dog`.
- Субкласс всегда может использоваться **вместо базового класса**, от которого он наследует; но базовый класс далеко не всегда может использоваться вместо расширяющего его субкласса.

Расширение базового класса

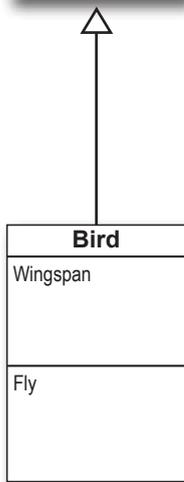
Чтобы объявить, что определяемый класс наследует от базового класса, используйте двоеточие (:). Класс становится субклассом и получает **все поля, свойства и методы** от класса, от которого он наследует. Класс Bird является субклассом Vertebrate:



```

class Vertebrate
{
    public int Legs { get; set; }

    public void Eat() {
        // код питания
    }
}
    
```



```

class Bird : Vertebrate
{
    public double Wingspan;
    public void Fly() {
        // code to make a bird fly
    }
}
    
```

Двоеточие в классе Bird означает, что класс наследует от класса Vertebrate. Это означает, что он наследует все поля, свойства и методы от Vertebrate.

Базовый класс указывается за двоеточием в объявлении класса. В данном случае Bird расширяет Vertebrate.



```

public void Main(string[] args) {
    Bird tweety = new Bird();
    Console.WriteLine(tweety.Wingspan);
    tweety.Fly();
    tweety.Legs = 2;
    Console.Write(tweety.Eat());
}
    
```

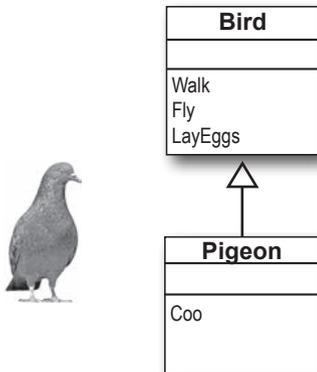
tweety является экземпляром Bird и поэтому содержит методы, свойства и поля Bird.

Так как класс Bird расширяет Vertebrate, каждый экземпляр Bird также содержит компоненты, определенные в классе Vertebrate.

Субкласс, расширяющий базовый класс, наследует все его компоненты: все поля, свойства и методы, содержащиеся в базовом классе. Все они автоматически добавляются в субкласс.

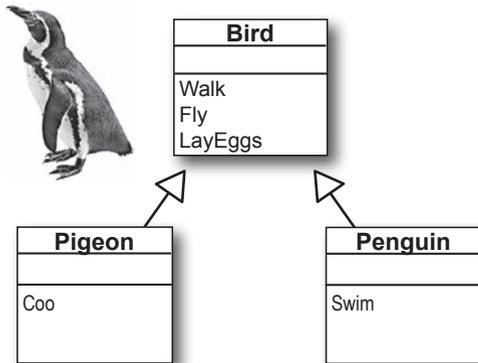
Мы знаем, что наследование добавляет в subclass поля, свойства и методы базового класса...

Вы уже сталкивались с наследованием, когда subclass должен унаследовать все методы, свойства и поля базового класса.



...но некоторые птицы не летают!

Что делать, если базовый класс содержит метод, который должен быть изменен вашим subclassом?



Кажется, у нас проблема. Пингвины — птицы, а класс Bird содержит метод Fly, но пингвины летать не должны. Было бы замечательно, если бы при вызове Fly для пингвина выводилось предупреждение.

```

class Bird {
    public void Fly() {
        /* Код реализации полетов
    }
    public void LayEggs() { ... };
    public void PreenFeathers() { ... };
}

class Pigeon : Bird {
    public void Coo() { ... }
}

public void SimulatePigeon() {
    Pigeon Harriet = new Pigeon();

    // Так как Pigeon является subclassом
    // Bird, мы можем вызывать методы обоих классов.

    Harriet.Walk();
    Harriet.LayEggs();
    Harriet.Coo();
    Harriet.Fly();
}

class Penguin : Bird {
    public void Swim() { ... }
}

public void SimulatePenguin() {
    Penguin Izzy = new Penguin();
    Izzy.Walk();
    Izzy.LayEggs();
    Izzy.Swim();
    Izzy.Fly();
}
  
```

Этот код компилируется, потому что Penguin расширяет Bird. Можно ли изменить класс Penguin так, чтобы при вызове Fly для пингвина вывело предупреждение?



Если бы эти классы входили в ваш симулятор зоопарка, как бы вы решали проблему летающих пингвинов?

Субкласс может переопределять методы для изменения или замены унаследованных компонентов

Иногда субкласс должен унаследовать большую часть поведения базового класса, но *не все*. Чтобы изменить поведение, унаследованное классом, можно **переопределить методы или свойства**, заменив их новыми методами или свойствами с теми же именами.

При **переопределении** метода новый метод должен обладать точно такой же сигнатурой, как у переопределяемого метода базового класса. Например, в случае с пингвинами это означает, что он называется Fly, возвращает void и вызывается без параметров.

пе-ре-о-пре-де-лить,
гл.

Определить заново или по-другому; дать новое определение.

① Добавьте ключевое слово **virtual** в метод базового класса.

Субкласс может переопределить только метод, помеченный ключевым словом `virtual`. Включение `virtual` в объявление метода Fly сообщает C#, что субклассу класса Bird разрешено переопределить метод Fly.

```
class Bird {
    public virtual void Fly() {
        // code to make the bird fly
    }
}
```

Добавление ключевого слова `virtual` в метод Fly сообщает C#, что субклассу разрешено переопределить метод.

② Добавьте ключевое слово **override** в одноименный метод субкласса.

Метод субкласса должен обладать точно такой же сигнатурой — той же комбинацией типа возвращаемого значения и параметров, и в объявлении должно использоваться ключевое слово `override`. Теперь объект Penguin выводит предупреждение при попытке вызова метода Fly.

```
class Penguin : Bird {
    public override void Fly() {
        Console.Error.WriteLine("WARNING");
        Console.Error.WriteLine("Flying Penguin Alert");
    }
}
```

Чтобы переопределить метод Fly, добавьте идентичный метод в субкласс и используйте ключевое слово `override`.

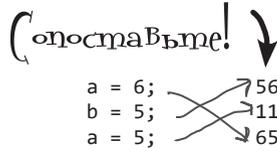
Мы использовали `Console.Error` для вывода сообщений об ошибках в стандартный поток ошибок (`stderr`), который обычно используется консольными приложениями для вывода описаний ошибок и важной диагностической информации.



Маши крыльями, Боб.
Уверен, мы скоро полетим!



Упражнение



Инструкции:

1. Заполните четыре пропуска в коде.
2. Сопоставьте предложенные варианты кода с выводом.

```

class A {
    public int ivar = 7;
    public _____ string m1() {
        return "A's m1, ";
    }
    public string m2() {
        return "A's m2, ";
    }
    public _____ string m3() {
        return "A's m3, ";
    }
}

class B : A {
    public _____ string m1() {
        return "B's m1, ";
    }
}
    
```

```

class C : B {
    public _____ string m3() {
        return "C's m3, " + (ivar + 6);
    }
}
    
```

Точка входа в программу

```

class Mixed5 {
    public static void Main(string[] args) {
        A a = new A();
        B b = new B();
        C c = new C();
        A a2 = new C();
        string q = "";
        _____
        Console.WriteLine(q);
    }
}
    
```

Подсказка: очень хорошо подумайте над тем, что означает эта строка.

Здесь вставляется блок кода (три строки).

Блоки кода:

```

q += b.m1();
q += c.m2();
q += a.m3();
-----
q += c.m1();
q += c.m2();
q += c.m3();
-----
q += a.m1();
q += b.m2();
q += c.m3();
-----
q += a2.m1();
q += a2.m2();
q += a2.m3();
    
```

Проведите линию от каждого блока из трех строк кода к строке вывода, которая будет получена при размещении блока в прямоугольнике.

Вывод:

- A's m1, A's m2, C's m3, 6
- B's m1, A's m2, A's m3,
- A's m1, B's m2, C's m3, 6
- B's m1, A's m2, C's m3, 13
- B's m1, C's m2, A's m3,
- A's m1, B's m2, A's m3,
- B's m1, A's m2, C's m3, 6
- A's m1, A's m2, C's m3, 13

(Не упрощайте свою задачу, вводя код в IDE, вы узнаете намного больше, если определите результат на бумаге!)



Сопоставьте!

```

a = 6;
b = 5;
a = 5;
    
```

→ 56
→ 11
→ 65

```

class A {
    public virtual string m1() {
    ...
    public virtual string m3() {
    }
    }
    
```

```

class B : A {
    public override string m1() {
    ...
class C : B {
    public override string m3() {
    }
    }
    
```

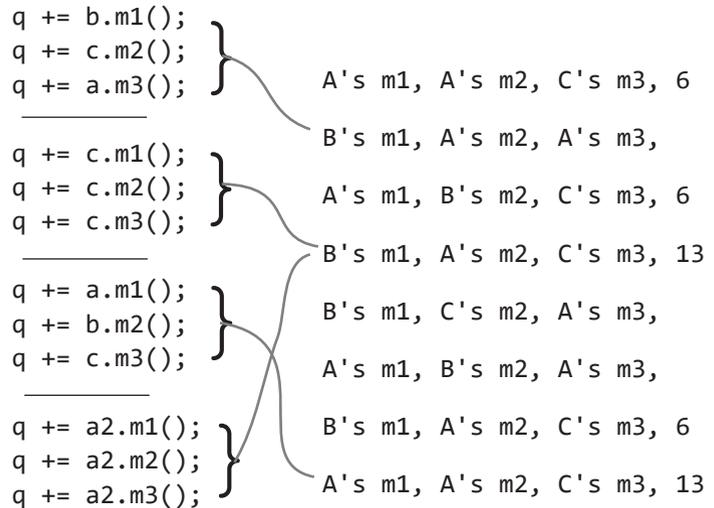
Ссылка на субкласс всегда может использоваться вместо ссылки на базовый класс, потому что конкретное может использоваться вместо общего. Таким образом, следующая строка:

```
A a2 = new C();
```

означает, что вы создаете новый объект C, затем создаете переменную a2, объявленную как ссылка на A, и присваиваете переменной ссылку на объект. Такие имена хороши разве что в головоломках — они слишком непонятны. Следующие строки следуют той же схеме, но используют более очевидные имена:

```

Canine fido = new Dog();
Bird pidge = new Pigen();
Feline rex = new Lion();
    
```



Часть Задаваемые Вопросы

В: Команда switch делает то же, что и серия команд if/else, верно? Получается, что она избыточна?

О: Совсем нет. Во многих ситуациях команды switch читаются лучше, чем команды if/else. Допустим, вы отображаете меню в консольном приложении и пользователь может нажать клавишу, чтобы выбрать один из десяти разных вариантов. Как будут смотреться 10 команд if/else подряд? Мы считаем, что команда switch будет более элегантно и удобочитаемой. Вы сразу видите, что с чем сравнивается, где обрабатывается каждый вариант и что происходит по умолчанию, если пользователь выбрал отсутствующий вариант. Кроме того, в if/else на удивление легко случайно пропустить else. Если пропущенный блок else находится где-то в середине длинной цепочки команд if/else, возникает коварная ошибка, которую на удивление трудно обнаружить. В одних случаях лучше читается команда switch, в других лучше читаются команды if/else. Вы сами пишете код в том виде, который (по вашему мнению) будет наиболее простым и наглядным.

В: Почему стрелка направлена вверх, от субкласса к базовому классу? Разве диаграмма не будет более логичной, если стрелка направлена сверху вниз?

О: На первый взгляд это кажется более логичным, но не так точно. Когда вы создаете класс, наследующий от другого класса, эти отношения встраиваются в субкласс — базовый класс остается неизменным. Его поведение несколько не изменяется, когда вы добавляете класс, наследующий от него. Базовый класс даже не знает о существовании нового класса. Его методы, поля и свойства остаются неизменными, но субкласс изменяет его поведение. Каждый экземпляр субкласса автоматически получает все свойства, поля и методы от базового класса — и все это делается простым добавлением двоеточия! Вот почему стрелка на диаграмме идет от субкласса к базовому классу, от которого он наследует.



Упражнение

Немного потренируемся в расширении базовых классов. Ниже приведен метод Main для программы, в которой моделируются птицы, несущие яйца. Ваша задача — реализовать два subclasses класса Bird.

1. Метод Main запрашивает у пользователя тип птицы и количество яиц:

```
static void Main(string[] args)
{
    while (true)
    {
        Bird bird;
        Console.WriteLine("\nPress P for pigeon, O for ostrich: ");
        char key = Char.ToUpper(Console.ReadKey().KeyChar);
        if (key == 'P') bird = new Pigeon();
        else if (key == 'O') bird = new Ostrich();
        else return;
        Console.WriteLine("\nHow many eggs should it lay? ");
        if (!int.TryParse(Console.ReadLine(), out int numberOfEggs)) return;
        Egg[] eggs = bird.LayEggs(numberOfEggs);
        foreach (Egg egg in eggs)
        {
            Console.WriteLine(egg.Description);
        }
    }
}
```

2. Добавьте класс Egg — конструктор задает размер и цвет яйца.

```
class Egg
{
    public double Size { get; private set; }
    public string Color { get; private set; }
    public Egg(double size, string color)
    {
        Size = size;
        Color = color;
    }
    public string Description {
        get { return $"A {Size:0.0}cm {Color} egg"; }
    }
}
```

3. Класс Bird, который вы будете расширять:

```
class Bird
{
    public static Random Randomizer = new Random();
    public virtual Egg[] LayEggs(int numberOfEggs)
    {
        Console.Error.WriteLine("Bird.LayEggs should never get called");
        return new Egg[0];
    }
}
```

4. Создайте класс Pigeon, расширяющий Bird. Переопределите метод LayEggs и настройте класс так, чтобы он нес яйца белого цвета ("white") размером от 1 до 3 сантиметров.
5. Создайте класс Ostrich, также расширяющий Bird. Переопределите метод LayEggs и настройте класс так, чтобы он нес яйца в крапинку ("speckled") размером от 12 до 13 сантиметров.

Вывод программы должен выглядеть так:

```
Press P for pigeon, O for ostrich: P
How many eggs should it lay? 4
A 3.0cm white egg
A 1.1cm white egg
A 2.4cm white egg
A 1.9cm white egg
```

```
Press P for pigeon, O for ostrich: O
How many eggs should it lay? 3
A 12.1cm speckled egg
A 13.0cm speckled egg
A 12.8cm speckled egg
```



Упражнение Решение

Ниже приведены классы Pigeon и Ostrich. Каждый из них содержит собственную версию метода LayEggs, которая использует ключевое слово `override` в объявлении метода. Ключевое слово `override` означает, что реализация subclasses заменяет реализацию, унаследованную от базового класса.

Pigeon является subclassом Bird, поэтому если вы переопределили метод LayEggs, а потом создали объект Pigeon и присвоили его переменной Bird с именем bird, при вызове `bird.LayEggs` будет вызван метод LayEggs, определенный в Pigeon.

```
class Pigeon : Bird
{
    public override Egg[] LayEggs(int numberOfEggs)
    {
        Egg[] eggs = new Egg[numberOfEggs];
        for (int i = 0; i < numberOfEggs; i++)
        {
            eggs[i] = new Egg(Bird.Randomizer.NextDouble() * 2 + 1, "white");
        }
        return eggs;
    }
}
```

Subclass Ostrich работает так же, как Pigeon. В обоих классах **ключевое слово** `override` в объявлении метода LayEggs означает, что новый метод заменяет реализацию LayEggs, унаследованную от Bird. Остается лишь создать набор яиц нужного цвета и размера.

```
class Ostrich : Bird
{
    public override Egg[] LayEggs(int numberOfEggs)
    {
        Egg[] eggs = new Egg[numberOfEggs];
        for (int i = 0; i < numberOfEggs; i++)
        {
            eggs[i] = new Egg(Bird.Randomizer.NextDouble() + 12, "speckled");
        }
        return eggs;
    }
}
```



Некоторые компоненты реализованы только в подклассе

Во всем коде, который вы видели до настоящего момента, обращения к компонентам производились извне, как, например, в методе Main в только что написанном коде вызывался метод LayEggs. Наследование по-настоящему раскрывает свой потенциал, когда базовый класс **использует метод или свойство, реализованные в подклассе**. Рассмотрим пример. В нашем симуляторе зоопарка установлены торговые автоматы, в которых посетители могут покупать газировку, леденцы и корм для животных в контактной зоне зоопарка.

```
class VendingMachine
```

```
{
    public virtual string Item { get; }

    protected virtual bool CheckAmount(decimal money) {
        return false;
    }

    public string Dispense(decimal money)
    {
        if (CheckAmount(money)) return Item;
        else return "Please enter the right amount";
    }
}
```

Класс использует ключевое слово `protected`. С этим модификатором доступ компонент класса становится **открытым только для его подклассов**, но остается приватным для всех остальных классов.

VendingMachine — базовый класс для всех торговых автоматов. Он содержит код продажи товаров, но сами товары не определяются. Метод для проверки того, внес ли покупатель правильную сумму, всегда возвращает `false`. Почему? Потому, что он **будет реализован в подклассе**. Подкласс для продажи корма для животных в контактной зоне выглядит так:

```
class AnimalFeedVendingMachine : VendingMachine
```

```
{
    public override string Item {
        get { return "a handful of animal feed"; }
    }

    protected override bool CheckAmount(decimal money)
    {
        return money >= 1.25M;
    }
}
```

Ключевое слово `override` со свойством `работает точно так же, как при переопределении метода`.

Для инкапсуляции используется ключевое слово `protected`. Метод `CheckAmount` объявляется с ключевым словом `protected`, потому что он ни при каких условиях не должен вызываться другим классом, так что обращение к нему возможно только из класса `VendingMachine` и его подклассов.

Анализ переопределения в отладчике

Отладчик поможет понять, что же именно происходит, когда мы создаем экземпляр `AnimalFeedVendingMachine` и обращаемся к нему с запросом на продажу корма. Создайте **новый проект консольного приложения**, после чего выполните следующие действия:



- 1 **Добавьте метод Main.** Метод должен содержать следующий код:

```
class Program
{
    static void Main(string[] args)
    {
        VendingMachine vendingMachine = new AnimalFeedVendingMachine();
        Console.WriteLine(vendingMachine.Dispense(2.00M));
    }
}
```

- 2 **Добавьте классы VendingMachine и AnimalFeedVendingMachine.** Когда они будут добавлены, попробуйте включить следующую строку кода в метод `Main`:

```
vendingMachine.CheckAmount(1F);
```

Вы получите сообщение об ошибке компиляции из-за ключевого слова `protected`, потому что к методам с таким модификатором может обращаться только сам класс `VendingMachine` или его субклассы:

```
✘ CS0122 'VendingMachine.CheckAmount(decimal)' is inaccessible due to its protection level
```

Удалите добавленную строку, чтобы приложение строилось нормально.

- 3 **Установите точку прерывания в первой строке метода Main.** Запустите программу. Когда она достигнет точки прерывания, воспользуйтесь командой **Step Into (F10)** для **пошагового выполнения кода**. Вот что при этом происходит:

- ★ Программа создает экземпляр `AnimalFeedVendingMachine` и вызывает его метод `Dispense`.
- ★ Метод определяется только в базовом классе, поэтому вызывается реализация `VendingMachine.Dispense`.
- ★ В первой строке `VendingMachine.Dispense` вызывается защищенный (`protected`) метод `CheckAmount`.
- ★ `CheckAmount` переопределяется в субклассе `AnimalFeedVendingMachine`, из-за чего `VendingMachine.Dispense` вызывает метод `CheckAmount`, определенный в `AnimalFeedVendingMachine`.
- ★ Эта версия `CheckAmount` возвращает `true`, поэтому `Dispense` возвращает свойство `Item`. `AnimalFeedVendingMachine` также переопределяет это свойство.

↑
Мы уже использовали отладчик `Visual Studio` для поиска ошибок в коде. Но этот инструмент также прекрасно подходит для изучения C# и анализа — как в этой врезке «Принципы отладки» (см. выше), где мы изучали работу переопределения. А вы сможете предложить другие способы экспериментирования с переопределением?

Что-то я не пойму, для чего нужны эти ключевые слова «virtual» и «override». Без них IDE выдает предупреждение, но это ничего не значит... **программа все равно работает!** То есть ключевые слова поставить можно, если это вроде как «положено», но по-моему, мы сами выдумываем себе трудности **на ровном месте.**



Для использования virtual и override есть веские причины!

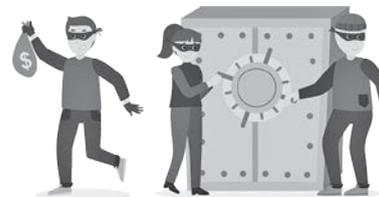
Ключевые слова virtual и override существуют не только для красоты. Они реально влияют на работу вашей программы. Ключевое слово virtual сообщает C#, что компонент (метод, свойство или поле) может расширяться — без него переопределение вообще невозможно. Ключевое слово override сообщает C#, что вы расширяете компонент. Если опустить ключевое слово override в подклассе, вы создадите *совершенно несвязанный* метод, который просто по случайности *имеет такое же имя*.

Звучит немного странно, не так ли? Но на самом деле все вполне разумно — чтобы понять, как работают ключевые слова virtual и override, лучше всего начать с написания кода. Построим реальный пример для экспериментов.

Если подкласс переопределяет метод в своем базовом классе, то всегда будет вызываться более конкретная реализация, определенная в подклассе, даже при вызове из метода базового класса.

Построение приложения для изучения *virtual* и *override*

Исключительно важной частью наследования в C# является расширение компонентов классов. Таким образом субкласс может унаследовать часть своего поведения от базового класса, переопределяя отдельные компоненты, и именно здесь в игру вступают **ключевые слова** *virtual* и *override*. Ключевое слово *virtual* определяет компоненты класса, которые могут расширяться. Если вы хотите расширить тот или иной компонент, он *должен* быть объявлен с ключевым словом *override*. Создадим набор классов для экспериментов с *virtual* и *override*. Первый класс представляет сейф с бриллиантами, а потом мы построим класс, представляющий хитрых взломщиков, которые пытаются эти бриллианты украсть.



1 Создайте новое консольное приложение и добавьте класс `Safe`.

Код класса `Safe`:

```
class Safe
{
    private string contents = "precious jewels";
    private string safeCombination = "12345";

    public string Open(string combination)
    {
        if (combination == safeCombination) return contents;
        return "";
    }

    public void PickLock(Locksmith lockpicker)
    {
        lockpicker.Combination = safeCombination;
    }
}
```

← (Делайте это!

← Объект `Safe` хранит ценности в поле `contents`. Чтобы он вернул их, метод `Open` должен быть вызван с правильной комбинацией, или... слесарь откроет замок.

← Мы добавим класс `Locksmith`, который может открыть кодовый замок и получить нужную комбинацию вызовом метода `PickLock` с передачей ссылки на себя. Чтобы предоставить `Locksmith` комбинацию, объект `Safe` использует свойство `Combination`, доступное только для записи.

2 Добавьте класс, представляющий владельца сейфа.

Владелец сейфа рассеян и часто забывает свой отлично защищенный пароль от сейфа. Добавьте класс `SafeOwner` для представления владельца:

```
class SafeOwner
{
    private string valuables = "";
    public void ReceiveContents(string safeContents)
    {
        valuables = safeContents;
        Console.WriteLine($"Thank you for returning my {valuables}!");
    }
}
```

3 Добавьте класс `Locksmith`, представляющий слесаря.

Если владелец сейфа воспользуется услугами профессионального слесаря для открытия сейфа, он ожидает, что все содержимое сейфа останется в неприкосновенности. Именно это делает метод `Locksmith.OpenSafe`:

```
class Locksmith
{
    public void OpenSafe(Safe safe, SafeOwner owner)
    {
        safe.PickLock(this);
        string safeContents = safe.Open(Combination);
        ReturnContents(safeContents, owner);
    }

    public string Combination { private get; set; }

    protected void ReturnContents(string safeContents, SafeOwner owner)
    {
        owner.ReceiveContents(safeContents);
    }
}
```

Метод `OpenSafe` класса `Locksmith` узнает комбинацию, открывает сейф и вызывает `ReturnContents` для безопасной передачи ценностей владельцу.

**4** Добавьте класс `JewelThief`, представляющий вора, который хочет украсть бриллианты.

В общей схеме появляется вор — и что самое неприятное, он также является исключительно опытным слесарем и умеет открывать сейфы. Добавьте класс `JewelThief`, расширяющий `Locksmith`:

```
class JewelThief : Locksmith
{
    private string stolenJewels;
    protected void ReturnContents(string safeContents, SafeOwner owner)
    {
        stolenJewels = safeContents;
        Console.WriteLine($"I'm stealing the jewels! I stole: {stolenJewels}");
    }
}
```

`JewelThief` расширяет `Locksmith`, наследуя метод `OpenSafe` и свойство `Combination`, но его метод `ReturnContents` крадет бриллианты, вместо того чтобы вернуть их владельцу. КАКОЕ КОВАРСТВО!

5 Добавьте метод `Main`, в котором экземпляр `JewelThief` крадет бриллианты.

Наступает момент для кражи века! В методе `Main` экземпляр `JewelThief` проникает в дом и использует унаследованный метод `Locksmith.OpenSafe` для получения комбинации. **Как вы думаете, что произойдет при его выполнении?**

```
static void Main(string[] args)
{
    SafeOwner owner = new SafeOwner();
    Safe safe = new Safe();
    JewelThief jewelThief = new JewelThief();
    jewelThief.OpenSafe(safe, owner);
    Console.ReadKey(true);
}
```

MINI Возьми в руку карандаш

Прочитайте код вашей программы. Прежде чем запускать его, напишите, что, по вашему мнению, он выведет на консоль. (Подсказка: проанализируйте, что именно класс `JewelThief` наследует от `Locksmith`.)

Субкласс может скрывать методы базового класса

Запустите программу JewelThief. Она выведет следующее сообщение:

Thank you for returning my precious jewels!

Вы ожидали другого? Возможно, чего-то такого:

I'm stealing the jewels! I stole: precious jewels

Похоже, объект JewelThief сработал как обычный объект Locksmith! Что же произошло?

MINI Возьми в руку карандаш
Решение

Вроде бы C# должен вызывать самый конкретный метод, не так ли? Тогда почему мы не назвали его JewelThief.ReturnContents?

Сокрытие методов и переопределение методов

Почему же объект JewelThief при вызове его метода ReturnContents действовал как объект Locksmith? Это связано с тем, как в классе JewelThief объявляется метод ReturnContents. В предупреждающем сообщении, которое выводится при компиляции программы, содержится ясная подсказка:

```
CS0108 'JewelThief.ReturnContents(string, SafeOwner)' hides inherited member 'Locksmith.ReturnContents(string, SafeOwner)'. Use the new keyword if hiding was intended.
```

Так как класс JewelThief наследует от Locksmith и замещает метод ReturnContents собственным методом, может показаться, что JewelThief переопределяет метод ReturnContents класса Locksmith, но на самом деле происходит нечто иное. Вероятно, вы ожидали, что JewelThief *переопределил* метод (сейчас мы обсудим эту возможность), но вместо этого JewelThief *скрывает* его.

JewelThief
Locksmith.ReturnContents
JewelThief.ReturnContents

Эти две ситуации очень сильно отличаются друг от друга. Когда субкласс скрывает метод, он замещает (с технической точки зрения — переобъявляет) одноименный метод базового класса. Таким образом, сейчас субкласс содержит два разных метода с одинаковыми именами: один наследуется от базового класса, а другой определяется в этом классе.

Используйте ключевое слово new для сокрытия методов

Присмотритесь к предупреждающему сообщению. Конечно, мы знаем, что предупреждения *нужно* читать, но иногда мы этого не делаем... верно? Так что теперь прочитайте, что же в нем сказано: «Используйте ключевое слово **new**, если предполагалось сокрытие».

Вернитесь к программе и добавьте ключевое слово **new**:

```
new public void ReturnContents(Jewels safeContents, Owner owner)
```

Как только вы добавите **new** в объявление метода ReturnContents класса JewelThief, предупреждение исчезнет — но ваш код все равно не будет работать так, как ожидалось!

В нем по-прежнему вызывается метод ReturnContents, определенный в классе Locksmith. Почему? Потому, что метод ReturnContents вызывается **из метода, определенного в классе Locksmith**, а конкретно из Locksmith.OpenSafe, при том что вызов был инициирован объектом JewelThief. Если класс JewelThief только скрывает метод ReturnContents класса Locksmith, то его собственный метод ReturnContents никогда не будет вызван.

Если субкласс просто добавляет метод с таким же именем, как у метода базового класса, он только скрывает метод базового класса вместо того, чтобы переопределять его.

Использование разных ссылок для вызова скрытых методов

Теперь мы знаем, что JewelThief только *скрывает* свой метод ReturnContents (вместо того, чтобы *переопределять* его). В результате он ведет себя как объект Locksmith каждый раз, *когда он вызывается как объект Locksmith*. JewelThief наследует одну версию ReturnContents от Locksmith и определяет вторую версию этого метода; отсюда следует, что в классе существуют два одноименных метода и они должны вызываться **двумя разными способами**.

Существуют два разных способа вызова метода ReturnContents. Если у вас имеется экземпляр JewelThief, вы можете воспользоваться переменной-ссылкой на JewelThief для вызова нового метода ReturnContents. Если использовать для вызова переменную-ссылку на Locksmith, будет вызван метод ReturnContents.

Вот как это делается:

```
// Субкласс JewelThief скрывает метод в базовом классе Locksmith, так что вы
// можете получить разное поведение для одного объекта в зависимости от того,
// какая ссылка использовалась для вызова!

// Объявление объекта JewelThief как ссылки на Locksmith заставляет его вызвать
// метод ReturnContents базового класса.
Locksmith calledAsLocksmith = new JewelThief();
calledAsLocksmith.ReturnContents(safeContents, owner);

// Если объект JewelThief объявляется в виде ссылки на JewelThief, то вызван
// будет метод ReturnContents() класса JewelThief, потому что он скрывает
// одноименный метод базового класса.
JewelThief calledAsJewelThief = new JewelThief();
calledAsJewelThief.ReturnContents(safeContents, owner);
```

Сможете ли вы понять, как заставить JewelThief переопределить метод ReturnContents, вместо того чтобы просто скрыть его? Удастся ли вам сделать это до того, как вы перейдете к следующему разделу?

Часть Задаваемые Вопросы

В: Я так и не понял, почему методы называются «виртуальными» (virtual), — мне они кажутся вполне реальными. Что в них виртуального?

О: Этот термин связан с особенностями внутренней реализации таких методов в .NET. В ней используется структура данных, называемая *таблицей виртуальных методов* (или *v-таблицей*). Эта таблица используется в .NET для отслеживания того, какие методы были унаследованы, а какие были переопределены. Не беспокойтесь: вам не нужно знать, как работает этот механизм, чтобы пользоваться виртуальными методами.

В: Вы говорили о замене суперкласса ссылкой на субкласс. Можно повторить еще разок?

О: Если на вашей диаграмме один класс расположен выше другого, класс, расположенный выше, **более абстрактен** по сравнению с нижним классом. **Более конкретные** классы (такие, как Shirt или Car) наследуют от более абстрактных (таких, как Clothing или Vehicle). Если вам подойдет любое транспортное средство, то подойдет и машина, и грузовик, и мотоцикл. Если вам нужна машина, то мотоцикл не подойдет.

Наследование работает точно так же. Если у вас есть метод, получающий параметр Vehicle, а класс Motorcycle наследует от класса Vehicle, то методу можно передать экземпляр Motorcycle. Если метод получает параметр Motorcycle, передать произвольный объект Vehicle не удастся, потому что это может быть экземпляр Van. Ведь тогда C# не будет знать, что делать, когда метод пытается обратиться к свойству Handlebars, которое есть только у мотоциклов.

Использование ключевых слов override и virtual для наследования поведения

Мы хотим, чтобы класс JewelThief всегда использовал свою реализацию ReturnContents, независимо от способа вызова. Именно так в нашем представлении должно работать наследование в большинстве случаев: субкласс может переопределить метод базового класса, чтобы вместо него вызывался метод субкласса. Начните с добавления ключевого слова override при объявлении метода ReturnContents:

```
class JewelThief {  
    protected override void ReturnContents  
        (string safeContents, SafeOwner owner)
```

Но это еще не все. Если вы ограничитесь добавлением ключевого слова override в объявление класса, компилятор выдаст сообщение об ошибке:

```
CS0506 'JewelThief.ReturnContents(string, SafeOwner)': cannot override inherited member 'Locksmith.ReturnContents(string, SafeOwner)' because it is not marked virtual, abstract, or override
```

Снова присмотритесь повнимательнее к тексту и прочитайте описание ошибки. JewelThief не может переопределить унаследованный метод ReturnContents, потому что он не помечен ключевым словом virtual, abstract или override в Locksmith. К счастью, эта ошибка исправляется очень быстро. Пометьте метод ReturnContents класса Locksmith ключевым словом virtual:

```
class Locksmith {  
    protected virtual void ReturnContents  
        (string safeContents, SafeOwner owner)
```

Снова запустите программу. На этот раз мы получаем вывод, к которому стремились:

I'm stealing the jewels! I stole: precious jewels



Возьми в руку карандаш

Соедините каждое из следующих описаний с ключевым словом, которое оно описывает.

1. Метод, **обращение к которому возможно только из экземпляра того же класса.**
2. Метод, который может быть заменен одноименным методом в субклассе.
3. Метод, **обращение к которому возможно из экземпляра любого другого класса.**
4. Метод, который **скрывает другой одноименный метод в суперклассе.**
5. Метод, который **заменяет метод, определенный в суперклассе.**
6. Метод, **обращение к которому возможно только из компонента класса или его субкласса.**

virtual
new
override
protected
private
public

1. private 2. virtual 3. public 4. new 5. override 6. protected

Когда я строю собственные иерархии классов, обычно я хочу переопределять методы, а не скрывать их. Но если я хочу их скрыть, то я всегда использую ключевое слово `new`, верно?



Точно. В большинстве случаев требуется переопределять методы, но вариант с сокрытием тоже возможен.

Когда вы работаете с субклассом, расширяющим базовый класс, то, скорее всего, вы хотите переопределять их, а не скрывать. Итак, когда вы получаете предупреждение компилятора о сокрытии метода, обратите на него внимание! Убедитесь в том, что вы планировали именно скрыть метод, а не просто забыли добавить ключевые слова `virtual` и `override`. Если вы всегда правильно используете ключевые слова `virtual`, `override` и `new`, подобные проблемы у вас никогда не возникнут!

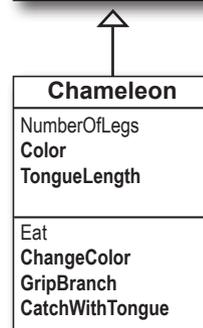
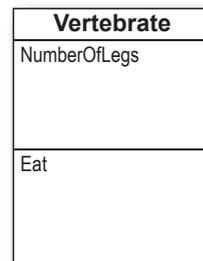
Если вы хотите переопределить метод в базовом классе, всегда помечайте его ключевым словом `virtual` и всегда используйте ключевое слово `override` каждый раз, когда вы хотите переопределить метод в субклассе. Если вы не будете следить за этим, то в конечном итоге это может привести к непреднамеренному сокрытию методов.

Субкласс может обратиться к своему базовому классу при помощи ключевого слова `base`

Даже когда вы переопределяете метод или свойство в базовом классе, иногда бывает нужно сохранить доступ к нему. К счастью, ключевое слово `base` позволяет обратиться к любому компоненту базового класса.

- ① Все животные едят, поэтому класс `Vertebrate` содержит метод `Eat`, получающий в параметре объект `Food`.

```
class Vertebrate {
    public virtual void Eat(Food morsel) {
        Swallow(morsel);
        Digest();
    }
}
```



- ② Хамелеоны едят, захватывая пищу длинным языком. Соответственно, класс `Chameleon` наследует от `Vertebrate`, но переопределяет `Eat`.

```
class Chameleon : Vertebrate {
    public override void Eat(Food morsel) {
        CatchWithTongue(morsel);
        Swallow(morsel);
        Digest();
    }
}
```

Код полностью совпадает с кодом из базового класса. Нам действительно нужны две одинаковые версии одного кода?

Метод `Chameleon.Eat` должен вызывать `CatchWithTongue`, но после этого он идентичен методу `Eat` переопределяемого базового класса `Vertebrate`.

- ③ Вместо того чтобы дублировать код, можно воспользоваться ключевым словом `base` для вызова переопределенного метода. Теперь нам доступна как старая, так и новая версия `Eat`.

```
class Chameleon : Vertebrate {
    public override void Eat(Food morsel) {
        CatchWithTongue(morsel);
        base.Eat(morsel);
    }
}
```

Обновленная версия метода из базового класса использует ключевое слово `base` для вызова метода `Eat` в базовом классе. Теперь дублирующийся код отсутствует, так что если когда-нибудь вам потребуется изменить то, как едят все позвоночные, хамелеоны получат изменения автоматически.

Нельзя просто воспользоваться записью «Eat(morsel)», потому что будет вызвана реализация Chameleon.Eat. Для обращения к Vertebrate.Eat необходимо использовать ключевое слово «base».

Если базовый класс содержит конструктор, ваш subclass должен его вызывать

Вернемся к коду, написанному с классами Bird, Pigeon, Ostrich и Egg. Мы хотим добавить класс BrokenEgg, который расширяет Egg; с ним 25% яиц, которые откладывают голуби, оказываются разбитыми. **Замечайте новую команду** в Pigeon.LayEgg следующей командой if/else, которая создает новый экземпляр Egg или BrokenEgg:

```
if (Bird.Randomizer.Next(4) == 0)
    eggs[i] = new BrokenEgg(Bird.Randomizer.NextDouble() * 2 + 1, "white");
else
    eggs[i] = new Egg(Bird.Randomizer.NextDouble() * 2 + 1, "white");
```



Осталось написать класс BrokenEgg, расширяющий Egg. Он будет идентичен классу Egg, не считая того, что у него есть конструктор, выводящий на консоль сообщение (о том, что яйцо разбито):

```
class BrokenEgg : Egg
{
    public BrokenEgg()
    {
        Console.WriteLine("A bird laid a broken egg");
    }
}
```



Простой возврат к старой версии проекта.

Чтобы загрузить в IDE предыдущую версию проекта, выберите команду *Recent Project and Solutions* (Windows) или *Recent Solutions* (Mac) из меню File.

Внесите эти два изменения в программу Egg. Ой-ой, кажется, следующие две строки кода порождают ошибки компиляции:

- ★ Первая ошибка в строке, в которой создается новый объект BrokenEgg: CS1729 - *'BrokenEgg' не содержит конструктора, который получает два аргумента.*
- ★ Вторая ошибка в конструкторе BrokenEgg: CS7036 – *не задан аргумент, соответствующий обязательному формальному параметру 'size' для 'Egg.Egg(double, string)'.*

Вам предоставляется еще одна отличная возможность прочитать описания ошибок и попытаться понять, что же пошло не так. Первая ошибка достаточно очевидна: команда, которая создает экземпляр BrokenEgg, пытается передать конструктору два аргумента, но класс BrokenEgg имеет конструктор без параметров. **Добавьте параметры в конструктор:**

```
public BrokenEgg(double size, string color)
```

Первая ошибка исчезает – метод Main нормально компилируется. Как насчет другой ошибки? Разделим описание ошибки на несколько частей:

- ★ В описании говорится о *Egg.Egg(double, string)*, т. е. о конструкторе класса Egg.
- ★ В нем упоминается параметр *'size'*, необходимый классу Egg для задания свойства Size.
- ★ Но *аргумент не задан*, потому что недостаточно изменить конструктор BrokenEgg для получения аргументов, соответствующих параметрам. Также необходимо **вызвать конструктор базового класса.**

Измените класс BrokenEgg и включите в него ключевое слово base для **вызова конструктора базового класса:**

```
public BrokenEgg(double size, string color) : base(size, color)
```

Наконец код нормально компилируется. Попробуйте запустить его – теперь, когда экземпляр Pigeon откладывает яйцо, приблизительно в четверти случаев при создании экземпляра будет выводиться сообщение о том, что яйцо разбито (но последующий вывод остается неизменным).

Субкласс и базовый класс могут иметь разные конструкторы

Когда мы изменяли BrokenEgg для вызова конструктора базового класса, мы привели его конструктор в соответствие с базовым классом Egg. А если мы хотим, чтобы все разбитые яйца имели нулевой размер, а их цвет начинался со слова «broken»? **Измените команду, создающую экземпляр BrokenEgg**, чтобы передавался только аргумент для цвета:

```
if (Bird.Randomizer.Next(4) == 0)
    eggs[i] = new BrokenEgg("white");
else
    eggs[i] = new Egg(Bird.Randomizer.NextDouble() * 2 + 1, "white");
```

← **Измените!**

При внесении этого изменения вы снова получите ошибку компиляции, в которой говорится об «обязательном формальном параметре», — и это логично, потому что конструктор BrokenEgg имеет два параметра, но ему передается только один аргумент.

Исправьте свой код — измените конструктор BrokenEgg, чтобы он получал один параметр:

```
class BrokenEgg : Egg
{
    public BrokenEgg(string color) : base(0, $"broken {color}")
    {
        Console.WriteLine("A bird laid a broken egg");
    }
}
```

← **Конструктор subclasses может получать любое количество параметров, но может не получать ни одного. Необходимо лишь использовать ключевое слово base для передачи правильного количества аргументов конструктору базового класса.**

Теперь снова запустите программу. Конструктор BrokenEgg по-прежнему выводит свое сообщение на консоль в цикле for в конструкторе Pigeon, но теперь он также заставляет Egg инициализировать поля Size и Color. Когда цикл foreach в методе Main выводит на консоль значение egg.Description, для каждого разбитого яйца выводится сообщение:

```
Press P for pigeon, O for ostrich:
p
How many eggs should it lay? 7
A bird laid a broken egg
A bird laid a broken egg
A bird laid a broken egg
A 2.4cm white egg
A 0.0cm broken White egg
A 3.0cm white egg
A 1.4cm white egg
A 0.0cm broken White egg
A 0.0cm broken White egg
A 2.7cm white egg
```

А вы знали, что голуби обычно откладывают только одно или два яйца? Как бы вы изменили класс Pigeon с учетом этого факта?

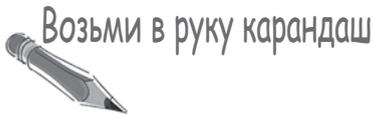
Какая погода там наверху?



Пора доделать приложение для Оуэна

В этой главе все началось с изменения программы — калькулятора повреждений, когда-то построенного для Оуэна, чтобы программа могла вычислять повреждения как от меча, так и от стрелы. Решение работало, а классы `SwordDamage` и `ArrowDamage` были хорошо инкапсулированы. Но кроме нескольких строк кода **два класса были почти идентичны**. Вы узнали, что повторение кода в разных классах неэффективно и небезопасно, особенно если вы собираетесь и далее расширять программу, добавляя в нее классы для новых видов оружия. Теперь в вашем арсенале появился новый инструмент для решения проблемы: **наследование**. А значит, пришло время завершить приложение-калькулятор. Это будет сделано за два шага: сначала новая модель классов будет спроектирована на бумаге, а затем мы реализуем ее в коде.

Построение модели классов на бумаге перед написанием кода помогает лучше понять суть проблемы и решить ее более эффективно.

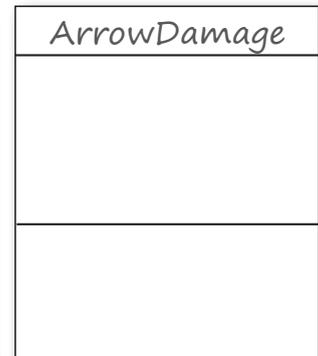
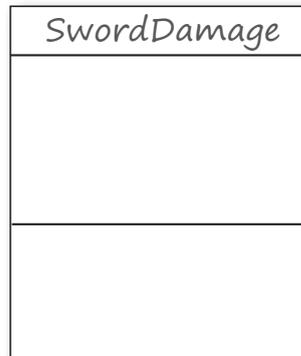
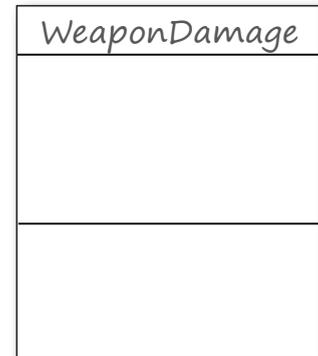
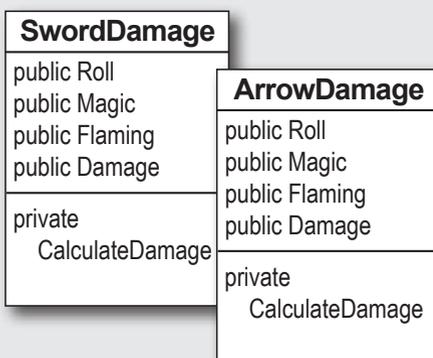


Хороший код начинает формироваться в голове, а не в IDE. А значит, стоит выделить время на проектирование модели классов на бумаге до того, как переходить к программированию.

Мы уже записали имена классов. Вам остается добавить компоненты во все три класса и соединить прямоугольники стрелками.

Для справки мы приводим диаграммы классов `SwordDamage` и `ArrowDamage`, которые были построены ранее. В каждый класс включен приватный метод `CalculateDamage`. Проследите за тем, чтобы при заполнении диаграммы классов были включены все открытые, приватные и защищенные компоненты классов. Укажите модификатор доступа (`public`, `private` или `protected`) рядом с каждым компонентом класса.

Так классы `SwordDamage` и `ArrowDamage` выглядели в начале этой главы. Они хорошо инкапсулированы, но большая часть кода `SwordDamage` дублируется в `ArrowDamage`.

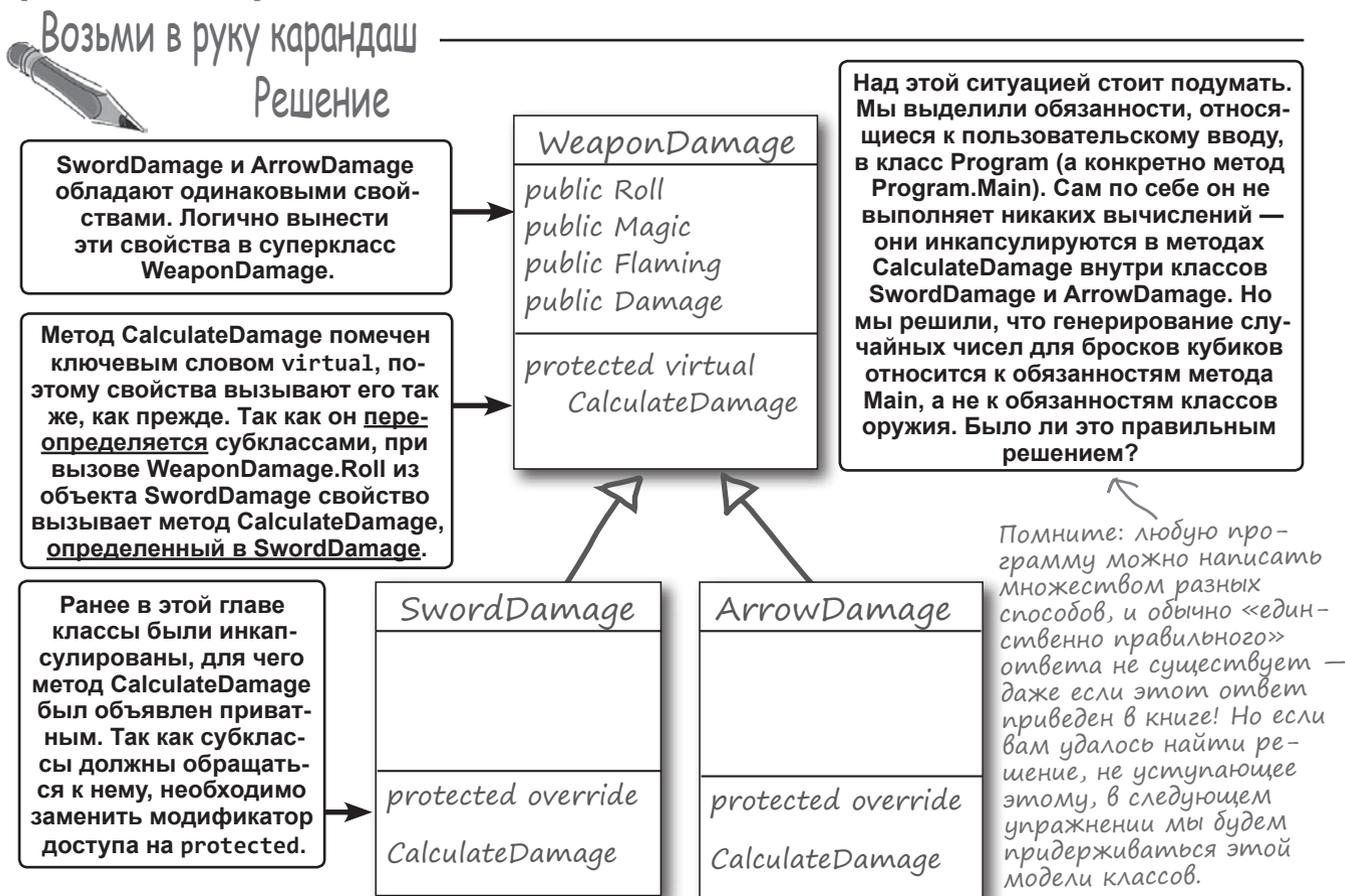


Минимальное перекрытие между классами — важный принцип проектирования, называемый разделением обязанностей

Если сегодня вы хорошо спроектируете свои классы, вам будет проще изменять их в будущем. Представьте, что у вас десяток разных классов для вычисления повреждений от разных видов оружия. Что, если вы захотите изменить тип Magic с bool на int, чтобы в игре могло присутствовать оружие с разными бонусами (например, волшебная булава +3 или волшебный кинжал +1)? С наследованием вам пришлось бы изменять свойство Magic в суперклассе. Конечно, вам придется изменить метод CalculateDamage для каждого класса, но объем работы будет намного меньше и пропадает риск того, что вы случайно забудете изменить один из классов. (В профессиональной разработке такое случается *сплошь и рядом!*)

Это пример **разделения обязанностей**, потому что каждый класс содержит только код, относящийся к одной конкретной части проблемы, решаемой вашей программой. Код, относящийся только к мечам, размещается в классе SwordDamage; код, относящийся только к стрелам, — в классе ArrowDamage; а общий код включается в класс WeaponDamage.

При проектировании классов разделение обязанностей должно стать одним из первоочередных факторов, которые вы должны учитывать. Если на один класс возложены две разные обязанности, попробуйте разбить его на два разных класса.





Упражнение

Итак, вы **спроектировали** модель классов, и мы можем перейти к написанию кода ее **реализации**. Это очень полезная привычка — сначала проектируйте свои классы, а затем преобразуйте их в код.

Ниже описана последовательность действий по завершению работы для Оуэна. Вы можете снова открыть проект, созданный в начале главы, или же создать новый проект и скопировать в него нужные части. Если ваш код сильно отличается от кода решения, приведенного ранее в этой главе, мы рекомендуем начать с кода решения. Если вы не хотите вводить его вручную, загрузите его по адресу <https://github.com/head-first-csharp/fourth-edition>.

- 1. Ничего не изменяйте в методе Main.** В нем будут использоваться классы `SwordDamage` и `ArrowDamage` (так же, как в начале главы).
- 2. Реализуйте класс `WeaponDamage`.** Добавьте новый класс `WeaponDamage` и приведите его в соответствие с диаграммой классов из раздела «Возьми в руку карандаш». При этом необходимо учитывать ряд факторов:
 - ★ Свойства в `WeaponDamage` *почти* идентичны свойствам в классах `SwordDamage` и `ArrowDamage` в начале главы. Изменилось только одно ключевое слово.
 - ★ Не включайте код в метод `CalculateDamage` (можно включить комментарий: `/* переопределяется в субклассе */`). Метод должен быть виртуальным, он не может быть приватным, в противном случае произойдет ошибка компиляции):

✘ CS0621 "WeaponDamage.CalculateDamage()": virtual or abstract members cannot be private
 - ★ Добавьте конструктор, который реализует начальный бросок.
- 3. Реализуйте класс `SwordDamage`.** При этом необходимо учитывать несколько факторов:
 - ★ Конструктор получает один параметр, который передается конструктору базового класса.
 - ★ `C#` всегда вызывает наиболее конкретную реализацию метода. Это означает, что вы должны переопределить метод `CalculateDamage` и вычислить в нем повреждения от меча.
 - ★ Также стоит задуматься над тем, как работает `CalculateDamage`. `Set`-методы свойств `Roll`, `Magic` и `Flaming` вызывают `CalculateDamage`, чтобы обеспечить автоматическое обновление поля `Damage`. Так как `C#` всегда вызывает наиболее конкретную реализацию метода, они вызовут `SwordDamage.CalculateDamage`, *несмотря на то что они являются частью суперкласса `WeaponDamage`*.
- 4. Реализуйте класс `ArrowDamage`.** Он работает точно так же, как `SwordDamage`, если не считать того, что метод `CalculateDamage` вычисляет повреждения для стрелы, а не для меча.

Мы можем внести довольно серьезные изменения в механизмы работы классов без изменения метода `Main`, из которого эти классы вызываются.

Если ваши классы хорошо инкапсулированы, это значительно упрощает изменение кода.

Если у вас есть знакомый профессиональный разработчик, спросите, какая задача за последний год вызвала у него больше всего негатива. С довольно высокой вероятностью он ответит что-то вроде: «Мне надо было внести изменения в класс, но для этого пришлось изменить два других класса, что потребовало еще трех изменений и т. д., — даже просто отслеживать все эти изменения было достаточно трудно». Проектирование классов с учетом правил инкапсуляции поможет избежать подобных ситуаций.





Упражнение Решение

Ниже приведен код класса `WeaponDamage`. Свойства почти идентичны свойствам старых классов `SwordDamage` и `ArrowDamage`. Также появился конструктор для моделирования исходного броска кубиков и метод `CalculateDamage`, переопределяемый в subclasses.

```
class WeaponDamage
{
    public int Damage { get; protected set; }

    private int roll;
    public int Roll
    {
        get { return roll; }
        set
        {
            roll = value;
            CalculateDamage();
        }
    }

    private bool magic;
    public bool Magic
    {
        get { return magic; }
        set
        {
            magic = value;
            CalculateDamage();
        }
    }

    private bool flaming;
    public bool Flaming
    {
        get { return flaming; }
        set
        {
            flaming = value;
            CalculateDamage();
        }
    }

    protected virtual void CalculateDamage() { /* Переопределяется subclasses */ }

    public WeaponDamage(int startingRoll)
    {
        roll = startingRoll;
        CalculateDamage();
    }
}
```

`WeaponDamage`

```
public Roll
public Magic
public Flaming
public Damage
```

```
protected virtual
    CalculateDamage
```

Set-метод свойства `Damage` должен быть снабжен пометкой `protected`. Таким образом он будет доступен для subclasses, но никакой другой класс задать значение свойства не сможет. Так как свойство защищено от случайной записи из других классов, subclasses остаются хорошо инкапсулированными.

Свойства могут вызвать метод `CalculateDamage`, который обновляет свойство `Damage`. И хотя они определяются в суперклассе, при наследовании subclasses они вызывают метод `CalculateDamage`, определенный в этом subclasses.

Полная аналогия с тем, как работал класс `JewelThief`, когда вы переопределяли метод из `Locksmith`. Это было нужно, чтобы вор крал бриллианты из сейфа, а не возвращал их владельцу.

Метод `CalculateDamage` сам по себе пуст — мы пользуемся тем фактом, что C# всегда вызывает наиболее конкретную реализацию. В новой иерархии класс `SwordDamage` расширяет `WeaponDamage`, когда set-метод его унаследованного свойства `Flaming` вызывает `CalculateDamage`, будет выполнена самая конкретная реализация этого метода, поэтому вместо этого вызывается реализация `SwordDamage.CalculateDamage`.

Отладчик поможет понять, как работают эти классы

Сделайте
это!

Одна из важнейших идей этой главы заключается в том, что при расширении класса можно переопределить его методы, чтобы внести весьма значительные изменения в его поведение. Чтобы понять, как работает этот механизм, мы воспользуемся отладчиком:

- ★ **Установите точки прерывания** в строках set-методов Roll, Magic и Flaming, в которых вызывается CalculateDamage.
- ★ Добавьте команду Console.WriteLine в WeaponDamage.CalculateDamage. Эта команда **никогда** не выполняется.
- ★ Запустите программу. Когда она достигает какой-либо из точек прерывания, используйте команду **Step Into** для входа в метод CalculateDamage. *Управление передается реализации subclasses* — метод WeaponDamage.CalculateDamage вызываться не будет.

Класс SwordDamage расширяет WeaponDamage и переопределяет его метод CalculateDamage, чтобы реализовать вычисление повреждений от меча. Код выглядит так:

```
class SwordDamage : WeaponDamage
{
    public const int BASE_DAMAGE = 3;
    public const int FLAME_DAMAGE = 2;

    public SwordDamage(int startingRoll) : base(startingRoll) { }

    protected override void CalculateDamage()
    {
        decimal magicMultiplier = 1M;
        if (Magic) magicMultiplier = 1.75M;

        Damage = BASE_DAMAGE;
        Damage = (int)(Roll * magicMultiplier) + BASE_DAMAGE;
        if (Flaming) Damage += FLAME_DAMAGE;
    }
}
```

От конструктора требуется совсем немного: вызвать конструктор суперкласса при помощи ключевого слова base и передать ему в аргументе параметр startingRoll.

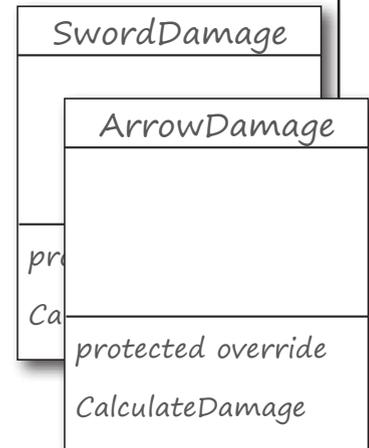
Перейдем к классу ArrowDamage. Он работает практически так же, как класс SwordDamage, не считая того, что он вычисляет повреждения для стрелы:

```
class ArrowDamage : WeaponDamage
{
    private const decimal BASE_MULTIPLIER = 0.35M;
    private const decimal MAGIC_MULTIPLIER = 2.5M;
    private const decimal FLAME_DAMAGE = 1.25M;

    public ArrowDamage(int startingRoll) : base(startingRoll) { }

    protected override void CalculateDamage()
    {
        decimal baseDamage = Roll * BASE_MULTIPLIER;
        if (Magic) baseDamage *= MAGIC_MULTIPLIER;
        if (Flaming) Damage = (int)Math.Ceiling(baseDamage + FLAME_DAMAGE);
        else Damage = (int)Math.Ceiling(baseDamage);
    }
}
```

Упражнение
Решение



Мы собираемся обсудить важный элемент разработки игр: динамику. На самом деле эта концепция настолько важна, что она отнюдь не ограничивается играми. Собственно, динамика в какой-то степени проявляется практически во всех видах приложений.



Динамика

Разработка игр... и не только

Динамика игры описывает, каким образом игровые механики комбинируются и взаимодействуют для управления игровым процессом. В любой ситуации, где есть игровые механики, они естественным образом формируют динамику. Концепция не ограничивается видеоиграми — механика присутствует во всех играх, а динамики возникают из всех механик.

- **Хороший пример механики** уже встречался нам ранее: в ролевой игре Оуэна для вычисления повреждений от разных видов оружия использовались формулы (встроенные в классы вычисления повреждений). Это хорошая отправная точка для анализа того, как изменения в механике влияют на динамику.
- Что произойдет, если вы измените механику формулы для стрелы, чтобы базовые повреждения умножались на 10? Это незначительное изменение в механике приводит к **огромным изменениям в динамике** игры. Внезапно стрелы становятся намного более смертоносными, чем мечи. Игроки перестанут пользоваться мечами и начнут стрелять из лука даже на минимальном расстоянии — и это изменение динамики.
- Когда **поведение игроков изменится**, Оуэн должен изменить сценарий кампании. Например, некоторые сражения, которые разрабатывались как сложные, внезапно становятся слишком простыми для игроков. Игроки снова меняют поведение.

Ненадолго остановитесь и поразмыслите над происходящим. Крошечные изменения в правилах приводят к **колоссальным изменениям в динамике**. Оуэн не вносил эти изменения непосредственно в игровой процесс; они стали побочными эффектами этого небольшого изменения в правилах. С технической точки зрения изменение динамики **проявилось** в результате изменения механики.

- Если вы еще не сталкивались с концепцией проявления, она может показаться немного странной, поэтому рассмотрим конкретный пример из классической видеоигры.
- **Механика игры Space Invaders проста.** Пришельцы двигаются влево-вправо и стреляют вниз; если выстрел попадает в корабль игрока, то игрок теряет одну жизнь. Игрок перемещает свой корабль влево-вправо и стреляет вверх. Если выстрел попадает в пришельца, этот пришелец уничтожается. Время от времени у верхнего края экрана пролетает командный корабль; его уничтожение приносит дополнительные очки игроку. Энергетическая защита постепенно ослабевает от выстрелов. За пришельцев разных видов игрок получает разное количество очков. С течением времени пришельцы начинают двигаться быстрее... В общем-то все.
- С **динамикой игры Space Invaders** дело обстоит сложнее. Поначалу игра проста: большинство игроков справляется с первой волной нападения, но она быстро усложняется. Изменяется только скорость перемещения пришельцев. Нарастание скорости пришельцев изменяет всю игру. **Темп**, т. е. субъективное восприятие скорости игры, радикально изменяется.
- Некоторые игроки стараются уничтожить пришельцев от края построения, потому что наличие «разрывов» замедляет снижение. Этот момент нигде не отражен в коде, в котором есть только простые правила перемещения пришельцев. Он относится к динамике и является побочным эффектом сочетания механик, а именно механики работы выстрелов с правилами перемещения пришельцев. Ничто из этого не запрограммировано в коде игры. Это не часть игровой механики, а динамика.



На первый взгляд динамика может показаться какой-то абстрактной концепцией. Позднее в этой главе мы еще займемся этой темой, а пока просто помните обо всем, что было сказано о динамике, во время работы над следующим проектом. Можете ли вы заметить, как динамика начинает влиять на игру, в процессе программирования?



Знаете что? Я уже **по горло сыта играми**. Игры на поиск пар, 3D-игры, игры с числами, классы для карт и пейнтбольных маркеров, которые должны использоваться в играх, модели классов для игр, проектирование игр...
Получается, **мы не занимаемся ничем, кроме игр**.

Да, мы все знаем, что разработчики **C#** могут найти очень неплохие вакансии на рынке труда. Я хочу изучить **C#**, чтобы использовать его в серьезной работе. Нельзя ли сделать хотя бы один проект, в котором мы займемся **серьезным бизнес-приложением?**

Видеоигры — это серьезная область бизнеса.

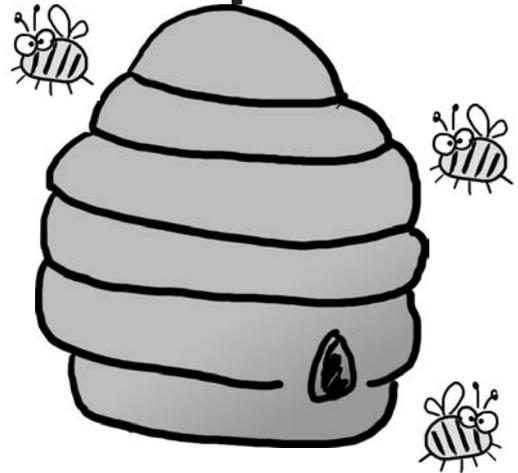
Индустрия видеоигр с каждым годом растет в глобальном масштабе. В ней работают сотни тысяч людей по всему миру, и талантливый разработчик непременно найдет себе в ней достойное место! Существует целая экосистема **независимых разработчиков**, которые создают и продают игры — в одиночку или небольшими группами.

Но вы правы — **C#** серьезный язык, и он используется для создания многих серьезных приложений, не имеющих отношения к играм. Собственно, хотя **C#** стал любимым языком многих разработчиков игр, он также стал одним из самых популярных языков во многих других отраслях.

Поэтому чтобы немного потренироваться в применении наследования, в следующем проекте мы создадим **серьезное бизнес-приложение**.

Построение системы управления ульем

Теперь ваша помощь нужна пчелиной матке! Улей вышел из-под контроля, и ей нужна программа, которая поможет управлять процессом производства меда. Улей полон рабочих пчел, имеется и список заданий. Нужно распределить задания между пчелами с учетом их специализации. Постройте систему, управляющую поведением рабочих пчел. Вот как она должна функционировать:

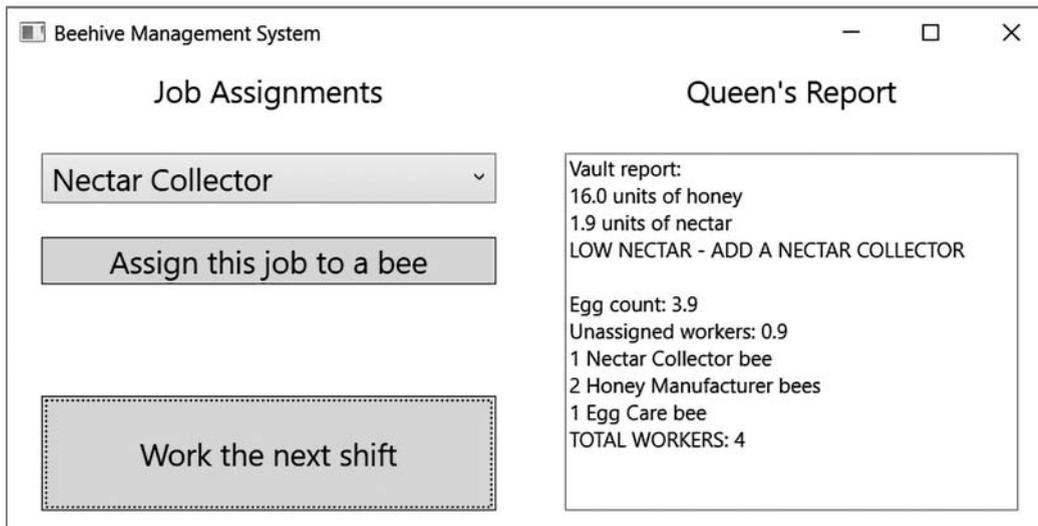


1 Матка раздает задания рабочим.

Существует три вида работ. Некоторые пчелы умеют вылетать из улья и приносить в него нектар. Другие перерабатывают нектар в мед, которым питаются пчелы. Наконец, матка постоянно откладывает яйца, пчелы-опекуны следят за тем, чтобы из яиц выросли новые рабочие.

2 Когда все задания будут распределены, время работать.

Раздав задания, матка заставляет пчел отработать очередную смену щелчком на кнопке «Work the next shift» в приложении. Программа строит отчет с информацией о том, сколько пчел еще работает над каждой задачей, а также о количестве нектара и меда в хранилище.



3 Управление ростом улья.

Как и все руководители бизнеса, пчелиная матка стремится к расширению своего предприятия. Улей — сложная система, и матка оценивает его размер по общему количеству рабочих. Как бы вы реализовали возможность добавления новых рабочих? До какого размера может вырасти улей, прежде чем в нем кончится мед и предприятие обанкротится?

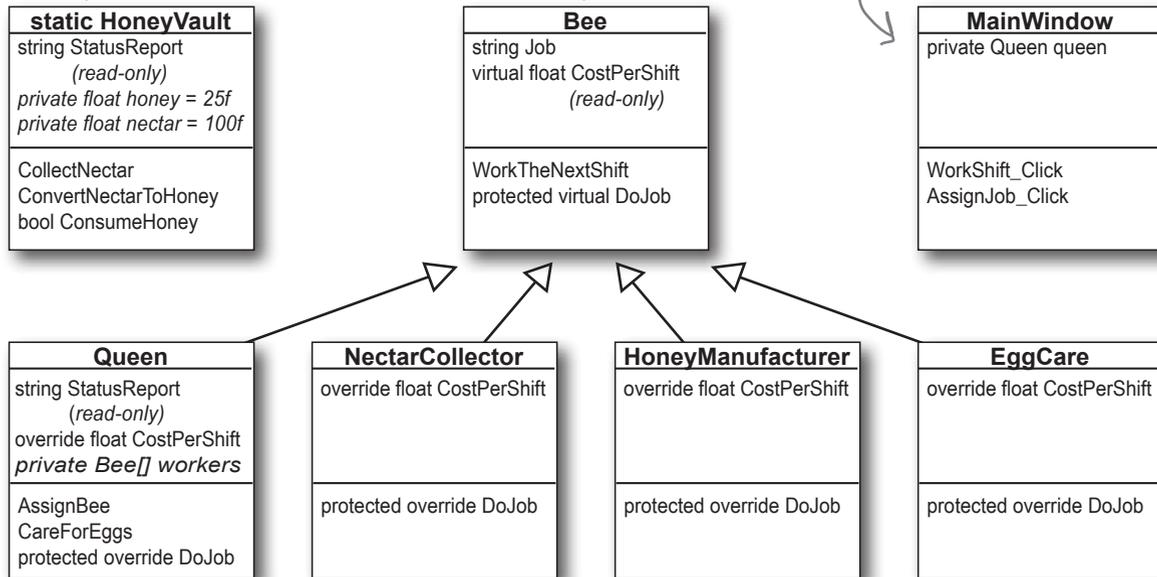
Модель классов системы управления ульем

Ниже представлены классы, которые мы построим для системы управления ульем. Будет использоваться модель с базовым классом и четырьмя subclasses, статический класс для управления объемами меда и нектара, а также класс `MainWindow`, содержащий код программной части главного окна.

`HoneyVault` — статический класс, который отслеживает текущие объемы меда и нектара в улье. Пчелы используют метод `ConsumeHoney`, который проверяет, достаточно ли меда для выполнения их задач, и если достаточно, — вычитает запрашиваемый объем.

`Bee` — базовый класс для всех классов пчел. Его метод `WorkTheNextShift` вызывает метод `ConsumeHoney` класса `HoneyVault`, и если тот вернет `true`, вызывает `DoJob`.

Код программной части главного окна решает несколько задач. Он создает экземпляр `Queen` и обработчики события `Click` для кнопок, которые вызывают методы `WorkTheNextShift` и `AssignBee` и выводят отчет.



Этот subclass `Bee` использует массив для отслеживания рабочих и переопределяет `DoJob` для вызова их методов `WorkTheNextShift`.

Этот подкласс `Bee` переопределяет `doJob` для вызова метода `HoneyVault` для сбора нектара.

Этот subclass `Bee` переопределяет `DoJob` для вызова метода `HoneyVault`, преобразующего нектар в мед.

Этот subclass `Bee` хранит ссылку на `Queen` и переопределяет `DoJob` для вызова метода `CareForEggs` класса `Queen`.



Эта модель классов — всего лишь начало. Мы приведем больше информации в процессе написания кода.

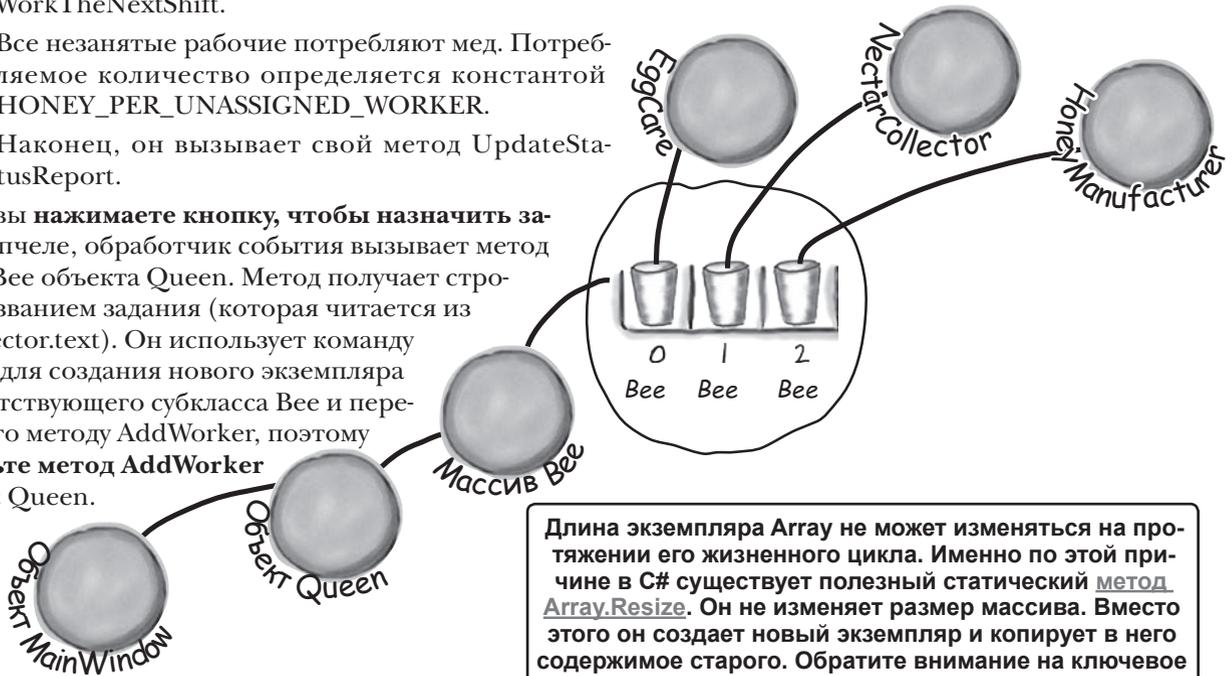
Очень внимательно проанализируйте эту модель классов. Она содержит много ценной информации о приложении, которое вам предстоит построить. Затем мы приведем всю информацию, необходимую для написания кода этих классов.

Класс Queen: как матка управляет рабочими

Когда вы нажимаете кнопку для отработки следующей смены, обработчик события Click кнопки вызывает метод WorkTheNextShift объекта Queen, унаследованный от базового класса Bee. Вот что происходит после этого:

- ★ Bee.WorkTheNextShift вызывает HoneyVault.ConsumeHoney(HoneyConsumed), используя свойство CostPerShift (которое переопределяется разными значениями в subclasses) для определения того, сколько меда потребуется для работы.
- ★ Затем Bee.WorkTheNextShift вызывает DoJob, который также переопределяется Queen.
- ★ Queen.DoJob увеличивает свое приватное поле eggs на 0.45 (с использованием константы EGGS_PER_SHIFT). Пчела EggCare вызывает свой метод CareForEggs, который уменьшает eggs и увеличивает unassignedWorkers.
- ★ Затем цикл foreach используется для вызова метода WorkTheNextShift.
- ★ Все незанятые рабочие потребляют мед. Потребляемое количество определяется константой HONEY_PER_UNASSIGNED_WORKER.
- ★ Наконец, он вызывает свой метод UpdateStatusReport.

Когда вы **нажимаете кнопку, чтобы назначить задание** пчеле, обработчик события вызывает метод AssignBee объекта Queen. Метод получает строку с названием задания (которая читается из jobSelector.text). Он использует команду switch для создания нового экземпляра соответствующего subclasses Bee и передачи его методу AddWorker, поэтому **добавьте метод AddWorker** в класс Queen.



Длина экземпляра Array не может изменяться на протяжении его жизненного цикла. Именно по этой причине в C# существует полезный статический [метод Array.Resize](#). Он не изменяет размер массива. Вместо этого он создает новый экземпляр и копирует в него содержимое старого. Обратите внимание на ключевое слово **ref** — вы еще узнаете о нем позднее в книге.

Метод AddWorker добавляет нового рабочего в массив рабочих класса Queen. Он вызывает метод Array.Resize для расширения массива, а затем добавляет в него нового рабочего.

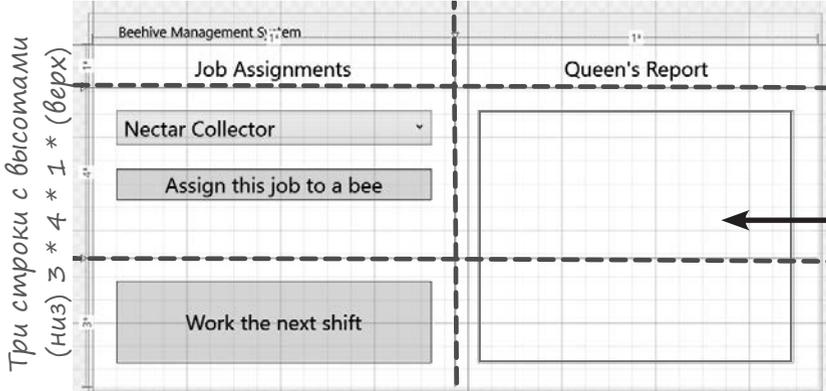
```

/// <summary>
/// Расширяет массив workers на один элемент и добавляет ссылку Bee.
/// </summary>
/// <param name="worker">Рабочий, добавляемый в массив workers.</param>
private void AddWorker(Bee worker)
{
    if (unassignedWorkers >= 1)
    {
        unassignedWorkers--;
        Array.Resize(ref workers, workers.Length + 1);
        workers[workers.Length - 1] = worker;
    }
}
    
```

Пользовательский интерфейс: добавление кода XAML главного окна

Создайте **новое приложение WPF** с именем **BeehiveManagementSystem**. Структура макета главного окна определяется сеткой со свойствами `Title = "Beehive Management System"` `Height="325"` `Width="625"`. В макете используются те же элементы `Label`, `StackPanel` и `Button`, которые встречались вам в предыдущих главах, а также добавляются два новых элемента. Раскрывающийся список в группе `Job Assignments` представляет собой элемент **ComboBox**, в котором пользователь может выбрать вариант из списка. Отчет о текущем состоянии из раздела `Queen's Report` выводится в элементе **TextBox**.

Сетка с двумя столбцами равной ширины



Это **элемент TextBox**. Обычно `TextBox` используется для ввода данных пользователем, но здесь его свойство `IsReadOnly` задано значением `True`, чтобы элемент был доступен только для чтения. Мы используем его вместо элемента `TextBlock`, который использовался в предыдущих проектах, по двум причинам. Во-первых, его граница обводится рамкой, а это хорошо смотрится. Во-вторых, он позволяет выделять и копировать текст, что очень полезно для отчета о текущем состоянии в бизнес-приложении.

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="1*" />
    <RowDefinition Height="4*" />
    <RowDefinition Height="3*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
```

Раскрывающийся список представляет собой **элемент ComboBox**. Это контейнерный элемент (как и `Grid`, например), у которого между открывающим и закрывающим тегами следуют элементы. В данном случае он содержит три элемента `ListBoxItem`, по одному для каждого варианта, который может быть выбран пользователем. Вообще говоря, вы можете раскрыть раздел `Common` в окне `Properties` и воспользоваться кнопкой  рядом с пунктом `Items` (выберите в списке `ListBoxItem`), но на самом деле проще ввести определения элементов в XAML вручную. Проследите за тем, чтобы содержимое каждого элемента точно соответствовало показанному ниже.

```
<Label Content="Job Assignments" FontSize="18" Margin="20,0"
  HorizontalAlignment="Center" VerticalAlignment="Bottom" />
```

```
<StackPanel Grid.Row="1" VerticalAlignment="Top" Margin="20">
  <ComboBox x:Name="jobSelector" FontSize="18" SelectedIndex="0" Margin="0,0,0,20">
    <ListBoxItem Content="Nectar Collector" />
    <ListBoxItem Content="Honey Manufacturer" />
    <ListBoxItem Content="Egg Care" />
  </ComboBox>
  <Button Content="Assign this job to a bee" FontSize="18px" Click="AssignJob_Click" />
</StackPanel>
```

Элементы `ListBoxItem` определяют варианты, которые пользователь видит в списке `ComboBox`.

```
<Button Grid.Row="2" Content="Work the next shift" FontSize="18px"
  Click="WorkShift_Click" Margin="20" />
```

```
<Label Content="Queen's Report" Grid.Column="1" FontSize="18" Margin="20,0"
  VerticalAlignment="Bottom" HorizontalAlignment="Center" />
```

```
<TextBox
  x:Name="statusReport" IsReadOnly="True"
  Grid.Row="1" Grid.RowSpan="2" Grid.Column="1" Margin="20" />
</Grid>
```

Присвойте элементу `TextBox` имя (`x:Name`), чтобы иметь возможность задать его свойство `Text` в коде программной части.



Длинные упражнения

Упражнение большое, но не падайте духом! Просто разбейте его на меньшие части. А когда вы начнете работать над ними, вы поймете, что многое из этого вам уже знакомо.

Постройте **систему управления ульем**. Цель системы — **максимизировать количество рабочих, которым поручено выполнение различных функций в улье**, и поддерживать работу улья до тех пор, пока в нем не кончится мед.

Правила улья

Рабочим может поручаться одно из трех заданий: сборщики нектара приносят нектар в хранилище, производители меда преобразуют нектар в мед, а опекуны превращают яйца в рабочих, которым могут поручаться задания. Во время смены матка откладывает яйца (чуть меньше двух смен уходит на откладывание одного яйца). Матка обновляет свой отчет текущего состояния в конце каждой смены. В отчете приводится информация о заполнении хранилища, количество яиц, нераспределенных рабочих и пчел, назначенных для каждого вида работы.

Начните с построения статического класса HoneyVault

- Класс HoneyVault станет хорошей отправной точкой, потому что он **не имеет зависимостей**, иначе говоря, он не вызывает методы и не использует свойства или поля других классов. Начните с создания нового класса HoneyVault. Объясните его с ключевым словом `static`, затем обратитесь к диаграмме класса и добавьте компоненты класса.
- HoneyVault содержит **две константы** (`NECTAR_CONVERSION_RATIO = .19f` и `LOW_LEVEL_WARNING = 10f`), которые используются в методах. Приватное поле `honey` инициализируется значением `25f`, а приватное поле `nectar` — значением `100f`.
- **Метод ConvertNectarToHoney** преобразует нектар в мед. Он получает параметр `float` с именем `amount`, уменьшает поле `nectar` на указанную величину и увеличивает поле `honey` на величину `amount × NECTAR_CONVERSION_RATIO`. (Если запрашиваемое значение больше количества нектара, оставшегося в хранилище, то преобразуется весь оставшийся нектар.)
- **Метод ConsumeHoney** определяет то, как пчелы потребляют мед для выполнения своих задач. Метод получает параметр `amount`. Если он меньше текущего содержимого поля `honey`, то метод вычитает `amount` из `honey` и возвращает `true`; в противном случае возвращается `false`.
- **Метод CollectNectar** вызывается пчелой `NectarCollector` каждую смену. Метод получает параметр `amount`. Если значение параметра положительное, оно прибавляется к полю `honey`.
- **Свойство StatusReport** имеет только `get`-метод доступа, который возвращает строку с количеством меда и нектара в хранилище. Если количество меда ниже порога `LOW_LEVEL_WARNING`, добавляется предупреждение ("LOW HONEY — ADD A HONEY MANUFACTURER"). То же самое происходит и с полем `nectar`.

Создайте класс Bee и начинайте строить классы Queen, HoneyManufacturer, NectarCollector и EggCare.

- Создайте базовый класс Bee. Его **конструктор** получает строку, которая используется для присваивания **свойства Job, доступного только для чтения**. Каждый субкласс Bee передает строку своему базовому конструктору — "Queen", "Nectar Collector", "Honey Manufacturer" или "Egg Care", — так что класс Queen содержит следующий код: `public Queen() : base("Queen")`.
- Виртуальное и доступное только для чтения **свойство CostPerShift** позволяет каждому субклассу Bee определить количество меда, потребляемого им за смену. **Метод WorkTheNextShift** передает `HoneyConsumed` методу `HoneyVault.ConsumeHoney`. Если `ConsumeHoney` возвращает `true`, значит, в улье остается достаточно меда и `WorkTheNextShift` вызывает `DoJob`.
- **Создайте пустые** классы `HoneyManufacturer`, `NectarCollector` и `EggCare`, которые просто расширяют Bee, — они потребуются вам для построения класса Queen. Сначала мы **достроим класс Queen**, а затем вернемся и доделаем другие субклассы Bee.
- Каждый субкласс Bee **переопределяет метод DoJob** кодом, реализующим его задание, и переопределяет свойство `CostPerShift` количеством меда, потребляемым за смену.
- Ниже приведены значения **свойства Bee.CostPerShift, доступного только для чтения**, для каждого субкласса Bee:
- `Queen.CostPerShift` возвращает `2.15f`, `NectarCollector.CostPerShift` возвращает `1.95f`, `HoneyManufacturer.CostPerShift` возвращает `1.7f`, и `EggCare.CostPerShift` возвращает `1.35f`.

Каждая отдельная часть этого упражнения вам уже знакома. Она вам ПО СИЛАМ!



Упражнение получилось длинным, **но это нормально!** Просто стройте его класс за классом. Сначала постройте класс `Queen`. Когда это будет сделано, вернитесь к другим субклассам `Bee`.

- Класс `Queen` содержит **приватное поле** `Bee[]` с именем `workers`. Изначально массив пуст. Мы привели метод `AddWorker` для добавления в массив ссылок на `Bee`.
- **Метод `AssignBee`** получает параметр с названием задания (например, "Egg Care"). Он содержит команду `switch(job)` с условиями, в которых вызывается `AddWorker`. Например, если `job` содержит "Egg Care", то будет вызван метод `AddWorker(new EggCare(this))`.
- Класс содержит **два приватных поля `float`** с именами `eggs` и `unassignedWorkers`, в которых хранится количество яиц (добавляемых при каждой смене) и количество рабочих, ожидающих назначения задания.
- Класс `Queen` переопределяет **метод `DoJob`** для добавления яиц, отдает приказ рабочим пчелам о начале работы и кормит медом рабочих, еще не получивших назначения на работу. Константа `EGGS_PER_SHIFT` (которой присваивается `0.45f`) прибавляется к полю `eggs`. Цикл `foreach` используется для вызова метода `WorkTheNextShift` каждого рабочего. Затем вызывается метод `HoneyVault.ConsumeHoney`, которому передается константа `HONEY_PER_UNASSIGNED_WORKER (0.5f) × workers.Length`.
- Изначально матка располагает тремя свободными рабочими — ее **конструктор** вызывает метод `AssignBee` три раза, чтобы создать трех рабочих пчел, по одной каждого типа.
- Пчелы-опекуны `EggCare` вызывают **метод `CareForEggs`** класса `Queen`. Он получает параметр `float` с именем `eggsToConvert`. Если поле `eggs >= eggsToConvert`, то значение `eggsToConvert` вычитается из `eggs` и прибавляется к `unassignedWorkers`.
- Внимательно посмотрите отчет о текущем состоянии на снимке — он генерируется приватным **методом `UpdateStatusReport`** (с использованием `HoneyVault.StatusReport`). Класс `Queen` вызывает `UpdateStatusReport` в конце своих методов `DoJob` и `AssignBee`.

Завершите построение других субклассов `Bee`

- **Класс `NectarCollector`** содержит константу `NECTAR_COLLECTED_PER_SHIFT = 33.25f`. Его метод `DoJob` передает эту константу `HoneyVault.CollectNectar`.
- **Класс `HoneyManufacturer`** содержит константу `NECTAR_PROCESSED_PER_SHIFT = 33.15f`. Его метод `DoJob` передает эту константу `HoneyVault.ConvertNectarToHoney`.
- **Класс `EggCare`** содержит константу `CARE_PROGRESS_PER_SHIFT = 0.15f`. Его метод `DoJob` передает эту константу `queen.CareForEggs`, используя приватную ссылку `Queen`, которая инициализируется в конструкторе `EgCare`.

Построение кода программной части главного окна

- Мы предоставили вам код XAML **главного окна**. Ваша задача — добавить код программной части. Он содержит приватное поле `Queen` с именем `queen`, которое инициализируется в конструкторе, и обработчики событий для кнопок и `ComboBox`.
- Подключите **обработчики событий**. Кнопка «Assign Job» вызывает `queen.AssignBee(jobSelector.Text)`. Кнопка «Work the Next Shift» вызывает `queen.WorkTheNextShift`. Обе кнопки присваивают `statusReport.Text` результату `queen.StatusReport`.

Подробнее о работе системы управления ульем

- Ваша цель — добиться как можно большего количества рабочих с назначенными заданиями (строка `TOTAL WORKERS` в отчете о текущем состоянии), а это **зависит от того, каких рабочих вы добавляете и когда это происходит**. Рабочие потребляют мед; если у вас будет слишком много рабочих одного типа, то уровень меда начнет падать. В процессе выполнения программы следите за уровнями нектара и меда. После нескольких первых смен вы получите предупреждение о нехватке меда (поэтому добавьте производителя меда); еще после нескольких — предупреждение о нехватке нектара (поэтому добавьте сборщика нектара), а после этого вам, возможно, придется расширять персонал улья. Какого значения `TOTAL WORKERS` вам удастся добиться, прежде чем в улье кончится мед?



Решение длинных упражнений

Это большой проект, и он состоит из **многих частей**. Если у вас возникнут затруднения, просто разделите текущую задачу на части. И это не волшебство — у вас уже есть все необходимое для понимания всех без исключения частей.

Код статического класса HoneyVault:

```
static class HoneyVault
{
    public const float NECTAR_CONVERSION_RATIO = .19f;
    public const float LOW_LEVEL_WARNING = 10f;
    private static float honey = 25f;
    private static float nectar = 100f;

    public static void CollectNectar(float amount)
    {
        if (amount > 0f) nectar += amount;
    }

    public static void ConvertNectarToHoney(float amount)
    {
        float nectarToConvert = amount;
        if (nectarToConvert > nectar) nectarToConvert = nectar;
        nectar -= nectarToConvert;
        honey += nectarToConvert * NECTAR_CONVERSION_RATIO;
    }

    public static bool ConsumeHoney(float amount)
    {
        if (honey >= amount)
        {
            honey -= amount;
            return true;
        }
        return false;
    }

    public static string StatusReport
    {
        get
        {
            string status = $"{honey:0.0} units of honey\n" +
                $"{nectar:0.0} units of nectar";
            string warnings = "";
            if (honey < LOW_LEVEL_WARNING) warnings +=
                "\nLOW HONEY - ADD A HONEY MANUFACTURER";
            if (nectar < LOW_LEVEL_WARNING) warnings +=
                "\nLOW NECTAR - ADD A NECTAR COLLECTOR";
            return status + warnings;
        }
    }
}
```

Константы из класса HoneyVault очень важны. Попробуйте повысить коэффициент преобразования нектара в мед — и с каждой сменой в улье будет появляться много меда. Попробуйте его уменьшить — и почти сразу же столкнетесь с нехваткой меда.

Пчелы NectarCollector выполняют свою работу, вызывая метод CollectNectar, чтобы добавить нектар в улей.

Пчелы HoneyManufacturer выполняют свою работу, вызывая ConvertNectarToHoney, что приводит к уменьшению количества нектара и увеличению количества меда в хранилище.

Каждая пчела старается потребить конкретное количество меда за каждую смену. Метод ConsumeToHoney возвращает true только в том случае, если в улье хватает меда для выполнения его работы.

Если ваш код не совпадает с нашим полностью, это нормально! Такие задачи могут решаться многими разными способами, и чем больше программа, тем больше возможных вариантов ее написания. Если ваш код работает, значит, упражнение решено правильно! Но выделите несколько минут на то, чтобы сравнить ваше решение с нашим, и попробуйте понять, почему мы приняли именно те, а не иные решения.

Попробуйте воспользоваться меню View для отображения в IDE представления Class View (панель будет пристыкована в окне Solution Explorer). Это полезный инструмент для исследования иерархии классов. Попробуйте раскрыть класс в окне Class View, затем разверните папку Base Types и просмотрите ее иерархию. Для переключения между представлением Class View и Solution Explorer используются вкладки в нижней части окна.



Решение длинных упражнений

Поведение этой программы определяется тем, как составляющие ее классы взаимодействуют друг с другом, особенно классы из иерархии Bee. На вершине этой иерархии располагается **суперкласс Bee**, расширяемый всеми остальными классами Bee:

```
class Bee
{
    public virtual float CostPerShift { get; }

    public string Job { get; private set; }

    public Bee(string job)
    {
        Job = job;
    }

    public void WorkTheNextShift()
    {
        if (HoneyVault.ConsumeHoney(CostPerShift))
        {
            DoJob();
        }
    }

    protected virtual void DoJob() { /* the subclass overrides this */ }
}

```

← Конструктор Bee получает один параметр, который используется для инициализации его свойства Job, доступного только для чтения. Класс Queen использует это свойство при построении отчета, чтобы определить, к какой разновидности относится эта конкретная пчела.

Класс NectarCollector каждую смену собирает нектар и приносит его в хранилище:

```
class NectarCollector : Bee
{
    public const float NECTAR_COLLECTED_PER_SHIFT = 33.25f;
    public override float CostPerShift { get { return 1.95f; } }
    public NectarCollector() : base("Nectar Collector") { }

    protected override void DoJob()
    {
        HoneyVault.CollectNectar(NECTAR_COLLECTED_PER_SHIFT);
    }
}

```

← Классы NectarCollector и HoneyManufacturer определяют константы, управляющие тем, сколько нектара собирается и какая его часть преобразуется в мед при каждой смене. Попробуйте изменить их — к изменению этих констант программа куда менее чувствительна, чем к изменению коэффициента преобразования нектара в мед.

Класс HoneyManufacturer преобразует нектар, находящийся в хранилище, в мед:

```
class HoneyManufacturer : Bee
{
    public const float NECTAR_PROCESSED_PER_SHIFT = 33.15f;
    public override float CostPerShift { get { return 1.7f; } }
    public HoneyManufacturer() : base("Honey Manufacturer") { }

    protected override void DoJob()
    {
        HoneyVault.ConvertNectarToHoney(NECTAR_PROCESSED_PER_SHIFT);
    }
}

```



Решение длинных упражнений

Каждый из subclasses Bee выполняет свою работу, но все они обладают **общим поведением**, даже Queen. Все они работают по сменам, но выполняют работу только при наличии достаточного количества меда.

Класс Queen управляет рабочими и генерирует отчеты о текущем состоянии:

```
class Queen : Bee
{
    public const float EGGS_PER_SHIFT = 0.45f;
    public const float HONEY_PER_UNASSIGNED_WORKER = 0.5f;

    private Bee[] workers = new Bee[0];
    private float eggs = 0;
    private float unassignedWorkers = 3;

    public string StatusReport { get; private set; }
    public override float CostPerShift { get { return 2.15f; } }

    public Queen() : base("Queen") {
        AssignBee("Nectar Collector");
        AssignBee("Honey Manufacturer");
        AssignBee("Egg Care");
    }

    private void AddWorker(Bee worker)
    {
        if (unassignedWorkers >= 1)
        {
            unassignedWorkers--;
            Array.Resize(ref workers, workers.Length + 1);
            workers[workers.Length - 1] = worker;
        }
    }

    private void UpdateStatusReport()
    {
        StatusReport = $"Vault report:\n{HoneyVault.StatusReport}\n" +
            $"Egg count: {eggs:0.0}\nUnassigned workers: {unassignedWorkers:0.0}\n" +
            $"{WorkerStatus("Nectar Collector")}\n{WorkerStatus("Honey Manufacturer")}" +
            $"Egg Care")\nTOTAL WORKERS: {workers.Length}";
    }

    public void CareForEggs(float eggsToConvert)
    {
        if (eggs >= eggsToConvert)
        {
            eggs -= eggsToConvert;
            unassignedWorkers += eggsToConvert;
        }
    }
}
```

Константы класса Queen чрезвычайно важны, потому что они определяют поведение программы на протяжении нескольких смен. Если матка отложит слишком много яиц, они будут есть больше меда, но также ускорят ход работ. Если рабочие, которым еще не были назначены задания, будут потреблять больше меда, это сделает актуальным более быстрое назначение заданий.

Изначально Queen назначает на работу по одной рабочей пчеле каждого типа в своем конструкторе.

Мы предоставили вам метод AddWorker. Он изменяет размер массива и добавляет объект Bee в конец массива. А вы заметили, что иногда в отчете о текущем состоянии указано, что количество рабочих, не имеющих задания, выводится как равно 1.0, но при этом добавить нового рабочего не удается? Установите точку прерывания в первую строку AddWorker — вы увидите, что значение unassignedWorkers равно 0.9999999999999999... Сможете ли вы предложить возможное решение этой проблемы?

Чтобы понять, какая информация здесь должна выводиться, следует внимательно присмотреться к отчету на снимке экрана.

Пчелы EggCare вызывают метод CareForEggs для преобразования яиц в рабочих, которым еще не назначено задание.



Решение длинных упражнений

Класс Queen управляет всей работой в программе — он отслеживает экземпляры рабочих Bee, создает новые экземпляры, когда возникает необходимость в назначении их на работу, и приказывает им начать отработку смен:

```
private string WorkerStatus(string job)
{
    int count = 0;
    foreach (Bee worker in workers)
        if (worker.Job == job) count++;
    string s = "s";
    if (count == 1) s = "";
    return $"{count} {job} bee{s}";
}

public void AssignBee(string job)
{
    switch (job)
    {
        case "Nectar Collector":
            AddWorker(new NectarCollector());
            break;
        case "Honey Manufacturer":
            AddWorker(new HoneyManufacturer());
            break;
        case "Egg Care":
            AddWorker(new EggCare(this));
            break;
    }
    UpdateStatusReport();
}

protected override void DoJob()
{
    eggs += EGGS_PER_SHIFT;
    foreach (Bee worker in workers)
    {
        worker.WorkTheNextShift();
    }
    HoneyVault.ConsumeHoney(unassignedWorkers * HONEY_PER_UNASSIGNED_WORKER);
    UpdateStatusReport();
}
}
```

← Приватный метод `WorkerStatus` использует цикл `foreach` для подсчета в массиве пчел, выполняющих конкретное задание. Обратите внимание на использование переменной «`s`» для множественного числа «`bees`», если количество пчел больше 1.

← Метод `AssignBee` использует команду `switch` для определения типа добавляемого рабочего. Строки в командах `case` должны точно соответствовать свойству `Content` каждого элемента `ListBoxItem` в `ComboBox`, в противном случае ни один из вариантов не подойдет.

← В ходе выполнения своей работы `Queen` добавляет яйца, приказывает каждому рабочему отработать следующую смену, а затем следит за тем, чтобы все свободные рабочие потребили мед. `Queen` обновляет отчет о текущем состоянии после назначения каждой пчелы на работу и отработки смены, чтобы информация постоянно была актуальной.

Queen не занимается мелочным администрированием. Класс всего лишь позволяет объектам рабочих Bee выполнить свои задания и потребить свою долю меда.

← Хороший пример разделения обязанностей: поведение, относящееся к деятельности пчелиной матки, инкапсулируется в классе `Queen`, а класс `Bee` содержит только поведение, общее для всех пчел.



Решение для длинных упражнений

Константы в начале каждого из субклассов Bee очень важны. Мы подбирали значения этих констант методом проб и ошибок: слегка изменяли одно из чисел, запускали программу и смотрели, к чему это приведет. Кажется, что нам удалось добиться неплохого баланса между классами. Как вы думаете, у нас получилось? *А может, у вас получится лучше? Да почти наверняка!*

Класс EggCare использует ссылку на объект Queen для вызова его метода CareForEggs с целью преобразования яиц в рабочих:

```
class EggCare : Bee
{
    public const float CARE_PROGRESS_PER_SHIFT = 0.15f;
    public override float CostPerShift { get { return 1.35f; } }

    private Queen queen;

    public EggCare(Queen queen) : base("Egg Care")
    {
        this.queen = queen;
    }

    protected override void DoJob()
    {
        queen.CareForEggs(CARE_PROGRESS_PER_SHIFT);
    }
}
```

Константа из EggCare определяет, с какой скоростью яйца превращаются в свободных рабочих. Большое количество рабочих может быть полезно для улья, но они также потребляют больше меда. Проблема в том, чтобы выдержать правильный баланс для рабочих разных типов.

Ниже приведен код программной части для главного окна. Он делает не так много — вся содержательная работа выполняется в других классах:

```
public partial class MainWindow : Window
{
    private Queen queen = new Queen();
    public MainWindow()
    {
        InitializeComponent();
        statusReport.Text = queen.StatusReport;
    }

    private void WorkShift_Click(object sender, RoutedEventArgs e)
    {
        queen.WorkTheNextShift();
        statusReport.Text = queen.StatusReport;
    }

    private void AssignJob_Click(object sender, RoutedEventArgs e)
    {
        queen.AssignBee(jobSelector.Text);
        statusReport.Text = queen.StatusReport;
    }
}
```

Код программной части обновляет элемент TextBox с отчетом о текущем состоянии в конструкторе, чтобы в программе всегда выводился самый актуальный отчет.

Кнопка «Assign Job» передает текст от выбранного элемента ComboBox методу Queen.AssignBee, поэтому очень важно, чтобы варианты из команды switch точно соответствовали элементам ComboBox.

Помните: если у вас возникнут проблемы с написанием кода, ничто не мешает вам заглянуть в решение!



Погодите-ка... Но разве это серьезное бизнес-приложение? Это же игра!

Обманщики.

Ладно, вы нас подловили. Да, признаем. Это игра.

А если конкретно, это **игра по управлению ресурсами**, т. е. игра, в которой механика сосредоточена на сборе, контроле и использовании ресурсов. Каждый, кто играл в симуляторы вроде SimCity или в стратегические игры вроде Civilization, хорошо понимает, что управление ресурсами является важной частью игры: игроку необходимы ресурсы (деньги, металл, топливо, дерево, вода и т. д.) для функционирования города или построения империи.

Игры по управлению ресурсами отлично подходят для экспериментов с отношениями между *механиками*, *динамикой* и *эстетикой*:

- ★ **Механика** проста: игрок назначает рабочих и запускает следующую смену. Затем каждая пчела добавляет нектар, уменьшает количество нектара/увеличивает количество меда или уменьшает количество яиц/увеличивает количество рабочих. Счетчик яиц увеличивается, и выводится отчет.
- ★ С **эстетикой** дело обстоит сложнее. Игроки испытывают стресс, когда уровни нектара или меда опускаются ниже установленных порогов и на экране появляется предупреждение о возможной нехватке. Они испытывают азарт, принимая решения, и удовлетворение от своего влияния на игру, а потом снова стресс, когда показатели перестают расти и начинают снова уменьшаться.
- ★ Игрой управляет ее динамика. В коде нет ничего, что могло бы привести к нехватке нектара или меда, — эти ресурсы только потребляются пчелами.

Не пожалейте времени и задумайтесь над этим, потому что здесь заложена суть игровой динамики. А вы найдете какие-нибудь возможности для применения этих идей в других программах (помимо игр)?

МОЗГОВОЙ ШТУРМ

Небольшое изменение в HoneyVault.NECTAR_CONVERSION_RATIO может существенно упростить или усложнить игру из-за ускорения или замедления потребления меда. Какие еще числовые характеристики влияют на игровой процесс? Как вы думаете, что определяет эти отношения?

Обратная связь направляет работу системы управления ульем

Выделим несколько минут на то, чтобы действительно разобраться, как работает игра. Коэффициент преобразования нектара серьезно влияет на игру. Если вы измените константы, это может привести к большим изменениям в игровом процессе. Если для преобразования яйца в рабочего будет достаточно небольшой порции меда, игра становится слишком простой. Если меда будет нужно много, игра значительно усложняется. Но при этом никакой настройки сложности в интерфейсе игры нет. Пчелиная матка не получает специальных улучшений, которые упрощают игру, или же опасных врагов либо сражений с боссами, усложняющих ее. Другими словами, в игре **нет кода, явно формирующего связь** между количеством яиц или рабочих и сложностью игры. Что же здесь происходит?

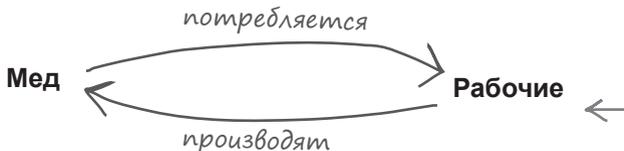
Когда вы направляете камеру на экран, на котором выводится ее видеозображение, вы тем самым создаете цикл обратной связи, порождающий эти странные узоры.

Вероятно, вам уже доводилось сталкиваться с **обратной связью**. Запустите видеозвонок между своим телефоном и компьютером. Поднесите телефон к компьютерной динамике, и вы услышите громкие эхо-шумы. Наведите камеру на экран компьютера, и вы увидите изображение экрана внутри изображения экрана внутри изображения экрана, а если повернуть телефон, то появится сюрреалистический узор. Перед вами явление обратной связи: вы берете живой видео- или аудиовывод и подаете его обратно *прямо на вход*. В коде приложения видеозвонков нет ничего, что бы конкретно генерировало эти странные изображения или звуки. Они **формируются** в результате обратной связи.



Рабочие и мед образуют цикл обратной связи

Ваша игра по управлению ульем основана на последовательности **циклов обратной связи**: множестве мелких циклов, в которых части игры взаимодействуют друг с другом. Например, производители меда добавляют мед в улей, где он потребляется производителями меда, делающими еще больше меда.

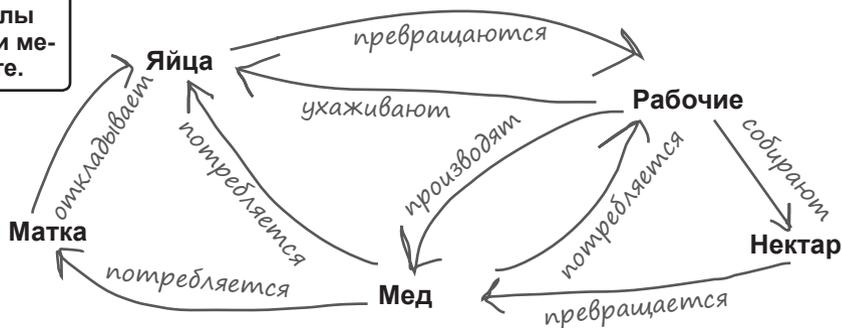


Цикл обратной связи между рабочими и медом — лишь одна крошечная часть всей системы, управляющей игрой. Удастся ли вам обнаружить ее на большей диаграмме внизу?

И это всего лишь один из циклов обратной связи. В игре присутствует много таких циклов, которые делают игру более сложной, более интересной и (хочется надеяться!) более занимательной.

Серия циклов обратной связи управляет динамикой игры. Код, построенный вами, не будет влиять на эти циклы напрямую. Они формируются теми механизмами, которые вы построите.

И эта концепция в действительности очень важна во многих реальных бизнес-приложениях, не только в играх. Все, что вы здесь узнаете, вы сможете использовать в своей повседневной работе профессионального разработчика программного обеспечения.





Механика, эстетика и динамика под увеличительным стеклом

Циклы обратной связи... Баланс... Неявное выполнение каких-то операций за счет создания системы... У вас голова еще кругом не идет? Что ж, вам предоставляется еще одна возможность применить проектирование игр для исследования более масштабной концепции из области программирования.

Ранее вы узнали о механике, динамике и эстетике — пришло время свести эти концепции воедино. **Фреймворк MDA (Mechanics-Dynamics-Aesthetics)** — формальный инструмент (здесь термин «формальный» означает «сформулированный в письменном виде»), который используется теоретиками и академиками для анализа и понимания игр. Он определяет отношения между механиками, динамикой и эстетикой и дает нам возможность обсуждать, как они формируют циклы обратной связи для взаимного влияния друг на друга.

Фреймворк MDA был разработан Робинот Ханике (Robin Hunicke), Марком Лебланом (Marc LeBlanc) и Робертом Зубеком (Robert Zubek), а его описание было опубликовано в статье «MDA: A Formal Approach to Game Design and Game Research» (2004 г.); статья вполне доступная и не содержит заумного академического жаргона. (Помните, как в главе 5 мы обсуждали, что эстетика включает испытания, повествование, тактильные ощущения, элементы фантастики и самовыражения? Все это взято из той статьи.) Не пожалейте времени и ознакомьтесь, это очень интересно: <http://bit.ly/mda-paper>.

Фреймворк MDA создавался для того, чтобы предоставить нам формальный механизм для рассмотрения и анализа видеоигр. Может показаться, что этот инструмент играет важную роль только в академической среде — например, институтском учебном курсе по проектированию игр. В действительности фреймворк MDA весьма полезен и для рядового разработчика игр, потому что он помогает нам лучше понять создаваемые игры и дает более глубокое представление о том, **что же делает эти игры интересными**.

Конечно, разработчики игр уже использовали термины «механика», «динамика» и «эстетика» на неформальном уровне, но в статье дается четкое определение, а также устанавливаются связи между ними.



В частности, фреймворк MDA помогает понять, чем различаются **взгляды на игру** у игроков и проектировщиков игр. Игрок прежде всего хочет, чтобы игра была интересной, — но вы уже знаете, что представления об «интересном» у разных игроков могут очень сильно различаться. С другой стороны, разработчики обычно рассматривают игру с позиций механики, потому что они тратят время на написание кода, проектирование уровней, создание графики и настройку механических аспектов игры.

Все разработчики (не только разработчики игр!) могут воспользоваться фреймворком MDA для более глубокого понимания циклов обратной связи

Воспользуемся фреймворком MDA для анализа классической игры Space Invaders, чтобы лучше понять циклы обратной связи.

- Начнем с механики игры: корабль игрока движется влево-вправо и стреляет снизу вверх. Пришельцы движутся строем и стреляют сверху вниз; энергетическое поле блокирует выстрелы. Чем меньше врагов остается на экране, тем быстрее они движутся.
- Игроки открывают различные стратегии: они стреляют с упреждением, отстреливают врагов на флангах вражеского построения, укрываются за защитным полем. В коде игры нет команд `if/else` или `switch` для таких стратегий; они открываются по мере того, как игрок начинает глубже понимать игру. Игроки изучают правила и получают более глубокое представление о системе, что помогает им более эффективно использовать правила. Другими словами, механики и динамики формируют цикл обратной связи.
- Движение пришельцев ускоряется, темп звукового сопровождения растет, игрок испытывает прилив адреналина. Игра становится более интересной, и в свою очередь, игроку приходится быстрее принимать решения, он совершает ошибки и меняет стратегии, что оказывает влияние на систему. Динамика и эстетика формируют другой цикл обратной связи.
- Ничто из этого не происходило по случайности. Скорость движения пришельцев, темп, звуки, графика... Все эти факторы были тщательно сбалансированы создателем игры Томохиро Нисикадо, который провел больше года за ее разработкой, черпая вдохновение из более ранних игр, из произведений Герберта Уэллса и даже своих собственных снов для создания классической игры.

Система управления ульем работает в пошаговом режиме... Преобразуем ее для работы в реальном времени

В **пошаговых играх** игровой процесс делится на части, в случае с системой управления ульем — на смены. Следующая смена начнется только после того, как вы нажмете кнопку, так что времени для назначения рабочих у вас будет более чем достаточно. Мы можем воспользоваться таймером DispatcherTimer (аналогичным тому, который использовался в главе 1) и **перевести игру в режим реального времени**, в котором игровой процесс идет непрерывно, причем для этого требуется лишь несколько строк кода.

Класс DispatcherTimer использовался в главе 1 для включения таймера в игру с поиском пар. Этот код очень похож на тот, который использовался в главе 1. Потратьте несколько минут и вернитесь к тому проекту, чтобы припомнить, как работает DispatcherTimer.

1 Добавьте команду using в начало файла MainWindow.xaml.cs.

Мы будем использовать таймер DispatcherTimer для принудительной отработки следующей смены через каждые 1.5 секунды. Класс DispatcherTimer принадлежит пространству имен System.Windows.Threading, поэтому в начало MainWindow.xaml.cs необходимо добавить следующую строку using:

```
using System.Windows.Threading;
```

2 Добавьте приватное поле, содержащее ссылку на DispatcherTimer.

Теперь нужно создать новый экземпляр DispatcherTimer. Сохраните его в приватном поле в начале класса MainWindow:

```
private DispatcherTimer timer = new DispatcherTimer();
```

3 Заставьте таймер вызывать метод-обработчик события Click кнопки WorkShift.

Таймер должен постоянно двигать игру вперед, так что если игрок не щелкнет на кнопке достаточно быстро, новая смена запускается автоматически. Начните с добавления следующего кода:

```
public MainWindow()
{
    InitializeComponent();
    statusReport.Text = queen.StatusReport;
    timer.Tick += Timer_Tick;
    timer.Interval = TimeSpan.FromSeconds(1.5);
    timer.Start();
}

private void Timer_Tick(object sender, EventArgs e)
{
    WorkShift_Click(this, new RoutedEventArgs());
}
```

Как только вы введете +=, Visual Studio предложит создать обработчик события Timer_Tick. Нажмите Tab, чтобы IDE сгенерировала метод за вас.

Таймер вызывает обработчик события Tick каждые 1.5 секунды, который, в свою очередь, вызывает обработчик события кнопки WorkShift.

А теперь запустите игру. Новая смена начинается каждые 1.5 секунды независимо от того, нажали вы кнопку или нет. Это не особо значительное изменение в механике **радикально меняет динамику игры**, что приводит к колоссальным изменениям в эстетике. А уж как игра работает лучше — в пошаговом режиме или в режиме реального времени — решать вам.

Для добавления таймера потребовалось **лишь несколько строк кода**, но от этого игра полностью изменилась. Не потому ли это произошло, что введение таймера сильно повлияло на **связи между механиками, динамикой и эстетикой?**



Да! Таймер изменил механику, это привело к изменению динамики, а та, в свою очередь, повлияла на эстетику.

Задержитесь на минуту и обдумайте этот цикл обратной связи. Изменение механики (таймер, который автоматически нажимает кнопку «Work the Next Shift» через каждые 1.5 секунды) создает совершенно новую динамику: временное окно, за которое игрок должен принять решение, иначе игра примет решение за него. Игра становится более напряженной, отчего некоторые игроки получают дозу адреналина, но у других это только вызывает стресс — эстетика изменилась. Игра становится более интересной для одних людей, но менее интересной для других.

При этом вы добавили в игру всего полдюжины строк кода, в которых не было логики «прими решение, иначе». Это еще один пример поведения, **проявляющегося** в результате совместной работы таймера и кнопки.

← И здесь тоже присутствует цикл обратной связи. Игроки, испытывающие стресс, обычно принимают менее эффективные решения, изменяя игру... Эстетика становится источником обратной связи для механики.

Похоже, все эти обсуждения циклов обратной связи важны, особенно та часть, где речь идет о **проявлении поведения**.

Циклы обратной связи и проявление поведения — важные концепции программирования.

Мы разрабатывали этот проект для того, чтобы вы могли потренироваться с наследованием, *а также* для того, чтобы предоставить вам возможность поэкспериментировать с проявляемым поведением. Речь идет о поведении, которое обусловлено не только тем, что делают ваши объекты сами по себе, но и тем, **как объекты взаимодействуют друг с другом**. Игровые константы (такие, как коэффициент преобразования нектара) являются важной частью таких проявляемых взаимодействий. Создавая это упражнение, мы сначала инициализировали эти константы некоторыми значениями, а затем стали вносить небольшие изменения, пока не получилась система, которая **не находилась** в равновесии (состояние, в котором все идеально сбалансировано), так что игрок вынужден принимать решения для того, чтобы игра продолжалась максимально возможное время. И все это обусловлено циклами обратной связи между яйцами, рабочими, нектаром, медом и пчелиной маткой.

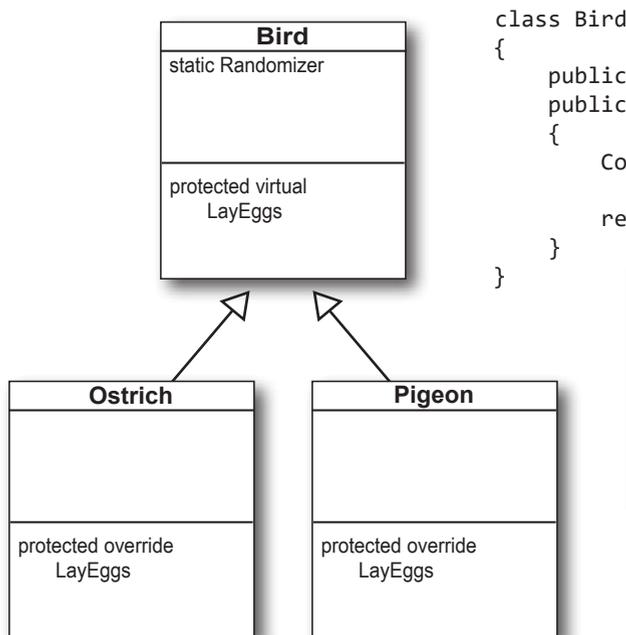


→ Попробуйте поэкспериментировать с этими циклами обратной связи. Например, увеличьте количество яиц на смену или исходный запас меда в улье — и игра станет проще. Не стесняйтесь, пробуйте! Незначительные изменения всего нескольких констант могут в корне изменить восприятие игры.

Экземпляры некоторых классов никогда не должны создаваться

Помните нашу иерархию классов из симулятора зоопарка? В любом нормальном зоопарке вы создадите экземпляры конкретных обитателей — классов Hippo, Dog или Lion... А как насчет классов Canine и Feline? Как насчет класса Animal? Оказывается, существуют классы, экземпляры которых вообще никогда не должны создаваться в программе... И даже если бы они были созданы, то никакого смысла в них не было бы!

Звучит странно? Вообще-то встречается сплошь и рядом — собственно, ранее в этой главе вы уже создали несколько классов, которые никогда не должны воплощаться в виде конкретных экземпляров.



```

class Bird
{
    public static Random Randomizer = new Random();
    public virtual Egg[] LayEggs(int numberOfEggs)
    {
        Console.Error.WriteLine
            ("Bird.LayEggs should never get called");
        return new Egg[0];
    }
}
    
```

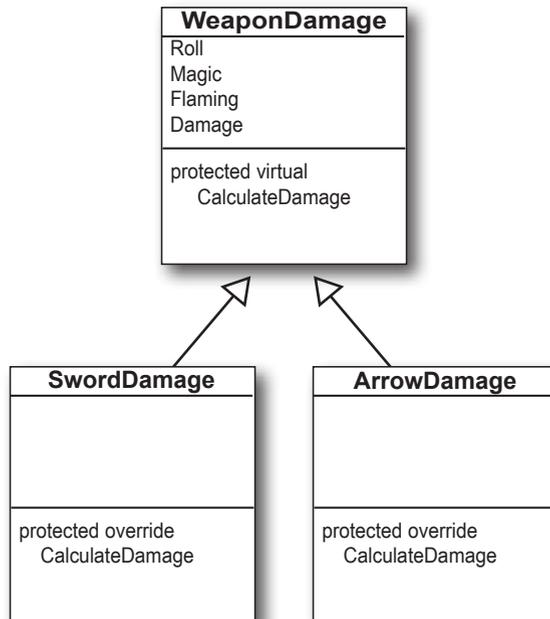
Класс Bird был совсем крошечным — он содержал только общий экземпляр Random и метод LayEggs, который существовал только для его переопределения в subclasses. Класс WeaponDamage был намного больше — он содержал целый набор свойств. Он также содержал метод CalculateDamage, предназначенный для переопределения subclasses, который вызывался из его метода WeaponDamage.

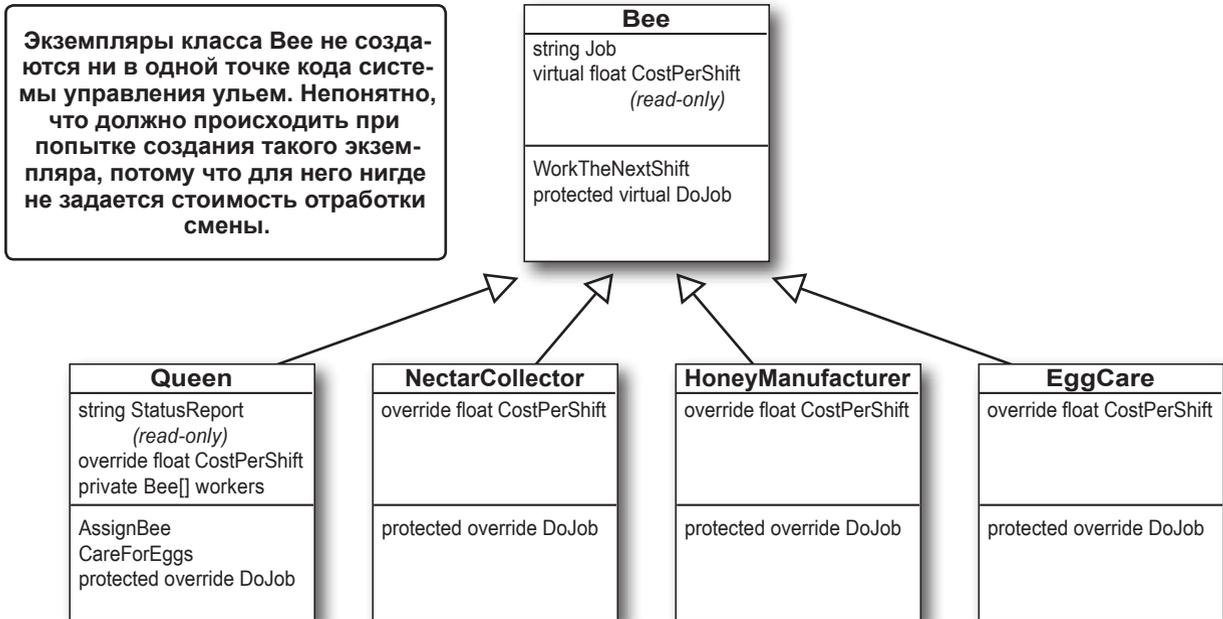
```

class WeaponDamage
{
    /* ... КОД СВОЙСТВ ... */

    protected virtual void CalculateDamage()
    {
        /* the subclass overrides this */
    }

    public WeaponDamage(int startingRoll)
    {
        roll = startingRoll;
        CalculateDamage();
    }
}
    
```





```

class Bee
{
    public virtual float CostPerShift { get; }

    public string Job { get; private set; }

    public Bee(string job)
    {
        Job = job;
    }

    public void WorkTheNextShift()
    {
        if (HoneyVault.ConsumeHoney(CostPerShift))
        {
            DoJob();
        }
    }

    protected virtual void DoJob() { /* переопределяется субклассом */ }
}
  
```

Класс `Bee` содержит метод `WorkTheNextShift`, который потребляет мед и после этого выполняет задание, назначенное пчеле, — поэтому предполагалось, что субкласс переопределяет метод `DoJob`, в котором непосредственно выполнялась работа.



Что же произойдет при попытке создания экземпляра класса `Bird`, `WeaponDamage` или `Bee`? Имеет ли это смысл хоть когда-нибудь? Будут ли работать методы таких классов?

Абстрактный класс — намеренно незавершенный класс

Достаточно часто в программе создаются классы с методами-«заполнителями», которые должны реализоваться в subclasses. Такой класс может находиться на вершине иерархии (как Bee, WeaponDamage или Bird) или же в ее середине (как Feline и Canine в модели классов симулятора зоопарка). Они пользуются тем фактом, что C# всегда вызывает самую конкретную реализацию метода — подобно тому, как WeaponDamage вызывает метод CalculateDamage, который реализован только в SwordDamage или ArrowDamage, или как Bee.WorkTheNextShift зависит от реализации метода DoJob subclassesами.

В C# для этой цели существует специальный инструмент: **абстрактный класс**. Это класс, который намеренно оставлен незавершенным; он содержит пустые компоненты, которые служат заполнителями и должны быть реализованы в subclasses. Чтобы объявить класс абстрактным, добавьте ключевое слово `abstract` в объявление класса. Об абстрактных классах необходимо знать следующее:

- ★ **Абстрактный класс работает так же, как и обычный.**
Абстрактный класс определяется точно так же, как и обычный. Он содержит поля и методы, он может наследовать от других классов и т. д.
- ★ **Абстрактный класс может иметь незавершенные компоненты-«заполнители».**
Абстрактный класс может включать объявления свойств и методов, которые должны быть реализованы наследующими классами. Метод, у которого есть объявление, но нет тела, называется **абстрактным методом**. Свойство, которое только объявляет свои методы доступа, но не определяет их, называется **абстрактным свойством**. Subclasses, расширяющие абстрактный класс, обязаны реализовать все абстрактные методы и свойства; в противном случае они также останутся абстрактными.
- ★ **Только абстрактные классы могут иметь абстрактные методы и свойства.**
Если вы включаете абстрактный метод или свойство в класс, вы должны пометить этот класс как абстрактный; в противном случае код компилироваться не будет. (Вскоре вы узнаете, как пометить класс как абстрактный.)
- ★ **Экземпляры абстрактных классов создаваться не могут.**
Конкретное — противоположность абстрактному. Конкретный метод имеет тело, и все классы, с которыми вы работали до сих пор, были конкретными. Главное различие между **абстрактным** и **конкретным** классом заключается в том, что экземпляр абстрактного класса невозможно создать командой `new`. Если вы попытаетесь это сделать, C# выдаст ошибку при компиляции кода.

Попробуйте и убедитесь! **Создайте новое консольное приложение**, добавьте в него пустой абстрактный класс и попробуйте создать экземпляр:

```
abstract class MyAbstractClass {  
  
class Program  
{  
    MyAbstractClass myInstance = new MyAbstractClass();  
}
```

Компилятор не позволяет создать экземпляр абстрактного класса, потому что абстрактные классы не предназначены для создания экземпляров.

Компилятор выдаст сообщение об ошибке и откажется строить ваш код:

```
✘ CS0144 Cannot create an instance of the abstract class or interface 'MyAbstractClass'
```



Погодите, что? Класс, экземпляр которого я даже не могу создать? Для чего вообще определять такие классы?

Например, если вы хотите предоставить часть кода, но при этом требуете, чтобы остальной код был обязательно определен subclasses.

Иногда при создании объектов, которые создаваться не должны, в программе могут происходить *разные неприятности*. Классы, находящиеся на вершине диаграммы классов, обычно содержат поля, которые должны инициализироваться subclasses. Класс Animal может выполнить вычисления, которые зависят от логического значения HasTail или Vertebrate, но при этом у него нет возможности задать это значение своими силами. *Короткий пример класса, создание экземпляра которого создает проблемы...*

```
class PlanetMission
{
    protected float fuelPerKm;
    protected long kmPerHour;
    protected long kmToPlanet;

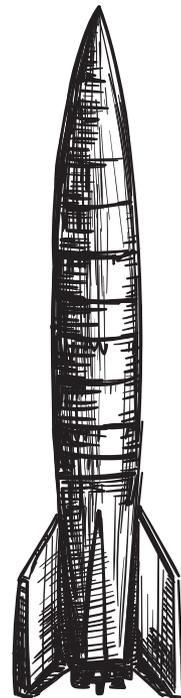
    public string MissionInfo()
    {
        long fuel = (long)(kmToPlanet * fuelPerKm);
        long time = kmToPlanet / kmPerHour;
        return $"We'll burn {fuel} units of fuel in {time} hours";
    }
}

class Mars : PlanetMission
{
    public Mars()
    {
        kmToPlanet = 92000000;
        fuelPerKm = 1.73f;
        kmPerHour = 37000;
    }
}

class Venus : PlanetMission
{
    public Venus()
    {
        kmToPlanet = 41000000;
        fuelPerKm = 2.11f;
        kmPerHour = 29500;
    }
}

class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine(new Venus().MissionInfo());
        Console.WriteLine(new Mars().MissionInfo());
        Console.WriteLine(new PlanetMission().MissionInfo());
    }
}
```

← (Делайте это!)



А вы сможете предсказать, что будет выведено на консоль, до запуска кода?

Как мы уже говорили, экземпляры некоторых классов не должны создаваться ни при каких условиях

Попробуйте запустить консольное приложение PlanetMission. Произошло ли то, что вы предполагали? На консоль выводятся две строки:

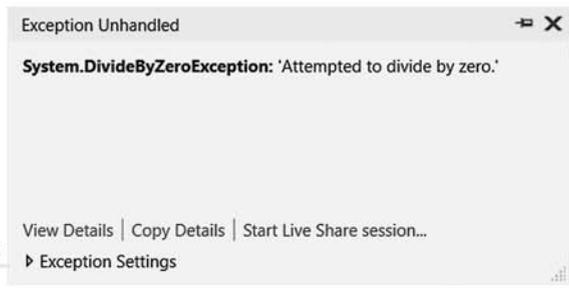
```
We'll burn 86509992 units of fuel in 1389 hours
We'll burn 159160000 units of fuel in 2486 hours
```

А потом происходит исключение.

Все проблемы начались с создания экземпляра класса PlanetMission. Его метод FuelNeeded ожидает, что значения полей будут заданы субклассом. Если этого не происходит, поля сохраняют значение по умолчанию — нуль. И когда C# пытается разделить число на нуль...

```
class PlanetMission
{
    protected float fuelPerKm;
    protected long kmPerHour;
    protected long kmToPlanet;

    public string MissionInfo()
    {
        long fuel = (long)(kmToPlanet * fuelPerKm);
        long time = kmToPlanet / kmPerHour;
        return $"We'll burn {fuel} units of fuel in {time} hours";
    }
}
```



Решение: используйте абстрактный класс

Если класс помечен ключевым словом `abstract`, C# не позволит вам написать код создания его экземпляра. Как же это решает проблему? По старой поговорке: профилактика лучше лечения. Добавьте ключевое слово `abstract` в объявление класса PlanetMission:

```
abstract class PlanetMission
{
    // Остальной код класса остается без изменений
}
```

Как только вы внесете это изменение, компилятор сообщит об ошибке:

```
CS0144 Cannot create an instance of the abstract class or interface 'PlanetMission'
```

Код вообще не компилируется, а если нет откомпилированного кода, то нет и исключения. Ситуация напоминает использование ключевого слова `private` в главе 5 или же ключевых слов `virtual` и `override` ранее в этой главе. Объявление компонентов класса приватными не изменяет его поведение, оно всего лишь не позволит вашему коду построиться при нарушении инкапсуляции. Ключевое слово `abstract` работает аналогичным образом: вы не получите исключение при создании экземпляра абстрактного класса, потому что компилятор C# не позволит вам создать этот экземпляр.

Если добавить ключевое слово **abstract** в объявление класса, то компилятор будет выдавать ошибку при любых попытках создания экземпляра этого класса.

У абстрактных методов нет тела

У класса `Bird`, который вы построили ранее, экземпляры создаваться не должны. Именно поэтому он использует `Console.Error` для вывода сообщения об ошибке, если программа попытается создать экземпляр и вызвать метод `LayEggs`:

```
class Bird
{
    public static Random Randomizer = new Random();
    public virtual Egg[] LayEggs(int numberOfEggs)
    {
        Console.Error.WriteLine
            ("Bird.LayEggs should never get called");
        return new Egg[0];
    }
}
```

Жизнь абстрактного метода ужасна.
Ведь это жизнь без тела.

Так как мы хотим предотвратить создание экземпляров класса `Bird`, добавим ключевое слово `abstract` в его объявление. Тем не менее этого недостаточно — кроме запрета на создание экземпляров, мы также хотим **потребовать**, чтобы каждый subclass, расширяющий `Bird`, **обязательно переопределял метод `LayEggs`**.

Именно это происходит при добавлении ключевого слова `abstract` к компоненту класса. **Абстрактный метод** существует только в виде объявления, но у него *нет тела метода*, которое должно быть реализовано каждым subclassом, расширяющим абстрактный класс. **Тело** метода состоит из кода, заключенного в фигурные скобки и следующего после объявления, — и это то, чего у абстрактных методов не бывает по определению.

Вернитесь к проекту `Bird` и **замените класс `Bird`** следующим абстрактным классом:

```
abstract class Bird
{
    public static Random Randomizer = new Random();
    public abstract Egg[] LayEggs(int numberOfEggs);
}
```

Ваша программа работает точно так же, как прежде! Но попробуйте включить следующую строку в метод `Main`:

```
Bird abstractBird = new Bird();
```

Компилятор выдает сообщение об ошибке:

```
CS0144 Cannot create an instance of the abstract class or interface 'Bird'
```

Попробуйте добавить тело к методу `LayEggs`:

```
public abstract Egg[] LayEggs(int numberOfEggs)
{
    return new Egg[0];
}
```

Вы снова получите ошибку компиляции, только другую:

```
CS0500 'Bird.LayEggs(int)' cannot declare a body because it is marked abstract
```



Если абстрактный класс содержит виртуальные компоненты, то каждый subclass должен переопределять все такие компоненты.

Абстрактные свойства работают как абстрактные методы

Вернемся к классу `Bee` из предыдущего примера. Мы уже знаем, что экземпляры этого класса создаваться не должны, поэтому преобразуем его в абстрактный класс. Для этого достаточно добавить модификатор `abstract` в объявление класса и преобразовать `DoJob` в абстрактный метод без тела:

```
abstract class Bee
{
    /* Остальной код класса остается без изменений */
    protected abstract void DoJob();
}
```

Однако существует еще один виртуальный компонент — и это не метод. Мы говорим о свойстве `CostPerShift`, которое вызывается методом `Bee.WorkTheNextShift` для определения того, сколько меда потребуется пчеле на эту смену:

```
public virtual float CostPerShift { get; }
```

В главе 5 вы узнали, что свойства в действительности представляют собой обычные методы, к которым вы обращаетесь как к полям. Для создания абстрактного свойства используется ключевое слово `abstract`, как и в случае с методом:

```
public abstract float CostPerShift { get; }
```

Абстрактные свойства могут иметь `get`-метод и/или `set`-метод доступа. `Set`- и `get`-методы в абстрактных свойствах **не могут иметь тела**. Их объявления выглядят как объявления автоматических свойств, но таковыми не являются, потому что не содержат никакой реализации. Абстрактные свойства, как и абстрактные методы, представляют собой «заполнители» для свойств, которые должны быть реализованы любым subclasses, расширяющим свой класс.

Ниже приведен полный код абстрактного класса `Bee`, вместе с абстрактным методом и свойством:

```
abstract class Bee
{
    public abstract float CostPerShift { get; }
    public string Job { get; private set; }

    public Bee(string job)
    {
        Job = job;
    }

    public void WorkTheNextShift()
    {
        if (HoneyVault.ConsumeHoney(CostPerShift))
        {
            DoJob();
        }
    }

    protected abstract void DoJob();
}
```

Замените!

Замените класс `Bee` в системе управления ульем новым абстрактным классом. Приложение все равно будет работать! А теперь попробуйте создать экземпляр класса `Bee` командой `new Bee()`; компилятор выдаст сообщение об ошибке. Но что еще важнее, *если вы попытаетесь расширить `Bee`, но забудете реализовать `CostPerShift`*, произойдет ошибка.



Упражнение

Пришло время потренироваться в использовании абстрактных классов. К счастью, искать кандидатов для преобразования в абстрактные классы долго не придется.

Ранее в этой главе мы изменили классы `SwordDamage` и `ArrowDamage` так, чтобы они расширяли новый класс с именем `WeaponDamage`. Преобразуйте класс `WeaponDamage` в абстрактный. В классе `WeaponDamage` также имеется хороший кандидат для преобразования в абстрактный метод — преобразуйте его.

Часто задаваемые вопросы

В: Когда я помечаю класс как абстрактный, как это влияет на его поведение? Методы или свойства абстрактного класса чем-то отличаются от методов или свойств конкретного класса?

О: Нет, абстрактные классы работают точно так же, как любые другие классы. Когда вы добавляете ключевое слово `abstract` в объявление класса, компилятор C# делает две вещи: (1) он запрещает использовать класс в командах `new`, (2) позволяет включать в класс абстрактные методы и свойства.

В: Некоторые абстрактные классы, которые вы показывали, были открытыми; другие были защищенными (`protected`). На что это влияет? И важен ли порядок этих ключевых слов в объявлении класса?

О: Абстрактные методы могут иметь любой модификатор доступа. Если вы объявите абстрактный метод приватным, то классы, реализующие этот абстрактный метод, также должны объявить его приватным. Порядок ключевых слов ни на что не влияет. Объявления `protected abstract void DoJob()`; и `abstract protected void DoJob()`; делают абсолютно одно и то же.

В: Меня смущает то, как вы используете термины «реализовать» или «реализация». Что вы имеете в виду, говоря о реализации абстрактного метода?

О: Когда вы используете ключевое слово `abstract` для объявления абстрактного метода или свойства, вы фактически **определяете** абстрактный компонент класса. Позднее при добавлении в конкретный класс завершеного метода или свойства с таким же объявлением вы **реализуете** этот компонент. Итак, вы определяете абстрактные методы или свойства в абстрактном классе и реализуете их в конкретных классах, которые расширяют этот абстрактный класс.

В: Я так и не понял, в чем смысл того, что ключевое слово `abstract` не позволяет моему коду компилироваться, если я пытаюсь создать экземпляр абстрактного класса. Я уже потратил достаточно времени на поиск и исправление всех ошибок компиляции. Зачем же мне усложнять построение своего кода?

О: Когда вы только изучаете программирование, ошибки компилятора «CS» становятся досадной помехой. Всем нам доводилось тратить время на поиски пропущенной запятой, точки или вопросительного знака, чтобы очистить список `Error List`. Тогда зачем использовать ключевые слова вроде `abstract` или `private`, которые только создают больше ограничений для вашего кода и повышают вероятность ошибок компиляции? На первый взгляд это выглядит противостественно. Если не использовать ключевое слово `abstract`, то вы никогда не получите ошибку компилятора «Невозможно создать экземпляр абстрактного класса». Тогда зачем его использовать?

Причина, по которой мы используем такие ключевые слова, как `abstract` и `private`, препятствующие построению вашего кода в некоторых случаях, проста. Исправить ошибку компилятора «Невозможно создать экземпляр абстрактного класса» гораздо проще, чем ту ошибку, которую это сообщение предотвращает. Если у вас имеется класс, экземпляры которого никогда не должны создаваться в программе, случайное создание экземпляра этого класса вместо одного из субклассов может стать исключительно коварной и трудноуловимой ошибкой. Добавление ключевого слова `abstract` в базовый класс **ускоряет проявление сбоя** с ошибкой, которая проще исправляется.

Ошибки, возникающие из-за создания экземпляра базового класса, который создаваться никогда не должен, бывают особенно коварными и нетривиальными. Объявление класса абстрактным ускоряет проявление сбоя в вашем коде, если вы попытаетесь создать экземпляр этого класса.



Упражнение Решение

Спасибо за рефакторинг!
Уверен, вы предотвратили немало противных ошибок в будущем. Теперь я могу больше думать о своей игре, а не о коде.

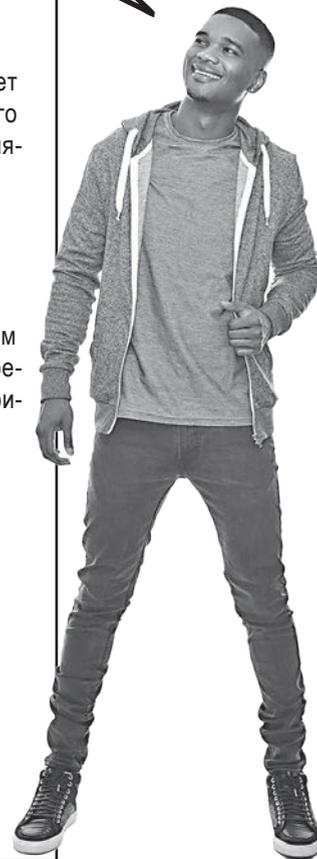
Отличная работа!

Экземпляры класса `WeaponDamage` создаваться не должны — этот класс существует только для того, чтобы классы `SwordDamage` и `ArrowDamage` могли наследовать его свойства и методы. А значит, будет логично объявить этот класс абстрактным. Взгляните на его метод `CalculateDamage`:

```
protected virtual void CalculateDamage() {  
    /* переопределяется субклассом */  
}
```

Этот метод становится отличным кандидатом для преобразования в абстрактном классе, потому что он существует только для того, чтобы субклассы могли переопределить его собственными реализациями, обновляющими свойство `Damage`. Ниже приведено все необходимое для внесения изменений в класс `WeaponDamage`:

```
abstract class WeaponDamage  
{  
    /* Свойства Damage, Roll, Flaming и Magic остаются  
    неизменными */  
  
    protected abstract void CalculateDamage();  
  
    public WeaponDamage(int startingRoll)  
    {  
        roll = startingRoll;  
        CalculateDamage();  
    }  
}
```



Это был первый раз, когда вы перечитывали код, написанный для предыдущих упражнений?

Возвращение к написанному ранее коду может восприниматься немного странно. Но на самом деле многие разработчики так поступают, и это полезная привычка. Вы находили в коде что-то такое, что со второго захода сделали бы иначе? Обнаруживали какие-то улучшения или изменения, которые стоило бы внести в код? Никогда не жалеете времени на рефакторинг вашего кода. Именно это было сделано в данном упражнении: мы изменили структуру кода без изменения его поведения. *Это и называется рефакторингом.*



o o

Наследование — очень полезная штука. Я могу определить метод в базовом классе, и он автоматически появится в каждом из subclasses. А если я захочу сделать это с двумя методами в двух разных классах? Может ли один subclass **расширять два базовых класса?**

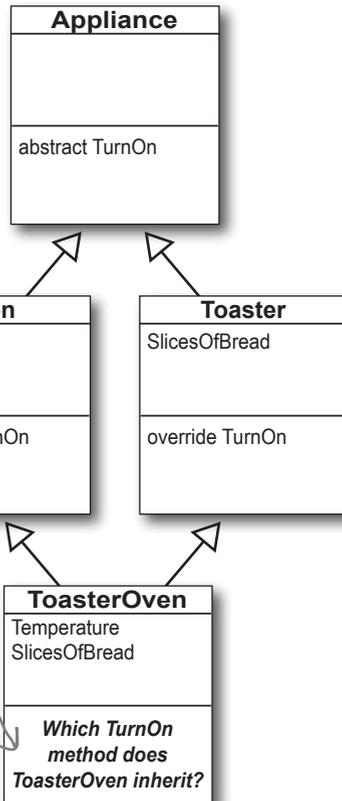
Звучит заманчиво! Но есть одна проблема.

Если бы C# допускал наследование от нескольких базовых классов, в программах открылась бы целая куча неприятностей. Если язык позволяет одному subclassу наследовать от двух базовых классов, это называется **множественным наследованием**. А если бы в C# поддерживалось множественное наследование, то это неизбежно привело бы к колоссальной и почти неразрешимой проблеме, которая называется...

Это настоящее название! Некоторые разработчики также называют ее «проблемой ромбовидного наследования».

Смертельный ромб

Печь (Oven) и Тостер (Toaster) наследуют от класса Электроприбор (Appliance) и переопределяют метод TurnOn. Если вам понадобится класс ToasterOven, было бы очень удобно, если бы мы могли унаследовать Temperature от Oven и SlicesOfBread от Toaster.



Метод TurnOn переопределяется обоими классами, Oven и Toaster. Если бы C# позволил расширить Oven одновременно с Toaster, какую версию TurnOn должен был бы получить класс ToasterOven?

Что происходило бы в БЕЗУМНОМ мире, в котором в C# поддерживалось бы множественное наследование? Давайте сыграем в «Что, если...» и выясним это.

Что, если... у вас имеется класс с именем Appliance, который содержит абстрактный метод с именем TurnOn?

И что, если... он имеет два subclassа: Oven со свойством Temperature и Toaster со свойством SlicesOfBread?

И что, если... вы хотите создать класс ToasterOven, наследующий как Temperature, так и SlicesOfBread?

И что, если... в C# поддерживалось бы множественное наследование и такое было бы возможно?

Остается последний вопрос...

Какую версию TurnOn унаследует ToasterOven?

Версию из Oven? Или версию из Toaster?

Это **НЕВОЗМОЖНО** определить заранее!

И именно поэтому множественное наследование в C# не поддерживается.



А хорошо бы, чтобы в C# было что-то вроде абстрактного класса, но только без проблемы ромбовидного наследования, чтобы C# позволял расширить сразу несколько таких псевдоклассов одновременно?

Но, наверное, об этом можно только мечтать...

КЛЮЧЕВЫЕ МОМЕНТЫ

- Субкласс может переопределять унаследованные компоненты, заменяя их новыми методами или свойствами с такими же именами.
- Чтобы переопределить метод или свойство, добавьте **ключевое слово** `virtual` в базовый класс, а затем добавьте **ключевое слово** `override` в одноименный метод или свойство в субклассе.
- **Ключевое слово** `protected` — модификатор доступа, с которым компонент класса является открытым только для своих субклассов, но остается приватным для всех остальных классов.
- Когда субкласс переопределяет метод из своего базового класса, то всегда вызывается **самая конкретная версия**, определенная в субклассе, даже если она вызывается базовым классом.
- Если субкласс просто добавляет метод с таким же именем, как у метода базового класса, он просто **скрывает** метод базового класса вместо того, чтобы переопределить его. Используйте **ключевое слово** `new` для сокрытия методов.
- **Динамика** игры описывает то, как механики комбинируются и взаимодействуют друг с другом для управления игровым процессом.
- Субкласс может обратиться к своему базовому классу при помощи **ключевого слова** `base`.
- Субкласс и базовый класс могут иметь **разные конструкторы**. Субкласс может выбрать, какие значения передать конструктору базового класса.
- Постройте **модель классов на бумаге**, прежде чем писать код. Это поможет вам лучше понять суть проблемы.
- Если перекрытие между классами минимально, это является проявлением важного принципа проектирования, называемого **разделением обязанностей**.
- **Проявляемое поведение** раскрывается при взаимодействии объектов друг с другом, за границами логики, напрямую запрограммированной в них.
- **Абстрактные классы** представляют собой намеренно незавершенные классы, экземпляры которых не могут создаваться в программе.
- Если вы добавите **ключевое слово** `abstract` в метод или свойство и исключите его тело, метод или свойство становятся абстрактными. Любой конкретный субкласс абстрактного класса должен реализовать этот метод или свойство.
- В процессе **рефакторинга** разработчик читает ранее написанный код и вносит в него улучшения, не изменяя его поведения.
- В C# не поддерживается множественное наследование из-за **проблемы ромбовидного наследования**: C# не может определить, какую версию компонента, унаследованного от двух базовых классов, следует использовать.

Лабораторный курс Unity № 3

Экземпляры GameObject

C# является объектно-ориентированным языком. А поскольку суть всех лабораторных работ Unity заключается в **получении практического опыта написания кода C#**, вполне разумно, что в этих лабораторных работах центральное место занимает создание объектов.

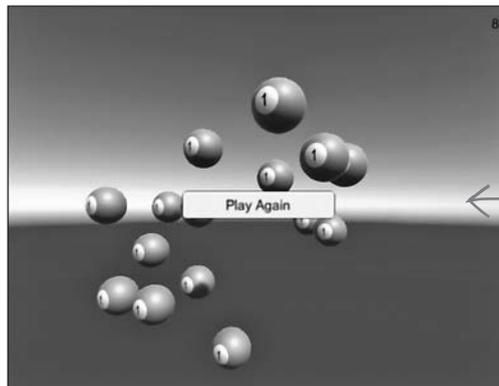
Вы уже создавали объекты в C# с того момента, когда узнали о существовании ключевого слова `new` в главе 3. В этой лабораторной работе Unity мы **создадим экземпляры объектов Unity GameObject** и воспользуемся ими в полноценной, работоспособной игре. Это станет отличной отправной точкой для написания Unity-игр на C#.

В следующих двух лабораторных работах Unity мы **создадим простую игру**, в которой будет задействован уже знакомый вам бильярдный шар. В этой игре мы возьмем за основу то, что вы ранее узнали об объектах C# и экземплярах. Мы воспользуемся заготовками (инструментом Unity для создания экземпляров GameObject) для создания множества экземпляров GameObject, а затем используем сценарии, чтобы заставить объекты GameObject летать в трехмерном пространстве игры.

Построим игру в Unity!

Платформа Unity предназначена для создания игр. Таким образом, в следующих двух лабораторных работах Unity вы примените то, что знаете о C#, для построения простой игры. Игра выглядит примерно так:

При запуске игры сцена медленно заполняется бильярдными шарами. Игрок должен щелкнуть на шарах, чтобы они исчезали с экрана. Если в сцене накопится 15 шаров, игра завершается.

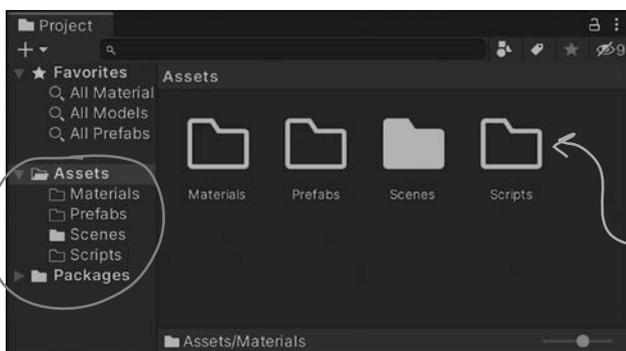


В правом верхнем углу выводится текущий счет. За каждый шар, на котором щелкнул игрок, ему начисляется победное очко.

После завершения игры кнопка Play Again запускает новую игру.

Итак, за дело! Как обычно, первое, что необходимо сделать, — создать проект Unity. На этот раз мы будем хранить файлы в чуть более структурированном виде, поэтому для материалов и сценариев будут созданы отдельные папки и еще одна папка — для заготовок (о них вы узнаете позднее в этой главе).

1. Прежде чем браться за дело, закройте все открытые проекты Unity. Также закройте среду Visual Studio — Unity откроет ее за вас.
2. **Создайте новый проект Unity** по 3D-шаблону, как это делалось в предыдущих лабораторных работах Unity. Присвойте ему имя, которое поможет вам запомнить, в каких лабораторных работах он используется (например, «Unity Labs 3 and 4»).
3. Выберите макет Wide, чтобы ваш экран соответствовал приводимым снимкам экрана.
4. Создайте папку для материалов внутри папки Assets. **Щелкните правой кнопкой мыши на папке Assets** в окне Project, выберите команду Create>>Folder. Присвойте ей имя **Materials**.
5. Создайте в Assets новую папку с именем **Scripts**.
6. Создайте в Assets еще одну папку с именем **Prefabs**.



Проследите за тем, чтобы папки Materials, Scripts и Prefabs были созданы внутри папки Assets.

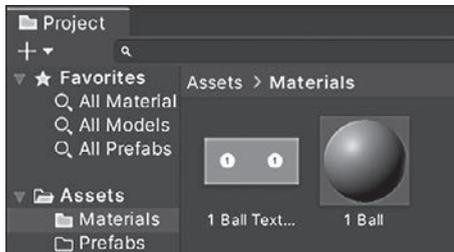
В окне Project пустые папки отображаются в контурном виде.

Создайте новый материал в папке Materials

Сделайте двойной щелчок на папке Materials, чтобы открыть ее. В этой папке мы создадим новый материал.

Перейдите по адресу <https://github.com/head-first-csharp/fourth-edition> и щелкните на ссылке Billiard Ball Textures (как это было сделано в первой лабораторной работе Unity), чтобы загрузить файл текстуры **1 Ball Texture.png** в папку на вашем компьютере. Перетащите загруженный файл в папку Materials — так же, как было сделано с загруженным файлом в первой лабораторной работе Unity, только на этот раз перетащите его в созданную вами папку Materials (вместо родительской папки Assets).

Теперь можно создать новый материал. Щелкните правой кнопкой мыши на папке Materials в окне Project и **выберите команду Create>>Material**. Присвойте новому материалу имя **1 Ball**. Он должен появиться в папке Materials в окне Project.



В предыдущих лабораторных работах Unity мы использовали текстуру — растровый графический файл, в который Unity может «оборачивать» объекты GameObject. При перетаскивании текстуры на сферу Unity автоматически создает материал, который используется Unity для хранения информации о том, как должен визуализироваться объект GameObject, содержащий ссылку на текстуру. На этот раз материал создается вручную. Как и в предыдущем упражнении, для загрузки PNG-файла текстуры необходимо щелкнуть на кнопке Download на странице GitHub.

Проследите за тем, чтобы в окне Materials был выбран материал 1 Ball; он должен отображаться в окне Inspector. Щелкните на файле *1 Ball Texture* и перетащите его на поле слева от метки Albedo.



Выберите материал 1 Ball в окне Project, чтобы просмотреть его свойства. Перетащите карту текстуры на поле слева от метки Albedo.

Крошечное изображение текстуры 1 Ball должно появиться в поле слева от пункта Albedo в окне Inspector.



Теперь при наложении на сферу вашего материала результат выглядит как бильярдный шар.



Объекты GameObject отражают свет своими поверхностями.

За сценой



Когда вы видите в игре Unity объект, обладающий цветом или текстурной картой, вы на самом деле видите поверхность объекта GameObject, отражающую свет в сцене. Цветом этой поверхности управляет **альбе́до**. Термин «альбе́до» происходит из физики (а конкретно из астрономии), и он обозначает цвет, отражаемый объектом. Дополнительную информацию об альбе́до можно найти в руководстве Unity. Выберите команду «Unity Manual» в меню Help, чтобы открыть руководство в браузере, и проведите поиск по строке «albedo» — одна из страниц руководства объясняет зависимость цвета и прозрачности от альбе́до.

Создание бильярдного шара в случайной точке сцены

Создайте новый объект Sphere при помощи сценария OneBallBehaviour:

- ★ Выберите команду 3D Object>>Sphere из меню GameObject, чтобы **создать сферу**.
- ★ Перетащите на нее новый **материал 1 Ball**, чтобы сфера выглядела как бильярдный шар.
- ★ Затем **щелкните правой кнопкой мыши на папке Scripts**, созданной вами в окне Project, и создайте **новый сценарий C#** с именем OneBallBehaviour.
- ★ **Перетащите сценарий на сферу** в окне Hierarchy. Выделите сферу и убедитесь в том, что в окне Inspector отображается компонент Script с именем «One Ball Behaviour».

Сделайте двойной щелчок на сценарии, чтобы отредактировать его в Visual Studio. Добавьте точно такой же код, который вы использовали в BallBehaviour из первой лабораторной работы Unity, а затем **закомментируйте строку Debug.DrawRay** в методе Update.

Ваш сценарий OneBallBehaviour должен выглядеть так:

```
public class OneBallBehaviour : MonoBehaviour
{
    public float XRotation = 0;
    public float YRotation = 1;
    public float ZRotation = 0;
    public float DegreesPerSecond = 180;

    // Start вызывается перед первым обновлением кадра
    void Start()
    {
    }

    // Update вызывается один раз на кадр
    void Update()
    {
        Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
        transform.RotateAround(Vector3.zero, axis, DegreesPerSecond * Time.deltaTime);
        // Debug.DrawRay(Vector3.zero, axis, Color.yellow);
    }
}
```

Мы не включаем строки using в код сценария, а просто считаем, что они есть.

Когда вы добавляете **метод Start** к объекту GameObject, Unity будет вызывать этот метод каждый раз, когда в сцену добавляется новый экземпляр этого объекта. Если метод Start находится в сценарии, присоединенном к объекту GameObject, который отображается в окне Hierarchy, этот метод будет вызван **сразу же после запуска игры**.

Unity часто создает экземпляр объекта GameObject за какое-то время до того, как он будет добавлен в сцену. Метод Start объекта GameObject вызывается только при фактическом добавлении GameObject в сцену.

Эта строка не понадобится, закомментируйте ее.

Теперь измените метод Start так, чтобы при создании сфера перемещалась в случайную позицию. Для этого мы воспользуемся свойством **transform.position**, которое изменяет позицию объекта GameObject в сцене. Ниже приведен код позиционирования шара в случайной точке — **добавьте его в метод Start** в сценарии OneBallBehaviour:

```
// Start вызывается перед первым обновлением кадра
void Start()
{
    transform.position = new Vector3(3 - Random.value * 6,
        3 - Random.value * 6, 3 - Random.value * 6);
}
```

Помните: кнопка Play не сохраняет вашу игру! Сохраняйтесь пораньше, сохраняйтесь почаще...

Запустите свою игру кнопкой Play в Unity. Шар должен вращаться по оси Y в случайной точке. Остановите и запустите игру несколько раз. Каждый раз шар должен появляться в новой точке сцены.

Применение отладчика для понимания Random.value

Мы уже неоднократно использовали класс Random из пространства имен .NET System – в частности, для распределения пар животных в игре из главы 1, а также для выбора случайных карт в главе 3. Но сейчас перед вами другой класс Random – попробуйте навести указатель мыши на ключевое слово Random в Visual Studio.

Оба этих класса называются Random, но если навести указатель мыши на них в Visual Studio, то вы увидите, что использовавшийся ранее класс принадлежит пространству имен System. А сейчас мы используем класс Random из пространства имен UnityEngine.

```

// Start is called before the first frame update
void Start()
{
    transform.position = new Vector3(3 - Random.value * 6,
        3 - Random.value * 6, 3 - Random.value * 6);
}
    
```

```

{
    static Random random = new Random();

    public static string[] pickedCards = new string[numberOfCards];
}
    
```

class System.Random
Represents a pseudo-random number generator, which is a device that produces a sequence of numbers that meet certain statistical requirements for randomness.

class UnityEngine.Random
Class for generating random data.

Из кода выбора случайных карт, написанного вами ранее.

Из кода видно, что новый класс Random отличается от того, который использовался ранее. Тогда для получения случайного значения вызывался метод Random.Next и полученное значение было целым числом. В новом коде используется **Random.value**, но это не метод, а свойство.

Воспользуйтесь отладчиком Visual Studio для просмотра видов значений, которые вам может предоставить новый класс Random. Щелкните на кнопке «Attach to Unity» (▶ Attach to Unity в Windows, ▶ Debug > Attach to Unity в macOS), чтобы присоединить Visual Studio к Unity. Установите точку прерывания в строке, добавленной в метод Start.

Теперь вернитесь в Unity и **запустите игру**. Она должна прерваться сразу же после нажатия кнопки Play. Задержите указатель над Random.value (проследите за тем, чтобы он располагался именно над value). Visual Studio выведет значение в подсказке:

```

13 void Start()
14 {
15     transform.position = new Vector3(3 - Random.value * 6,
16         3 - Random.value * 6, 3 - Random.value * 6);
17 }
    
```

Random.value 0.4680484

Оставьте среду Visual Studio присоединенной к Unity, затем вернитесь в редактор Unity и **остановите игру** (в редакторе Unity, не в Visual Studio). Снова запустите игру. Проведите это еще несколько раз. Каждый раз вы будете получать новое случайное значение. Так работает класс UnityEngine.Random: он выдает новое случайное значение от 0 до 1 при каждом обращении к его свойству value.

Нажмите кнопку Continue (▶ Continue), чтобы продолжить игру. Точка прерывания была установлена только в методе Start, который вызывается один раз для каждого экземпляра GameObject, поэтому повторного прерывания не будет. Вернитесь в Unity и остановите игру.

Вы не сможете редактировать сценарии в среде Visual Studio, пока она присоединена к Unity, поэтому щелкните на кнопке Stop Debugging, чтобы отсоединить отладчик Visual Studio от Unity.

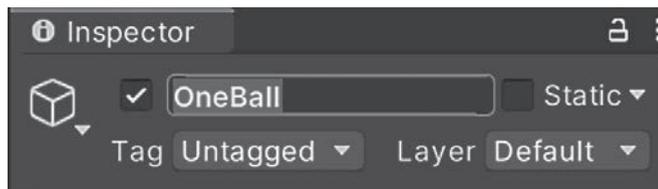
Возможно, Unity предложит вам включить отладку, как это произошло в последней лабораторной работе Unity.

Оставьте среду Visual Studio присоединенной к Unity и несколько раз перезапустите игру. При каждом перезапуске вы будете получать новое случайное число в зоне от 0 до 1.

Преобразование объекта GameObject в заготовку

В Unity **заготовка** (prefab) представляет собой объект GameObject, экземпляр которого можно создать в сцене. В нескольких последних главах мы работали с экземплярами объектов и создавали объекты посредством создания экземпляров классов. Unity предоставляет возможность использования объектов и экземпляров, чтобы вы могли строить игры, повторно использующие одни и те же объекты GameObject. Преобразуем объект GameObject в заготовку.

У объектов GameObject есть имена. Переименуйте свой объект GameObject в *OneBall*. Для начала **выделите сферу**, щелкнув на ней в окне Hierarchy или в сцене. Затем в окне Inspector **измените ее имя на OneBall**.



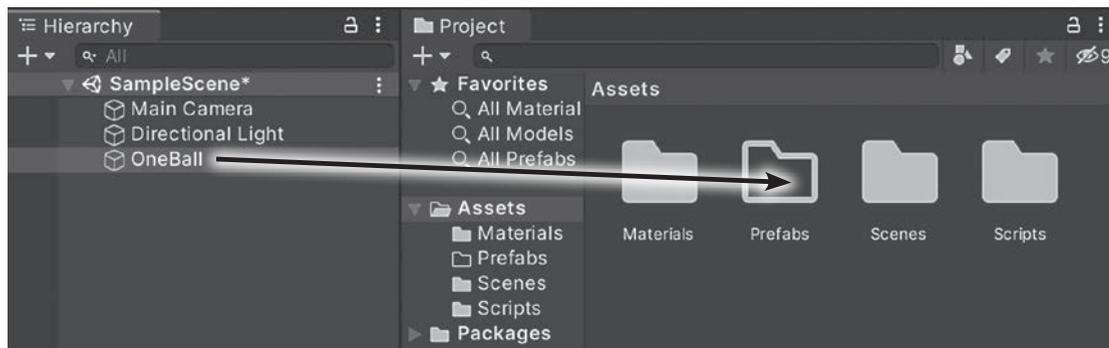
Будьте осторожны!

Visual Studio не позволяет редактировать код во время соединения с Unity.

Если вы пытаетесь редактировать код, но обнаруживаете, что Visual Studio не дает вносить изменения, скорее всего, среда Visual Studio все еще присоединена к Unity! Нажмите квадратную кнопку Stop Debugging, чтобы разорвать связь между ними.

Также объект GameObject можно переименовать еще одним способом: щелкните на нем правой кнопкой мыши в окне Hierarchy и выберите команду Rename.

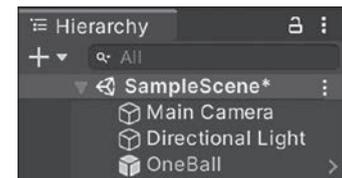
Теперь объект GameObject можно преобразовать в заготовку. **Перетащите OneBall из окна Hierarchy в папку Prefabs.**



Объект OneBall должен появиться в папке Prefabs. Обратите внимание: OneBall в окне Hierarchy окрашивается в синий цвет. Это означает, что он стал заготовкой (prefab). Для некоторых игр это нормально, но в игре, которую мы строим, все экземпляры сфер должны создаваться сценариями.

Щелкните правой кнопкой мыши на OneBall в окне Hierarchy, **удалите объект GameObject OneBall из сцены**. Вы должны видеть его только в окне Project, но не в окне Hierarchy или в сцене.

**А вы не забываете сохранять свою сцену в ходе работы?
«Сохраняйтесь пораньше, сохраняйтесь почаще!»**



Когда объект GameObject окрашен в синий цвет в окне Hierarchy, Unity тем самым показывает, что перед вами экземпляр заготовки.

Создание сценария для управления игрой

Игра должна каким-то образом добавлять шары в сцену (а также вести текущий счет и следить за тем, не завершилась ли она).

Щелкните правой кнопкой мыши на папке Scripts в окне Project и **создайте новый сценарий с именем GameController**. В новом сценарии будут использоваться два метода, доступные в любом сценарии GameObject:

- ★ **Метод Instantiate** создает **новый экземпляр GameObject**. При создании объектов GameObject в Unity ключевое слово new обычно не используется (как это делалось в главе, например). Вместо этого вызывается метод Instantiate, который мы будем вызывать из метода AddABall.
- ★ **Метод InvokeRepeating** снова и снова вызывает другой метод в сценарии. В данном случае он ожидает 1.5 секунды, а затем вызывает метод AddABall один раз в секунду все оставшееся время игры.

Исходный код выглядит так:

```
public class GameController : MonoBehaviour
{
    public GameObject OneBallPrefab;

    void Start()
    {
        InvokeRepeating("AddABall", 1.5F, 1);
    }

    void AddABall()
    {
        Instantiate(OneBallPrefab);
    }
}
```

К какому типу относится второй аргумент, передаваемый InvokeRepeating?

Метод InvokeRepeating в Unity снова и снова вызывает другой метод. В первом параметре передается строка с именем вызываемого метода.

Метод с именем AddABall. Его единственная задача — создание экземпляра заготовки.

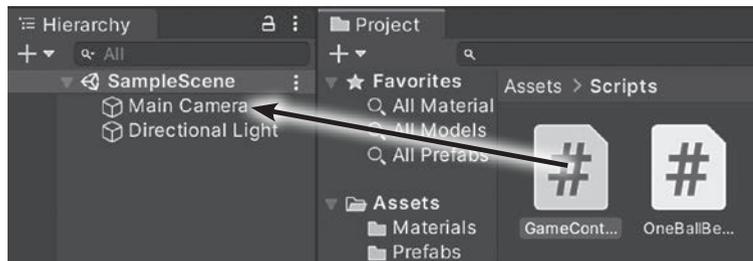
Мы передаем поле OneBallPrefab в параметре метода Instantiate, который будет использоваться Unity для создания экземпляра вашей заготовки.



Unity выполняет только сценарии, присоединенные к объектам GameObject в сцене. Сценарий GameController будет создавать экземпляры нашей заготовки OneBall, но его необходимо к чему-то присоединить. К счастью, вы уже знаете, что камера представляет собой обычный объект GameObject с компонентом Camera (а также AudioListener). Главная камера (Main Camera) всегда доступна в сцене. Как вы думаете, как нам следует поступить с только что созданным сценарием GameController?

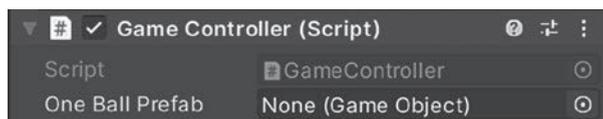
Присоединение сценария к главной камере

Чтобы сценарий GameController работал, его необходимо присоединить к какому-то объекту GameObject. К счастью, главная камера Main Camera тоже является объектом GameObject — просто этот объект снабжен компонентом Camera и компонентом AudioListener, поэтому мы присоединим новый сценарий к главной камере. **Перетащите сценарий GameController из папки Scripts в окне Project на главную камеру Main Camera в окне Hierarchy.**



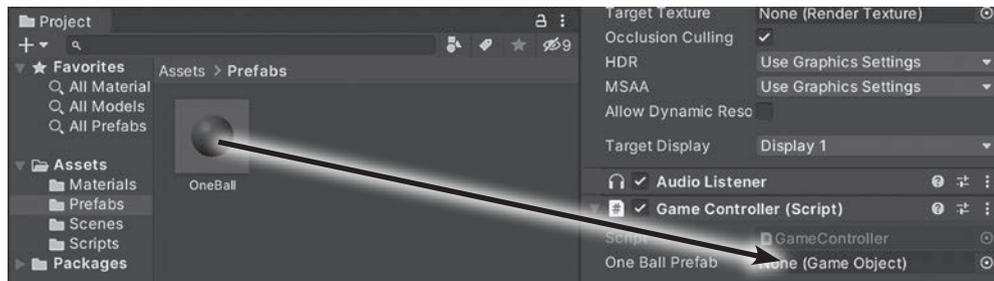
Об открытых и частных полях вы узнали в главе 5. Если класс сценария содержит открытое поле, то редактор Unity отображает это поле в компоненте Script в Inspector. Между буквами верхнего регистра он вставляет пробелы, чтобы имя проще читалось.

Взгляните на окно Inspector — вы увидите в нем компонент для сценария (точно так же, как для любого другого объекта GameObject). Сценарий содержит *открытое поле с именем OneBallPrefab*, поэтому Unity отображает его в компоненте Script.

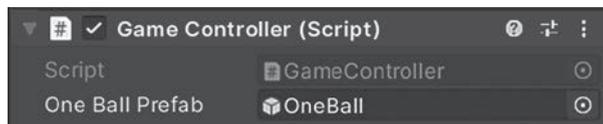


← Поле OneBallPrefab в классе GameController. Unity добавляет пробелы между буквами верхнего регистра, чтобы имя проще читалось (как и в предыдущей лабораторной работе).

У поля OneBallPrefab по-прежнему нет значения (None), его необходимо задать. **Перетащите объект OneBall из папки Prefabs на поле рядом с меткой One Ball Prefab.**



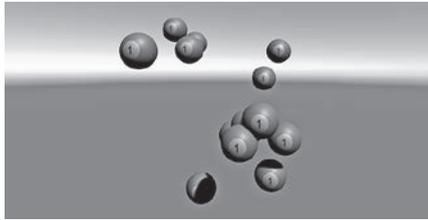
Теперь поле OneBallPrefab сценария GameController содержит ссылку на заготовку OneBall:



Вернитесь к коду и **внимательно присмотритесь к методу AddABall**. Он вызывает метод Instantiate, передавая ему в аргументе поле OneBallPrefab. Вы можете инициализировать это поле так, чтобы в нем хранилась нужная заготовка. Таким образом, каждый раз, когда сценарий GameController будет вызывать метод AddABall, *он будет создавать новый экземпляр заготовки OneBall*.

Запустите свой код кнопкой Play

Игра полностью готова к запуску. Сценарий GameController, присоединенный к главной камере, ожидает 1.5 секунды, после чего создает экземпляр заготовки OneBall каждую секунду. Метод Start каждого созданного экземпляра OneBall перемещает его в случайную позицию в сцене, а метод Update поворачивает его вокруг оси Y каждые 2 секунды с использованием полей OneBallBehaviour (как в предыдущей лабораторной работе). Проследите за тем, как игровая область постепенно заполняется вращающимися шарами.

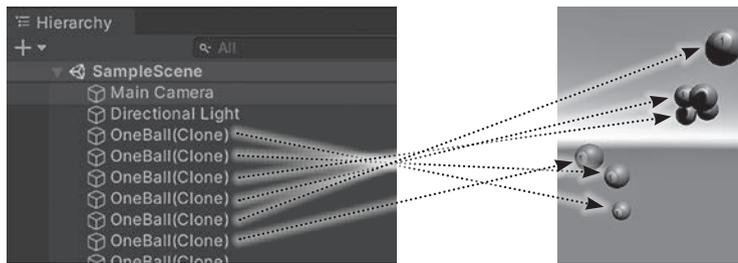


Unity вызывает метод Update каждого объекта GameObject перед каждым кадром. Это называется циклом обновления.

Когда вы создаете экземпляры объектов GameObject в своем коде, они появляются в окне Hierarchy при запуске игры.

Проследите за созданием экземпляров в окне Hierarchy

Каждый шар, летающий в сцене, представляет собой экземпляр заготовки OneBall. Каждый из экземпляров имеет собственный экземпляр класса OneBallBehaviour. В окне Hierarchy можно отслеживать все экземпляры OneBall — при создании очередного экземпляра в Hierarchy добавляется новый пункт «OneBall(Clone)».



Мы включили в лабораторный курс Unity ряд упражнений по программированию. Они ничем не отличаются от упражнений, приведенных в оставшейся части книги, — и помните, что подглядывать в решение — не значит жульничать.

Щелкните на любом из пунктов OneBall(Clone), чтобы просмотреть его в окне Inspector. Вы увидите, что его значения Transform изменяются в процессе вращения, как и в последней лабораторной работе.



Упражнение

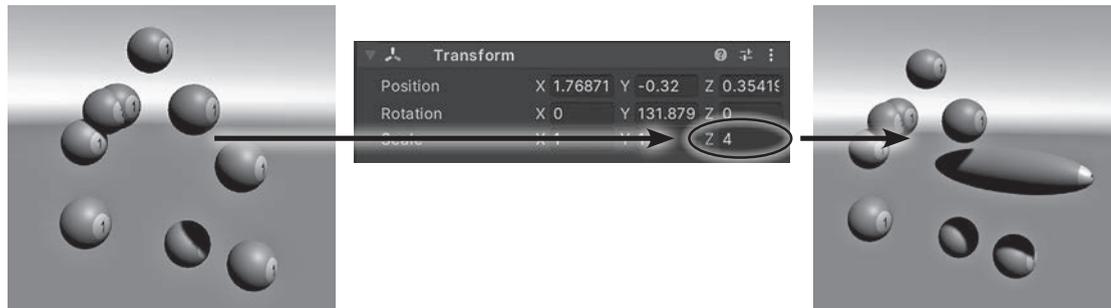
Разберитесь, как добавить поле BallNumber в сценарий OneBallBehaviour, чтобы, когда вы в следующий раз щелкнете на экземпляре OneBall в окне Hierarchy и проверите его компонент OneBallBehaviour (Script), под метками X Rotation, Y Rotation, Z Rotation и Degrees Per Second отображалось поле Ball Number:

Ball Number 11

У первого экземпляра OneBall полю Ball Number должно быть присвоено значение 1. У второго экземпляра ему присваивается 2, у третьего — 3 и т. д. Подсказка: вам понадобится хранить счетчик, **общий для всех экземпляров OneBall**. Измените метод Start, чтобы увеличить значение счетчика, а затем используйте его для присваивания полю BallNumber.

Работа с экземплярами GameObject в окне Inspector

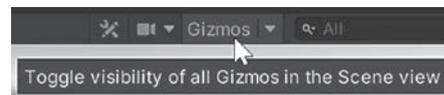
Запустите игру. После того как в ней будут созданы несколько экземпляров шара, щелкните на кнопке Pause — редактор Unity вернется к представлению Scene. Щелкните на одном из экземпляров OneBall в окне Hierarchy, чтобы выбрать его. Редактор Unity выделяет его в окне Scene, чтобы показать, какой объект был выбран. Перейдите к компоненту Transform в окне Inspector и **задайте его свойству Z scale значение 4**, чтобы шар растянулся по оси.



Снова запустите игру — теперь вы будете видеть, к какому шару применяются изменения. Попробуйте изменить значения его полей DegreesPerSecond, XRotation, YRotation и ZRotation, как в предыдущей лабораторной работе.

Пока игра работает, попробуйте переключаться между представлениями Game и Scene. Вы можете использовать манипуляторы в представлении Scene **даже во время игры** и даже для экземпляров GameObject, созданных методом Instantiate (а не добавленных в окно Hierarchy).

Попробуйте пощелкать на кнопке Gizmos в верхней части панели инструментов, чтобы включать и отключать манипуляторы. Вы можете включить их в представлении Game и одновременно отключить в представлении Scene.



Упражнение Решение

Чтобы добавить в сценарий OneBallBehaviour поле BallNumber для хранения количества шаров, добавленных ранее, можно воспользоваться статическим полем (которое мы назвали BallCount). Каждый раз, когда создается новый экземпляр шара, Unity вызывает его метод Start; таким образом мы можем увеличить статическое поле BallCount и присвоить результат полю BallNumber этого экземпляра.

```
static int BallCount = 0;
public int BallNumber;

void Start()
{
    transform.position = new Vector3(3 - Random.value * 6,
        3 - Random.value * 6, 3 - Random.value * 6);

    BallCount++;
    BallNumber = BallCount;
}
```

Все экземпляры OneBall совместно используют одно статическое поле BallCount, так что метод Start первого экземпляра увеличивает его до 1, второй экземпляр увеличивает BallCount до 2, третий — до 3 и т. д.

Предотвращение перекрытия шаров

А вы заметили, что некоторые шары перекрываются друг другом?

Unity содержит мощный **физический движок**, с помощью которого вы можете заставить свои объекты GameObject вести себя так, словно они являются реальными твердыми телами, а твердые тела не могут занимать одну область пространства. Чтобы предотвратить перекрытие, необходимо сообщить Unity, что заготовка OneBall является твердым объектом.

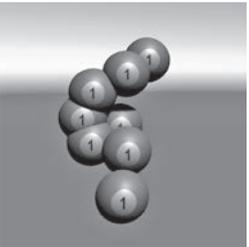
Остановите игру и **щелкните на заготовке OneBall в окне Project**, чтобы выделить ее. Перейдите к окну Inspector и прокрутите список до кнопки Add Component:



Щелкните на кнопке, откроется окно Component. **Выберите Physics**, чтобы просмотреть физические компоненты, а затем **выберите компонент Rigidbody**, чтобы добавить его к компоненту.



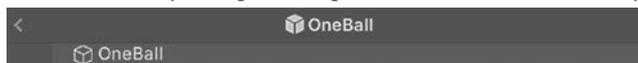
Проследите за тем, чтобы флажок Use Gravity был снят. В противном случае шары будут реагировать на гравитацию и начнут падать. А поскольку их ничто не остановит, падение будет длиться вечно.



Раз уж мы занялись физическими экспериментами, есть один опыт, который бы оценил Галилей. Попробуйте установить флажок Use Gravity во время выполнения игры. Новые шары, которые будут создаваться в сцене, начнут падать вниз, время от времени сталкиваясь с другими шарами и сбивая их с места.

Снова запустите игру; вы увидите, что шары в ней уже не перекрываются. Время от времени шар может быть создан поверх уже существующего. Когда это происходит, новый шар «сталкивает» старый со своего места.

Проведем небольшой физический эксперимент, который докажет, что шары действительно стали твердыми телами. Запустите игру и подождите, когда будут созданы более двух шаров. Перейдите в окно Hierarchy. Если оно имеет вид:

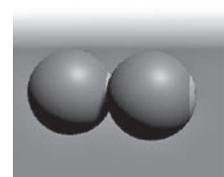


значит, вы редактируете заготовку — щелкните на кнопке со стрелкой назад (←) в правом верхнем углу окна Hierarchy, чтобы вернуться к сцене (возможно, вам придется снова развернуть SampleScene).

- ★ Удерживая клавишу Shift, щелкните на первом экземпляре OneBall в окне Hierarchy, а затем щелкните на втором. В результате должны быть выделены два экземпляра OneBall.
- ★ В полях Position на панели Transform выводятся дефисы (—). Задайте в поле Position значение (0, 0, 0), чтобы задать положение экземпляров OneBall одновременно.
- ★ Используя Shift+щелчок для выделения других экземпляров OneBall, щелкните правой кнопкой мыши и выберите Delete, чтобы удалить их из сцены, в которой должны остаться только два перекрывающихся шара.
- ★ Снимите игру с паузы — шары теперь не могут перекрываться, поэтому они будут вращаться рядом друг с другом.

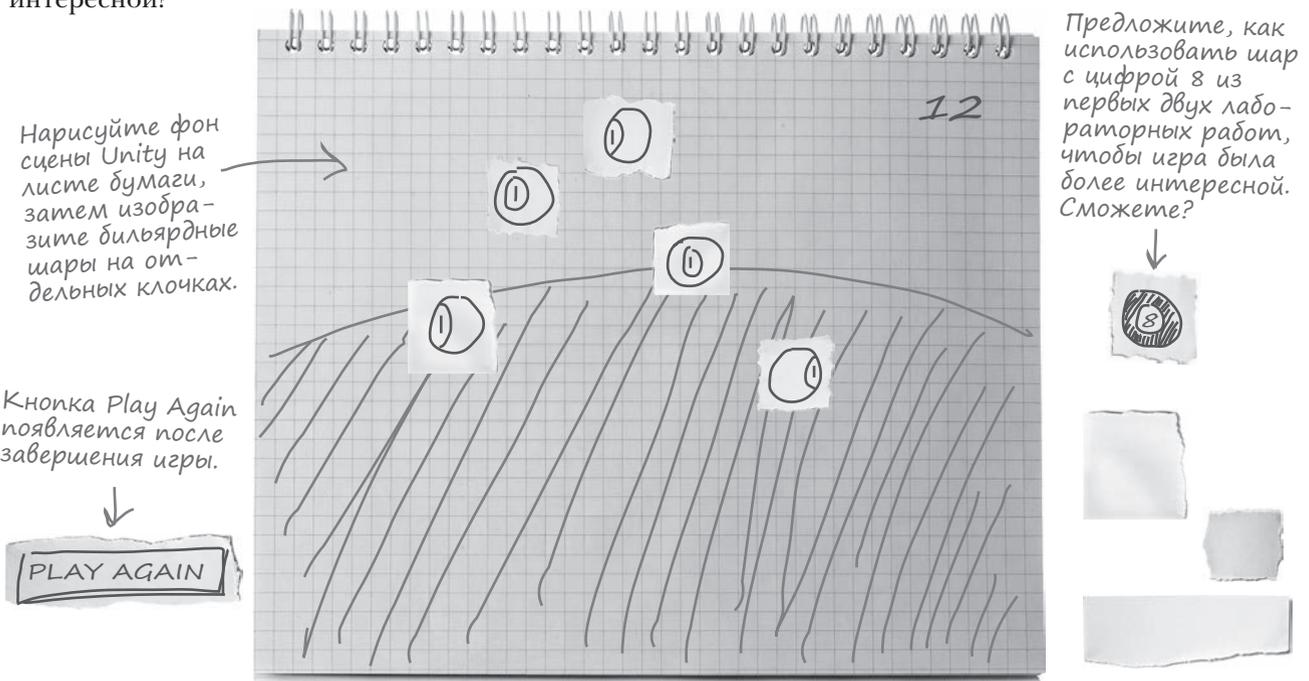
Остановите игру в Unity и Visual Studio, сохраните сцену.
«Сохраняйтесь пораньше, сохраняйтесь почаще!»

Окно Hierarchy может использоваться для удаления объектов GameObject из сцены во время выполнения игры.



Проявите фантазию!

Мы находимся уже на середине построения игры! Она будет завершена в следующей лабораторной работе. А пока перед вами открылась отличная возможность потренироваться в построении **бумажных прототипов**. Описание игры было приведено в начале этой лабораторной работы. Попробуйте создать ее бумажный прототип. А может, у вас будут какие-нибудь предложения, которые сделают игру более интересной?



КЛЮЧЕВЫЕ МОМЕНТЫ

- **Альbedo** — термин из области физики, обозначающий цвет, отражаемый объектом. Unity может использовать текстурные карты с альbedo в качестве материалов.
- Unity использует собственный **класс Random** в пространстве имен UnityEngine. Статический метод Random.value возвращает случайное число в диапазоне от 0 до 1.
- **Заготовка (prefab)** представляет собой объект GameObject, экземпляр которого вы можете создать в своей сцене. Любой объект GameObject может быть преобразован в заготовку.
- **Метод Instantiate** создает новый экземпляр объекта GameObject. Метод Destroy уничтожает его. Экземпляры создаются и уничтожаются в конце цикла обновления.
- **Метод InvokeRepeating** вызывает другой метод из сценария снова и снова.
- Unity вызывает метод Update каждого объекта GameObject перед каждым кадром. Это называется **циклом обновления**.
- Вы можете просмотреть «живые» экземпляры своих заготовок, щелкая на них в окне Hierarchy.
- Когда вы добавляете компонент **Rigidbody** в объект GameObject, физический движок Unity заставляет его вести себя как реальный твердый физический объект.
- Компонент Rigidbody позволяет включать и отключать **гравитацию** для объектов GameObject.

7 Интерфейсы, приведение типов и «is»

Классы должны держать обещания

Ладно-ладно. Я знаю, что я реализую интерфейс **Букмекер**. Но у меня не хватает времени, и я не смогу реализовать метод `ВыплатаДенег()` до следующей пятницы.



У тебя два дня. А потом я пошлю к тебе пару объектов **Бандит**, чтобы убедиться, что ты реализуешь метод `ХожуНаКостылях`.

Вам нужен объект для выполнения конкретной задачи? Используйте интерфейс. Иногда возникает необходимость сгруппировать объекты по **выполняемым ими функциям**, а не по классам, от которых они наследуют. На помощь приходят **интерфейсы**. Интерфейсы могут использоваться для определения конкретных задач. Любой экземпляр класса, **реализующего** интерфейс, гарантированно выполняет эту задачу независимо от того, с какими другими классами он связан. Чтобы эта схема работала, каждый класс, реализующий интерфейс, должен гарантировать **выполнение всех своих обязательств**... иначе программа компилироваться не будет.

Улей под атакой!

Вражеский улей пытается захватить территорию пчелиной матки и посылает боевых пчел, которые нападают на ваших рабочих. Из-за этого в иерархию классов был добавлен новый элитный субкласс `Bee` с именем `HiveDefender`; пчелы этого класса занимаются защитой улья.

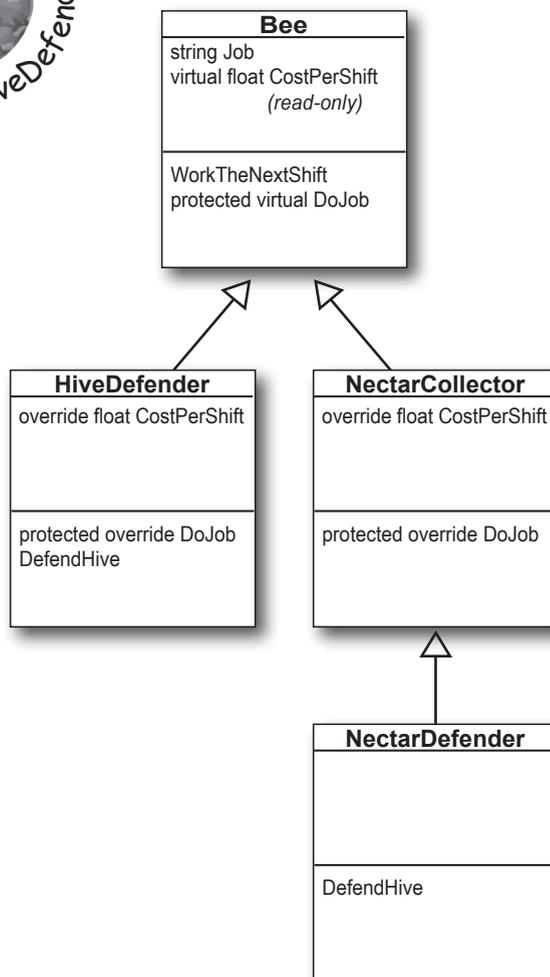


Нам понадобится метод `DefendHive`, потому что враги могут напасть в любой момент

Можно добавить в иерархию классов `Bee` субкласс `HiveDefender`; для этого мы расширим класс `Bee`, переопределим `CostPerShift` количеством меда, потребляемого каждым защитником за смену, и переопределим метод `DoJob` так, чтобы пчела летела к вражескому улью и нападала на вражеских пчел.

Но вражеские пчелы могут напасть в любой момент. Мы хотим, чтобы защитники могли защищать улей *независимо от того, выполняют они свое нормальное задание или нет*.

Из-за этого в дополнение к `DoJob` мы добавим метод `DefendHive` к каждому классу `Bee`, способному защищать улей, — не только к классу элитных рабочих `HiveDefender`, но и ко всем его братьям и сестрам, способным защищать матку. Матка вызывает методы `DefendHive` своих рабочих каждый раз, когда она видит, что улей атакует.



Мы можем воспользоваться приведением типов для вызова метода DefendHive...

Когда мы программировали метод `Queen.DoJob`, мы использовали цикл `foreach` для получения всех ссылок `Bee` в массиве `workers`. Далее эти ссылки использовались для вызова `worker.DoJob`. Если улей атакуют, матка `Queen` хочет вызвать методы `DefendHive` своих защитников. Мы предоставим метод `HiveUnderAttack`, который будет вызываться каждый раз, когда улей подвергается нападению вражеских пчел. Матка в цикле `foreach` приказывает рабочим защищать улей, пока все нападающие не исчезнут.

Но тут возникает проблема. Матка может использовать ссылки `Bee` для вызова `DoJob`, потому что каждый субкласс переопределяет `Bee.DoJob`, но она не может использовать ссылку `Bee` для вызова метода `DefendHive`, потому что этот метод не является частью класса `Bee`. Как же ей вызвать `DefendHive`?

Так как `DefendHive` определяется только в субклассах, для вызова метода `DefendHive` нам придется выполнить **приведение типа** для преобразования ссылки на `Bee` к правильному субклассу:

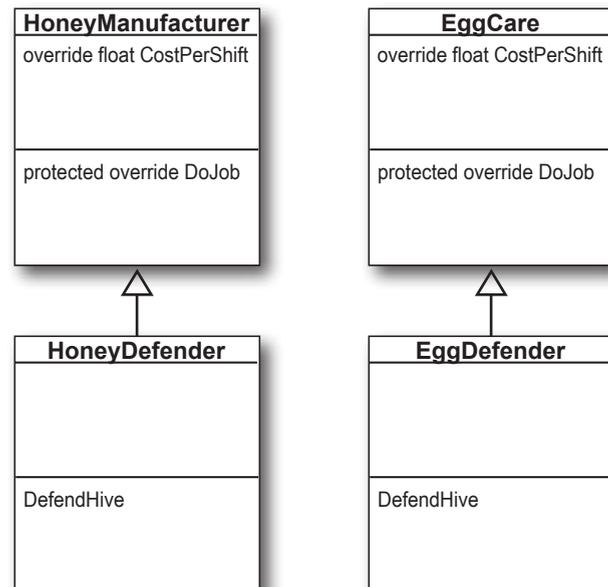
```
public void HiveUnderAttack() {
    foreach (Bee worker in workers) {
        if (EnemyHive.AttackingBees > 0) {
            if (worker.Job == "Hive Defender") {
                HiveDefender defender = (HiveDefender) worker;
                defender.DefendHive();
            } else if (worker.Job == "Nectar Defender") {
                NectarDefender defender = (NectarDefender) defender;
                defender.DefendHive();
            }
        }
    }
}
```

...но что, если потребуется добавить новые субклассы `Bee`, способные защищать улей?

Некоторые производители меда и опекуны тоже хотят встать в строй и защищать улей. Это означает, что нам придется добавить новые блоки `else` в свой метод `HiveUnderAttack`.

Архитектура усложняется. Метод `Queen.DoJob` остается простым и удобным — очень короткий цикл `foreach`, который использует модель классов `Bee` для вызова конкретной версии метода `DoJob`, реализованной в субклассе. Но с методом `DefendHive` это невозможно, потому что он не является частью класса `Bee` — и мы не хотим добавлять его, потому что не все пчелы могут защищать улей. Существует ли более эффективный способ выполнения одной задачи классами, не связанными отношениями наследования?

Производители меда и опекуны тоже хотят участвовать в защите улья. Придется ли `Queen` создавать `if/else` для каждого субкласса?



Интерфейс определяет методы и свойства, которые должны быть реализованы классом...

Интерфейс работает практически так же, как абстрактный класс: вы используете абстрактные методы, а затем двоеточие (:), чтобы класс реализовал этот интерфейс.

Таким образом, если вы хотите добавить защитников улья, для этого можно определить интерфейс с именем IDefend. Ниже показано, как это выглядит. **Ключевое слово interface** определяет интерфейс, в который входит единственный компонент — абстрактный метод с именем Defend. Все компоненты интерфейса по умолчанию являются открытыми и абстрактными, поэтому C# для простоты позволяет *опустить ключевые слова public и abstract*.

```
interface IDefend
{
    void Defend();
}
```

Интерфейс содержит один компонент — открытый абстрактный метод с именем Defend. Он работает точно так же, как абстрактные методы, приведенные в главе 6.

Любой класс, реализующий интерфейс IDefend, должен включать метод Defend, объявление которого соответствует объявлению в интерфейсе. Если он этого не сделает, компилятор выдает ошибку.

...но количество интерфейсов, которые могут быть реализованы классом, не ограничено

Мы уже сказали, что для реализации интерфейса классом используется двоеточие (:). А что, если класс уже использует двоеточие для расширения базового класса? Никаких проблем! **Класс может реализовать много разных интерфейсов, даже если он уже расширяет базовый класс:**

```
class NectarDefender : NectarCollector, IDefend
{
    void Defend() {
        /* Код защиты улья */
    }
}
```

Так как метод Defend является частью интерфейса IDefend, класс NectarDefender обязан реализовать его; в противном случае он не будет компилироваться.

Если класс реализует интерфейс, он должен включать все методы и свойства, перечисленные в интерфейсе; в противном случае код не построится.

Итак, теперь у нас имеется класс, который работает как NectarCollector, но также может защищать улей. NectarCollector расширяет Bee, так что **при использовании его по ссылке на Bee** он действует как Bee:

```
Bee worker = new NectarCollector();
Console.WriteLine(worker.Job);
worker.WorkTheNextShift();
```

Но при использовании по ссылке на IDefend он действует как защитник улья:

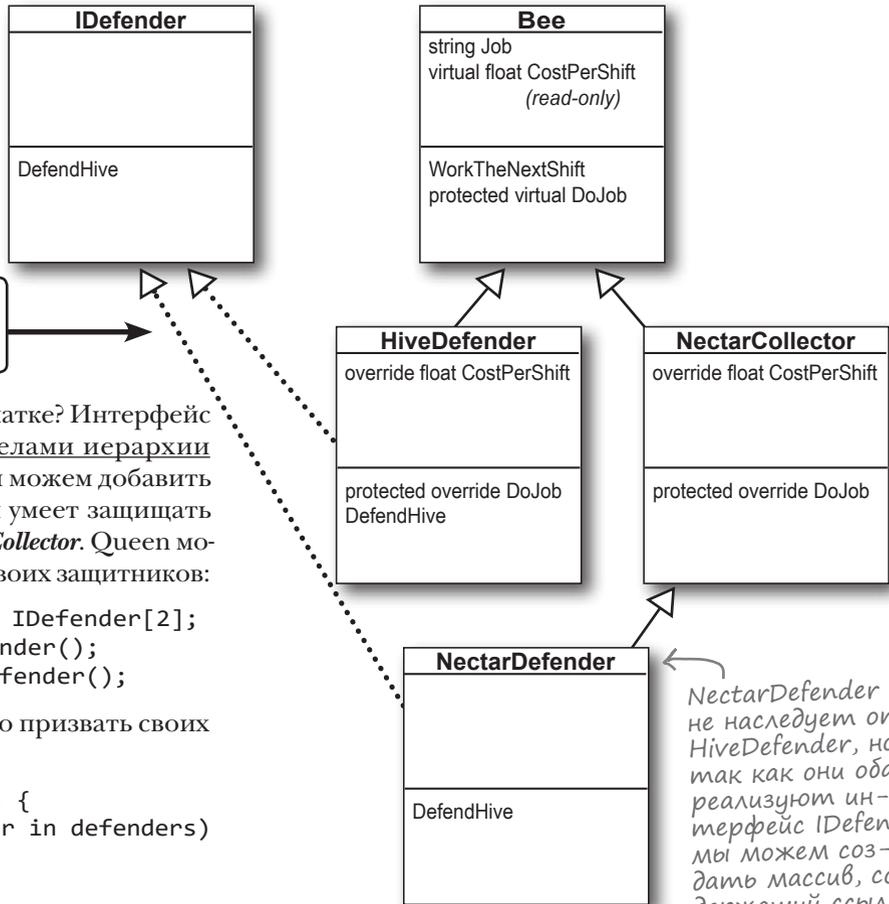
```
IDefend defender = new NectarCollector();
defender.Defend();
```

Интерфейсы позволяют несвязанным классам выполнять одну задачу

Интерфейсы – чрезвычайно мощный инструмент для проектирования кода C#, простого для понимания и построения. Для начала продумайте **конкретные задачи, которые должны выполняться классами**, потому что именно в них заключается суть интерфейсов.

Любой класс *Bee* может реализовать интерфейс *IDefender* независимо от своего положения в иерархии классов. Если он содержит метод *DefendHive*, код успешно построится.

Реализация интерфейса обозначается на диаграмме классов **пунктирными линиями**.



Как же это помогает пчелиной матке? Интерфейс *IDefender* существует за пределами иерархии классов *Bee*. Таким образом, мы можем добавить класс *NectarDefender*, который умеет защищать улей и *может расширять *NectarCollector**. *Queen* может поддерживать массив всех своих защитников:

```

IDefender[] defenders = new IDefender[2];
defenders[0] = new HiveDefender();
defenders[1] = new NectarDefender();
    
```

В таком случае она сможет легко призвать своих защитников:

```

private void DefendTheHive() {
    foreach (IDefender defender in defenders)
    {
        defender.Defend();
    }
}
    
```

И поскольку эта функциональность существует за пределами модели классов *Bee*, это можно сделать **без изменения существующего кода**.

NectarDefender не наследует от *HiveDefender*, но так как они оба реализуют интерфейс *IDefender*, мы можем создать массив, содержащий ссылки на обе разновидности объектов.



Мы приведем много примеров интерфейсов.

Все еще не до конца понимаете, как работают интерфейсы и почему их стоит использовать? Не беспокойтесь, это нормально! Синтаксис интерфейсов несложен, но **в нем много нюансов**. Поэтому мы посвятим интерфейсам дополнительное время... рассмотрим множество примеров и **потренируемся** на многочисленных упражнениях.



Теперь я знаю, что ты можешь защитить улей, и всем нам стало гораздо безопаснее!



Потренируемся в использовании интерфейсов

Лучший способ разобраться в интерфейсах — начать пользоваться ими. **Создайте новый проект консольного приложения.**

- 1 **Добавьте метод Main.** В следующем коде определяется класс TallGuy, а также код Main, который создает его экземпляр с использованием инициализатора объекта и вызывает его метод TalkAboutYourself. Ничего нового в этом коде нет:

```
class TallGuy {
    public string Name;
    public int Height;

    public void TalkAboutYourself() {
        Console.WriteLine($"My name is {Name} and I'm {Height} inches tall.");
    }
}

class Program
{
    static void Main(string[] args)
    {
        TallGuy tallGuy = new TallGuy() { Height = 76, Name = "Jimmy" };
        tallGuy.TalkAboutYourself();
    }
}
```

- 2 **Добавьте интерфейс.** Мы заставим класс TallGuy реализовать интерфейс. Добавьте в проект интерфейс IClown: щелкните правой кнопкой мыши на проекте в окне Solution Explorer, выберите **Add>>New Item... (Windows)** или **Add>>New File... (Mac)** и выберите команду Interface. Проследите за тем, чтобы файлу было присвоено имя IClown.cs. IDE создаст интерфейс, который включает объявление интерфейса. Добавьте метод Honk:

```
interface IClown
{
    void Honk();
}
```

Добавлять public или abstract в интерфейс не нужно, потому что все свойства и методы автоматически становятся открытыми и абстрактными.

- 3 **Попробуйте запрограммировать остальной код интерфейса IClown.** Прежде чем переходить к следующему шагу, попробуйте создать остальную часть интерфейса IClown и измените класс TallGuy для реализации этого интерфейса. Помимо void-метода с именем Honk, не получающего параметров, интерфейс IClown также должен содержать доступное только для чтения свойство с именем FunnyThingIHave, которое имеет get-метод, но не имеет set-метода.

Имена интерфейсов начинаются с I

Когда вы создаете интерфейс, присвойте ему имя, начинающееся с буквы I верхнего регистра. Нет никаких правил, которые бы это требовали, но при соблюдении этого соглашения ваш код станет намного более понятным. В этом несложно убедиться: перейдите в IDE в любую пустую строку в любом методе и введите «I» — в окне IntelliSense выводится список интерфейсов .NET.

- 4** Ниже приведен интерфейс `IClown`. Вы правильно написали код? Если вы разместили метод `Honk` на первом месте, это нормально — в интерфейсах, как и в классах, порядок компонентов интерфейса роли не играет.

```
interface IClown
{
    string FunnyThingIHave { get; }
    void Honk();
}
```

← Интерфейс `IClown` требует, чтобы любой реализующий его класс содержал `void`-метод с именем `Honk` и строковое свойство с именем `FunnyThingIHave`, имеющее `get`-метод.

- 5** Измените класс `TallGuy`, чтобы он реализовал `IClown`. Напоминаем: за оператором `:` всегда указывается базовый класс, от которого наследует класс (если он есть), а затем идет список интерфейсов, которые он реализует (интерфейсы разделяются запятыми). Так как в данном случае базового класса нет и реализуется всего один интерфейс, объявление выглядит так:

```
class TallGuy : IClown
```

Убедитесь в том, что остальной код класса выглядит так же, включая два поля и метод. Выберите команду `Build Solution` из меню `Build` в IDE, чтобы откомпилировать и построить программу. Вы получите две ошибки:

```

❌ CS0535 'TallGuy' does not implement interface member 'IClown.FunnyThingIHave'
❌ CS0535 'TallGuy' does not implement interface member 'IClown.Honk()'

```

- 6** Исправьте ошибки, добавив недостающие компоненты интерфейса. Ошибки исчезнут, как только вы добавите все методы и свойства, определенные в интерфейсе. Реализуйте интерфейс: добавьте строковое свойство `FunnyThingsIHave`, доступное только для чтения, и `get`-метод, который всегда возвращает строку «big shoes». Затем добавьте метод `Honk`, который выводит на консоль сообщение «Honk honk!».

Код должен выглядеть так:

```
public string FunnyThingIHave {
    get { return "big shoes"; }
}

public void Honk() {
    Console.WriteLine("Honk honk!");
}
```

← Любой класс, реализующий интерфейс `IClown`, обязан содержать `void`-метод с именем `Honk` и строковое свойство с именем `FunnyThingsIHave`, имеющее `get`-метод. Свойство `FunnyThingsIHave` также может иметь `set`-метод. В интерфейсе о нем ничего не сказано, поэтому возможны оба варианта.

- 7** Теперь ваш код компилируется. Обновите метод `Main`, чтобы он выводил свойство `FunnyThingsIHave` объекта `TallGuy`, а затем вызывал свой метод `Honk`.

```
static void Main(string[] args) {
    TallGuy tallGuy = new TallGuy() { Height = 76, Name = "Jimmy" };
    tallGuy.TalkAboutYourself();
    Console.WriteLine($"The tall guy has {tallGuy.FunnyThingIHave}");
    tallGuy.Honk();
}
```



Возьми в руку карандаш

Вам предоставляется возможность продемонстрировать свои художественные способности. Слева приводятся объявления интерфейсов и классов. Ваша задача — нарисовать соответствующую диаграмму класса справа. Не забудьте обозначать реализацию интерфейса пунктирной линией, а наследование от класса — сплошной.

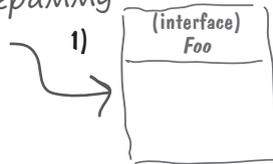
Вы получаете...

Мы нарисовали первую диаграмму за вас.

Как это выглядит?

1)

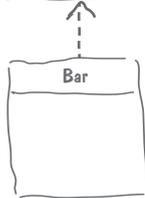
```
interface Foo { }
class Bar : Foo { }
```



2)

2)

```
interface Vinn { }
abstract class Vout : Vinn { }
```



3)

```
abstract class Muffie : Whuffie { }
class Fluffie : Muffie { }
interface Whuffie { }
```

3)

4)

4)

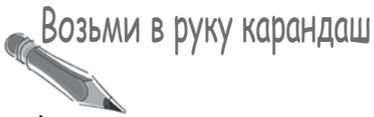
```
class Zoop { }
class Boop : Zoop { }
class Goop : Boop { }
```

Для 5) вам понадобится чуть больше места

5)

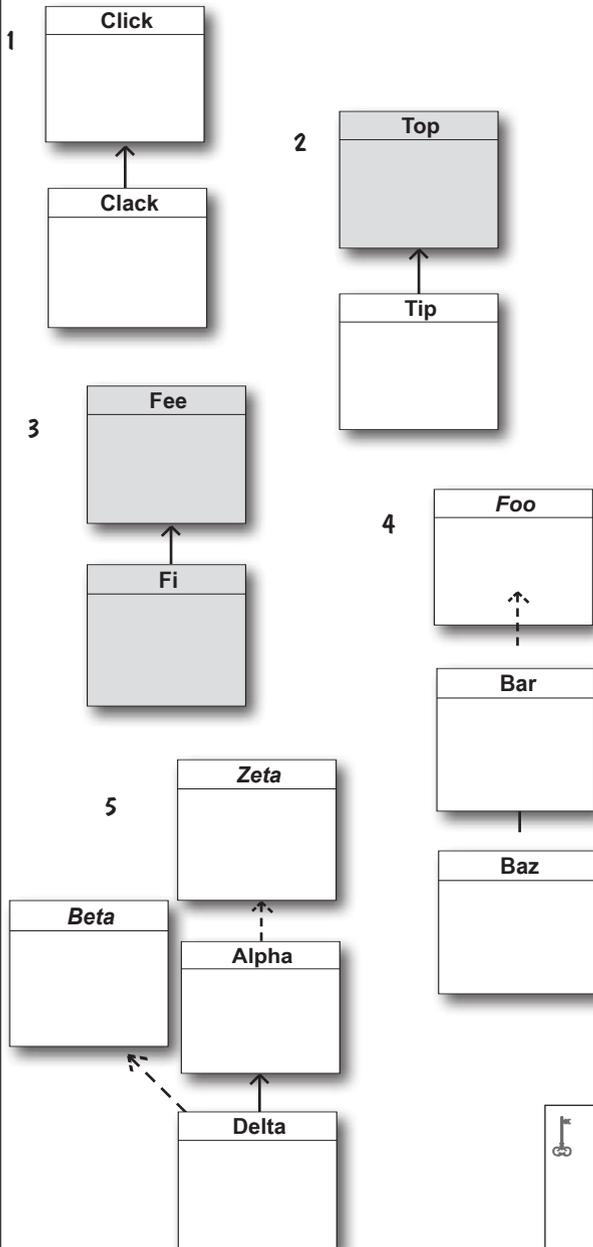
```
class Gamma : Delta, Epsilon { }
interface Epsilon { }
interface Beta { }
class Alpha : Gamma, Beta { }
class Delta { }
```





Слева изображены наборы диаграмм классов. Ваша задача — преобразовать их в допустимые объявления С#. Мы выполнили упражнение 1 за вас. А вы заметили, что объявления классов представляют собой пары фигурных скобок {}? Дело в том, что классы пока не содержат компонентов. (Но при этом они остаются действительными классами, которые успешно строятся!)

Вы получаете...



Как выглядит объявление?

- 1) `public class Click { }`
- 2) `public class Clack : Click { }`
- 2)
- 3)
- 4)
- 5)

		расширяет		класс
		реализует		интерфейс
				абстрактный класс

Беседа у камина



Кто важнее: абстрактный класс или интерфейс?

Абстрактный класс:

Я думаю, вполне очевидно, кто из нас важнее. Без меня программист не выполнит свою работу. Посмотрим правде в глаза: ты и близко ко мне не можешь подойти.

Как ты вообще мог подумать, что можешь быть важнее меня? Ты даже не в состоянии наследовать как полагается, тебя можно только реализовать.

Превосходит? Да ты сошел с ума. Я намного более гибок. Да, мои экземпляры не могут создаваться — но этого не можешь и ты. Зато, в отличие от тебя, я могу пользоваться **невероятной мощью** наследования. Нищebroды, которые тебя расширяют, даже не могут пользоваться ключевыми словами `virtual` и `override`!

Интерфейс:

Прекрасно. Другого я от тебя и не ожидал!

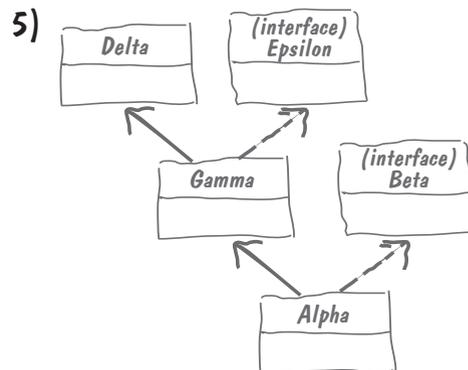
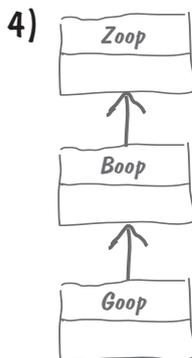
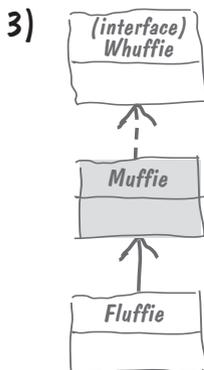
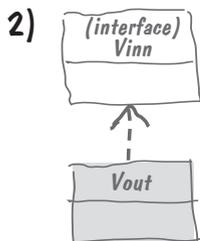
Отлично, вечная история. «Интерфейсы не используют механизм наследования», «интерфейсы только реализуются». Все это обычное невежество. Реализация ничем не хуже наследования, а в чем-то даже превосходит его!

Да? А если класс захочет наследовать у тебя **и** у твоего товарища? **Наследовать от двух классов нельзя.** Нужно выбрать кого-то одного. А вот число реализуемых интерфейсов может быть любым, так что не нужно говорить мне о гибкости! С моей помощью программист заставит классы делать что угодно.

Возьми в руку карандаш



Решение



Как это выглядит?

Абстрактный класс:

Кажется, ты несколько переоцениваешь свою роль.

И ты думаешь, что это хорошо? Ха! Когда ты используешь меня и мои subclasses, ты точно знаешь, что происходит внутри всех нас. Я могу реализовать любое поведение, необходимое всем моим subclassам, а им остается только наследовать его. Прозрачность — мощная штука!

Да ну? По моим наблюдениям, программистов очень даже волнует содержимое свойств и методов.

Да, конечно... расскажи кодеру, что он не может писать код.

Интерфейс:

Серьезно? Ну давай посмотрим, насколько мощным я могу быть для разработчиков, которые используют меня. Вся моя суть — выполнение конкретной задачи. Если у разработчика имеется ссылка на интерфейс, ему вообще не нужно ничего знать о том, что происходит внутри объекта.

Девять из десяти, что программисту нужны определенные свойства и методы, но его при этом не волнует, как именно они реализуются.

Да, конечно. Когда-нибудь потом. Только вспомни, как часто программисты пишут методы с объектами в качестве параметров, которые просто должны включать в себя определенные методы. И в этот момент никого не волнует, как именно эти методы построены. Главное, чтобы они были. Программисту достаточно воспользоваться интерфейсом, и проблема решена!

И что ты с ним будешь делать?

Возьми в руку карандаш
Решение

2) `abstract class Top { }`
`class Tip : Top { }`

3) `abstract class Fee { }`
`abstract class Fi : Fee { }`

4) `interface Foo { }`
`class Bar : Foo { }`
`class Baz : Bar { }`

5) `interface Zeta { }`
`class Alpha : Zeta { }`
`interface Beta { }`
`class Delta : Alpha, Beta { }`

Delta наследует от Alpha и реализует Beta.

Как выглядит объявление?

Создать экземпляр интерфейса невозможно, но можно получить ссылку на интерфейс

Допустим, вам нужен объект с методом Defend, чтобы вы могли использовать его в цикле для защиты улья. Для этого подойдет любой объект, реализующий интерфейс IDefender. Это может быть объект HiveDefender, объект NectarDefender или даже совершенно посторонний объект HelpfulLadyBug. Если объект реализует интерфейс IDefender, он гарантированно содержит метод Defend. Вам остается только вызвать его.

И здесь в игру вступают **ссылки на интерфейсы**. Вы можете воспользоваться такой ссылкой для обращения к объекту, который реализует нужный вам интерфейс, и всегда можете быть уверены в том, что он содержит нужные вам методы, даже если вы не знаете о нем ничего более.

При попытке создания экземпляра интерфейса код не будет строиться

Вы можете создать массив ссылок IWorker, но создать экземпляр интерфейса невозможно. Вы *можете* связать эти ссылки с новыми экземплярами классов, реализующих IWorker. Так у вас появляется массив, содержащий объекты разных видов!

При попытке создания экземпляра интерфейса компилятор протестует:

```
IDefender barb = new IDefender(); ← НЕ КОМПИЛИРУЕТСЯ
```

Вы не сможете использовать ключевое слово new с интерфейсом, и это логично — методы и свойства не имеют реализаций. Если бы вы создали объект на базе интерфейса, то как бы определялось поведение такого объекта?

Используйте интерфейсы для ссылок на уже существующие объекты

Итак, создать экземпляр интерфейса невозможно... Но интерфейс *можно использовать для создания ссылочной переменной* и использовать эту переменную для обращения к объекту, *реализующему* интерфейс.

Помните, что ссылку Tiger можно передать любому методу, который рассчитывает получить Animal, потому что Tiger расширяет Animal? Здесь та же самая ситуация — экземпляр класса, реализующего IDefender, может использоваться в любом методе или команде, рассчитывающем получить IDefender.

```
IDefender susan = new HiveDefender(); ←  
IDefender ginger = new NectarDefender();
```

И хотя этот объект способен на большее, при использовании ссылки на интерфейс вам доступны только методы, входящие в интерфейс.

Это обычные команды new — точно такие же, как те, что встречались вам ранее в книге. Единственное различие заключается в том, что для обращения к ним **используется переменная типа IDefender**.



← Интерфейс используется для объявления переменных «susan» и «ginger», но это обычные ссылки, которые работают точно так же, как любые другие.



У бассейна

Возьмите фрагменты кода из бассейна и разместите их в пустых строках. Каждый фрагмент можно использовать несколько раз. В бассейне есть и лишние фрагменты. **Ваша задача** — получить набор классов, которые будут компилироваться, запускаться и давать показанный ниже результат.

```

_____ INose {
    _____;
    string Face { get; }
}

abstract class _____ : _____ {
    private string face;
    public virtual string Face {
        _____ { _____ _____ ; }
    }

    public abstract int Ear();

    public Picasso(string face)
    {
        _____ = face;
    }
}

class _____ : _____ {
    public Clowns() : base("Clowns") { } } \n";

    public override int Ear() {
        return 7;
    }
}

class _____ : _____ {
    public Acts() : base("Acts") { }
    public override _____ {
        return 5;
    }
}

class _____ : _____ {
    public override string Face {
        get { return "Of2016"; }
    }

    public static void Main(string[] args) {
        string result = "";
        INose[] i = new INose[3];
        i[0] = new Acts();
        i[1] = new Clowns();
        i[2] = new Of2016();
        for (int x = 0; x < 3; x++) {
            result +=
                $"{_____} {_____}
        }
        Console.WriteLine(result);
        Console.ReadKey();
    }
}

```

Точка входа — перед вами полноценная программа C#

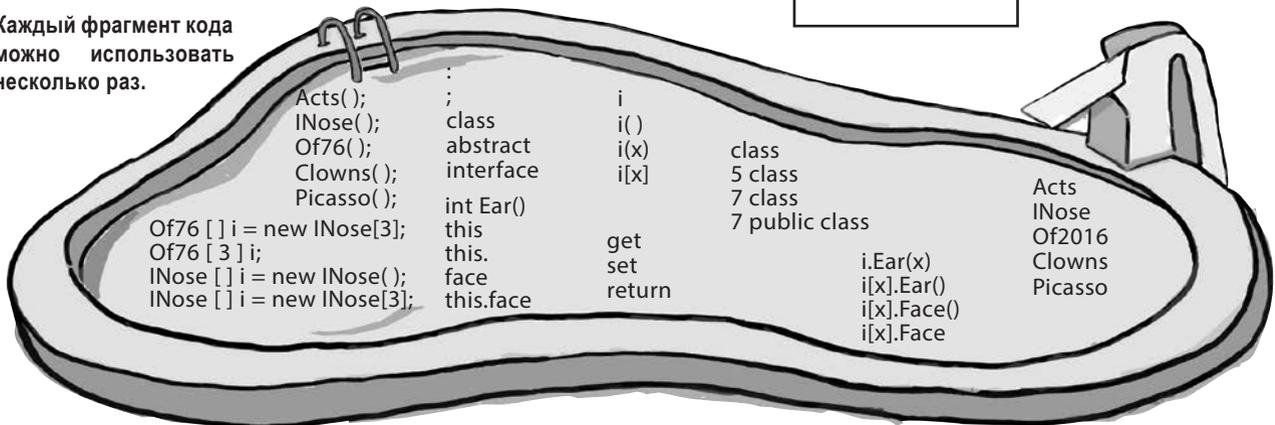
Результат

```

5 Acts
7 Clowns
7 Of2016

```

Каждый фрагмент кода можно использовать несколько раз.





У бассейна. Решение

Возьмите фрагменты кода из бассейна и разместите их в пустых строках. Каждый фрагмент можно использовать несколько раз. В бассейне есть и лишние фрагменты. **Ваша задача** — получить набор классов, которые будут компилироваться, запуститься и давать показанный ниже результат.

Результат
5 Acts
7 Clowns
7 Of2016

Face — get-метод, возвращающий значение свойства face. Оба компонента определяются в Picasso и наследуются subclasses.

Здесь класс Acts вызывает конструктор из класса Picasso, от которого он наследует. Конструктору передается значение "Acts", которое сохраняется в свойстве Face.

```
interface INose {
    int Ear();
    string Face { get; }
}

abstract class Picasso : INose {
    private string face;
    public virtual string Face {
        get { return face ; }
    }

    public abstract int Ear();

    public Picasso(string face)
    {
        this.face = face;
    }
}

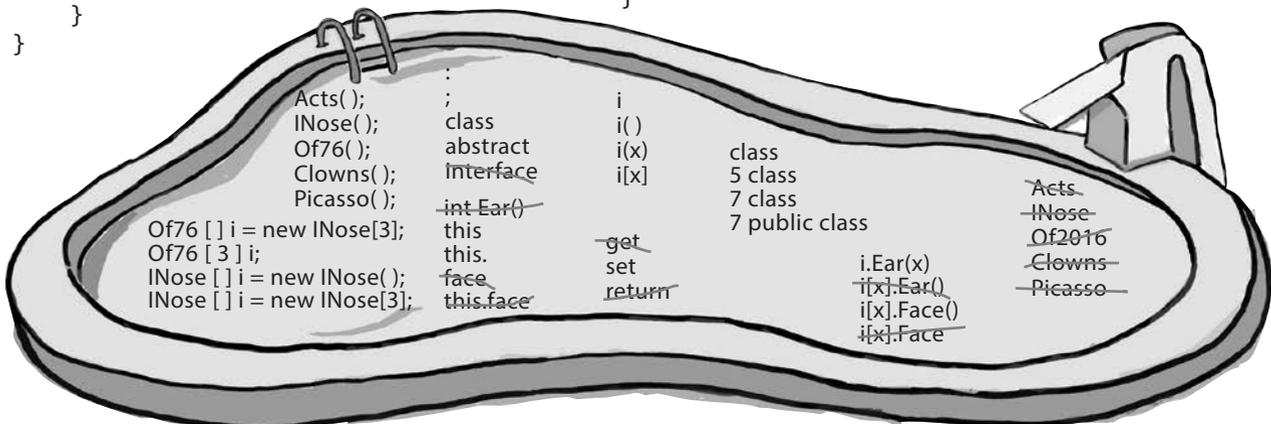
class Clowns : Picasso {
    public Clowns() : base("Clowns") { }

    public override int Ear() {
        return 7;
    }
}
```

```
class Acts : Picasso {
    public Acts() : base("Acts") { }
    public override int Ear() {
        return 5;
    }
}

class Of2016 : Clowns {
    public override string Face {
        get { return "Of2016"; }
    }

    public static void Main(string[] args) {
        string result = "";
        INose[] i = new INose[3];
        i[0] = new Acts();
        i[1] = new Clowns();
        i[2] = new Of2016();
        for (int x = 0; x < 3; x++) {
            result +=
                $"{i[x].Ear()} {i[x].Face}\n";
        }
        Console.WriteLine(result);
        Console.ReadKey();
    }
}
```

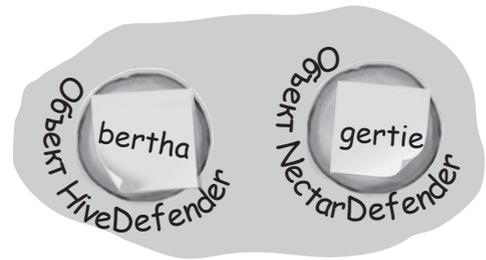


Ссылки на интерфейсы являются обычными ссылками на объекты

Вы уже знаете, что все объекты существуют в куче. Когда вы работаете со ссылкой на интерфейс, она всего лишь становится новым способом обращения к объектам, с которыми вы уже работали. Давайте повнимательнее присмотримся к тому, как интерфейсы используются для обращения к объектам из кучи.

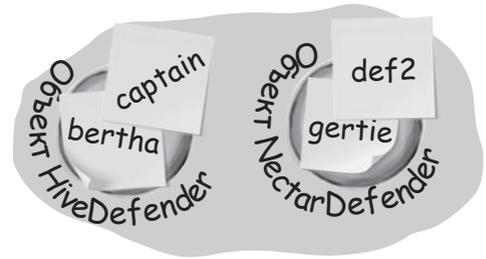
- 1 **Все начинается с обычного создания объектов.** Следующий код создает экземпляр `HiveDefender` и экземпляр `NectarDefender` — оба этих класса реализуют интерфейс `IDefender`.

```
HiveDefender berth = new HiveDefender();
NectarDefender gertie = new NectarDefender();
```



- 2 **Затем добавляются ссылки на `IDefender`.** Ссылки на интерфейсы используются точно так же, как ссылки на любые другие типы. Следующие две команды используют интерфейсы для создания **новых ссылок на существующие объекты**. Ссылка на интерфейс может указывать только на экземпляр класса, который этот интерфейс реализует.

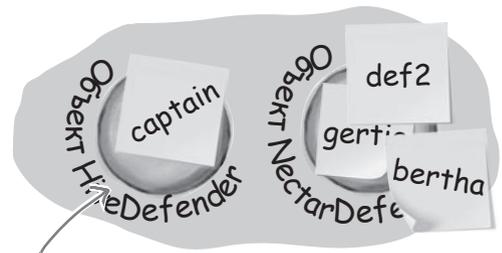
```
IDefender def2 = gertie;
IDefender captain = berth;
```



- 3 **Ссылка на интерфейс поддерживает существование объекта.** Если в системе не осталось ни одной ссылки, указывающей на объект, этот объект уничтожается. Нет никаких правил, требующих, чтобы ссылки относились к одному типу! В том, что касается жизни объектов, ссылка на интерфейс ничем не отличается от любых других ссылок, поэтому она предотвращает уничтожение объекта в ходе сборки мусора.

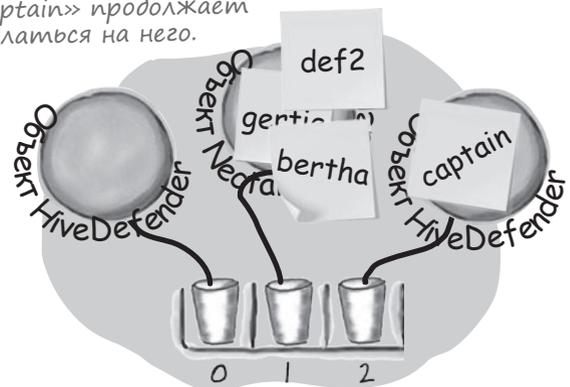
```
berth = gertie; ← Теперь berth указываем на NectarDefender.
// Ссылка captain все еще указывает на объект
// HiveDefender
```

Этот объект не исчезает из кучи, потому что «captain» продолжает ссылаться на него.



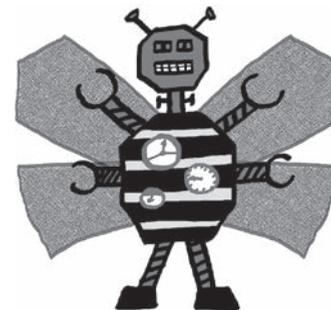
- 4 **Интерфейс используется так же, как любой другой тип.** Вы можете создать новый объект командой `new` и присвоить его интерфейсной ссылочной переменной в одной строке кода. **Интерфейсы могут использоваться для создания массивов**, которые содержат любые объекты, реализующие интерфейс.

```
IDefender[] defenders = new IDefender[3];
defenders[0] = new HiveDefender();
defenders[1] = berth;
defenders[2] = captain;
```



RoboBee 4000 может выполнять работу пчел без расхода драгоценного меда

В последнем квартале бизнес процветал, и у пчелиной матки хватило средств для приобретения последнего технологического достижения: RoboBee 4000. Робот умеет выполнять работу трех разных пчел, а самое замечательное, что он не расходует мед! Впрочем, есть некоторые претензии к экологичности — робот расходует бензин. Как же при помощи интерфейсов интегрировать RoboBee в повседневную работу улья?



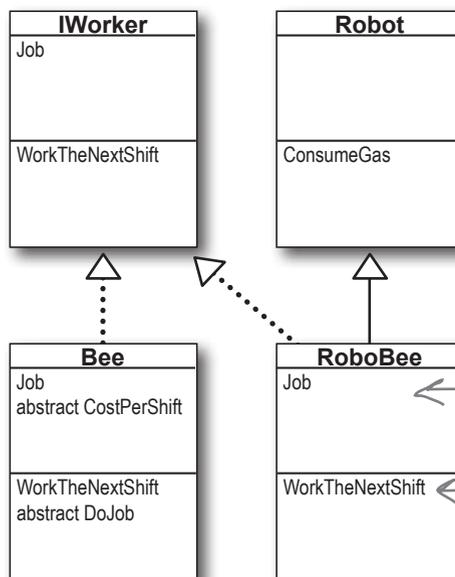
```
class Robot
{
    public void ConsumeGas() {
        // Неэкологично
    }
}

class RoboBee4000 : Robot, IWorker
{
    public string Job {
        get { return "Egg Care"; }
    }
    public void WorkTheNextShift()
    {
        // Выполняет работу трех пчел
    }
}
```

Повнимательнее присмотритесь к диаграмме классов, чтобы понять, как использовать интерфейс для интеграции класса RoboBee в систему управления ульем. Помните: пунктирные линии используются для обозначения того, что объект реализует интерфейс.

Можно создать интерфейс IWorker, который состоит из двух компонентов, относящихся к выполнению работы в улье.

Класс Bee реализует интерфейс IWorker, тогда как класс RoboBee наследует от Robot и реализует IWorker. Это означает, что объект является роботом, но может выполнять задачи рабочих пчел.



Начнем с простейшего класса Robot — все знают, что роботы работают на бензине, поэтому класс содержит метод ConsumeGas.

Класс RoboBee реализует оба компонента интерфейса IWorker. В этом отношении у нас нет выбора — если класс RoboBee не реализует все содержимое интерфейса IWorker, код компилироваться не будет.

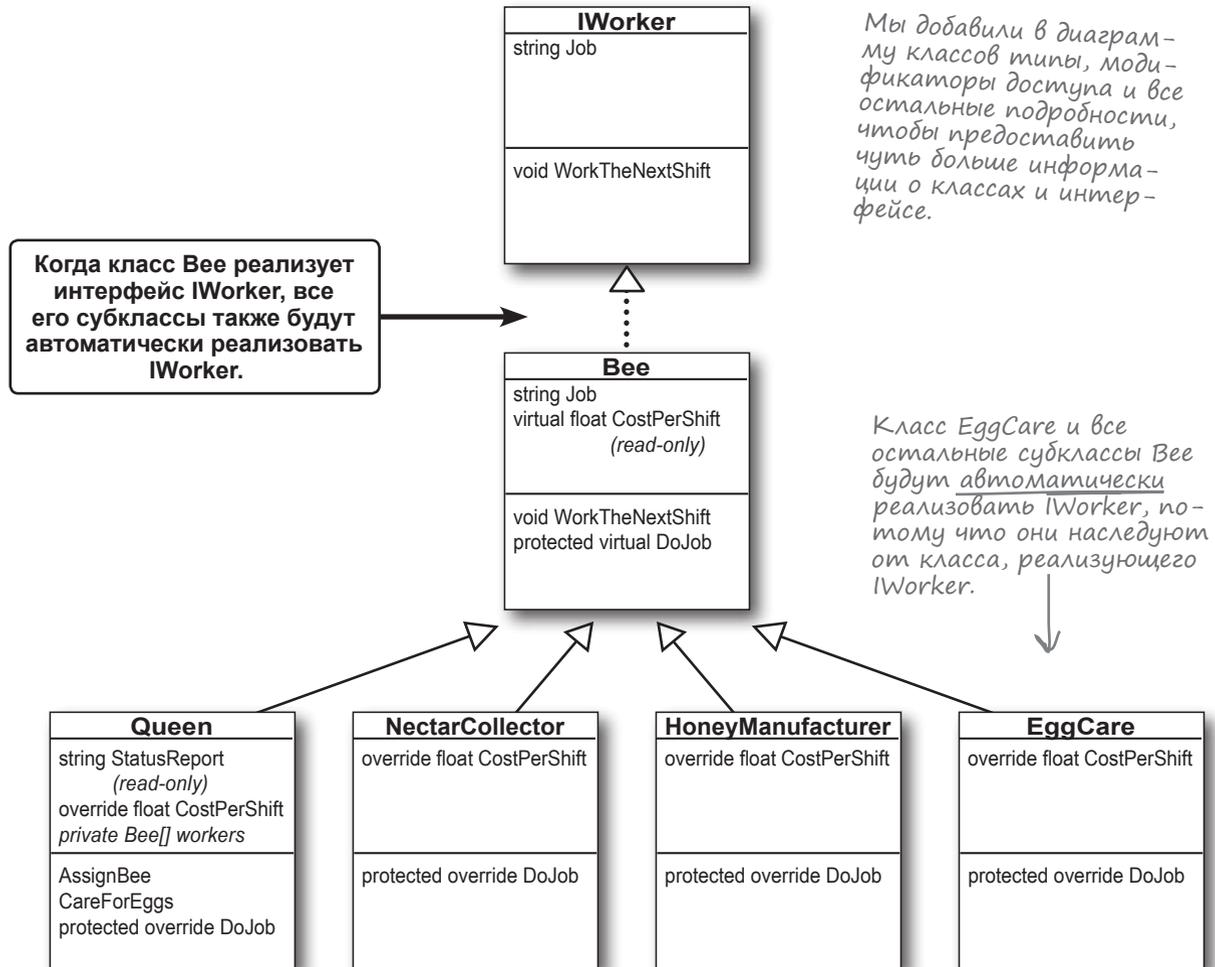
Остается изменить систему управления ульем так, чтобы в каждом обращении к рабочему в ней использовался интерфейс IWorker вместо абстрактного класса Bee.



Упражнение

Измените систему управления ульем так, чтобы для обращений к рабочим в ней использовался интерфейс `IWorker` вместо абстрактного класса `Bee`.

Ваша задача — добавить в проект интерфейс `IWorker`, а затем провести рефакторинг кода, чтобы заставить класс `Bee` реализовать его, а также изменить класс `Queen`, чтобы в нем использовались только ссылки `IWorker`. Обновленная диаграмма классов должна выглядеть так:



Что нужно сделать:

- Добавьте интерфейс `IWorker` в проект системы управления ульем.
- Измените класс `Bee`, чтобы он реализовал интерфейс `IWorker`.
- Измените класс `Queen`, чтобы все ссылки на `Bee` были заменены ссылками на `IWorker`.

Вроде бы работы не так уж много, потому что... ее действительно немного. После добавления интерфейсов остается только изменить одну строку кода в классе `Bee` и три строки кода в классе `Queen`.



Упражнение Решение

Измените систему управления ульем так, чтобы для обращений к рабочим в ней использовался интерфейс `IWorker` вместо абстрактного класса `Bee`. Для этого пришлось добавить интерфейс `IWorker`, а также изменить классы `Bee` и `Queen`. Значительных изменений в коде не потребовалось — ведь использование интерфейсов не требует большого объема лишнего кода.

Вы начали с добавления интерфейса `IWorker` в проект

```
interface IWorker
{
    string Job { get; }
    void WorkTheNextShift();
}
```

Затем вы изменили `Bee` для реализации интерфейса `IWorker`

```
abstract class Bee : IWorker
{
    /* Остальной код класса остается без изменений */
}
```

И наконец, вы изменили класс `Queen` так, чтобы вместо ссылок на `Bee` в нем использовались ссылки на `IWorker`

```
class Queen : Bee
{
    private IWorker[] workers = new IWorker[0];

    private void AddWorker(IWorker worker)
    {
        if (unassignedWorkers >= 1)
        {
            unassignedWorkers--;
            Array.Resize(ref workers, workers.Length + 1);
            workers[workers.Length - 1] = worker;
        }
    }

    private string WorkerStatus(string job)
    {
        int count = 0;
        foreach (IWorker worker in workers)
            if (worker.Job == job) count++;
        string s = "s";
        if (count == 1) s = "";
        return $"{count} {job} bee{s}";
    }

    /* Остальной код класса Queen остается без изменений */
}
```

Любой класс может реализовать ЛЮБОЙ интерфейс при условии, что он соблюдает гарантии по реализации методов и свойств интерфейса.

Попробуйте изменить `WorkerStatus` так, чтобы заменить `IWorker` в цикле `foreach` на `Bee`:

```
foreach (Bee worker in workers)
```

Запустите код — он нормально работает! Теперь попробуйте заменить его на `NectarCollector`. На этот раз вы получите исключение `System.InvalidCastException`. Как вы думаете, почему это происходит?

Часть Задаваемые Вопросы

В: Когда я помещаю свойство в интерфейс, оно выглядит как автоматическое свойство. Означает ли это, что при реализации интерфейсов могут использоваться только автоматические свойства?

О: Совсем нет. Действительно, свойство в интерфейсе очень похоже на автоматическое свойство (как свойство `Job` в интерфейсе `IWorker` на следующей странице), но они определенно не являются автоматическими свойствами. Свойство `Job` можно было бы реализовать так:

```
public Job {
    get; private set;
}
```

Приватный `set`-метод необходим, потому что автоматическое свойство требует наличия как `get`-, так и `set`-метода (даже если они являются приватными). Также реализация могла бы выглядеть так:

```
public Job {
    get {
        return "Egg Care";
    }
}
```

Компилятор это полностью устроит. Также можно добавить `set`-метод — интерфейс требует наличия `get`-метода, но он не запрещает иметь `set`-метод. (Если использовать автоматическое свойство для реализации, вы можете самостоятельно решить, должен `set`-метод быть приватным или открытым.)

В: Разве не странно, что в моих интерфейсах нет модификаторов доступа? Почему не пометить методы и свойство открытыми?

О: Модификаторы доступа не нужны, потому что все компоненты интерфейса по умолчанию являются открытыми. Допустим, у вас имеется интерфейс вида:

```
void Honk();
```

Объявление говорит, что класс должен содержать открытый `void`-метод с именем `Honk`, но ничего не сообщает о том, что этот метод должен делать. Он может делать все что угодно — что бы он ни делал, код будет нормально компилироваться при условии, что в нем присутствует метод с подходящей сигнатурой.

Выглядит знакомо? Правильно, потому что вы уже видели этот принцип — в абстрактных классах в главе 6. Когда вы объявляете в интерфейсе методы или свойства без тел, они **автоматически** становятся **открытыми и абстрактными**, как абстрактные компоненты, используемые в абстрактных классах. Они работают точно так же, как любые другие методы или свойства, — хотя ключевое слово `abstract` явно не включается, его присутствие подразумевается. Именно по этой причине каждый класс, реализующий интерфейс, **должен реализовать каждый его компонент**.

Люди, занимавшиеся проектированием C#, могли бы заставить вас пометить все компоненты как открытые и абстрактные, но это было бы излишне. Поэтому они сделали все компоненты открытыми и абстрактными по умолчанию, чтобы код программы был более понятным.

Все компоненты открытого интерфейса автоматически являются открытыми, потому что они используются для определения открытых методов и свойств любого класса, реализующего этот интерфейс.

Свойство Job в интерфейсе IWorker — костыль

В системе управления ульем свойство Worker.Job используется по схеме: `if (worker.Job == job)`.

Вам здесь ничего не кажется странным? Нам кажется. Мы считаем, что это **костыль** — неэлегантное, примитивное решение. Почему использование Job кажется нам костылем? Представьте, что произойдет, если вы допустите опечатку:

```
class EggCare : Bee {  
    public EggCare(Queen queen) : base("Egg Crae")  
  
    // В классе EggCare появляется ошибка, хотя  
    // остальной код класса остается неизменным.  
}
```

← Мы совершили опечатку в «Egg Care» — согласитесь, такую опечатку может сделать каждый. Вы представляете, насколько трудно будет найти подобную ошибку, вызванную простой опечаткой?

Теперь код не может определить, указывает ли ссылка Worker на экземпляр EggCare. Возникает крайне коварная ошибка, которую очень трудно обнаружить. В этом коде определенно повышается риск ошибок... Но почему мы называем его «костылем»?

Ранее говорилось о **разделении обязанностей**: весь код, относящийся к решению конкретной задачи, должен храниться вместе. Свойство Job *нарушает принцип разделения обязанностей*. Если у вас имеется ссылка на Worker, то вам уже не нужно проверять строку, чтобы узнать, указывает ссылка на объект EggCare или на объект NectarCollector. Свойство Job возвращает «Egg Care» для объекта EggCare, «Nectar Collector» для объекта NectarCollector и используется только для проверки типа объекта. Но ведь эта информация уже отслеживается *в типе объекта*.

Кажется, я понимаю, к чему вы клоните.
Наверняка C# дает мне возможность проверить тип объекта
без использования костылей, верно?



Верно! C# предоставляет вам необходимые средства для работы с типами.

Для проверки типов классов не нужно добавлять лишние свойства (такие, как Job) и строки «Egg Care» или «Nectar Collector». C# предоставляет в ваше распоряжение средства для проверки типа объекта.

Ко-стыль, суц.

В технике — уродливое, неуклюжее или неэлегантное решение задачи, которое создает трудности с сопровождением.

Использование is для проверки типа объекта

Как избавиться от костыля со свойством Job? Непосредственно сейчас объект Queen содержит массив workers; это означает, что он может получить ссылку на IWorker. При помощи свойства Job Queen определяет, какие рабочие относятся к категории EggCare, а какие — к категории NectarCollector:

```
foreach (IWorker worker in workers) {
    if (worker.Job == "Egg Care") {
        WorkNightShift((EggCare)worker);
    }
}

void WorkNightShift(EggCare worker) {
    // Код обработки смены
}
```

Как вы уже видели, если случайно ввести «Egg Crae» вместо «Egg Care», код терпит крах самым позорным образом. А если случайно задать свойству Job объекта HoneyManufacturer значение «Egg Care», вы получите ошибку InvalidCastException. Было бы замечательно, если бы компилятор мог выявлять подобные проблемы на стадии написания кода — подобно тому, как мы используем приватные и открытые компоненты для обнаружения других проблем.

C# предоставляет специальное средство для этой цели: **ключевое слово is** проверяет тип объекта. Если у вас имеется ссылка на объект, **воспользуйтесь is** для проверки того, относится ли она к конкретному типу:

objectReference is ObjectType newVariable

Если объект, на который указывает objectReference, относится к типу ObjectType, то is возвращает true и создается новая ссылка этого типа с именем newVariable.

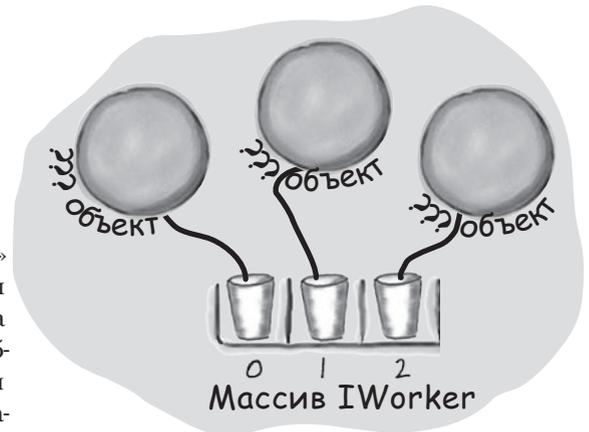
Таким образом, если пчелиная матка хочет найти всех своих рабочих-опекунов EggCare и заставить их работать в ночную смену, она может воспользоваться ключевым словом is:

```
foreach (IWorker worker in workers) {
    if (worker is EggCare eggCareWorker) {
        WorkNightShift(eggCareWorker);
    }
}
```

Команда if в этом цикле использует is для проверки каждой ссылки на IWorker. Присмотритесь повнимательнее к проверке условия:

```
worker is EggCare eggCareWorker
```

Если объект, на который ссылается переменная worker, является объектом EggCare, условие возвращает true, а команда is присваивает ссылку новой переменной EggCare с именем eggCareWorker. Происходящее напоминает приведение типов, но команда is **безопасно выполняет приведение за вас**.



Ключевое слово is возвращает true, если объект соответствует заданному типу, и C# может объявить переменную со ссылкой на этот объект.



Использование is для обращения к методам subclasses

Объединим все, о чем говорилось ранее, в новом проекте. Мы создадим простую модель классов, на вершине которой находится Animal, затем идут классы Hippo и Canine, расширяющие Animal, и класс Wolf расширяет Canine.

Создайте новое консольное приложение и добавьте в него классы **Animal**, **Hippo**, **Canine** и **Wolf**:

```

abstract class Animal
{
    public abstract void MakeNoise();
}

class Hippo : Animal
{
    public override void MakeNoise()
    {
        Console.WriteLine("Grunt.");
    }

    public void Swim()
    {
        Console.WriteLine("Splash! I'm going for a swim!");
    }
}

abstract class Canine : Animal
{
    public bool BelongsToPack { get; protected set; } = false;
}

class Wolf : Canine
{
    public Wolf(bool belongsToPack)
    {
        BelongsToPack = belongsToPack;
    }

    public override void MakeNoise()
    {
        if (BelongsToPack)
            Console.WriteLine("I'm in a pack.");
        Console.WriteLine("Aroooooo!");
    }

    public void HuntInPack()
    {
        if (BelongsToPack)
            Console.WriteLine("I'm going hunting with my pack!");
        else
            Console.WriteLine("I'm not in a pack.");
    }
}
    
```

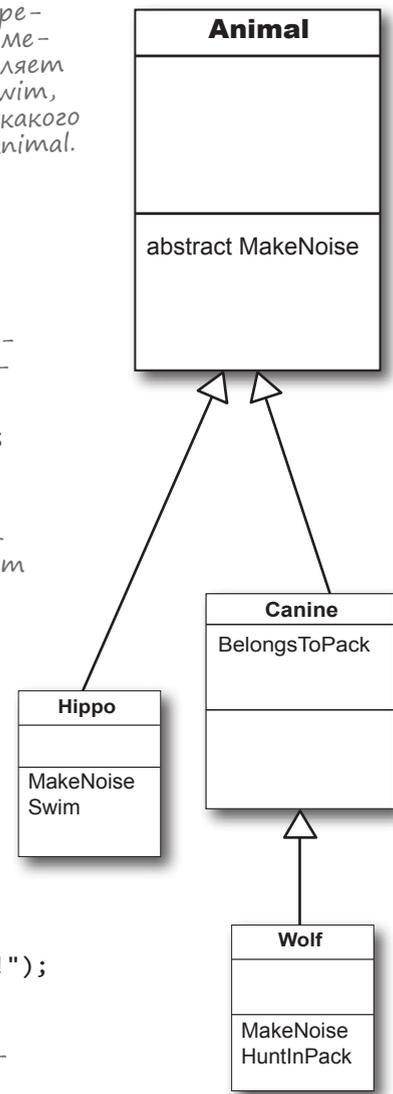
← Абстрактный класс Animal находится на вершине иерархии.

Субкласс Hippo переопределяет абстрактный метод MakeNoise и добавляет собственный метод Swim, который не имеет никакого отношения к классу Animal.

← Абстрактный класс Canine расширяет Animal. Он содержит свое абстрактное свойство BelongsToPack.

← Класс Wolf расширяет Canine и добавляет собственный метод HuntInPack.

← Метод HuntInPack присутствует только в классе Wolf. Он не наследуется от суперкласса.



Заполним код метода Main. Он делает следующее:

- * Он создает массив объектов Hippo и Wolf, а затем перебирает все объекты в цикле foreach.
- * Он использует ссылку на Animal для вызова метода MakeNoise.
- * Если текущим объектом является Hippo, метод Main вызывает его метод Hippo.Swim.
- * Если текущим объектом является Wolf, метод Main вызывает его метод Wolf.HuntInPack.

В главе 6 вы узнали, что разные ссылки могут использоваться для вызова разных методов одного объекта. Когда вы не использовали ключевые слова `override` и `virtual`, а ваша ссылочная переменная относилась к типу `Locksmith`, она вызывала метод `Locksmith.ReturnContents`; но когда переменная относилась к типу `JewelThief`, вызывался метод `JewelThief.ReturnContents`. Здесь происходит нечто похожее.

Проблема в том, что если у вас имеется ссылка на `Animal`, которая указывает на объект `Hippo`, вы не сможете использовать ее для вызова `Hippo.Swim`:

```
Animal animal = new Hippo();
animal.Swim(); // <-- Эта строка не компилируется!
```

Неважно, что вы работаете с объектом `Hippo`. Если вы не используете переменную `Animal`, то вы сможете обращаться только к полям, методам и свойствам `Animal`.

К счастью, у проблемы есть обходное решение. Если вы на сто процентов уверены в том, что работаете с объектом `Hippo`, то вы можете преобразовать ссылку на `Animal` в ссылку на `Hippo` — и после этого обратиться к методу `Hippo.Swim`:

```
Hippo hippo = (Hippo)animal;
hippo.Swim(); // Объект остался тем же, но теперь вы можете вызвать метод Hippo.Swim.
```

А вот как выглядит метод `Main`, в котором ключевое слово `is` используется для вызова `Hippo.Swim` или `Wolf.HuntInPack`:

```
class Program
{
    static void Main(string[] args)
    {
        Animal[] animals =
        {
            new Wolf(false),
            new Hippo(),
            new Wolf(true),
            new Wolf(false),
            new Hippo()
        };

        foreach (Animal animal in animals)
        {
            animal.MakeNoise();
            if (animal is Hippo hippo)
            {
                hippo.Swim();
            }

            if (animal is Wolf wolf)
            {
                wolf.HuntInPack();
            }

            Console.WriteLine();
        }
    }
}
```

Цикл `foreach` перебирает элементы массива «`animals`». Ему нужно объявить переменную типа `Animal`, соответствующего типу массива, но по этой ссылке нельзя будет обратиться к методам `Hippo.Swim` или `Wolf.HuntInPack`.

Цикл `foreach` перебирает элементы массива «`animals`». Ему нужно объявить переменную типа `Animal`, соответствующего типу массива, но по этой ссылке нельзя будет обратиться к методам `Hippo.Swim` или `Wolf.HuntInPack`.

Воспользуйтесь отладчиком и постарайтесь досконально разобраться в том, что же здесь происходит. Установите точку прерывания в первой строке цикла `foreach`; добавьте отслеживания для `animal`, `hippo` и `wolf`; выполните программу в пошаговом режиме.

А если мы захотим, чтобы другие животные плавали или охотились в стае?

А вы знаете, что львы тоже охотятся стаями (HuntInPack)? И что тигры умеют плавать (Swim)? А как насчет собак, которые охотятся стаями и плавают? Если вы попытаетесь добавить методы Swim и HuntInPack ко всем животным в нашем зоопарке, которым они могут понадобиться, цикл foreach будет становиться все длиннее и длиннее.

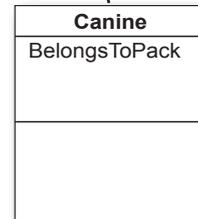
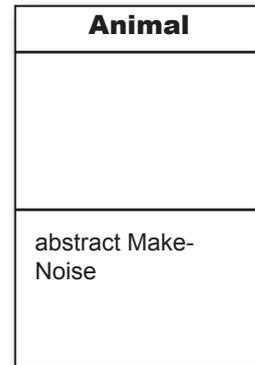
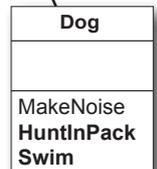
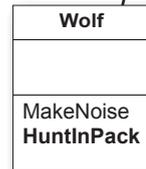
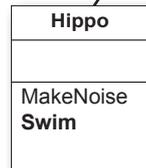
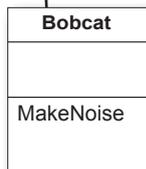
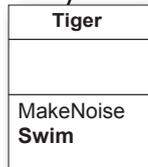
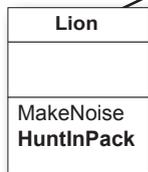
Красота определения абстрактного метода или свойства в базовом классе и его переопределения в subclasses заключается в том, что вам **не нужно ничего знать о subclasses**, чтобы использовать его. Можно добавить сколько угодно subclasses Animal, и цикл все равно будет работать:

```
foreach (Animal animal in animals) {  
    animal.MakeNoise();  
}
```

Метод MakeNoise всегда будет реализован объектом.

Собственно, этот факт можно рассматривать как **контракт**, соблюдение которого обеспечивается компилятором.

Итак, если бы методы HuntInPack и Swim тоже могли рассматриваться как контракты, мы бы могли использовать с ними другие обобщенные переменные — подобно тому, как это делается с классом Animal?



Объекты Wolf и Dog едят и спят одинаково, но издают разные звуки.

Использование интерфейсов для работы с классами, выполняющими одну задачу

Классы плавающих животных содержат метод Swim, а классы животных, охотящихся стаей, содержат метод HuntInPack. Допустим, это неплохое начало. Теперь мы хотим написать код, работающий с объектами, которые плавают или охотятся стаей, — и здесь по-настоящему проявляется мощь интерфейсов. Воспользуйтесь ключевым словом `interface` для определения двух интерфейсов и **добавьте абстрактный компонент** в каждый интерфейс:

```
interface ISwimmer {
    void Swim();
}

interface IPackHunter {
    void HuntInPack();
}
```

Добавьте!



Затем **заставьте классы Hippo и Wolf реализовать интерфейсы**, добавив интерфейс в конец объявления каждого класса. Используйте двоеточие (`:`) для обозначения реализации интерфейса, по аналогии с тем, как делается при расширении класса. Если класс уже расширяет другой класс, достаточно поставить запятую после имени суперкласса и указать имя. После этого необходимо проследить за тем, чтобы класс **реализовал все компоненты интерфейса**, иначе компилятор выдаст сообщение об ошибке.

```
class Hippo : Animal, ISwimmer {
    /* Код остается неизменным — и он ДОЛЖЕН включать метод Swim. */
}

class Wolf : Canine, IPackHunter {
    /* Код остается неизменным — и он ДОЛЖЕН включать метод HuntInPack. */
}
```

Используйте ключевое слово «is» для проверки того, что животное плавает или охотится стаей

При помощи **ключевого слова is** можно проверить, реализует ли конкретный объект заданный интерфейс; этот способ работает независимо от того, какие еще интерфейсы реализуются объектом. Если переменная `animal` ссылается на объект, реализующий интерфейс `ISwimmer`, то проверка `animal is ISwimmer` дает результат `true` и вы можете безопасно привести его к ссылке на `ISwimmer` для вызова метода `Swim`.

```
foreach (Animal animal in animals)
{
    animal.MakeNoise();
    if (animal is ISwimmer swimmer)
    {
        swimmer.Swim();
    }
    if (animal is IPackHunter hunter)
    {
        hunter.HuntInPack();
    }
    Console.WriteLine();
}
```

Как будет выглядеть ваш код, если у вас появятся 20 разных плавающих subclasses Animal? Понадобится 20 разных команд `if (animal is...)`, которые будут преобразовывать `animal` к разным subclassам для вызова метода `Swim`. При использовании `ISwimmer` достаточно одной проверки.

Как и прежде, мы используем ключевое слово `is`, но на этот раз оно используется с интерфейсами. При этом работает оно точно так же, как прежде.

is и безопасная навигация по иерархии классов

Помните, как вы выполняли упражнение с заменой Bee на IWorker в системе управления ульем? Тогда еще вы столкнулись с исключением InvalidCastException. *Теперь мы объясним, почему это произошло.*

- ✔ **Ссылку на NectarCollector можно безопасно преобразовать в ссылку на IWorker.**
Все экземпляры NectarCollector являются Bee (т. е. расширяют базовый класс Bee), поэтому вы всегда можете воспользоваться оператором =, чтобы получить ссылку на NectarCollector и присвоить ее переменной Bee.

```
HoneyManufacturer lily = new HoneyManufacturer();  
Bee hiveMember = lily;
```

А поскольку объект Bee реализует интерфейс IWorker, его тоже можно безопасно преобразовать к ссылке на IWorker.

```
HoneyManufacturer daisy = new HoneyManufacturer();  
IWorker worker = daisy;
```

Преобразования такого рода безопасны: они никогда не приводят к выдаче исключения IllegalCastException, потому что они только присваивают более конкретные объекты переменным более общего типа *в той же иерархии классов*.

- ✘ **Ссылку на Bee невозможно безопасно преобразовать в ссылку на NectarCollector.**
Безопасное преобразование возможно в другом направлении (преобразование Bee в NectarCollector), потому что не все объекты Bee являются экземплярами NectarCollector. Например, HoneyManufacturer *определенно не является* NectarCollector. Таким образом, следующее преобразование:

```
IWorker pearl = new HoneyManufacturer();  
NectarCollector irene = (NectarCollector)pearl;
```

является недействительным: вы пытаетесь преобразовать объект к переменной, которая не соответствует его типу.

- ! **Ключевое слово is позволяет безопасно выполнять преобразования типов.**
К счастью, ключевое слово **is** безопаснее приведения типов с круглыми скобками. Оно позволяет проверить, что тип соответствует заданному и выполняет приведение к новой переменной только при соответствии типов.

```
if (pearl is NectarCollector irene) {  
    /* Код, использующий объект NectarCollector. */  
}
```

Этот код никогда не выдает исключение InvalidCastException, потому что код, использующий объект NectarCollector, выполняется только в том случае, если pearl является объектом NectarCollector.

В C# также существует другой инструмент для безопасного преобразования типов: ключевое слово as

C# предоставляет еще один инструмент для безопасных преобразований: **ключевое слово as**. Также as выполняет безопасные преобразования типов. Вот как оно работает: предположим, у вас имеется ссылка на IWorker с именем pearl и вы хотите безопасно преобразовать ее в переменную NectarCollector с именем irene. Безопасное преобразование в NectarCollector может быть выполнено следующим образом:

```
NectarCollector irene = pearl as NectarCollector;
```

Если типы совместимы, то эта команда присваивает переменной irene ссылку на тот же объект, на который указывает переменная pearl. Если тип объекта не соответствует типу переменной, то исключение не выдается. Вместо этого **переменной присваивается null**, что можно проверить командой if:

```
if (irene != null) {  
    /* Код, в котором используется объект NectarCollector */  
}
```



**Будьте
осторожны!**

В очень старых версиях C# ключевое слово is работает иначе.

Ключевое слово is существует в C# в течение долгого времени, но только в версии C# 7.0, выпущенной в 2017 году, is позволяет объявить новую переменную. Таким образом, если вы работаете в Visual Studio 2015, следующая конструкция будет вам недоступна: if (pearl is NectarCollector irene) {...}.

Вместо этого придется использовать ключевое слово as для выполнения преобразований, а затем проверить результат и определить, равен ли он null:

```
NectarCollector irene = pearl as NectarCollector;  
if (irene != null) { /* Код, в котором используется ссылка irene */ }
```

Возьми в руку карандаш

В массиве слева используются типы из модели классов Бее. Два из этих типов не компилируются — вычеркните их. Справа приведены три команды с использованием ключевого слова is. Запишите, при каких значениях i каждая из них даст результат true.

```
IWorker[] bees = new IWorker[8];  
bees[0] = new HiveDefender();  
bees[1] = new NectarCollector();  
bees[2] = bees[0] as IWorker;  
bees[3] = bees[1] as NectarCollector;  
bees[4] = IDefender;  
bees[5] = bees[0];  
bees[6] = bees[0] as Object;  
bees[7] = new IWorker();
```

1. (bees[i] is IDefender)
-
2. (bees[i] is IWorker)
-
3. (bees[i] is Bee)
-

Использование повышающего и понижающего приведения типа для перемещения вверх и вниз по иерархии классов

На диаграммах классов базовый класс обычно размещается наверху, под ним идут subclasses, ниже располагаются subclasses subclasses, и т. д. Чем выше класс на диаграмме, тем более абстрактна его природа; чем ниже класс на диаграмме, тем он конкретнее. «Абстрактное наверху, конкретное внизу» не является непреложным правилом; это **соглашение**, благодаря которому вы можете с первого взгляда определить, как работают наши модели классов.

В главе 6 мы говорили о том, что subclass всегда может использоваться вместо базового класса, от которого он наследует, но базовый класс далеко не всегда может использоваться вместо расширяющего его subclass. На это правило также можно взглянуть иначе: в каком-то смысле **вы перемещаетесь вверх и вниз по иерархии классов**. Допустим, вы начинаете со следующей команды:

```
NectarCollector ida = new NectarCollector();
```

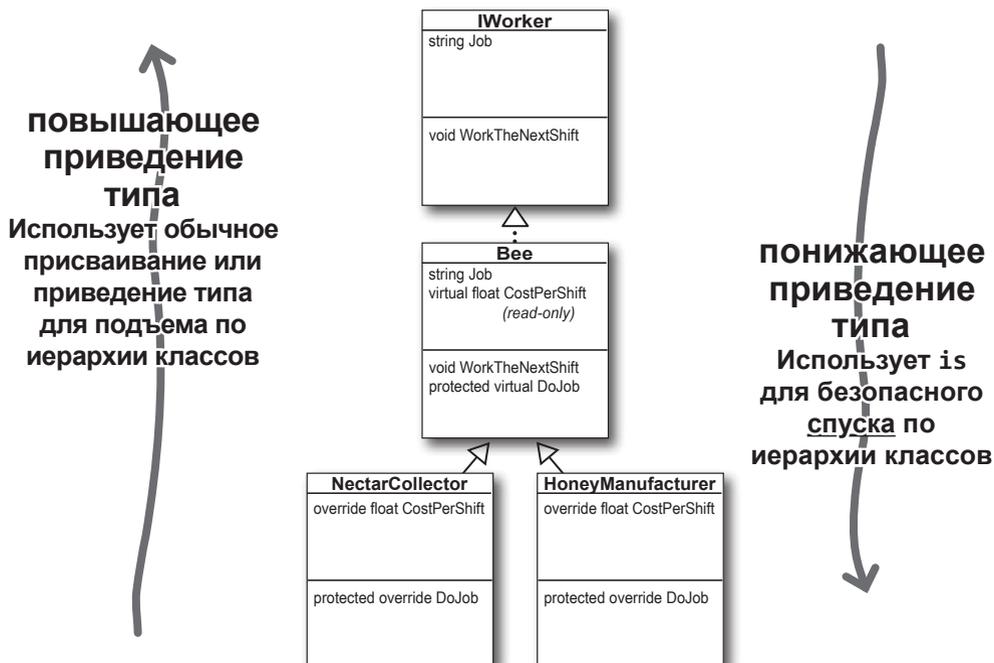
Оператор = может использоваться для нормального присваивания (для суперклассов) или приведения типа (для интерфейсов). Эти операции означают **перемещение вверх** по иерархии и называются **повышающим приведением типа** (upcasting):

```
// Выполнить повышающее преобразование NectarCollector в Bee
Bee beeReference = ida;

// Это повышающее преобразование безопасно, потому что все объекты Bee реализуют IWorker
IWorker worker = (IWorker)beeReference;
```

Также можно перемещаться по иерархии в другом направлении, используя оператор is для безопасного **перемещения вниз** по иерархии классов. Такое преобразование называется **понижающим приведением типа** (downcasting).

```
// Выполнить понижающее преобразование IWorker в NectarCollector
if (worker is NectarCollector rose) { /* Код, использующий ссылку rose */ }
```



Пример повышающего приведения типа

Если вы пытаетесь вычислить, как бы вам сэкономить на счетах за электричество, вас на самом деле не интересует, что делает каждый из ваших электроприборов, — важно лишь то, что они потребляют энергию. Таким образом, если вы пишете программу для отслеживания потребления электричества, то скорее всего, вы ограничитесь написанием класса `Appliance` (Электроприбор). Но если вам нужно отличать кофеварку от печи, вы построите иерархию классов и добавите методы и свойства, специфические для кофеварки или печи, в классы `CoffeeMaker` и `Oven`; эти классы наследуют от класса `Appliance`, который содержит их общие методы и свойства.

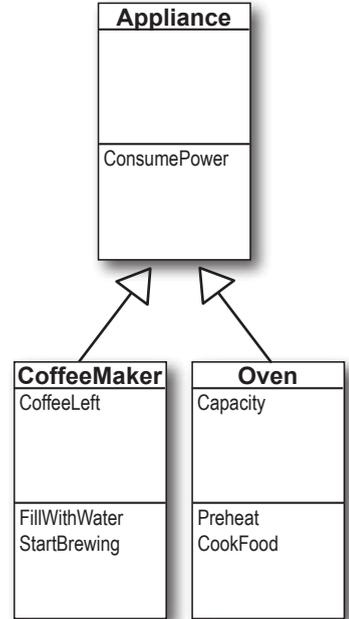
Затем пишется метод для отслеживания энергопотребления:

```
void MonitorPower(Appliance appliance) {
    /* код, добавляющий информацию в базу данных
       энергопотребления домашних электроприборов */
}
```

Чтобы воспользоваться этим методом для отслеживания энергопотребления кофеварки, вы создаете экземпляр `CoffeeMaker` и передаете ссылку на него непосредственно методу:

```
CoffeeMaker mrCoffee = new CoffeeMaker();
MonitorPower(mrCoffee);
```

Отличный пример повышающего преобразования. Хотя метод `MonitorPower` получает ссылку на объект `Appliance`, ему также можно передать ссылку `mrCoffee`, потому что `CoffeeMaker` является subclassом `Appliance`.



Возьми в руку карандаш

Решение

В массиве слева используются типы из модели классов `Bee`. Два из этих типов не компилируются — вычеркните их. Справа приведены три команды с использованием ключевого слова `is`. Запишите, при каких значениях `i` каждая из них даст результат `true`.

```
IWorker[] bees = new IWorker[8];
bees[0] = new HiveDefender();
bees[1] = new NectarCollector();
bees[2] = bees[0] as IWorker;
bees[3] = bees[1] as NectarCollector;
bees[4] = IDefender;
bees[5] = bees[0];
bees[6] = bees[0] as Object;
bees[7] = new IWorker();
```

Элементы 0, 2 и 6 в массиве указывают на один и тот же объект `HiveDefender`.

1. (bees[i] is IDefender)

2. (bees[i] is IWorker)

→ 0, 2 и 6

0, 1, 2, 3, 5, 6

3. (bees[i] is Bee)

0, 1, 2, 3, 5, 6

Все эти объекты расширяют `Bee`, а `Bee` реализует `IWorker`, так что все они являются объектами `Bee` и `IWorker`.

Повышающее приведение преобразует CoffeeMaker в Appliance

Когда вы заменяете базовый класс субклассом, например используете CoffeeMaker вместо Appliance или Hippo вместо Animal, это называется **повышающим приведением типа**. Это чрезвычайно мощный инструмент, применяемый при построении иерархий классов. Единственный недостаток повышающего приведения заключается в том, что вы можете использовать только свойства и методы базового класса. Иначе говоря, когда вы рассматриваете CoffeeMaker как Appliance, вы не сможете приказать кофеварке сварить кофе или заполнить ее водой. Вы *можете* проверить, включен прибор или нет, потому что это можно сделать с любым объектом Appliance (а свойство PluggedIn является частью класса Appliance).

1 Создайте несколько объектов.

Начнем с создания нескольких экземпляров CoffeeMaker и Oven:

```
CoffeeMaker misterCoffee = new CoffeeMaker();
Oven oldToasty = new Oven();
```

2 Как создать массив объектов Appliance?

Вы не сможете поместить CoffeeMaker в массив Oven[], как и поместить Oven в массив CoffeeMaker[]. При этом **оба** вида объектов можно разместить в массиве Appliance[]:

```
Appliance[] kitchenWare = new Appliance[2];
kitchenWare[0] = misterCoffee;
kitchenWare[1] = oldToasty;
```

Вы можете воспользоваться повышающим приведением типа для создания массива с элементами Appliance, в котором могут храниться экземпляры как CoffeeMaker, так и Oven.

3 Однако вы не можете рассматривать любой экземпляр Appliance как Oven.

Если у вас имеется ссылка на Appliance, вы сможете по ней обращаться **только** к методам и свойствам, связанным с электроприборами вообще. Вы **не сможете** использовать методы и свойства CoffeeMaker по ссылке Appliance, *даже если вы точно знаете, что имеете дело с CoffeeMaker*. Таким образом, следующие команды работают нормально, потому что они работают с объектом CoffeeMaker как с Appliance:

```
Appliance powerConsumer = new CoffeeMaker();
powerConsumer.ConsumePower();
```

Но при попытке использовать объект как CoffeeMaker:

```
powerConsumer.StartBrewing();
```

Эта строка не компилируется, потому что powerConsumer является ссылкой на Appliance и может использоваться только для выполнения операций Appliance.

код компилироваться не будет и в IDE будет выведено сообщение об ошибке:

```
CS1061 'Appliance' does not contain a definition for 'StartBrewing' and no accessible extension method 'StartBrewing' accepting a first argument of type 'Appliance' could be found (are you missing a using directive or an assembly reference?)
```

После повышающего приведения от субкласса к базовому классу вам будут доступны только методы и свойства, **соответствующие ссылке**, используемой для обращения к объекту.

powerConsumer представляет собой ссылку на Appliance, указывающую на объект CoffeeMaker.

Включать этот код в приложение необязательно — просто прочитайте его и постарайтесь понять, как работают повышающие и понижающие приведения типа. Вы еще **неоднократно** потренируетесь в их применении.



Понижающее приведение преобразует Appliance в CoffeeMaker

Повышающее приведение типа – замечательный инструмент, потому что он позволяет использовать CoffeeMaker или Oven везде, где нужен объект Appliance. Впрочем, у него есть большой недостаток: если вы используете ссылку на Appliance, которая указывает на объект CoffeeMaker, вы сможете использовать только методы и свойства, принадлежащие Appliance. В таких ситуациях на помощь приходит **понижающее** приведение типа: вы берете **ранее повышенную ссылку** и изменяете ее обратно. Чтобы проверить, что Appliance в действительности является объектом CoffeeMaker и его можно преобразовать обратно в CoffeeMaker, используйте ключевое слово is.

1 Начнем со ссылки CoffeeMaker, уже подвергнутой повышающему приведению типа.

Для этого использовался следующий код:

```
Appliance powerConsumer = new CoffeeMaker();
powerConsumer.ConsumePower();
```

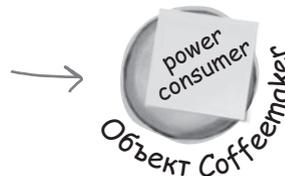
2 А если потребуется преобразовать Appliance обратно в CoffeeMaker?

Допустим, вы строите приложение, которое обращается к массиву ссылок на Appliance, чтобы объект CoffeeMaker мог начать варить кофе. Вы не можете воспользоваться ссылкой на Appliance для вызова метода CoffeeMaker:

```
Appliance someAppliance = appliances[5];
someAppliance.StartBrewing();
```

Эта команда не будет компилироваться – вы получите ошибку компиляции «'Appliance' не содержит определения 'StartBrewing'», потому что StartBrewing является компонентом CoffeeMaker, а вы используете ссылку на Appliance.

Ссылка на Appliance, которая указывает на объект CoffeeMaker. Она может использоваться только для обращения к компонентам класса Appliance.



3 Но так как мы точно знаем, что имеем дело с объектом CoffeeMaker, мы работаем с объектом соответствующим образом.

Первым шагом становится ключевое слово is. Когда вы точно знаете, что ссылка Appliance указывает на объект CoffeeMaker, вы можете воспользоваться is для ее понижающего приведения. Это позволит вам использовать методы и свойства класса CoffeeMaker. Так как класс CoffeeMaker наследует от Appliance, он содержит все методы и свойства Appliance.

```
if (someAppliance is CoffeeMaker javaJoe) {
    javaJoe.StartBrewing();
}
```



Ссылка javaJoe указывает на тот же объект CoffeeMaker, что и powerConsumer, но является ссылкой на CoffeeMaker и может использоваться для вызова метода StartBrewing.

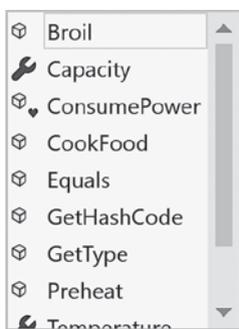


Повышающие и понижающие приведения типов также работают и с интерфейсами

Интерфейсы отлично работают с повышающими и понижающими приведениями типов. Добавим интерфейс ICooksFood для каждого класса, который умеет разогревать еду. Затем добавим класс Microwave — и Microwave, и Oven реализуют интерфейс ICooksFood. Теперь ссылкой на объект Oven может быть ссылка на ICooksFood, ссылка на Microwave или ссылка на Oven. Это означает, что вы можете создать ссылки трех разных типов, указывающие на объект Oven, и каждая из них может использоваться для обращения к разным компонентам в зависимости от типа ссылки. К счастью, функция IntelliSense в IDE поможет точно определить, что можно или нельзя делать с каждой из них:

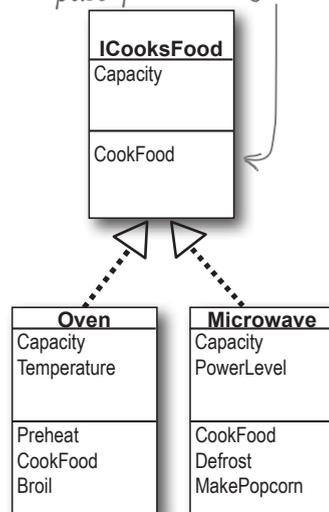
```
Oven misterToasty = new Oven();
misterToasty.
```

По ссылке на Oven можно обращаться ко всем компонентам Oven.



Как только вы введете точку, на экране появляется окно IntelliSense со списком всех компонентов, которые вы можете использовать. misterToasty — ссылка на Oven, указывающая на объект Oven; соответственно, по ней можно обращаться ко всем методам и свойствам. Это **самый конкретный тип**, который может указывать только на объекты Oven.

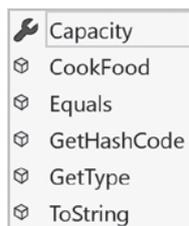
Любой класс, реализующий ICooksFood, представляет электроприбор, способный разогревать еду.



Чтобы обратиться к компонентам интерфейса ICooksFood, преобразуйте ссылку в ссылку на ICooksFood:

```
if (misterToasty is ICooksFood cooker) {
    cooker.
```

По ссылке на ICooksFood можно обращаться только к компонентам, которые являются частью интерфейса.



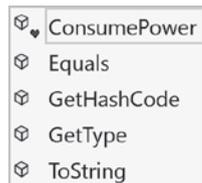
cooker — ссылка на ICooksFood, указывающая на тот же объект Oven. По ней также можно обращаться только к компонентам ICooksFood, но ссылка также может указывать на объект Microwave.

Три разные ссылки, указывающие на один объект, могут обращаться к разным методам и свойствам в зависимости от типа ссылки.

Это тот же класс Oven, который использовался ранее, поэтому он также расширяет базовый класс Appliance. Если вы можете использовать ссылку на Appliance для обращения к объекту, видны будут только компоненты класса Appliance:

```
if (misterToasty is Appliance powerConsumer)
    powerConsumer.
```

Appliance содержит только один компонент ConsumePower, и только он присутствует в раскрывающемся списке.



powerConsumer является ссылкой на Appliance. По такой ссылке можно обращаться только к открытым полям, методам и свойствам в Appliance. Эта ссылка является более общей, чем ссылка на Oven (так что при желании она также может указывать на объект CoffeeMaker).

Часто Задаваемые Вопросы

В: Еще раз: вы говорите, что повышающее приведение типа возможно всегда, но понижающее приведение возможно не всегда. Почему?

О: Потому, что повышающее приведение не сработает, если вы попытаетесь назначить объекту класс, от которого он не наследует, или интерфейс, который он не реализует. Компилятор немедленно определяет, что повышающее приведение выполняется некорректно, и выдает ошибку. Когда мы говорим «повышающее приведение возможно всегда, но понижающее приведение возможно не всегда», фактически мы говорим: «Каждая печь является электроприбором, но не каждый электроприбор является печью».

В: Я читал в интернете, что интерфейсы определяют контракты, но не понимаю почему. Что это значит?

О: Да, многие люди сравнивают интерфейсы с контрактами. («В чем интерфейс похож на контракт?») — этот вопрос очень часто задается на собеседованиях при приеме на работу.) Когда вы указываете, что ваш класс реализует интерфейс, вы тем самым обещаете компилятору, что он будет содержать определенные методы. Компилятор следит за тем, чтобы вы сдержали свое обещание. Если такая точка зрения поможет вам понять интерфейсы — пожалуйста, рассматривайте их с этой точки зрения.

Но на наш взгляд, лучше представить себе интерфейс как некий контрольный список. Компилятор проходит по списку и убеждается в том, что вы включили все методы интерфейса в ваш класс. Если вы этого не сделали, то компилятор протестует и программа компилироваться не будет.

В: Зачем мне использовать интерфейсы? Такое впечатление, что мы просто добавляем новые ограничения, а класс при этом никак не изменяется.

О: Потому что когда ваш класс реализует интерфейс, вы можете использовать этот интерфейс как тип для объявления ссылки, которая может указывать на любой экземпляр класса, реализующего этот интерфейс. Данная возможность очень полезна — она позволяет создать один ссылочный тип, который может работать с множеством разнообразных объектов.

Рассмотрим небольшой пример. Лошадь, вол, мул и бык могут тянуть повозку. В нашей модели зоопарка `Horse`, `Ox`, `Mule` и `Steer` будут разными классами. Допустим, в вашем зоопарке имеется аттракцион, на котором посетители катаются на повозках, и вы хотите создать массив любых животных, способных тянуть повозку. И тут выясняется, что создать массив, в котором могут храниться все эти классы, невозможно. Это было бы возможно, если бы они наследовали от одного базового класса, но это не так. Что же делать?

На помощь приходят интерфейсы. Вы можете создать интерфейс `IPuller` с методами для перемещения повозок. Тогда массив объявляется следующим образом:

```
IPuller[] pullerArray;
```

Теперь в массив можно поместить ссылку на любое животное — при условии, что оно реализует интерфейс `IPuller`.

Интерфейс напоминает контрольный список, по которому компилятор проверяет, что ваш класс реализует необходимый набор методов.

Интерфейсы могут наследовать от других интерфейсов

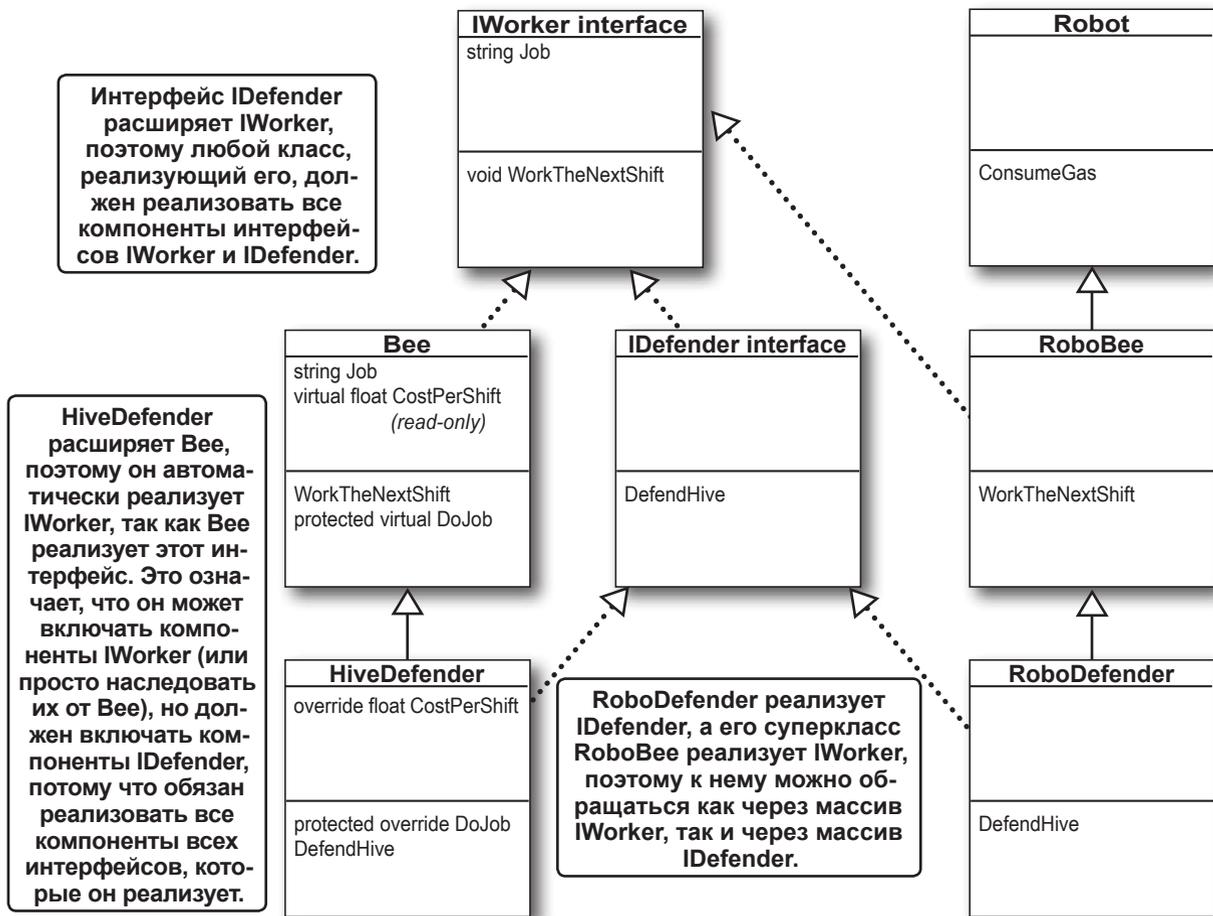
Как упоминалось ранее, один класс, наследующий от другого, получает все методы и свойства базового класса. **Наследование интерфейсов** устроено проще. Так как интерфейс не содержит реальных тел методов, вам не нужно беспокоиться о вызове конструкторов или методов базового класса. Наследующие интерфейсы просто **накапливают все компоненты** расширяемых интерфейсов.

Как это выглядит в коде? Добавим интерфейс IDefender, наследующий от IWorker:

```
interface IDefender : IWorker {
    void DefendHive();
}
```

← Используйте двоеточие (:), чтобы обозначить, что интерфейс расширяет другой интерфейс.

Если класс реализует интерфейс, он должен реализовать каждое свойство и каждый метод этого интерфейса. Если интерфейс наследует от другого интерфейса, то также должны быть реализованы все свойства и методы *этого* интерфейса. Таким образом, любой класс, реализующий IDefender, должен реализовать не только все компоненты IDefender, но и все компоненты IWorker. Следующая модель классов включает IWorker и IDefender, а также **две отдельные иерархии**, в которых они реализуются.





Упражнение

Создайте новое консольное приложение с классами, реализующими интерфейс IClown. Сможете ли вы разобраться в том, как должен строиться код нижних уровней?

1

Начните с создания интерфейса IClown, созданного ранее:

```
interface IClown {
    string FunnyThingIHave { get; }
    void Honk();
}
```

2

Расширьте новый интерфейс IScaryClown, расширяющий IClown. Он должен содержать строковое свойство с именем ScaryThingIHave с get-методом, но без set-метода и void-метод с именем ScareLittleChildren.

3

Создайте классы, реализующие эти интерфейсы:

- * **Класс с именем FunnyFunny**, реализующий IClown. Он использует частную строковую переменную с именем funnyThingHave. Get-метод FunnyThingIHave использует funnyThingHave в качестве резервного поля. Используйте конструктор, который получает параметр и использует его для инициализации частного поля. Метод Honk выводит сообщение: «Hi kids! I have a», значение funnyThingIHave и точку.

- * **Класс с именем ScaryScary**, реализующий IScaryClown. Он использует частную переменную с именем scaryThingCount для хранения целого числа. Конструктор задает как поле scaryThingCount, так и funnyThingIHave, наследуемое ScaryScary от FunnyFunny. Get-метод ScaryThingIHave возвращает строку с числом из конструктора, за которым следует слово «spiders». Метод ScareLittleChildren выводит на консоль сообщение «Boo! Gotcha! Look at my...!» (... заменяется соответствующим значением поля.)

4

Ниже приведен новый код метода Main — к сожалению, он не работает. Сможете ли вы определить, как исправить его, чтобы он строил и выводил сообщения на консоль?

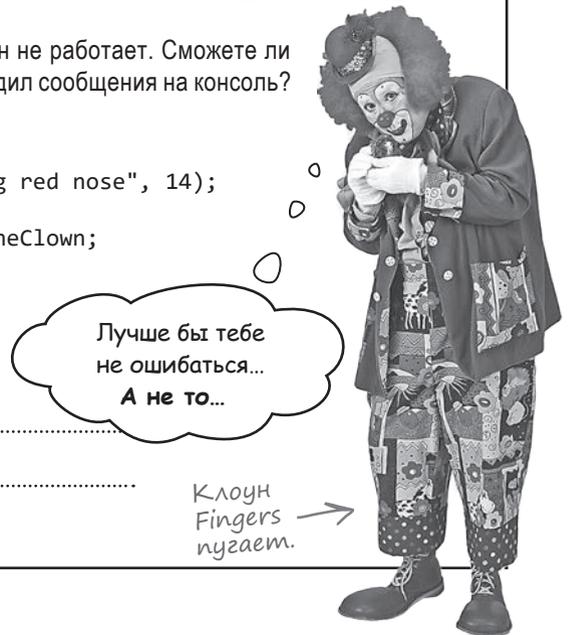
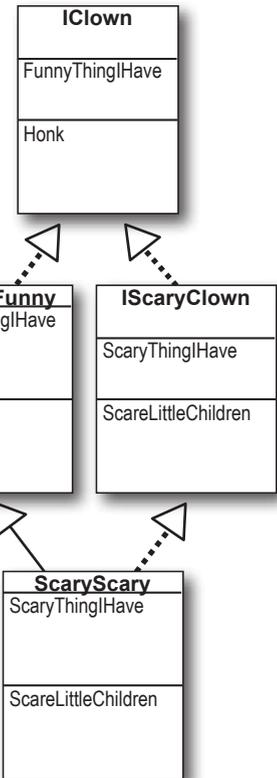
```
static void Main(string[] args)
{
    IClown fingersTheClown = new ScaryScary("big red nose", 14);
    fingersTheClown.Honk();
    IScaryClown iScaryClownReference = fingersTheClown;
    iScaryClownReference.ScareLittleChildren();
}
```

Прежде чем запускать этот код, **запишите результат**, который метод Main выведет на консоль (после исправления):

.....

.....

Затем выполните код и проверьте свой ответ.





Упражнение Решение

Создайте новое консольное приложение с классами, реализующими интерфейс `IClown`. Можете ли вы разобраться в том, как должен строиться код нижних уровней?

Интерфейс `IScaryClown` расширяет `IClown`, добавляя свойство и метод:

```
interface IScaryClown : IClown
{
    string ScaryThingIHave { get; }
    void ScareLittleChildren();
}
```

Класс `FunnyFunny` реализует интерфейс `IClown` и использует конструктор для инициализации резервного поля:

```
class FunnyFunny : IClown
{
    private string funnyThingIHave;
    public string FunnyThingIHave { get { return funnyThingIHave; } }

    public FunnyFunny(string funnyThingIHave)
    {
        this.funnyThingIHave = funnyThingIHave;
    }

    public void Honk()
    {
        Console.WriteLine($"Hi kids! I have a {funnyThingIHave}.");
    }
}
```

Точно такие же конструкторы и резервные поля, как использовались в главе 5.

Класс `ScaryScary` расширяет класс `FunnyFunny` и реализует интерфейс `IScaryClown`. Его конструктор использует ключевое слово `base` для вызова конструктора `FunnyFunny`, инициализирующего приватное резервное поле:

```
class ScaryScary : FunnyFunny, IScaryClown
{
    private int scaryThingCount;

    public ScaryScary(string funnyThing, int scaryThingCount) : base(funnyThing)
    {
        this.scaryThingCount = scaryThingCount;
    }

    public string ScaryThingIHave { get { return $"{scaryThingCount} spiders"; } }

    public void ScareLittleChildren()
    {
        Console.WriteLine($"Boo! Gotcha! Look at my {ScaryThingIHave}!");
    }
}
```

`FunnyFunny.funnyThingIHave` — приватное поле, и класс `ScaryScary` не может обратиться к нему — он должен использовать ключевое слово `base` для вызова конструктора `FunnyFunny`.

Чтобы исправить метод `Main`, замените строки 3 и 4 метода следующими строками, использующими оператор `is`:

```
if (fingersTheClown is IScaryClown iScaryClownReference)
{
    iScaryClownReference.ScareLittleChildren();
}
```

Вы можете присвоить ссылке на `FunnyFunny` объект `ScaryScary`, потому что `ScaryScary` наследует от `FunnyFunny`. Ссылке на `IScaryClown` невозможно присвоить произвольный объект клоуна, потому что вы не знаете, является ли этот клоун страшным (`Scary`). Именно по этой причине необходимо использовать ключевое слово `is`.

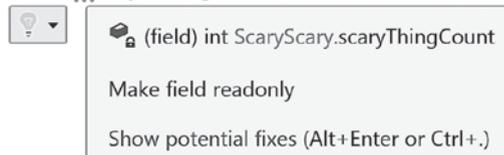


Я заметил, что IDE часто спрашивает меня, хочу ли я сделать поле **доступным только для чтения**. Стоит ли это делать?

Безусловно! Ограничение доступа к полям только для чтения способствует предотвращению ошибок.

Вернитесь к полю `ScaryScary.scaryThingCount` — IDE выводит точки под первыми двумя буквами имени поля. Наведите указатель мыши на точки, чтобы в IDE открылось окно:

```
private int scaryThingCount;
```



Нажмите `Ctrl+`, чтобы вызвать список действий, и выберите команду «**Add readonly modifier**», чтобы добавить **ключевое слово** `readonly` в объявление:

```
private readonly int scaryThingCount;
```

Теперь значение поля может быть задано только при объявлении или в конструкторе. Если вы попытаетесь изменить его значение в любой другой точке метода, компилятор выдаст сообщение об ошибке:

CS0191 A readonly field cannot be assigned to (except in a constructor or a variable initializer)

Ключевое слово `readonly`... еще один механизм, используемый C# для улучшения безопасности ваших данных.

Ключевое слово `readonly`

Важная причина для использования инкапсуляции — предотвращение случайной перезаписи данных одного класса со стороны другого класса. Что помещает классу перезаписать свои собственные данные? В этом может помочь ключевое слово `readonly`. Любое поле с пометкой `readonly` может быть изменено только при объявлении или в конструкторе.

Чаще Задаваемые Вопросы

В: Зачем использовать интерфейс, если можно просто записать все необходимые методы прямо в классе?

О: При использовании интерфейсов вы также записываете методы в своем классе. Интерфейсы позволяют группировать эти классы в соответствии с работой, которую они выполняют. Они помогают следить за тем, чтобы каждый класс, выполняющий определенную разновидность работы, делал это с использованием одних и тех же методов. При этом класс может выполнять работу так, как считает нужным, и из-за интерфейса вам не нужно беспокоиться о том, как он это будет делать.

Пример: в системе могут присутствовать классы `Truck` (Грузовик) и `Sailboat` (Яхта), реализующие интерфейс `ICarryPassenger`. Допустим, интерфейс `ICarryPassenger` требует, чтобы любой класс, реализующий этот интерфейс, содержал метод `ConsumeEnergy`. Тогда программа может использовать обе разновидности классов для перевозки пассажиров, несмотря на то что метод `ConsumeEnergy` класса `Sailboat` использует силу ветра, а метод класса `Truck` использует дизельное топливо.

Теперь представим, что интерфейс `ICarryPassenger` отсутствует. Тогда вам придется туго — нужно будет как-то сообщить программе, какие транспортные средства могут перевозить пассажиров, а какие нет. Вам придется просматривать каждый класс, который может использоваться программой, и определять, присутствует ли в нем метод для перевозки пассажиров из одного места в другое. И тогда пришлось бы вызывать для транспортных средств, которые могут использоваться в вашей программе, тот метод, который был в них определен для перевозки пассажиров. При отсутствии стандартного интерфейса методам могут быть присвоены самые разные имена, они могут быть спрятаны в других методах. Как видите, ситуация очень быстро усложняется.

В: Зачем использовать свойства в интерфейсах? Почему не ограничиться полями?

О: Хороший вопрос. Интерфейс определяет только механизм выполнения классом конкретной разновидности работы. Сам по себе объектом он не является, поэтому вы не сможете создать его экземпляр и сохранить в нем информацию. Если вы добавите поле, которое является простым объявлением переменной, `C#` придется где-то хранить эти данные, ведь интерфейс этого не может. Свойство — механизм, при котором нечто выглядит для других объектов как поле, но так как в действительности это метод, никакие данные на самом деле не сохраняются.

В: Чем ссылка на обычный объект отличается от ссылки на интерфейс?

О: Вы уже знаете, как работают обычные ссылки на объекты. Если создать экземпляр `Skateboard` с именем `vertBoard`, а потом новую ссылку на него с именем `halfPipeBoard`, обе ссылки будут указывать на одно и то же. Но если `Skateboard` реализует интерфейс `IStreetTricks` и вы создадите ссылку на интерфейс, указывающую на `Skateboard`, с именем `streetBoard`, то по ней будут доступны только методы класса `Skateboard`, которые также входят в интерфейс `IStreetTricks`.

Все три ссылки в действительности указывают на один и тот же объект. Если вы обратитесь к объекту по ссылке `halfPipeBoard` или `vertBoard`, вы сможете обратиться к любому методу или свойству объекта. Если же вы обратитесь к нему по ссылке `streetBoard`, вам будут доступны только методы и свойства из интерфейса.

В: И зачем тогда использовать ссылки на интерфейсы, если они только ограничивают возможности работы с объектом?

О: Ссылки на интерфейсы позволяют работать с группами разнородных объектов, которые выполняют одну операцию. Вы можете создать массив с типами ссылок на интерфейсы, которые позволяют передавать информацию методам `ICarryPassenger` и обратно независимо от того, работаете вы с объектом `Truck`, `Horse`, `Unicycle` или `Car`. Вероятно, каждый из этих объектов будет решать свою задачу не так, как другие, но со ссылками на интерфейсы вы точно знаете, что они содержат одни и те же методы, которые получают одни и те же параметры и имеют те же возвращаемые типы. Итак, вы можете обращаться к ним и передавать информацию абсолютно одинаково.

В: Напомните, для чего мне стоит объявить компонент класса защищенным (`protected`) вместо приватного (`private`) или открытого (`public`)?

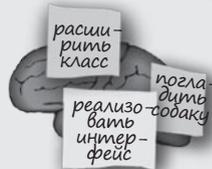
О: Потому что это помогает улучшить инкапсуляцию классов. Во многих случаях субклассу необходим доступ к некоторой внутренней части своего базового класса. Например, если вы хотите переопределить свойство, достаточно часто в `get`-методе используется резервное поле базового класса. При построении классов объявляйте компоненты открытыми, только если для этого есть веские причины. Модификатор доступа `protected` позволяет раскрыть компонент только для субкласса, которому он необходим, и оставить его приватным для всех остальных.

Ссылкам на интерфейсы известны только методы и свойства, определенные в интерфейсе.

КЛЮЧЕВЫЕ МОМЕНТЫ

- **Интерфейс** определяет методы и свойства, которые должны быть реализованы классом.
- Интерфейсы определяют **обязательные компоненты** в виде абстрактных методов и свойств.
- По умолчанию все компоненты интерфейсов являются **открытыми и абстрактными** (поэтому ключевые слова `public` и `abstract` для компонентов не используются).
- Если вы указываете, что класс реализует интерфейс при помощи **двоеточия** (:), класс должен реализовать **все компоненты интерфейса**; в противном случае код не будет компилироваться.
- Класс может **реализовать несколько интерфейсов** (и проблема ромбовидного наследования не возникнет, потому что интерфейсы не имеют реализации).
- Интерфейсы очень полезны, потому что они позволяют **несвязанным классам** выполнять **одну задачу**.
- Имена интерфейсов должны начинаться с буквы `I` верхнего регистра (это всего лишь соглашение, компилятор не следит за его выполнением.)
- Отношения реализации интерфейсов обозначаются на диаграммах классов **пунктирными стрелками**.
- Экземпляр интерфейса **невозможно создать ключевым словом** `new`, потому что его компоненты являются абстрактными.
- **Интерфейс может использоваться как тип** для ссылки на объект, который его реализует.
- Любой класс может **реализовать любой интерфейс** при условии, что он соблюдает обязательства по реализации методов и свойств этого интерфейса.
- Все, что входит в открытый интерфейс, **автоматически становится открытым**, потому что интерфейс используется для определения открытых методов и свойств любого класса, который его реализует.
- **Костыль** — уродливое, неуклюжее или незлегантное решение, которое создает трудности с сопровождением.
- **Ключевое слово** `is` возвращает `true`, если объект соответствует заданному типу. Также оно может использоваться для объявления переменной и присваивания ей по ссылке проверяемого объекта.
- Под **повышающим приведением типа** обычно понимается обычное присваивание или приведение типа с перемещением вверх по иерархии классов или присваивание переменной суперкласса ссылки на объект субкласса.
- Ключевое слово `is` позволяет выполнять **понижающее приведение типа** — безопасное перемещение вниз по иерархии классов — для использования переменной субкласса для обращения к объекту суперкласса.
- Повышающие и понижающие приведения типа также **работают с интерфейсами** — вы можете повысить ссылку на объект до ссылки на интерфейс или наоборот.
- Ключевое слово `as` похоже на приведение типа, кроме того что при недействительности приведения типа оно возвращает `null` (вместо выдачи исключения).
- Если поле помечается **ключевым словом** `readonly`, его значение может быть задано только в инициализаторе поля или в конструкторе.

Закрепляем в памяти



Запомните, как работают интерфейсы: вы расширяете класс, но реализуете интерфейс. Когда речь идет о «расширении», обычно имеется в виду, что вы берете нечто существующее и добавляете к нему что-то новое (в данном случае — поведение). Под реализацией имеется в виду воплощение в жизнь соглашения — вы обязуетесь добавить все компоненты интерфейса (и компилятор следит за выполнением этого соглашения).



Я думаю, что у интерфейсов есть **огромный недостаток**. Когда я пишу абстрактный класс, я могу включить в него код. Не означает ли это, что абстрактные классы лучше интерфейсов?

Вообще-то вы можете добавить код в интерфейсы. Для этого включите в них статические компоненты и реализации по умолчанию.

Возможности интерфейсов не сводятся к проверке того, что реализующие их классы включают некоторые компоненты. Конечно, это их основная задача. Но интерфейсы также могут содержать код, как и любые другие инструменты, используемые для создания модели классов.

Самый простой способ добавления кода в интерфейс основан на включении **статических методов, свойств и полей**. Они работают точно так же, как статические компоненты классов: в них могут храниться данные любого типа (включая ссылки на объекты) и их можно вызывать как любые другие статические методы: `Interface.MethodName()`;

Также можно включить код в интерфейсы, добавляя в методы реализации по умолчанию. Чтобы добавить реализацию по умолчанию, просто добавьте тело метода в интерфейс. Этот метод не является частью объекта (этот механизм отличен от наследования!), и обратиться к нему можно только по ссылке на интерфейс. Он может вызывать методы, реализуемые объектом, при условии, что они являются частью интерфейса.



Будьте
осторожны!

Реализации интерфейсов по умолчанию — относительно новая возможность C#.

Если вы работаете в старой версии Visual Studio, возможно, вы не сможете использовать реализации по умолчанию, потому что они появились только в версии C# 8.0, которая была опубликована в Visual Studio 2019 версии 16.3.0, выпущенной в сентябре 2019 года. Возможности текущей версии C# могут не поддерживаться в старых версиях Visual Studio.

Интерфейсы могут содержать статические компоненты

Один из известных клоунских трюков — множество клоунов, втиснувшихся в маленький клоунский автомобиль! Обновим интерфейс `IClown` и добавим в него статические методы, генерирующие описание клоунского автомобиля. Вот что для этого в него нужно добавить:

- ★ Мы будем использовать случайные числа, поэтому добавим статическую ссылку на экземпляр `Random`. Пока она применима только в `IClown`, но вскоре также будет использоваться в `IScaryClown`, поэтому ссылка будет помечена модификатором `protected`.
- ★ Клоунский автомобиль выглядит смешно только в том случае, если он битком набит клоунами, поэтому мы добавим статическое свойство `int` с приватным статическим резервным полем и `set`-методом, который принимает только значения больше 10.
- ★ Метод с именем `ClownCarDescription` возвращает строку с описанием клоунского автомобиля.

IClown
FunnyThingIHave static CarCapacity protected static Random
Honk static ClownCarDescription

Ниже приведен код — в нем используется статическое поле, свойство и метод, как в обычном классе:

```
interface IClown
{
    string FunnyThingIHave { get; }
    void Honk();

    protected static Random random = new Random();
    private static int carCapacity = 12;

    public static int CarCapacity {
        get { return carCapacity; }
        set {
            if (value > 10) carCapacity = value;
            else Console.Error.WriteLine($"Warning: Car capacity {value} is too small");
        }
    }

    public static string ClownCarDescription()
    {
        return $"A clown car with {random.Next(CarCapacity / 2, CarCapacity)} clowns";
    }
}
```

Добавьте!

Статическое поле `random` помечено модификатором доступа `protected`. Это означает, что к нему можно обращаться только из `IClown` или из любого интерфейса, расширяющего `IClown` (например, `IScaryClown`).

Теперь можно обновить метод `Main` для обращения к статическим компонентам `IClown`:

```
static void Main(string[] args)
{
    IClown.CarCapacity = 18;
    Console.WriteLine(IClown.ClownCarDescription());

    // Остальной код метода Main остается без изменений
}
```

Попробуйте добавить приватное поле в свой интерфейс. Вы сможете его добавить — но только если оно является статическим! Если убрать ключевое слово `static`, компилятор сообщит, что интерфейсы не содержат полей экземпляров.

Эти статические компоненты интерфейсов ведут себя точно так же, как статические компоненты классов, встречавшиеся в предыдущих главах. Открытые компоненты могут использоваться из любого класса, приватные компоненты могут использоваться только из `IClown`, а защищенные компоненты могут использоваться из `IClown` и любого интерфейса, который его расширяет.

Реализации по умолчанию определяют тело методов интерфейса

Все методы, которые встречались вам в интерфейсах до настоящего времени, кроме статических, были абстрактными: они не имели тела, поэтому любой класс, реализующий интерфейс, должен предоставить реализацию этого метода.

Но вы также можете предоставить **реализацию по умолчанию** для любых методов своего интерфейса. Пример:

```
interface IWorker {
    string Job { get; }
    void WorkTheNextShift();

    void Buzz() {
        Console.WriteLine("Buzz!");
    }
}
```

При желании вы даже можете добавить в интерфейс приватные методы, но они будут вызываться только из открытых реализаций по умолчанию.

Вы можете вызвать реализацию по умолчанию, но для вызова должна использоваться ссылка на интерфейс:

```
IWorker worker = new NectarCollector();
worker.Buzz();
```

Этот код не компилируется — вы получите сообщение об ошибке «*'NectarCollector' не содержит определение 'Buzz'*»:

```
NectarCollector pearl = new NectarCollector();
pearl.Buzz();
```

Дело в том, что если метод интерфейса имеет реализацию по умолчанию, он становится виртуальным методом (таким же, как методы, которые вы использовали в своих классах). Любой класс, реализующий интерфейс, имеет возможность реализовать метод. Виртуальный метод связывается с интерфейсом. Как и любая другая реализация интерфейса, он не наследуется. И это хорошо: если бы класс наследовал реализации по умолчанию от всех интерфейсов, которые он реализует, и в двух таких интерфейсах присутствовали методы с одинаковыми именами, в классе бы возникла проблема ромбовидного наследования.

Используйте @ для создания буквальных строковых литералов

Символ @ имеет специальный смысл в программах C#. Если поместить его в начало строкового литерала, этот символ сообщает компилятору C#, что литерал должен интерпретироваться буквально. В частности, это означает, что символы \ не будут интерпретироваться как служебные последовательности, таким образом, строка @"\n" содержит обратную косую черту и символ n, а не символ новой строки. Кроме того, @ сообщает компилятору C# о включении всех разрывов строк. Таким образом, строка @"Line 1

Line 2" эквивалентна "Line1\nLine2" (включая разрыв строки.)

Буквальные строковые литералы могут использоваться для создания «много-строчных» строк, включающих разрывы. Они нормально работают со строковой интерполяцией — просто добавьте \$ в начало.

Добавление метода ScareAdults с реализацией по умолчанию

Наш интерфейс IScaryClown идеально моделирует страшных клоунов. Но тут возникает проблема: он содержит только метод, который позволяет пугать детей. А если мы хотим, чтобы клоуны доводили до полного ужаса еще и взрослых?

Для этого *можно* было бы включить абстрактный метод ScareAdults в интерфейс IScaryClown. Но что, если у вас уже есть десяток классов, реализующих IScaryClown? И что, если большинству из них прекрасно подойдет одна реализация метода ScareAdults? Именно в таких ситуациях реализация по умолчанию оказывается по-настоящему полезной. Реализация по умолчанию позволяет добавить метод в уже используемый интерфейс **без необходимости обновления каких-либо классов, реализующих его**. Добавьте в IScaryClown метод ScareAdults с реализацией по умолчанию:

```
interface IScaryClown : IClown
{
    string ScaryThingIHave { get; }
    void ScareLittleChildren();

    void ScareAdults()
    {
        Console.WriteLine($"I am an ancient evil that will haunt your dreams.
Behold my terrifying necklace with {random.Next(4, 10)} of my last victim's fingers.
Oh, also, before I forget...");
        ScareLittleChildren();
    }
}
```

Здесь используется буквальный литерал. Также можно было использовать нормальный строковый литерал и добавить \n для разрывов строк. Такой синтаксис намного проще читается.

Добавьте!

Присмотритесь к тому, как работает метод ScareAdults. Он содержит только две команды, но в них упакована значительная функциональность. Проанализируем, что же здесь происходит:

- * Команда Console.WriteLine использует буквальный литерал со строковой интерполяцией. Литерал начинается с символов \$@, которые сообщают компилятору C# два факта: \$ приказывает использовать строковую интерполяцию, а @ — использовать формат буквального литерала. Это означает, что строка будет содержать три внутренних разрыва строк.
- * Литерал использует строковую интерполяцию для вызова random.Next(4, 10), который использует приватное статическое поле random, наследуемое IScaryClown от IClown.
- * Как упоминалось ранее, если класс содержит статическое поле, это означает, что существует только одна копия этого поля. Таким образом, существует только один экземпляр Random, который совместно используется как IClown, так и IScaryClown.
- * В последней строке метода ScareAdults вызывается метод ScareLittleChildren. Этот метод является абстрактным в интерфейсе IScaryClown, а следовательно, он будет вызывать версию ScareLittleChildren из класса, реализующего IScaryClown.
- * Это означает, что ScareAdults будет вызывать версию ScareLittleChildren, определенную в классе, реализующем IScaryClown.

Вызовите новую реализацию по умолчанию, изменив блок после команды if в методе Main так, чтобы в нем вызывался метод ScareAdults вместо ScareLittleChildren:

```
if (fingersTheClown is IScaryClown iScaryClownReference)
{
    iScaryClownReference.ScareAdults();
}
```



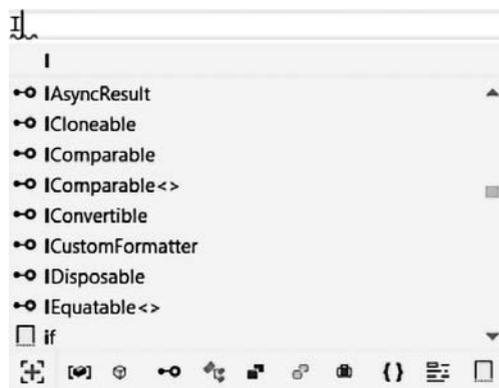
Мне все время кажется, что интерфейсы — это какая-то **теория**. Я понимаю, как они работают, в маленьких примерах книги. Но используются ли они разработчиками в реальных проектах?

Разработчики C# постоянно пользуются интерфейсами, особенно при работе с библиотеками, фреймворками и API.

Разработчики всегда «стоят на плечах гигантов». Вы уже перешли ко второй половине книги, и в первой половине вы писали код, который выводит текст на консоль, рисует окна с кнопками и строит трехмерные объекты. Вам не нужно было писать код для вывода конкретных байтов на консоль или рисования линий и текста для прорисовки кнопок в окне либо выполнять вычисления для вывода сферы — вы воспользовались кодом, написанным ранее другими людьми:

- ★ Вы пользовались **фреймворками** (такими, как .NET Core и WPF).
- ★ Вы пользовались **API** (например, API сценариев Unity).
- ★ Фреймворки и API содержат **библиотеки классов**, к которым вы обращаетесь при помощи директив `using` в начале кода.

При использовании библиотек, фреймворков и API вы часто используете интерфейсы. Убедитесь сами: откройте приложение .NET Core или WPF, щелкните внутри любого метода и введите `I`, чтобы вызвать окно IntelliSense. Любое потенциальное совпадение, помеченное значком , является интерфейсом. Всё это интерфейсы, которые могут использоваться для работы с фреймворком.



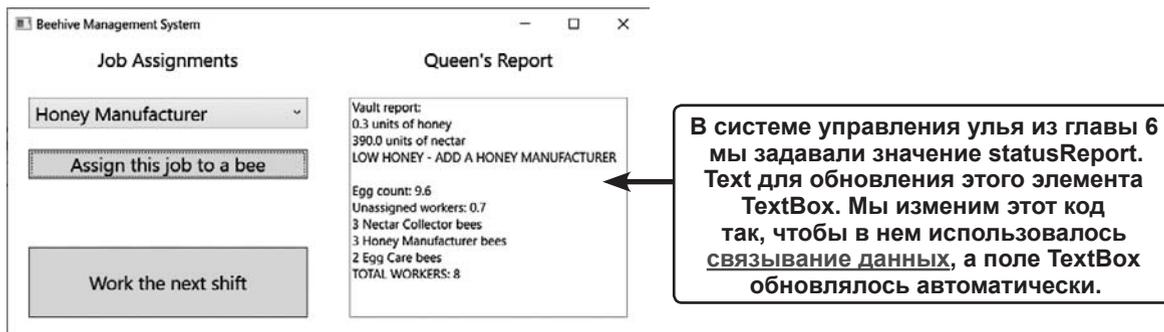
Небольшая часть интерфейсов, входящих в .NET Core.

У рассмотренной ниже функциональности WPF не существует эквивалента для Mac, поэтому в приложении «Visual Studio для пользователей Mac» этот раздел пропущен.

приведение типов интерфейсов и is

Связывание данных обеспечивает автоматическое обновление элементов WPF

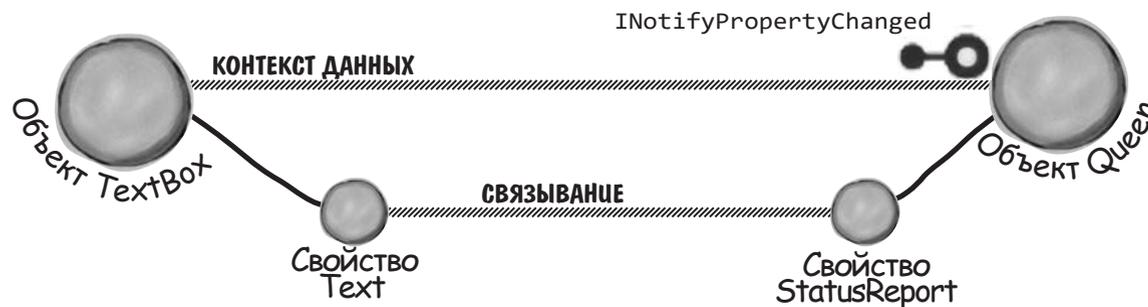
Рассмотрим отличный пример использования интерфейсов для реальных целей: **связывание данных**. Связывание данных — исключительно полезный механизм WPF, который позволяет настроить элементы управления так, что их свойства будут автоматически задаваться на основании значения свойства объекта. При изменении этого свойства происходит автоматическое изменение свойств элементов управления.



Ниже приведен краткий обзор действий по изменению системы управления ульем — вскоре они будут рассмотрены более подробно:

- 1 Измените класс `Queen` и реализуйте интерфейс `INotifyPropertyChanged`.**
Этот интерфейс позволяет `Queen` объявить, что отчет о текущем состоянии изменился.
- 2 Измените код XAML, чтобы в нем создавался экземпляр `Queen`.**
Мы свяжем свойство `TextBox.Text` со свойством `StatusReport` класса `Queen`.
- 3 Измените код программной части, чтобы в поле «queen» использовался только что созданный экземпляр `Queen`.**
В настоящий момент поле `queen` в файле `MainWindow.xaml.cs` использует инициализатор поля с командой `new` для создания экземпляра `Queen`. Мы изменим его так, чтобы вместо этого использовался экземпляр, созданный в XAML.

Связывание данных начинается с **контекста данных** — объекта, содержащего данные для отображения в `TextBox`. Мы будем использовать экземпляр `Queen` в качестве контекста данных.



Класс `Queen` должен сообщать `TextBox` об обновлении своего свойства `StatusReport`. Для этого мы обновим класс `Queen` так, чтобы он реализовал **интерфейс `INotifyPropertyChanged`**.

Связывание данных в системе управления ульем

Чтобы добавить связывание данных в приложение WPF, необходимо внести ряд изменений.

Сделайте это!

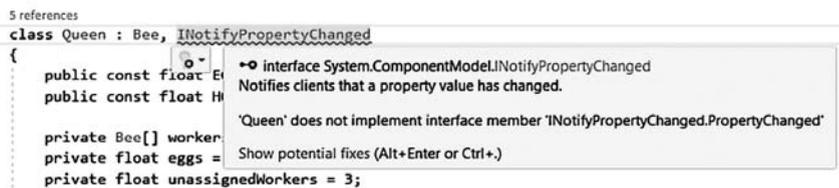
1

Измените класс Queen, чтобы он реализовал интерфейс INotifyPropertyChanged.

Обновите объявление класса Queen, чтобы он реализовал интерфейс INotifyPropertyChanged. Этот интерфейс принадлежит пространству имен System.ComponentModel, поэтому в начало класса следует добавить директиву using:

```
using System.ComponentModel;
```

Теперь можно добавить INotifyPropertyChanged в конец объявления класса. IDE подчеркивает объявление красной волнистой линией, чего и следовало ожидать, так как вы еще не реализовали интерфейс и не добавили его компоненты.



Нажмите Alt+Enter или Ctrl+, чтобы вывести потенциальные исправления, и выберите в контекстном меню команду «Implement Interface». IDE добавляет в класс строку кода с ключевым словом event, которое вам еще не встречалось:

```
public event PropertyChangedEventHandler PropertyChanged;
```

И знаете что? Вы уже пользовались событиями! У класса Dispatcher, использованного в главе 1, было событие Tick, а элементы кнопок WPF использовали событие Click. *Теперь класс Queen содержит событие PropertyChanged.* Любой класс, который используется для связывания данных, инициирует (выдает) событие PropertyChanged, чтобы сообщить WPF об изменении свойства.

Класс Queen должен инициировать свое событие по аналогии с тем, как Dispatcher инициирует свое событие Tick с заданным интервалом, а кнопка Button инициирует свое событие Click, когда пользователь щелкает на ней. **Добавьте метод OnPropertyChanged:**

```
protected void OnPropertyChanged(string name)
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
}
```

Теперь необходимо **изменить метод UpdateStatusReport** для вызова OnPropertyChanged:

```
private void UpdateStatusReport()
{
    StatusReport = $"Vault report:\n{HoneyVault.StatusReport}\n" +
        $"Egg count: {eggs:0.0}\nUnassigned workers: {unassignedWorkers:0.0}\n" +
        $"{WorkerStatus("Nectar Collector")}\n{WorkerStatus("Honey Manufacturer")}" +
        $"Egg Care")\nTOTAL WORKERS: {workers.Length}";
    OnPropertyChanged("StatusReport");
}
```

Мы добавили событие в класс Queen, а также метод, использующий оператор ?. для инициирования событий. Вот и все, что необходимо знать о событиях, — в конце книги приводится ссылка на главы, в которой события рассматриваются более подробно.

2

Измените код XAML, чтобы создать экземпляр Queen.

Ранее вы создавали объекты ключевым словом `new`, а также пользовались методом `Unity Instantiate`. XAML предоставляет еще один способ создания новых экземпляров классов. **Добавьте следующий фрагмент в XAML непосредственно перед** тегом `<Grid>`:

```
<Window.Resources>
  <local:Queen x:Key="queen"/>
</Window.Resources>
```

Этот тег создает новый экземпляр `Queen` и добавляет его в ресурсы вашего окна (механизм, применяемый окнами WPF для хранения ссылок на объекты, используемые элементами управления).

Затем **измените тег** `<Grid>` и добавьте к нему атрибут `DataContext`:

```
<Grid DataContext="{StaticResource queen}">
```

Наконец, **добавьте атрибут** `Text` к тегу `<TextBox>`, чтобы связать его со свойством `StatusReport` класса `Queen`:

```
<TextBox Text="{Binding StatusReport, Mode=OneWay}">
```

Теперь элемент `TextBox` будет автоматически обновляться каждый раз, когда объект `Queen` будет инициировать свое событие `PropertyChanged`.

3

Измените код программной части, чтобы использовать экземпляр Queen в ресурсах окна.

Пока поле `queen` в файле `MainWindow.xaml.cs` содержит инициализатор поля с командой `new` для создания экземпляра `Queen`. Мы изменим его, чтобы вместо этого использовался экземпляр, созданный в XAML.

Для начала закомментируйте (или удалите) три вхождения строки, присваивающей `statusReport.Text`. Одна строка находится в конструкторе `MainWindow`, а две другие — в обработчиках событий `Click`:

```
// statusReport.Text = queen.StatusReport;
```

Затем измените объявление поля `Queen` и удалите инициализатор поля (`new Queen();`) в конце: `private readonly Queen queen;`

Наконец, измените конструктор, чтобы он задавал значение поля `queen` следующим образом:

```
public MainWindow()
{
  InitializeComponent();
  queen = Resources["queen"] as Queen;
  //statusReport.Text = queen.StatusReport;
  timer.Tick += Timer_Tick;
  timer.Interval = TimeSpan.FromSeconds(1.5);
  timer.Start();
}
```

Теперь в приложении WPF используется связывание данных, и нам не нужно применять свойство `Text` для обновления элемента `TextBox` с отчетом. Закомментируйте или удалите эту строку.

В коде используется **словарь** с именем `Resources`. (Мы немного *забегаем вперед* — вы узнаете о словарях в следующей главе.) Запустите игру. Она ведет себя точно так же, как и прежде, но теперь `TextBox` обновляется автоматически каждый раз, когда `Queen` обновляет отчет о текущем состоянии.

Поздравляем! Вы только что использовали интерфейс для добавления связывания данных в приложение WPF.

Часто Задаваемые Вопросы

В: Кажется, я понимаю все, что было сделано. А вы можете кратко пройтись по изменениям на случай, если я что-то упустил?

О: Безусловно. Приложение системы управления ульем, построенное в главе 6, обновляет элемент TextBox (statusReport), задавая его свойство Text в коде:

```
statusReport.Text = queen.StatusReport;
```

Мы изменили приложение, чтобы в нем использовалось связывание данных для автоматического обновления TextBox каждый раз, когда объект Queen обновляет свое свойство StatusReport. Для этого были внесены три изменения. Прежде всего, класс Queen реализовал интерфейс INotifyPropertyChanged, чтобы уведомлять пользовательский интерфейс о любых изменениях свойства. Затем в коде XAML был создан экземпляр Queen, а свойство TextBox.Text связано со свойством StatusReport объекта Queen. Наконец, мы изменили код программной части, чтобы использовать экземпляр, созданный в XAML, и удалили строки, в которых задавалось значение statusReport.Text.

В: Какую именно задачу решает этот интерфейс?

О: Интерфейс INotifyPropertyChanged предоставляет возможность сообщить WPF о том, что свойство изменилось, чтобы приложение могло обновить связанные с этим свойством элементы. Реализуя этот интерфейс, вы строите класс, который может решать конкретную задачу: оповещение приложений WPF об изменениях свойств. Интерфейс состоит из одного компонента — события с именем PropertyChanged. Когда класс используется для связывания данных, WPF проверяет, реализует ли он интерфейс INotifyPropertyChanged, и если реализует — присоединяет обработчик события к событию PropertyChanged вашего класса (по аналогии с тем, как вы связывали обработчики событий с обработчиками Click элементов Button).

В: Я заметил, что при открытии окна в визуальном конструкторе элемент TextBox с отчетом о текущем состоянии уже не пуст. Почему — из-за связывания данных?

О: Вот это наблюдательность! Да, когда вы изменили XAML и добавили секцию <Window.Resources> для создания нового экземпляра объекта Queen, конструктор XAML в Visual Studio создал экземпляр объекта. Когда вы изменили Grid, добавили контекст данных и создали связывание со свойством Text элемента TextBox, визуальный конструктор использовал эту информацию для вывода текста. Следовательно, когда вы используете связывание данных, экземпляры ваших классов создаются не только при выполнении программы. Visual Studio создает экземпляры объектов **при редактировании окна XAML**. Это чрезвычайно мощная возможность IDE, потому что она позволяет вам изменить свойства в коде и увидеть результаты в конструкторе сразу же при построении кода.



Будьте
осторожны!

Связывание данных работает со свойствами, но не с полями.

Связывание данных может использоваться только с открытыми свойствами. Если вы попытаетесь связать атрибут элемента WPF с открытым полем, в программе ничего не изменится, — впрочем, исключения вы тоже не получите.

«Полиморфизм» означает, что один объект может существовать в разных формах

Каждый раз, когда вы используете RoboBee вместо IWorker, или Wolf вместо Animal, или даже выдержанный вермонтский чеддер в рецепте, в котором требуется любой сыр, вы используете **полиморфизм**. И это же происходит при каждом повышающем или понижающем приведении типа. Вы берете объект и используете его в методе или в команде, которые ожидают получить что-то другое, — это называется полиморфизмом.

Полиморфизм вокруг нас

Вы уже давно пользуетесь полиморфизмом — просто мы не использовали этот термин. Когда вы будете писать код для оставшихся глав, обращайтесь внимание на разнообразные возможности его использования.

Ниже перечислены четыре типичных сценария использования полиморфизма. Мы приведем пример для каждого случая, хотя вы не увидите эти конкретные строки в упражнениях. Когда вам попадется похожий код в упражнениях последних глав книги, вернитесь к этой странице и сверьтесь со списком:

- Ссылочной переменной, использующей один класс, присваивается экземпляр другого класса.

```
NectarStinger berth = new NectarStinger();
INectarCollector gatherer = berth;
```

- Повышающее приведение с использованием subclasses в команде или в методе, рассчитанными на его базовый класс.

```
spot = new Dog();
zooKeeper.FeedAnimal(spot);
```

← Если FeedAnimal ожидает получить объект Animal, а Dog наследует от Animal, вы можете передать Dog FeedAnimal.

- Создание ссылочной переменной с типом интерфейса и присваивание ей объекта, реализующего этот интерфейс.

```
IStingPatrol defender = new StingPatrol();
```

← Это тоже повышающее приведение типа!

- Понижающее приведение типа с ключевым словом is.

```
void MaintainTheHive(IWorker worker) {
    if (worker is HiveMaintainer) {
        HiveMaintainer maintainer = worker as HiveMaintainer;
        ...
    }
}
```

← Метод MaintainTheHive получает любую реализацию IWorker в параметре. Он использует ключевое слово «as» для того, чтобы ссылка на HiveMaintainer указывала на worker.

Вы используете полиморфизм каждый раз, когда берете экземпляр одного класса и используете его в команде или в методе, рассчитанных на другой тип, например родительский класс или интерфейс, реализуемый классом.



Кажется, я начинаю понимать, как работать с объектами!

Идея объединения данных казалась революционной в тот момент, когда она была впервые предложена. Впрочем, мы строили все программы C# именно так, и вы можете относиться к ней как к обычному программированию.

Вы занимаетесь объектно-ориентированным программированием.

Программирование, которым вы занимаетесь, называется **объектно-ориентированным (ООП)**. До широкого распространения таких языков, как C#, программисты не использовали объекты и методы при написании своего кода. Они просто вызывали функции (аналоги методов в неориентированных программах), которые хранились в одном месте, словно каждая программа была одним большим статическим классом, состоявшим только из статических методов. Такой подход серьезно усложнял написание программ, моделировавших решаемые задачи. К счастью, вам уже не придется писать программы без использования ООП, являющегося ключевой частью C#.

Четыре основных принципа объектно-ориентированного программирования

Когда программисты говорят об ООП, они обычно имеют в виду четыре важных принципа. Каждый из этих принципов должен быть вам хорошо знаком, потому что вы использовали каждый из них в своих программах. Мы уже упоминали о полиморфизме, а первые три принципа знакомы вам по главам 5 и 6: **наследование**, **абстракция** и **инкапсуляция**.

Означает создание объекта, который хранит внутреннюю информацию о своем состоянии в частных полях. При этом открытые свойства и методы используются для того, чтобы другие классы могли работать только с той частью внутренних данных, которую им разрешено видеть.

Означает, что класс или интерфейс наследует от другого класса или интерфейса.

*** Наследование**

*** Инкапсуляция**

*** Абстракция**

*** Полиморфизм**

Абстракция используется при создании модели классов, которая начинается с самых общих (или абстрактных) классов, а затем переходит к более конкретным классам, наследующим от них.

Термин «полиморфизм» буквально означает «много форм». Можете ли вы представить ситуацию, в которой объект принимает несколько разных форм в вашем коде?

8 Перечисления и Коллекции

Организация данных

... И в этой сцене массовка выстраивается
в очередь по росту.
Всем занять свои места!

Эй!
Почему никто не строится?
Шевелитесь, время — деньги. Э-э-э...
Кто-нибудь меня слышит?

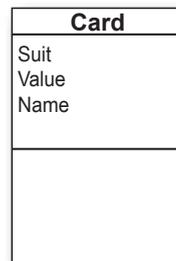
Данные не всегда бывают такими аккуратными и ухоженными, как нам хотелось бы. В реальном мире данные, как правило, не хранятся маленькими аккуратными кусочками. Нет, данные поступают **вагонами, штабелями и кучами**. Для их систематизации нужны мощные инструменты, и тут вам на помощь приходят **перечисления и коллекции**. Перечисления — типы, позволяющие определять значения для классификации ваших данных. Коллекции — специальные объекты, способные **хранить и сортировать** данные, которые обрабатывает программа, и **управлять** ими. В результате вы можете сосредоточиться на основной идее программирования, оставив задачу управления данных коллекциям.

Строки не всегда подходят для хранения категорий данных

В нескольких следующих главах мы будем работать с игральными картами; давайте создадим для них класс Card. Для начала создайте новый класс Card с конструктором, который получает масть и номинал карты и сохраняет их в строковом виде:

```
class Card
{
    public string Value { get; set; }
    public string Suit { get; set; }
    public string Name { get { return $"{Value} of {Suit}"; } }

    public Card(string value, string suit)
    {
        Value = value;
        Suit = suit;
    }
}
```



Класс Card использует строковые свойства для хранения мастей и номиналов.

Пока выглядит неплохо. Вы можете создать объект Card и использовать его следующим образом:

```
Card aceOfSpades = new Card("Ace", "Spades");
Console.WriteLine(aceOfSpades); // Выводит Ace of Spades
```

Но тут возникает проблема. Использование строк для хранения мастей и номиналов может иметь непредвиденные значения:

```
Card duchessOfRoses = new Card("Duchess", "Roses");
Card fourteenOfBats = new Card("Fourteen", "Bats");
Card dukeOfOxen = new Card("Duke", "Oxen");
```

Этот код компилируется, но такие «масти» и «номиналы» совершенно бессмысленны. Класс Card не должен допускать такие типы, как действительные данные.

В принципе, *можно* было бы добавить в конструктор код, который проверяет каждую строку и убеждается в том, что она представляет действительную масть или номинал и выдает исключение при некорректных входных данных. Такое решение вполне допустимо — конечно, если вы организуете корректную обработку исключений.

Но *разве не было бы удобнее*, если бы компилятор C# мог автоматически обнаруживать недействительные значения? Если бы компилятор мог гарантировать, что все масти и номиналы действительны, еще *до выполнения кода*? Представьте, такая возможность *существует*! Все, что для этого нужно, — использовать **перечисление** для допустимых значений.

перечислять, глагол.
указывать элементы в определенном порядке.



Карта «Герцог быков». В природе не встречается.

Перечисления предназначены для работы с наборами допустимых значений

Перечисление (или **перечисляемый тип**) — тип данных, допускающий ограниченный набор допустимых значений для некоторых данных. Таким образом, мы можем определить для карточных мастей перечисление с именем `Suits` и следующие допустимые масти:

```
enum Suits {
    Diamonds,
    Clubs,
    Hearts,
    Spades,
}
```

← Перечисление начинается с ключевого слова `enum`, за которым следует имя. Это перечисление называется `Suits`.

← Далее в перечислении указываются элементы, разделенные запятыми и заключенные в фигурные скобки. Для каждого уникального значения — в данном случае для каждой масти — определяется один элемент.

← Вообще говоря, запятая за последним элементом перечисления необязательна, но она упрощает возможные перемещения элементов методом копирования/вставки.

Перечисление определяет новый тип

Используя ключевое слово `enum`, вы **определяете новый тип**. Несколько полезных фактов, которые необходимо знать о перечислениях:

- ✓ Перечисление может использоваться как тип в определении переменной — такой же, как строка, `int` или любой другой тип:


```
Suits mySuit = Suits.Diamonds;
```
- ✓ Так как перечисление является типом, оно может использоваться для создания массива:


```
Suits[] myVals= new Suits[3] { Suits.Spades, Suits.Clubs, mySuit };
```
- ✓ Для сравнения значений из перечислений может использоваться оператор `==`. Следующий метод получает перечисление `Suit` в параметре и использует `==` для проверки его на равенство с `Suits.Hearts`:


```
void IsItAHeart(Suits suit) {
    if (suit == Suits.Hearts) {
        Console.WriteLine("You pulled a heart!");
    } else {
        Console.WriteLine($"You didn't pull a heart: {suit}");
    }
}
```

Метод `ToString` перечисления возвращает эквивалентную строку, т. е. `Suits.Spades.ToString` возвращает `"Spades"`.
- ✓ В перечисление нельзя добавить новое значение. Если вы попытаетесь это сделать, программа не будет компилироваться — это позволит избежать некоторых неприятных ошибок:

```
IsItAHeart(Suits.Oxen);
```

При попытке использовать значение, не входящее в перечисление, компилятор выдаст ошибку:

✗ CS0117 'Suits' does not contain a definition for 'Oxen'

Перечисление определяет новый тип, который может содержать значения из фиксированного набора. Любое значение, которое не входит в перечисление, нарушит работоспособность кода, что может предотвратить возможные ошибки в будущем.

Перечисления позволяют представлять числа именами

Иногда бывает удобнее работать с числами, если присвоить им имена. Вы можете связать числа со значениями в перечислении и использовать имена для обращения к ним. В этом случае вам не придется иметь дело с загадочными числами, неожиданно появляющимися в вашем коде. Следующее перечисление позволяет отслеживать оценки за выполнение различных команд в конкурсе для собак:

```
enum TrickScore {
```

```
    Sit = 7,
    Beg = 25,
    RollOver = 50,
    Fetch = 10,
    ComeHere = 5,
    Speak = 30,
}
```

Элементы не обязательно указывать в каком-то определенном порядке, и с одним числом может быть связано несколько имен.

Укажите имя, за ним «=», а потом число, которое представляется этим именем.

int можно преобразовать в перечисление, и перечисление (на базе int) можно преобразовать в int.

Некоторые перечисления используют другие типы вместо int, например byte или long (см. ниже). Их можно преобразовать к их базовому типу вместо int.

Фрагмент метода, который использует перечисление TrickScore преобразованием его к int и обратно:

```
int score = (int)TrickScore.Fetch * 3;
// Следующая строка выводит: The score is 30
Console.WriteLine($"The score is {score}");
```

Перечисление можно преобразовать в число и выполнить с ним вычисления. Его даже можно преобразовать в строку — метод ToString перечисления возвращает строку с именем элемента:

```
TrickScore whichTrick = (TrickScore)7;
// The next line prints: Sit
Console.WriteLine(whichTrick.ToString());
```

Если какое-то имя не связано с числом, элементам списка присваиваются значения по умолчанию. Первому элементу будет присвоено значение 0, второму — 1, и т. д. Но что произойдет, если одному из перечислений потребуется присвоить очень большое число? По умолчанию для типов перечислений используется тип int, поэтому нужный тип необходимо задать оператором (:), как в следующем примере:

```
enum LongTrickScore : long {
    Sit = 7,
    Beg = 2500000000025
}
```

Число слишком велико для типа int

Сообщает компилятору, что значения в TrickScore должны интерпретироваться с типом long, а не int.

Приведение типа (int) приказывает компилятору преобразовать имя в число, которое оно представляет. Таким образом, поскольку TrickScore.Fetch имеет значение 10, (int) TrickScore.Fetch преобразует его в значение 10 типа int.

int можно преобразовать обратно в TrickScore, а TrickScore.Sit представляет значение 7.

Console.WriteLine вызывает метод ToString перечисления; этот метод возвращает строку с именем элемента.

Если вы попытаетесь использовать это перечисление без указания типа long, произойдет ошибка:

CS0266 Cannot implicitly convert type 'long' to 'int'.



Упражнение

Используйте то, что вы узнали о перечислениях, для построения класса, представляющего игральную карту. Создайте новый проект консольного приложения **.NET Core Console App** и добавьте в него класс с именем **Card**.

Card
Value
Suit
Name

Добавьте в **Card** два открытых свойства: **Suit** (допустимые значения **Spades**, **Clubs**, **Diamonds** и **Hearts**) и **Value** (**Ace**, **Two**, **Three**... **Ten**, **Jack**, **Queen**, **King**). Также понадобится еще одно свойство: открытое свойство, доступное только для чтения, с именем **Name**, которое возвращает строку вида «**Ace of Spades**» или «**Five of Diamonds**».

Добавьте два перечисления для определения мастей и номиналов в отдельных файлах *.cs

Добавьте перечисления. В **Windows** используйте знакомую функцию **Add>>Class**, а затем **замените class на enum** в каждом файле. В **macOS** используйте команду **Add>>New File...** и выберите **Empty Enumeration**. Скопируйте **перечисление Suits**, которое мы привели, затем создайте перечисление для номиналов. Проследите за тем, чтобы номиналы соответствовали картам: значение **(int)Values.Ace** всегда должно быть равно **1**, **Two** — **2**, **Three** — **3** и т. д. Номинал **Jack** должен соответствовать **11**, **Queen** — **12** и **King** — **13**.

Добавьте конструктор и свойство Name, возвращающее строку с именем карты

Добавьте конструктор, который получает два параметра, **Suit** и **Value**:

```
Card myCard = new Card(Values.Ace, Suits.Spades);
```

Свойство **Name** должно быть доступно только для чтения. **Get-метод** должен возвращать строку с описанием карты. Таким образом, код:

```
Console.WriteLine(myCard.Name);
```

должен выводить следующий результат:

```
Ace of Spades
```

Чтобы добавить перечисление в Visual Studio для Mac, добавьте файл и выберите тип файла «Empty Enumeration».

Выведите имя случайной карты в методе Main

Чтобы ваша программа сгенерировала карту со случайной мастью и номиналом, преобразуйте случайное число от **0** до **3** для перечисления **Suits** и еще одно случайное число от **1** до **13** для перечисления **Values**. Для этого можно воспользоваться встроенным классом **Random**, который предоставляет три разных способа вызова следующего метода **Next**:

Если один метод может быть вызван несколькими способами, это называется **перегрузкой (overloading)**.

```
Random random = new Random();
int numberBetween0and3 = random.Next(4);
int numberBetween1and13 = random.Next(1, 14);
int anyRandomInteger = random.Next();
```

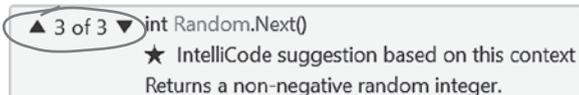
Вы делали это в главе 3. Этот вызов приказывает **Random** вернуть значение в диапазоне от **1** до **14**.

Часто задаваемые вопросы

В: Я помню, что ранее в книге метод **Random.Next** вызывался с двумя аргументами. Я заметил, что при вызове метода появляется окно **IntelliSense**, в углу которого написано «**3 из 3**». Это как-то связано с перегрузкой?

О: Да! Когда класс содержит переопределенный метод — или метод, который может вызываться несколькими способами, — IDE сообщает вам обо всех имеющихся возможностях. В данном случае класс **Random** содержит три возможные реализации метода **Next**. Как только вы вводите в окне кода **random.Next()**, IDE открывает окно **IntelliSense** с параметрами разных переопределенных методов.

Стрелки **▲** и **▼** рядом с надписью «**3 из 3**» позволяют просматривать варианты. Это особенно полезно при работе с методами, имеющими десятки перегруженных определений. Таким образом, когда вы вызываете **Random.Next**, убедитесь в том, что вы выбираете правильный перегруженный метод. Но пока не стоит отвлекаться на это — перегрузка гораздо подробнее рассматривается позднее в этой главе.





Упражнение
Решение

Колода карт — отличный пример программы, в которой очень важно ограничивать возможные значения. Никто не захочет взять свои карты и увидеть среди них 28 червей или джокер треф. Ниже приведена наша реализация класса Card — она будет несколько раз использована в следующих главах.

Перечисление Suits определяется в файле с именем Suits.cs. Код перечисления у вас уже есть — он идентичен перечислению Suits, которое было приведено ранее в этой главе. Перечисление Values хранится в файле с именем Values.cs:

```
enum Values {
    Ace = 1,
    Two = 2,
    Three = 3,
    Four = 4,
    Five = 5,
    Six = 6,
    Seven = 7,
    Eight = 8,
    Nine = 9,
    Ten = 10,
    Jack = 11,
    Queen = 12,
    King = 13,
}
```

Здесь Value. Ace присваивается значение 1.

A Values.King присваивается значение 13.

Мы выбрали для перечислений имена Suits и Values, тогда как свойства класса Card, используемые этими перечислениями для типов, называются Suit и Value. Что вы скажете об этих именах? Найдите имена других перечислений, которые встречаются в книге. Не лучше ли было бы присвоить им имена Suit и Value?

Правильного или неправильного ответа не существует — собственно, в справочнике Microsoft по языку C# присутствуют формы как одиночного (например, Season), так и множественного числа (например, Days):

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum>.

Класс Card имеет конструктор, задающий свойства Suit и Value, и свойство Name, генерирующее строку с описанием карты:

```
class Card {
    public Values Value { get; private set; }
    public Suits Suit { get; private set; }

    public Card(Values value, Suits suit) {
        this.Suit = suit;
        this.Value = value;
    }

    public string Name {
        get { return $"{Value} of {Suit}"; }
    }
}
```

Пример инкапсуляции. Мы объявили set-методы для свойств Value и Suit приватными, потому что они должны вызываться только из конструктора. Таким образом предотвращается их случайное изменение.

Get-метод свойства Name использует тот факт, что метод ToString перечисления возвращает свое имя, преобразованное в строку.

В классе Program используется статическая ссылка Random с приведением типов Suits и Values для создания случайной карты Card:

```
class Program
{
    private static readonly Random random = new Random();

    static void Main(string[] args)
    {
        Card card = new Card((Values)random.Next(1, 14), (Suits)random.Next(4));
        Console.WriteLine(card.Name);
    }
}
```

Перегруженный метод Random.Next используется для генерирования случайного числа в диапазоне от 1 до 13. Результат преобразуется в значение Values.

Для создания колоды карт можно воспользоваться массивом...

А если вы хотите создать класс, представляющий колоду карт? Для этого необходимо отслеживать каждую карту в колоде, а также знать, в каком порядке следуют карты. В принципе, для этого можно было бы воспользоваться массивом Cards — верхняя карта колоды хранится в элементе с индексом 0, следующая — в элементе с индексом 1, и т. д. Следующий класс мог бы стать отправной точкой — объект Deck, который начинается с полной колоды из 52 карт:

```
class Deck
{
    private readonly Card[] cards = new Card[52];

    public Deck() {
        int index = 0;
        for (int suit = 0; suit <= 3; suit++)
        {
            for (int value = 1; value <= 13; value++)
            {
                cards[index++] = new Card((Values)value, (Suits)suit);
            }
        }
    }

    public void PrintCards()
    {
        for (int i = 0; i < cards.Length; i++)
            Console.WriteLine(cards[i].Name);
    }
}
```

Мы использовали два цикла «for» для перебора всех возможных комбинаций масти и номинала.

...но что, если массива окажется недостаточно?

Представьте все, что можно сделать с колодой карт. Если вы играете в карточную игру, вам постоянно придется изменять порядок карт, добавлять и удалять карты из колоды. С обычным массивом делать это не так просто. Например, возьмем метод AddWorker из системы управления ульем в главе 6:

```
private void AddWorker(Bee worker) {
    if (unassignedWorkers >= 1) {
        unassignedWorkers--;
        Array.Resize(ref workers, workers.Length + 1);
        workers[workers.Length - 1] = worker;
    }
}
```

Этот код использовался для добавления элемента в массив в главе 6. А что будет, если вам потребуется добавить ссылку на Bee в середину, а не в конец массива?

Приходилось вызывать Array.Resize для расширения массива, а потом добавлять нового рабочего в конец. Слишком много лишней работы.

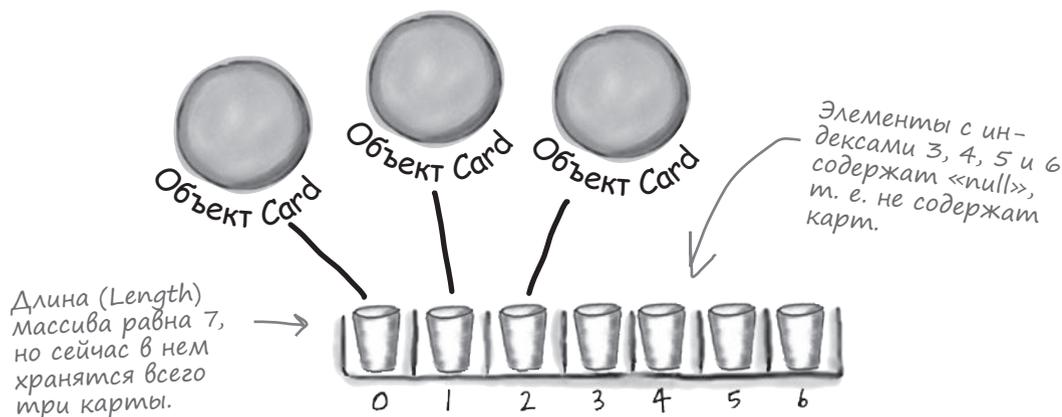


А как вы реализуете метод Shuffle, который переставляет карты в случайном порядке? Как насчет метода для сдачи первой карты, который возвращает карту и удаляет ее из колоды? Как реализовать добавление карты в колоду?

С массивами бывает неудобно работать

Массив хорошо подходит для фиксированного набора значений или ссылок. Но если вам потребуется перемещать элементы или добавлять новые элементы, не помещающиеся в массиве, начинаются проблемы. Рассмотрим несколько ситуаций, в которых работа с массивами может оказаться неудобной.

У каждого массива имеется длина. Она не изменяется при изменении размера массива, а значит, вы должны знать длину массива для работы с ним. Допустим, вы хотите использовать массив для хранения ссылок на Card. Если количество ссылок, которые необходимо хранить, меньше длины массива, можно воспользоваться null-ссылками для представления пустых элементов.



Необходимо отслеживать количество карт, хранимых в массиве. Можно добавить поле `int` (допустим, с именем `cardCount`), в котором будет храниться индекс последней карты в массиве. Таким образом, массив с тремя картами будет иметь длину (Length) 7, но поле `cardCount` будет содержать 3.



Ситуация усложняется. Вы можете легко добавить метод `Peek`, который возвращает ссылку на верхнюю карту, и вы можете узнать карту сверху колоды. А если потребуется добавить карту? Если `cardCount` меньше длины массива, можно просто поместить карту в массив с этим индексом и увеличить `cardCount` на единицу. Но если массив заполнен, придется создать новый массив большего размера и скопировать в него существующие карты. Удалить карту несложно — но после того, как `cardCount` уменьшится на единицу, необходимо позаботиться о том, чтобы по индексу удаленной карты хранилось значение `null`. А если потребуется удалить карту **в середине списка**? Если вы удаляете карту 4, необходимо сдвинуть карту 5 на открывшееся место, затем карту 6, карту 7... Нет, только не это!

В методе `AddWorker` из главы 6 для этой цели использовался метод `Array.Resize`.

В списках можно хранить коллекции... чего угодно

В C# и .NET существует множество классов **коллекций**, позволяющих легко решить неприятные вопросы с добавлением и удалением элементов массива. Чаще всего используется коллекция `List<T>`. Создав объект `List<T>`, вы сможете легко добавлять и удалять элементы из произвольной позиции списка, получать значение отдельного элемента и даже перемещать элемент из одной позиции в другую. Посмотрим, как работают списки.

Иногда мы будем опускать `<T>` в упоминаниях `List`. Когда вы увидите в тексте `List`, считайте, что это `List<T>`.

- 1 Начните с создания нового экземпляра `List<T>`.** Вспомните, что у каждого массива есть тип: вы работаете не просто с массивом, а с массивом `int`, массивом `Card` и т. д. Со списками дело обстоит аналогично. Вы должны указать тип объектов или значений, которые должны храниться в списке, в угловых скобках (`<>`) при создании списка ключевым словом `new`:

```
List<Card> cards = new List<Card>();
```



При создании списка был указан тип `<Card>`, поэтому в списке могут храниться только ссылки на объекты `Card`.

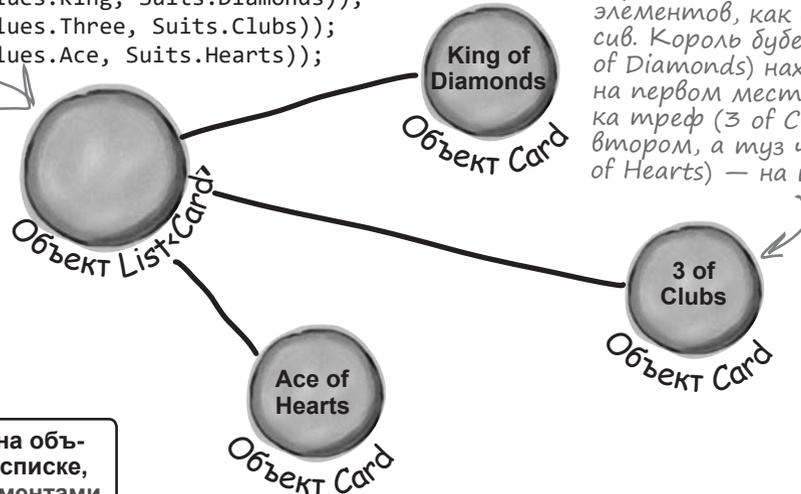


`<T>` в конце `List<T>` означает, что тип является обобщением. `T` заменяется конкретным типом — таким образом, `List<int>` попросту означает `List` с элементами `int`. На нескольких ближайших страницах у вас будет возможность потренироваться в использовании обобщений.

- 2 Теперь можно перейти к добавлению элементов в `List<T>`.** После того как у вас появится объект `List<T>`, вы можете добавить в него сколько угодно элементов — при условии, что они полиморфны по типу, заданному при создании `List<T>`, иначе говоря, что они совместимы с типом по присваиванию (к этой категории относятся интерфейсы, абстрактные и базовые классы).

```
cards.Add(new Card(Values.King, Suits.Diamonds));
cards.Add(new Card(Values.Three, Suits.Clubs));
cards.Add(new Card(Values.Ace, Suits.Hearts));
```

В `List` можно добавить сколько угодно объектов `Card` — для этого достаточно вызвать метод `Add`. Объект `List` сам позаботится о том, чтобы для элементов было достаточно «ячеек». Если свободное место в `List` будет исчерпано, объект автоматически расширится.



Список поддерживает определенный порядок своих элементов, как и массив. Король будет (King of Diamonds) находится на первом месте, тройка треф (3 of Clubs) — на втором, а туз червей (Ace of Hearts) — на третьем.

Значения или ссылки на объекты, содержащиеся в списке, обычно называются элементами.

Списки обладают большей гибкостью, чем массивы

Класс List встроен в .NET Framework. Он позволяет делать с объектами много такого, что было невозможно с традиционными массивами. Перечислим некоторые возможности List<T>:

`new List<egg>()`; создает список List с объектами Egg. В исходном состоянии список пуст. Вы можете добавлять и удалять из него объекты, но так как список предназначен для объектов Egg, добавлять можно только ссылки на объекты Egg или любые объекты, которые могут быть преобразованы в Egg.

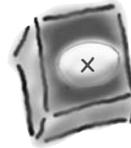
- 1 **Использование ключевого слова new для создания экземпляра List (как и следовало ожидать):**

```
List<Egg> myCarton = new List<Egg>();
```

Ссылка на объект Egg.

- 2 **Добавление элементов в список.**

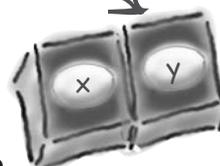
```
Egg x = new Egg();  
myCarton.Add(x);
```



Список расширяется для хранения объекта Egg...

- 3 **Добавление других элементов в список.**

```
Egg y = new Egg();  
myCarton.Add(y);
```



Другой объект Egg.

...и снова расширяется для хранения второго объекта Egg.

- 4 **Определение количества элементов в списке.**

```
int theSize = myCarton.Count;
```

- 5 **Проверка наличия конкретного объекта в List.**

```
bool isIn = myCarton.Contains(x);
```

Теперь вы можете провести поиск конкретного объекта Egg в List. Разумеется, поиск вернет true, потому что вы только что добавили этот объект Egg в List.

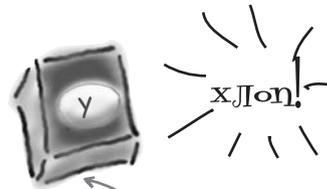
- 6 **Определение позиции этого объекта в List.**

```
int index = myCarton.IndexOf(x);
```

Индекс x будет равен 0, а индекс y будет равен 1.

- 6 **Удаление объекта из списка.**

```
myCarton.Remove(x);
```



При удалении x в списке остается только y, размер списка уменьшился! Если удалить y, список вскоре будет уничтожен сборщиком мусора.

Возьми в руку карандаш



Несколько строк из середины программы. Предполагается, что эти команды выполняются последовательно, одна после другой, а переменные были объявлены ранее.

Заполните пустые ячейки таблицы. Присмотритесь к коду List в левом столбце и запишите, как бы, по вашему мнению, выглядел этот код при использовании обычного массива. Мы не ожидаем, что вы идеально справитесь со всеми заданиями, так что постарайтесь предложить наиболее вероятные варианты.

List

Обычный массив

Мы заполнили за вас пару ответов.

<code>List<String> myList = new List <String>();</code>	<code>String [] myList = new String[2];</code>
<code>String a = "Yay!";</code>	<code>String a = "Yay!";</code>
<code>myList.Add(a);</code>	
<code>String b = "Bummer";</code>	<code>String b = "Bummer";</code>
<code>myList.Add(b);</code>	
<code>int theSize = myList.Count;</code>	
<code>Guy o = guys[1];</code>	
<code>bool foundIt = myList.Contains(b);</code>	
<p>Подсказка: здесь одной строки кода будет недостаточно.</p>	



Возьми в руку карандаш
Решение

Вам было предложено заполнить пустые ячейки: просмотрите код с List в левом столбце и попробуйте написать эквивалентный код с обычным массивом.

List	Обычный массив
<code>List<String> myList = new List <String>();</code>	<code>String[] myList = new String[2];</code>
<code>String a = "Yay!";</code>	<code>String a = "Yay!";</code>
<code>myList.Add(a);</code>	<code>myList[0] = a;</code>
<code>String b = "Bummer";</code>	<code>String b = "Bummer";</code>
<code>myList.Add(b);</code>	<code>myList[1] = b;</code>
<code>int theSize = myList.Count;</code>	<code>int theSize = myList.Length;</code>
<code>Guy o = guys[1];</code>	<code>Guy o = guys[1];</code>
<code>bool foundIt = myList.Contains(b);</code>	<code>bool foundIt = false; for (int i = 0; i < myList.Length; i++) { if (b == myList[i]) { isIn = true; } }</code>

Списки List – объекты, содержащие методы, которые ничем не отличаются от других классов, использовавшихся до сих пор. Чтобы просмотреть список доступных методов в IDE, просто введите . рядом с именем List. Параметры передаются этим методам точно так же, как и классам, написанным вами.

Элементы в списке упорядочены; позиция элемента в списке называется **индексом**. Как и в случае с массивом, индексы списков начинаются с 0. Для обращения к элементу списка с конкретным индексом используется **индексатор**:

```
Guy o = guys[1];
```

С массивами ваши возможности намного более ограничены. Размер массива должен быть задан при его создании, и всю логику выполняемых операций вам придется написать самостоятельно.

Класс Array содержит статические методы, которые немного упрощают выполнение некоторых операций — например, вы уже видели метод Array.Resize, использованный в методе AddWorker. Но мы сосредоточимся на объектах List, потому что с ними гораздо проще работать.

Построим приложение для хранения обуви

Пришло время применить список List на практике. Построим консольное приложение .NET Core, которое предлагает пользователю добавить или удалить из списка пару обуви. Пример запуска приложения с добавлением двух пар обуви и их последующим удалением: 

Начнем с класса Shoe, в котором хранится фасон и цвет обуви. Затем мы создадим класс с именем ShoeCloset, который хранит данные обуви в списке List<Shoe>, с методами AddShoe и RemoveShoe для добавления и удаления обуви.

делайте это!

1 Добавьте перечисление для фасона обуви.

Обувь бывает разная: кроссовки, сандалии и т. д., так что перечисление имеет смысл:

```
enum Style
{
    Sneaker,
    Loafer,
    Sandal,
    Flipflop,
    Wingtip,
    Clog,
}
```

Вспомните, о чем говорилось ранее: перечисление можно привести к int и обратно. Таким образом, Sneaker равно 0, Loafer - 1, и т. д.

2 Добавьте класс Shoe. В нем используется перечисление Style для фасона и строка для цвета; в целом класс работает так же, как класс Card, созданный ранее в этой главе:

```
class Shoe
{
    public Style Style {
        get; private set;
    }
    public string Color {
        get; private set;
    }
    public Shoe(Style style, string color)
    {
        Style = style;
        Color = color;
    }
    public string Description
    {
        get { return $"A {Color} {Style}"; }
    }
}
```



The shoe closet is empty.

```
Press 'a' to add or 'r' to remove a shoe: a
Add a shoe
Press 0 to add a Sneaker
Press 1 to add a Loafer
Press 2 to add a Sandal
Press 3 to add a Flipflop
Press 4 to add a Wingtip
Press 5 to add a Clog
Enter a style: 1
Enter the color: black
```



The shoe closet contains:
Shoe #1: A black Loafer

```
Press 'a' to add or 'r' to remove a shoe: a
Add a shoe
Press 0 to add a Sneaker
Press 1 to add a Loafer
Press 2 to add a Sandal
Press 3 to add a Flipflop
Press 4 to add a Wingtip
Press 5 to add a Clog
Enter a style: 0
Enter the color: blue and white
```

Нажмите 'a', чтобы добавить обувь, затем выделите фасон и введите цвет.

The shoe closet contains:
Shoe #1: A black Loafer
Shoe #2: A blue and white Sneaker

```
Press 'a' to add or 'r' to remove a shoe: r
Enter the number of the shoe to remove: 2
Removing A blue and white Sneaker
```

The shoe closet contains:
Shoe #1: A black Loafer

Нажмите 'r', чтобы удалить обувь, затем введите номер удаляемой пары.

```
Press 'a' to add or 'r' to remove a shoe: r
Enter the number of the shoe to remove: 1
Removing A black Loafer
```

The shoe closet is empty.

```
Press 'a' to add or 'r' to remove a shoe:
```



- 3** Класс `ShoeCloset` использует `List<Shoe>` для управления данными обуви. Класс `ShoeCloset` содержит три метода: метод `PrintShoes` выводит список обуви на консоль, метод `AddShoe` запрашивает у пользователя данные обуви, включаемой в список, а метод `RemoveShoe` предлагает пользователю удалить данные:

```
using System.Collections.Generic;
```

← Не забудьте включить строку `using` в начало кода, без нее вы не сможете использовать класс `List`.

```
class ShoeCloset
```

```
{
```

```
    private readonly List<Shoe> shoes = new List<Shoe>();
```

```
    public void PrintShoes()
```

```
    {
```

```
        if (shoes.Count == 0)
```

```
        {
```

```
            Console.WriteLine("\nThe shoe closet is empty.");
```

```
        }
```

```
        else
```

```
        {
```

```
            Console.WriteLine("\nThe shoe closet contains:");
```

```
            int i = 1;
```

```
            foreach (Shoe shoe in shoes)
```

```
            {
```

```
                Console.WriteLine($"Shoe #{i++}: {shoe.Description}");
```

```
            }
```

```
        }
```

```
    public void AddShoe()
```

```
    {
```

```
        Console.WriteLine("\nAdd a shoe");
```

```
        for (int i = 0; i < 6; i++)
```

```
        {
```

```
            Console.WriteLine($"Press {i} to add a {(Style)i}");
```

```
        }
```

```
        Console.Write("Enter a style: ");
```

```
        if (int.TryParse(Console.ReadKey().KeyChar.ToString(), out int style))
```

```
        {
```

```
            Console.Write("\nEnter the color: ");
```

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

```
            Shoe shoe = new Shoe((Style)style, color);
```

```
            shoes.Add(shoe);
```

```
        }
```

```
    }
```

```
    public void RemoveShoe()
```

```
    {
```

```
        Console.Write("\nEnter the number of the shoe to remove: ");
```

```
        if (int.TryParse(Console.ReadKey().KeyChar.ToString(), out int shoeNumber) &&
```

```
            (shoeNumber >= 1) && (shoeNumber <= shoes.Count))
```

```
        {
```

```
            Console.WriteLine($"Removing {shoes[shoeNumber - 1].Description}");
```

```
            shoes.RemoveAt(shoeNumber - 1);
```

```
        }
```

```
    }
```

```
}
```

Цикл `foreach` перебирает список «`shoes`», и для каждой пары обуви на консоль выводится строка.

Список `List`, в котором хранятся ссылки на объекты `Shoe`.

ShoeCloset
private List<Shoe> shoes
PrintShoes AddShoe RemoveShoe

Цикл `for` присваивает «`i`» целое число в диапазоне от 0 до 5. Интерполируемая строка использует «`{(Style)i}`» для приведения типа к перечислению `Style`, а затем вызывает метод `ToString` для вывода имени элемента.

Здесь мы создаем новый экземпляр `Shoe` и добавляем его в список.

Такой код уже встречался вам ранее: он вызывает `Console.ReadKey`, а затем использует `KeyChar` для получения нажатой клавиши в виде символа. Метод `int.TryParse` получает строку, а не символ, поэтому `char` преобразуется в строку вызовом `ToString`.

Здесь экземпляр `Shoe` удаляется из списка.

- 4 Добавьте класс Program с точкой входа. Пока вроде ничего особенного не происходит. Дело в том, что все интересное поведение инкапсулируется в классе ShoeCloset:

```
class Program
{
    static ShoeCloset shoeCloset = new ShoeCloset();
    static void Main(string[] args)
    {
        while (true)
        {
            shoeCloset.PrintShoes();
            Console.WriteLine("\nPress 'a' to add or 'r' to remove a shoe: ");
            char key = Console.ReadKey().KeyChar;

            switch (key)
            {
                case 'a':
                case 'A':
                    shoeCloset.AddShoe();
                    break;
                case 'r':
                case 'R':
                    shoeCloset.RemoveShoe();
                    break;
                default:
                    return;
            }
        }
    }
}
```



После варианта 'a' нет команды break, поэтому происходит сквозная передача управления к варианту 'A' — оба варианта обрабатываются одним методом shoeCloset.AddShoe.

Для обработки пользовательского ввода используется команда switch. Команда 'A' в верхнем регистре должна работать точно так же, как и команда 'a' в нижнем регистре, поэтому мы разместили два варианта case подряд, не разделяя их командой break:

```
case 'a':
case 'A':
```

Когда switch переходит к новому варианту case, перед которым не была выполнена команда break, происходит сквозной переход к следующему варианту. Между двумя командами case даже могут располагаться команды. Но будьте внимательны — при этом очень легко пропустить нужную команду break.

- 5 Запустите приложение и добейтесь повторения приведенных результатов. Попробуйте отладить приложение и разобраться в том, как работают списки. Ничего запоминать пока не нужно — у вас будет масса возможностей потренироваться в их использовании!

Элементы списка под увеличительным стеклом



Класс коллекции List содержит метод Add для добавления элемента в конец списка. Метод AddShoe создает экземпляр Shoe, а затем вызывает метод shoes.Add со ссылкой на этот экземпляр:

```
shoes.Add(shoe);
```

Класс List также содержит метод RemoveAt для удаления из списка элемента с заданным индексом. У списков, как и у массивов, индексы **начинаются с нуля**; иначе говоря, первому элементу соответствует индекс 0, второму — индекс 1, и т. д.

```
shoes.RemoveAt(shoeNumber - 1);
```

Наконец, метод PrintShoes использует свойство List.Count для проверки того, пуст ли список:

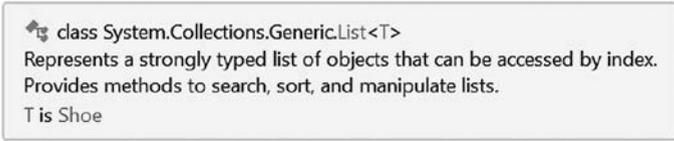
```
if (shoes.Count == 0)
```

В обобщенных коллекциях могут храниться любые типы

Вы уже видели, что в списке могут храниться строки или объекты Shoe. Также можно создавать списки целых чисел или любых объектов, которые вы можете создать. В таком случае список становится **обобщенной коллекцией**. При создании нового объекта List вы привязываете его к конкретному типу: можно создать список для хранения элементов int, строк или объектов Shoe. Такой подход упрощает работу со списками: после того, как список будет создан, вы всегда знаете тип данных, которые в нем хранятся.

Но что имеется в виду под термином «обобщенный»? Исследуем обобщенные коллекции в Visual Studio. Откройте файл *ShoeCloset.cs* и наведите указатель мыши на List:

```
private readonly List<Shoe> shoes = new List<Shoe>();
```



class System.Collections.Generic.List<T>
Represents a strongly typed list of objects that can be accessed by index.
Provides methods to search, sort, and manipulate lists.
T is Shoe

Несколько фактов, на которые стоит обратить внимание:

- ★ Класс List из пространства имен System.Collections.Generic — это пространство имен содержит классы обобщенных коллекций (именно поэтому необходима строка using).
- ★ В описании сказано, что List предоставляет «методы для поиска, сортировки и выполнения операций со списками». Некоторые из этих методов использовались в нашем классе ShoeCloset.
- ★ В верхней строке упоминается List<T>, а в нижней — T is Shoe. Так определяются обобщения — по сути, в описании сказано, что List может работать с любым типом, но для этого конкретного списка таким типом становится класс Shoe.

Обобщенные списки объявляются с <угловыми скобками>

Когда вы объявляете список, какой бы тип в нем ни хранился, это всегда делается одинаково: тип объектов, хранящихся в списке, задается в <угловых скобках>.

Часто обобщенные классы (не только List) записываются в виде: List<T>. По этой записи сразу видно, что класс может определяться с любым типом элементов.

На самом деле это не означает, что вы добавляете букву T. Это всего лишь условная запись, которая встречается с классами и интерфейсами, работающими с любыми типами. На место части <T> можно поместить любой тип (например, List<Shoe>); тем самым вы ограничиваете элементы указанным типом.

```
List<T> name = new List<T>();
```

↑
Списки могут быть очень гибкими (с возможностью хранения любого типа), а могут быть предельно ограниченными. Они делают все, что могут делать массивы, и немало того, на что массивы не способны.

В обобщенных коллекциях могут храниться объекты любого типа. Они предоставляют целостный набор методов для работы с объектами в коллекции независимо от того, объекты какого типа в ней хранятся.

об-об-щен-ный, прил.
Характеристика класса или группы объектов; неконкретный.

Подсказки для IDE: Go To Definition (Windows) / Go To Declaration (macOS)

Класс List является частью .NET Core — огромной подборки очень полезных классов, интерфейсов, типов и т. д. В Visual Studio существует чрезвычайно мощный инструмент, при помощи которого можно исследовать эти классы и любой другой код, написанный вами. Откройте Program.cs и найдите следующую строку: `static ShoeCloset shoeCloset = new ShoeCloset();`

Щелкните правой кнопкой мыши на ShoeCloset и выберите команду **Go To Definition (Windows)** или **Go To Declaration (macOS)**.

В Windows также можно перейти к определению класса, компонента или переменной, щелкнув на нем с нажатой клавишей Ctrl.

Quick Actions and Refactorings...	Ctrl+.
Rename...	Ctrl+R, Ctrl+R
Remove and Sort Usings	Ctrl+R, Ctrl+G
Peek Definition	Alt+F12
Go To Definition	F12
Go To Base	Alt+Home

Quick Fix...	⌘↵
Rename...	⌘R
Remove and Sort Usings	
Go to Declaration	⌘D
Go to Implementation	

IDE немедленно переходит к определению класса ShoeCloset. Вернитесь к Program.cs и перейдите к определению PrintShoes в следующей строке: `shoeCloset.PrintShoes();` IDE переходит прямо к определению метода в классе ShoeCloset. Команда Go To Definition/Go To Declaration позволяет быстро перемещаться по вашему коду.

Используйте Go To Definition/Declaration для исследования обобщенных коллекций

А теперь самое интересное. Откройте ShoeCloset.cs и перейдите к определению List. IDE открывает отдельную вкладку с определением класса List. Не беспокойтесь, если новая вкладка содержит много сложного кода! Понимать все не обязательно — просто найдите следующую строку, из которой видно, что List<T> реализует набор интерфейсов:

```
public class List<NullableAttribute(2) T> : ICollection<T>, IEnumerable<T>, IEnumerable,
    IList<T>, IReadOnlyCollection<T>, IReadOnlyList<T>, ICollection, IList
```

Заметили, что на первом месте стоит интерфейс ICollection<T>? Этот интерфейс используется всеми обобщенными коллекциями. Вероятно, вы уже поняли, что делать дальше — перейти к определению/объявлению ICollection<T>. Ниже показано, что вы увидите в Visual Studio для Windows (комментарии XML свернуты и заменены кнопками ; на Mac их можно раскрыть:

```
namespace System.Collections.Generic
{
    ...public interface ICollection<NullableAttribute(2) T> : IEnumerable<T>, IEnumerable
    {
        ...int Count { get; }
        ...bool IsReadOnly { get; }

        ...void Add(T item);
        ...void Clear();
        ...bool Contains(T item);
        ...void CopyTo(T[] array, int arrayIndex);
        ...bool Remove(T item);
    }
}
```

Обобщенная коллекция предоставляет возможность узнать, сколько элементов в ней хранится; добавить новые элементы; очистить; проверить, содержит ли она заданный элемент; а также удалить элемент. Также поддерживаются другие возможности (например, List позволяет удалить элемент с конкретным индексом), но этот минимальный стандартный набор поддерживается каждой обобщенной коллекцией.

В предыдущей главе мы говорили о том, что суть интерфейсов — выполнение классами конкретных задач. Функциональность обобщенной коллекции является конкретной задачей. Любой класс может обладать этой функциональностью, для чего достаточно реализовать интерфейс ICollection<T>. Этот интерфейс реализуется классом List<T>; в этой главе вы увидите еще несколько классов коллекций, которые тоже его реализуют. Все они работают по-разному, но поскольку все они обладают функциональностью обобщенной коллекции, вы можете быть уверены в том, что все они поддерживают базовые возможности сохранения значений или ссылок.

КЛЮЧЕВЫЕ МОМЕНТЫ

- **List** — класс .NET для хранения, управления и удобной работы с наборами значений или ссылок на объекты. Значения или ссылки, хранящиеся в списке, называются **элементами**.
- Размер List **изменяется динамически** под нужный размер. При добавлении данных List расширяется по мере надобности.
- Чтобы поместить объект в List, используйте метод Add. Для удаления используется метод Remove.
- Для удаления объектов из List по индексу используется метод RemoveAt.
- Тип List объявляется с **аргументом-типом** в угловых скобках. Например, List<Frog> означает, что список List сможет хранить только объекты типа Frog.
- **Метод Contains** проверяет, присутствует ли конкретный объект в List. Метод IndexOf возвращает индекс конкретного элемента List.
- Свойство **Count** возвращает количество элементов в списке.
- Используйте **индексатор** (конструкция вида guys[3]) для обращения к элементу коллекции с заданным индексом.
- Для перебора List списка можно воспользоваться **циклом foreach** (по аналогии с массивами).
- List является **обобщенной коллекцией**; это означает, что List может хранить любой тип.
- Все обобщенные коллекции реализуют обобщенный **интерфейс ICollection<T>**.
- <T> в определении обобщенного класса или интерфейса **заменяется типом** при создании экземпляра.
- Используйте команду Go To Definition (Windows) или Go To Declaration (macOS) в Visual Studio для исследования вашего кода и других используемых классов.



Будьте
осторожны!

Не изменяйте коллекцию в процессе ее перебора в цикле foreach!

Если коллекция будет изменена, выдается исключение *InvalidOperationException*. Вы можете убедиться в этом сами. Создайте новое консольное приложение

.NET Core, затем добавьте код для создания нового списка *List<string>*, добавьте значение, воспользуйтесь циклом *foreach* для перебора и добавьте новое значение в коллекцию **внутри** цикла. При запуске кода цикл *foreach* выдаст исключение. И помните: при использовании обобщенных классов всегда необходимо указывать тип — таким образом *List<string>* обозначает список строк.

```
static void Main(string[] args)
{
    List<string> values = new List<string>();
    values.Add("a value");
    foreach (string s in values)
    {
        values.Add("another value");
    }
}
```

Exception Unhandled

System.InvalidOperationException: 'Collection was modified; enumeration operation may not execute.'

[View Details](#) | [Copy Details](#) | [Start Live Share session...](#)

▸ [Exception Settings](#)



Развлечения с Магнитами

Сможете ли вы расставить фрагменты кода, чтобы создать работоспособное консольное приложение, которое выводит приведенный результат на консоль?

```
static void Main(string[] args)
```

```
{
```

```
}
```

```
string zilch = "zero";
string first = "one";
string second = "two";
string third = "three";
string fourth = "4.2";
string twopointtwo = "2.2";
```

```
}
```

```
a.Add(zilch);
a.Add(first);
a.Add(second);
a.Add(third);
```

```
}
```

```
static void PppPppL (List<string> a){
```

```
foreach (string element in a)
{
    Console.WriteLine(element);
}
```

```
List<string> a = new List<string>();
```

```
if (a.IndexOf("four") != 4)
{
    a.Add(fourth);
}
```

```
a.RemoveAt(2);
```

```
if (a.Contains("three"))
{
    a.Add("four");
}
```

```
PppPppL(a);
```

```
if (a.Contains("two")) {
    a.Add(twopointtwo);
}
```

Результат

```
zero
one
three
four
4.2
```



Развлечения с Магнитами Решение

Если вы хотите выполнить этот код, не забудьте включить строку «using System. Collection.Generic» в начало.

Помните, как мы обсуждали использование содержательных имен в главе 3? Да, они улучшают код, но с ними эти упражнения получились бы слишком простыми. Только не используйте загадочные имена вида PppPppL в реальных программах!

```
static void Main(string[] args)
```

```
{
```

```
List<string> a = new List<string>();
```

```
string zilch = "zero";  
string first = "one";  
string second = "two";  
string third = "three";  
string fourth = "4.2";  
string twopointtwo = "2.2";
```

```
a.Add(zilch);  
a.Add(first);  
a.Add(second);  
a.Add(third);
```

```
if (a.Contains("three"))  
{  
    a.Add("four");  
}
```

```
a.RemoveAt(2);
```

```
if (a.IndexOf("four") != 4)  
{  
    a.Add(fourth);  
}
```

```
if (a.Contains("two")) {  
    a.Add(twopointtwo);  
}
```

```
PppPppL(a);
```

```
}
```

```
static void PppPppL(List<string> a){
```

```
foreach (string element in a)  
{  
    Console.WriteLine(element);  
}
```

```
}
```

```
}
```

Результат

```
zero  
one  
three  
four  
4.2
```

А вы сможете определить, почему значение "2.2" не добавляется в список, хотя оно объявляется здесь и передается Add ниже? Воспользуйтесь отладчиком, чтобы разобраться в происходящем!

Метод RemoveAt удаляет элемент с индексом 2 — это третий элемент в списке.

Цикл foreach перебирает все элементы списка и выводит их.

Метод PppPppL использует цикл foreach для перебора списка строк, их объединения в одну большую строку, которая выводится в окне сообщения.

Часть Задаваемые Вопросы

В: С чего бы мне использовать перечисление вместо коллекции? Разве они не решают похожие задачи?

О: Перечисления очень сильно отличаются от коллекций. Прежде всего перечисления являются **типами**, тогда как коллекции являются **объектами**.

Перечисления можно рассматривать как удобный механизм хранения **списков констант**, к которым можно обращаться по имени. Они упрощают чтение вашего кода и гарантируют, что для обращения к часто используемым значениям всегда будут использоваться правильные имена.

В коллекции могут храниться практически любые данные, потому что в коллекции хранятся **ссылки на объекты**, по которым можно обращаться к компонентам объектов. С другой стороны, перечислению должен быть присвоен один из **типов значений C#** (см. главу 4). Их можно преобразовать в значения, но не в ссылки.

Кроме того, перечисления не могут динамически изменять свой размер. Они не могут реализовывать интерфейсы или содержать методы, и для сохранения значения из перечисления в другой переменной придется преобразовать их к другому типу. С учетом всего сказанного мы получаем два сильно различающихся способа хранения данных. Каждый из них полезен по-своему.

В: Похоже, класс List — довольно мощная штука. Тогда зачем использовать массивы?

Массивы потребляют меньше памяти и процессорного времени в программах, но прирост производительности очень невелик. Если вы выполняете одну и ту же операцию, скажем, миллионы раз в секунду, то лучше использовать массив вместо списка. Но если ваша программа работает медленно, крайне маловероятно, что переход со списка на массив решит эту проблему.

О: Для хранения коллекции объектов обычно используется список, а не массив. Одна из ситуаций, в которых предпочтение отдается массивам (примеры будут приведены позднее в книге), — чтение последовательности байтов (например, из файла). В этом случае часто вызывается метод класса .NET, возвращающий `byte[]`. К счастью, списки легко преобразуются в массивы (вызовом метода `ToArray`), а массивы преобразуются в списки (при помощи перегруженного конструктора).

В: Я не понимаю, почему коллекции называются «обобщенными»?

О: Обобщенная коллекция представляет собой объект коллекции (или встроенный объект, который позволяет хранить наборы других объектов и управлять ими), настроенный для хранения только одного типа (или нескольких типов, как вы вскоре увидите).

В: Ладно, с «коллекцией» понятно. Но что делает ее «обобщенной»?

О: Когда-то в супермаркетах продавались обобщенные товары в белой упаковке с черной надписью, которая сообщала только вид товара («Картофельные чипсы», «Кола», «Мыло» и т. д.).

То же самое происходит с обобщенными типами данных. Список `List<T>` будет работать одинаково, какие бы данные в нем ни хранились. Список объектов `Card`, `int`, `long` и даже другие списки — все они могут действовать на уровне контейнера. Вы всегда можете добавлять,

удалять, вставлять элементы и т. д., какие бы данные ни хранились в списке.

В: Можно ли создать список без типа?

О: Нет. С каждым списком, а на самом деле с каждой обобщенной коллекцией (вскоре вы узнаете о других обобщенных коллекциях), должен быть связан тип. В C# поддерживаются необобщенные списки `ArrayList`, в которых могут храниться любые объекты. Если вы хотите использовать `ArrayList`, в программу необходимо включить строку `using System.Collections;`. Впрочем, делать это приходится нечасто, потому что `List<Object>` обычно хорошо подходит для ситуации, в которой можно было бы использовать нетипизованный объект `ArrayList`.

При создании нового объекта List всегда указывается тип — он сообщает C# тип данных, которые будут храниться в списке. В List могут храниться как типы значений (int, bool, decimal и т. д.), так и классы.

↑
Термин «обобщенный» обозначает тот факт, что хотя в конкретном экземпляре List могут храниться элементы одного конкретного типа, сам класс List работает с любым типом. Именно на это указывает `<T>` в имени — эта часть сообщает, что список содержит набор ссылок типа T.

Инициализаторы коллекций похожи на инициализаторы объектов

C# предоставляет довольно удобную сокращенную запись для ситуаций, в которых требуется создать список и немедленно занести в него набор элементов. При создании нового объекта List можно воспользоваться инициализатором коллекции для определения начального списка элементов. Элементы будут добавлены сразу же после создания списка.

Этот код создает новый экземпляр List<Shoe> и заполняет его объектами Shoe, для чего многократно вызывается метод Add.

```
List<Shoe> shoeCloset = new List<Shoe>();
shoeCloset.Add(new Shoe() { Style = Style.Sneakers, Color = "Black" });
shoeCloset.Add(new Shoe() { Style = Style.Clogs, Color = "Brown" });
shoeCloset.Add(new Shoe() { Style = Style.Wingtips, Color = "Black" });
shoeCloset.Add(new Shoe() { Style = Style.Loafers, Color = "White" });
shoeCloset.Add(new Shoe() { Style = Style.Loafers, Color = "Red" });
shoeCloset.Add(new Shoe() { Style = Style.Sneakers, Color = "Green" });
```

Тот же код, переписанный с инициализатором коллекции

А вы заметили, что каждый объект Shoe инициализируется собственным инициализатором объекта? Да, инициализаторы объектов могут вкладываться в инициализатор коллекции.

Чтобы создать инициализатор коллекции, возьмите каждый элемент, который добавлялся вызовом Add, и включите его в команду создания списка.

За командой создания списка следуют фигурные скобки с отдельными командами «new», разделенными запятыми.

```
List<Shoe> shoeCloset = new List<Shoe>() {
    new Shoe() { Style = Style.Sneakers, Color = "Black" },
    new Shoe() { Style = Style.Clogs, Color = "Brown" },
    new Shoe() { Style = Style.Wingtips, Color = "Black" },
    new Shoe() { Style = Style.Loafers, Color = "White" },
    new Shoe() { Style = Style.Loafers, Color = "Red" },
    new Shoe() { Style = Style.Sneakers, Color = "Green" },
};
```

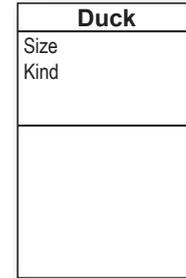
Вы не ограничены использованием команд «new» в инициализаторе — также можно включать сюда переменные.

Инициализатор коллекции делает код более компактным, так как он позволяет объединить операцию создания списка с добавлением исходного набора элементов.

Создание списка уток

Сделайте это!

Ниже приведен класс Duck, в котором хранится информация об утках, живущих с вами по соседству. **Создайте новый проект консольного приложения** и включите в него класс Duck и перечисление KindOfDuck.



```
class Duck {
    public int Size {
        get; set;
    }
    public KindOfDuck Kind {
        get; set;
    }
}

enum KindOfDuck {
    Mallard,
    Muscovy,
    Loon,
}
```

Добавьте Duck и KindOfDuck в проект. Перечисление KindOfDuck используется для отслеживания пород уток, поддерживаемых в приложении. Обратите внимание на то, что значения элементам перечисления не присваиваются — это очень типично. Числовые значения нас не интересуют, и значения по умолчанию (0, 1, 2...) прекрасно подойдут.

Выполните свой код — он выводит на консоль набор объектов Duck.

Инициализатор для списка уток

В коллекции изначально хранятся данные шести уток, поэтому мы создадим список List<Duck> с инициализатором коллекции из шести команд. Каждая команда в инициализаторе создает новый объект Duck, используя инициализатор объекта для определения значений полей Size и Kind объекта Duck. Убедитесь в том, что в начале Program.cs располагается соответствующая **директива using**:

```
using System.Collections.Generic;
```

Затем добавьте следующий метод PrintDucks в класс Program:

```
public static void PrintDucks(List<Duck> ducks)
{
    foreach (Duck duck in ducks) {
        Console.WriteLine($"{duck.Size} inch {duck.Kind}");
    }
}
```

Наконец, добавьте следующий код в метод Main из файла Program.cs. Код создает список List с объектами Duck и выводит их:

```
List<Duck> ducks = new List<Duck>() {
    new Duck() { Kind = KindOfDuck.Mallard, Size = 17 },
    new Duck() { Kind = KindOfDuck.Muscovy, Size = 18 },
    new Duck() { Kind = KindOfDuck.Loan, Size = 14 },
    new Duck() { Kind = KindOfDuck.Muscovy, Size = 11 },
    new Duck() { Kind = KindOfDuck.Mallard, Size = 14 },
    new Duck() { Kind = KindOfDuck.Loan, Size = 13 },
};

PrintDucks(ducks);
```

Списки удобны, но с СОРТИРОВКОЙ могут возникнуть проблемы

Нетрудно представить различные способы сортировки чисел или строк. Но как отсортировать два объекта, особенно если они содержат несколько полей? В каких-то случаях можно упорядочить объекты по значению поля Name, тогда как в других случаях имеет смысл упорядочить объекты на основании поля размера или даты рождения. Существует множество способов упорядочения объектов, и все они поддерживаются списками.

Список уток можно отсортировать по размеру...

От маленьких к большим...



...или по породе.

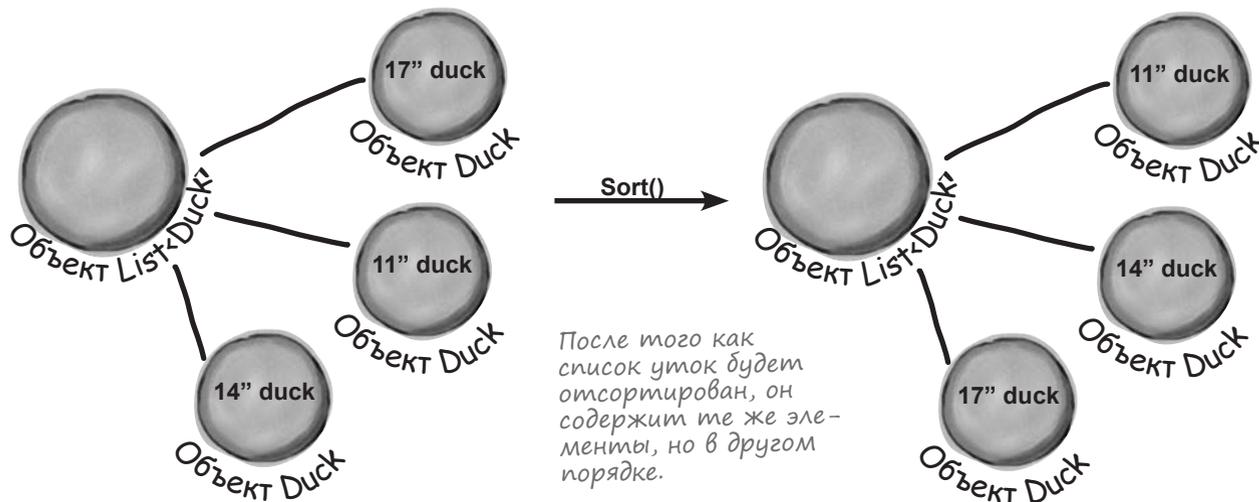
По породе...



Списки умеют сортировать свое содержимое

Каждый список List содержит метод **Sort**, который переставляет все элементы списка в некотором порядке. Списки уже умеют сортировать многие встроенные типы и классы, и вы можете легко обучить их сортировать классы, написанные вами.

Формально сортировкой занимается не объект List<T>. Это задача объекта IComparer<T>, о котором вы вскоре узнаете.



IComparable<Duck> помогает snucky List сортировать объекты Duck

Если у вас имеется список List с числами и вы вызываете его метод Sort, список будет отсортирован от меньших чисел к большим. Как список List определит, как нужно сортировать объекты Duck? Мы сообщаем List.Sort, что объекты класса Duck могут участвовать в сортировке, и делается это так, как мы обычно указываем, что класс может выполнять определенную задачу: *при помощи интерфейса*.

Метод List.Sort умеет сортировать любые типы или классы, реализующие интерфейс IComparable<T>. Интерфейс содержит всего один компонент — метод CompareTo. Sort использует метод CompareTo объекта для сравнения его с другими объектами и по возвращенному значению (int) определяет, какой из объектов должен предшествовать другому.

Метод CompareTo объекта сравнивает его с другим объектом

Чтобы наделить объект List возможностью сортировать уток, можно **изменить класс Duck и реализовать IComparable<Duck>**, т. е. добавить его единственный компонент — метод CompareTo, получающий ссылку на Duck в параметре.

Обновите класс Duck и реализуйте интерфейс IComparable<Duck>, чтобы сортировка производилась по размеру:

```
class Duck : IComparable<Duck> {
    public int Size { get; set; }
    public KindOfDuck Kind { get; set; }

    public int CompareTo(Duck duckToCompare) {
        if (this.Size > duckToCompare.Size)
            return 1;
        else if (this.Size < duckToCompare.Size)
            return -1;
        else
            return 0;
    }
}
```

Многие методы CompareTo похожи на этот. Метод сначала сравнивает поле Size с полем Size другого объекта Duck. Если у текущего объекта Duck оно больше, возвращается 1; если меньше, возвращается -1. Если значения равны, то возвращается 0.

Тип сравниваемых объектов задается при реализации интерфейса IComparable<T>.

Если вы хотите отсортировать список от меньших значений к большим, метод CompareTo должен возвращать положительное значение при сравнении с меньшим объектом и отрицательное — при сравнении с большим.

Если объект, с которым сравнивается текущий объект, должен следовать после текущего в отсортированном списке, метод CompareTo должен возвращать положительное число. Если он должен предшествовать текущему объекту, метод CompareTo должен возвращать отрицательное число. Если они равны, метод возвращает нуль.

Добавьте следующую строку в конец метода Main, непосредственно перед вызовом PrintDucks. Она приказывает списку уток отсортировать себя. Теперь утки упорядочиваются по размеру перед выводом на консоль

```
ducks.Sort();
PrintDucks(ducks);
```



Использование IComparer для определения порядка сортировки

Класс Duck реализует IComparable, так что ListSort знает, как отсортировать список List объектов Duck. Но что, если вы захотите отсортировать их способом, отличным от стандартного? Или если потребуется отсортировать объекты типа, не реализующего IComparable? В таком случае в аргументе List.Sort можно передать объект-компаратор для определения другого способа сортировки объектов. Обратите внимание на перегрузку List.Sort:

```
void List<Duck>.Sort(IComparer<Duck>? comparer)
Sorts the elements in the entire List<T> using the specified comparer.
comparer: The IComparer<in T> implementation to use when comparing elements, or null to use the default comparer Comparer<T>.Default.
```

Перегруженная версия List.Sort получает ссылку на IComparer<T>, где T заменяется обобщенным типом вашего списка (таким образом, для List<Duck> передается аргумент IComparer<Duck>, для List<string> — IComparer<string>, и т. д.). Так как передается ссылка на объект, реализующий интерфейс, мы знаем, что это означает: что объект *выполняет конкретную задачу*. В данном случае задачей является сравнение пар элементов списка и определение порядка их сортировки для List.Sort.

Интерфейс IComparer<T> содержит только один компонент: **метод с именем Compare**. Этот метод практически не отличается от метода CompareTo в IComparable<T>: он получает два параметра-объекта x и y и возвращает положительное значение, если x предшествует y; отрицательное значение, если x следует после y; или ноль, если они равны.

Включение IComparer в проект

Добавьте класс DuckComparerBySize в проект. Это объект-компаратор, который будет передаваться в параметре List.Sort для того, чтобы утки сортировались по размеру.

Интерфейс IComparer принадлежит пространству имен System.Collections.Generic, поэтому при добавлении класса в новый файл следует проследить за тем, чтобы он содержал необходимую директиву using:

```
using System.Collections.Generic;
```

Код класса компаратора выглядит так:

```
class DuckComparerBySize : IComparer<Duck>
{
    public int Compare(Duck x, Duck y)
    {
        if (x.Size < y.Size)
            return -1;
        if (x.Size > y.Size)
            return 1;
        return 0;
    }
}
```

Если Compare возвращает отрицательное число, это означает, что объект x в порядке сортировки должен предшествовать объекту y. Иначе говоря, x «меньше» y.

Любое положительное значение означает, что объект x должен следовать за объектом y, т. е. x «больше» y. Ноль означает, что два значения «равны».

Объект-компаратор представляет собой экземпляр класса, реализующего интерфейс IComparer<T>, который можно передать в ссылке List.Sort. Метод Compare работает так же, как метод CompareTo интерфейса IComparable<T>. Когда метод List.Sort сравнивает элементы для сортировки, он передает пары объектов методу Compare объекта-компаратора, так что список List будет сортироваться по-разному в зависимости от того, как вы реализовали компаратор.

А вы сможете изменить DuckComparerBySize, чтобы утки сортировались по убыванию размера, т. е. от больших к меньшим?

Создание экземпляра компаратора

Если вы хотите выполнить сортировку с использованием `IComparer<T>`, необходимо создать новый экземпляр класса, реализующего этот интерфейс, в данном случае `Duck`. Это объект-компаратор, который помогает `List.Sort` определить порядок сортировки элементов. Как и с любым другим (нестатическим) классом, перед использованием необходимо создать его экземпляр:

```
IComparer<Duck> sizeComparer = new DuckComparerBySize();  
ducks.Sort(sizeComparer);  
PrintDucks(ducks);
```

Замените `ducks.Sort` в методе `Main` двумя строками кода. Список по-прежнему сортируется, но теперь для сортировки используется компаратор.

Методу `Sort` в параметре передается ссылка на новый объект `DuckComparerBySize`.



Несколько реализаций IComparer, несколько способов сортировки объектов

Вы можете создать несколько классов `IComparer<Duck>` с разной логикой сортировки, чтобы сортировать список уток разными способами. Далее остается лишь воспользоваться нужным компаратором, когда потребуется отсортировать список каким-то определенным образом. Ниже приведена еще одна реализация компаратора `Duck`, которую следует добавить в проект

Этот компаратор сортирует уток по породе. Не забывайте, что при сравнении значений из перечисления `Kind` вы сравниваете их индексы в перечислении. Мы не присваивали конкретные значения при объявлении перечисления `KindOfDuck`, поэтому им присваиваются значения 0, 1, 2 и т. д. в порядке их следования в объявлении перечисления (`Mallard` соответствует 0, `Muscovy` — 1, `Loon` — 2).

```
class DuckComparerByKind : IComparer<Duck> {  
    public int Compare(Duck x, Duck y) {  
        if (x.Kind < y.Kind)  
            return -1;  
        if (x.Kind > y.Kind)  
            return 1;  
        else  
            return 0;  
        }  
}
```

Мы сравнили свойства `Kind` объектов `Duck`, так что объекты будут отсортированы по значению индекса свойства `Kind`.

Обратите внимание: понятия «больше» и «меньше» здесь имеют другой смысл. Мы использовали `<` и `>` для сравнения значений индексов перечисления, что позволяет нам разместить уток в нужном порядке.

Пример совместного использования перечислений и списков. Перечисления заменяют обычные числа и используются при сортировке списков.

Измените свою программу, чтобы в ней использовался компаратор. Следующий фрагмент сортирует объекты `Duck` перед выводом на консоль.

```
IComparer<Duck> kindComparer = new DuckComparerByKind();  
ducks.Sort(kindComparer);  
PrintDucks(ducks);
```



Компараторы могут выполнять сложные сравнения

У создания отдельного класса для сортировки объектов Duck есть одно важное преимущество: вы можете встроить в класс более сложную логику — и добавить компоненты, которые помогут определить, как именно должен быть отсортирован список.

```
enum SortCriteria {
    SizeThenKind,
    KindThenSize,
}
```

Перечисление сообщает объекту-компаратору, каким способом следует отсортировать объекты Duck.

Более сложный класс для сравнения Duck. Его метод Compare получает те же параметры, но для определения способа сортировки он обращается к открытому полю SortBy.

```
class DuckComparer : IComparer<Duck> {
    public SortCriteria SortBy = SortCriteria.SizeThenKind;

    public int Compare(Duck x, Duck y) {
        if (SortBy == SortCriteria.SizeThenKind)
            if (x.Size > y.Size)
                return 1;
            else if (x.Size < y.Size)
                return -1;
            else
                if (x.Kind > y.Kind)
                    return 1;
                else if (x.Kind < y.Kind)
                    return -1;
                else
                    return 0;
        else
            if (x.Kind > y.Kind)
                return 1;
            else if (x.Kind < y.Kind)
                return -1;
            else
                if (x.Size > y.Size)
                    return 1;
                else if (x.Size < y.Size)
                    return -1;
                else
                    return 0;
    }
}
```

Команда if проверяет поле SortBy. Если поле содержит SizeThenKind, то утки сначала сортируются по размеру, а при одинаковых размерах — по породе.

Вместо того чтобы просто вернуть 0 при совпадении размеров, компаратор проверяет породу и возвращает 0 только в том случае, если у двух уток совпадает как размер, так и порода.

Если SortBy не содержит SizeThenKind, компаратор сначала выполняет сортировку по породе. Если у двух уток порода совпадает, тогда компаратор сравнивает их размеры.

```
DuckComparer comparer = new DuckComparer();
Console.WriteLine("\nSorting by kind then size\n");
comparer.SortBy = SortCriteria.KindThenSize;
ducks.Sort(comparer);
PrintDucks(ducks);
Console.WriteLine("\nSorting by size then kind\n");
comparer.SortBy = SortCriteria.SizeThenKind;
ducks.Sort(comparer);
PrintDucks(ducks);
```

Добавьте этот код в конец метода Main. В нем используется объект-компаратор, а перед вызовом ducks.Sort задается значение поля SortBy. Теперь можно изменить способ сортировки списков своих объектов простым изменением свойства в компараторе.





Упражнение

Постройте консольное приложение, которое создает список карт в случайном порядке, выводит их на консоль, использует объект-компаратор для сортировки карт, а затем выводит отсортированный список.



1 Напишите метод для создания перетасованного набора карт.

Создайте новое консольное приложение. Добавьте перечисление `Suits`, перечисление `Values` и класс `Card`, приведенный ранее в этой главе. Затем добавьте в `Program.cs` два статических метода: метод `RandomCard`, возвращающий ссылку на карту со случайной мастью и номиналом, и `PrintCards`, который выводит `List<Card>`.

2 Создайте класс, реализующий `IComparer<Card>` для сортировки карт.

Вам предоставляется хорошая возможность воспользоваться меню `Quick Actions` среды IDE для реализации интерфейса. Добавьте класс с именем `CardComparerByValue`, а затем реализуйте интерфейс `IComparer<Card>`:

```
class CardComparerByValue : IComparer<Card>
```

Щелкните на `IComparer<Card>` и наведите указатель мыши на `I`. На экране появляется значок с изображением лампочки (💡 или 🧠). Если щелкнуть на значке, IDE открывает меню `Quick Actions`:

В IDE существует удобный способ ускоренного вызова меню `Quick Actions`: нажмите `Ctrl+` (Windows) или `Option+Enter` (Mac).



Ваш объект `IComparer` должен сортировать карты по номиналу, чтобы карты с малыми номиналами располагались в начале списка.

Выберите команду «Implement interface» — она приказывает IDE автоматически заполнить все методы и свойства интерфейса, которые необходимо реализовать. В данном случае создается пустой метод `Compare` для сравнения двух карт, `x` и `y`. Метод должен возвращать `1`, если `x` больше `y`; `-1`, если `x` меньше `y`; `0`, если они равны. Сначала карты упорядочиваются по масти: первыми идут бубны (`Diamonds`), затем трефы (`Clubs`), червы (`Hearts`) и пики (`Spades`). Проследите за тем, чтобы любой король следовал после любого валета, который, в свою очередь, следует после любой четверки, которая следует после любого туза. Значения перечислений можно сравнивать без приведения типа: `if (x.Suit < y.Suit)`.

3 Убедитесь в том, что программа выдает правильный результат.

Напишите метод `Main`, чтобы вывод выглядел так: →

- * Программа запрашивает количество карт.
- * Если пользователь вводит допустимое число и нажимает `Enter`, программа генерирует список случайных карт и выводит их.
- * Список карт сортируется с использованием компаратора.
- * Программа выводит отсортированный список карт.

```
Enter number of cards: 9
Eight of Spades
Nine of Hearts
Four of Hearts
Nine of Hearts
King of Diamonds
King of Spades
Six of Spades
Seven of Clubs
Seven of Clubs
```

... sorting the cards ...

```
King of Diamonds
Seven of Clubs
Seven of Clubs
Four of Hearts
Nine of Hearts
Nine of Hearts
Six of Spades
Eight of Spades
King of Spades
```



Упражнение
Решение

Постройте консольное приложение, которое создает список карт в случайном порядке, выводит их на консоль, использует объект-компаратор для сортировки карт, а затем выводит отсортированный список. Не забудьте добавить директиву `using System.Collections.Generic`; в начало файла с точкой входа.

```
class CardComparerByValue : IComparer<Card>
{
    public int Compare(Card x, Card y)
    {
        if (x.Suit < y.Suit)
            return -1;
        if (x.Suit > y.Suit)
            return 1;
        if (x.Value < y.Value)
            return -1;
        if (x.Value > y.Value)
            return 1;
        return 0;
    }
}

class Program
{
    private static readonly Random random = new Random();

    static Card RandomCard()
    {
        return new Card((Values)random.Next(1, 14), (Suits)random.Next(4));
    }

    static void PrintCards(List<Card> cards)
    {
        foreach (Card card in cards)
        {
            Console.WriteLine(card.Name);
        }
    }

    static void Main(string[] args)
    {
        List<Card> cards = new List<Card>();
        Console.Write("Enter number of cards: ");
        if (int.TryParse(Console.ReadLine(), out int numberOfCards))
            for (int i = 0; i < numberOfCards; i++)
                cards.Add(RandomCard());

        PrintCards(cards);

        cards.Sort(new CardComparerByValue());
        Console.WriteLine("\n... sorting the cards ...\n");

        PrintCards(cards);
    }
}
```

← Внутренняя реализация сортировки карт использует встроенный метод `List.Sort`. `Sort` получает объект `IComparer`, который содержит всего один метод: `Compare`. Такая реализация получает две карты и сначала сравнивает их номиналы, а потом масти.

Мы хотим, чтобы все бубны (Diamonds) предшествовали всем трефам (Clubs), поэтому начинать нужно со сравнения мастей. При этом стоит воспользоваться перечислениями.

Эти команды выполняются только в том случае, если `x` и `y` имеют одинаковые значения, а следовательно, первые две команды `return` не были выполнены.

Если ни одна из четырех других команд не сработает, карты должны быть одинаковыми — вернуть ноль.

← Здесь создается обобщенный список `List` объектов `Card` для хранения карт. Когда карты окажутся в списке, их можно легко отсортировать с использованием `IComparer`.

Мы опустили фигурные скобки. Как вы считаете, это упрощает или усложняет чтение кода?

Переопределение метода ToString позволяет объекту описать себя

Каждый объект содержит метод **ToString**, который преобразует его в строку. На самом деле вы уже пользовались им — каждый раз, когда вы использовали { фигурные скобки } при строковой интерполяции, при этом вызывался метод ToString для содержимого фигурных скобок. IDE тоже использует ToString(). Когда вы создаете класс, он наследует метод ToString от Object — базового класса верхнего уровня, расширяемого всеми остальными классами.

Метод Object.ToString выводит **полное имя класса**, т. е. пространство имен, за которым следует точка и имя класса. Так как мы использовали пространство имен DucksProject при написании этой главы, классу Duck будет соответствовать полное имя DucksProject.Duck:

Console.WriteLine(new Duck().ToString()); → "DucksProject.Duck"

IDE также вызывает метод ToString — например, при просмотре или отслеживании значений переменных:

Когда вы наводите указатель мыши на «ducks», IDE выводит содержимое списка List, как это делалось ранее с массивами.

IDE вызывает метод ToString при просмотре или отслеживании переменной, но метод ToString, унаследованный Duck от Object, просто возвращает имя класса. В данном случае такое «описание» бесполезно.

Хм-м, информация оказалась менее полезной, чем можно было надеяться. Мы видим, что список содержит шесть объектов Duck. Раскрыв узел Duck, вы увидите его значения Kind и Size. Но не лучше ли увидеть все данные немедленно?

Переопределение метода ToString для просмотра объектов Duck в IDE

К счастью, ToString является виртуальным методом Object, базового класса всех объектов. Следовательно, вам достаточно **переопределить метод ToString**, а когда это будет сделано, результаты немедленно появятся в окне Watch в IDE!

```

override
  Equals(object? obj)
  GetHashCode()
  ToString() string? object.ToString()
             Returns a string that represents the current object.
    
```

Когда отладчик IDE выводит содержимое объекта, он при этом вызывает метод ToString объекта.

Щелкните на **toString()**, чтобы приказать IDE добавить новый метод ToString. Измените его код, чтобы он выглядел так:

```

public override string ToString()
{
    return $"A {Size} inch {Kind}";
}
    
```

Запустите программу и снова просмотрите список. Теперь в IDE выводится содержимое ваших объектов Duck.

Обновите циклы `foreach`, чтобы объекты `Duck` и `Card` выводили свои описания на консоль

Вы уже видели два примера перебора списка объектов в цикле и вызова `Console.WriteLine` для вывода на консоль строки для каждого объекта. Вспомните цикл `foreach`, который выводил все объекты `Card` из коллекции `List<Card>`:

```
foreach (Card card in cards)
{
    Console.WriteLine(card.Name);
}
```

Метод `PrintDucks` делал нечто подобное с объектами `Duck` в `List`:

```
foreach (Duck duck in ducks) {
    Console.WriteLine($"{duck.Size} inch {duck.Kind}");
}
```

Это операция довольно часто встречается при работе с объектами. Теперь, когда ваш объект `Duck` содержит метод `ToString`, метод `PrintDucks` должен воспользоваться этим обстоятельством. Используйте функцию IDE `IntelliSense` для просмотра перегруженных версий метода `Console.WriteLine`, а конкретно этой:

▲ 10 of 18 ▼ `void Console.WriteLine(object value)`



Если передать `Console.WriteLine` ссылку на объект, метод автоматически вызовет метод `ToString` этого объекта.

Если передать `Console.WriteLine` любой объект, будет вызван его метод `ToString`. Замените метод `PrintDucks` следующей реализацией, в которой вызывается перегруженная версия:

```
public static void PrintDucks(List<Duck> ducks) {
    foreach (Duck duck in ducks) {
        Console.WriteLine(duck);
    }
}
```

Замените метод `PrintDucks` этим методом и снова запустите код. Программа выдаст тот же результат. Если, допустим, вы захотите добавить в `Duck` свойство `Color` или `Weight`, будет достаточно обновить метод `ToString`, и это изменение отразится на всем, что использует этот метод (включая метод `PrintDucks`).

Добавьте метод `ToString` в объект `Card`

Ваш объект `Card` уже содержит свойство `Name`, возвращающее имя карты:

```
public string Name { get { return $"{Value} of {Suit}"; } }
```

Но ведь именно это должен делать метод `ToString`. Добавьте метод `ToString` в класс `Card`:

```
public override string ToString()
{
    return Name;
}
```

Мы решили обращаться к свойству `Name` из метода `ToString`. Как вы оцениваете этот выбор? Не лучше было бы удалить свойство `Name` и переместить его код в метод `ToString`? Когда вы возвращаетесь к изменению своего кода, часто приходится принимать подобные решения, и не всегда очевидно, какой выбор лучше.

Это упростит отладку программ, в которых используются объекты `Card`.

Возьми в руку карандаш



Прочитайте этот код и напишите, какой результат он будет выдавать.

```
enum Breeds
{
    Collie = 3,
    Corgi = -9,
    Dachshund = 7,
    Pug = 0,
}
class Dog : IComparable<Dog>
{
    public Breeds Breed { get; set; }
    public string Name { get; set; }

    public int CompareTo(Dog other)
    {
        if (Breed > other.Breed) return -1;
        if (Breed < other.Breed) return 1;
        return -Name.CompareTo(other.Name);
    }

    public override string ToString()
    {
        return $"A {Breed} named {Name}";
    }
}
class Program
{
    static void Main(string[] args)
    {
        List<Dog> dogs = new List<Dog>()
        {
            new Dog() { Breed = Breeds.Dachshund, Name = "Franz" },
            new Dog() { Breed = Breeds.Collie, Name = "Petunia" },
            new Dog() { Breed = Breeds.Pug, Name = "Porkchop" },
            new Dog() { Breed = Breeds.Dachshund, Name = "Brunhilda" },
            new Dog() { Breed = Breeds.Collie, Name = "Zippy" },
            new Dog() { Breed = Breeds.Corgi, Name = "Carrie" },
        };
        dogs.Sort();
        foreach (Dog dog in dogs)
            Console.WriteLine(dog);
    }
}
```

Подсказка: обратите внимание на минус!

Приложение выводит шесть строк на консоль. Можете ли вы заранее определить, что она выведет, и записать результаты? Попробуйте определить результат исключительно на основании чтения кода, без запуска приложения.

.....

.....

.....

.....

.....

.....

.....

Ниже приведен результат выполнения приложения. А вы правильно записали его? Если вы ошиблись — ничего страшного! Просто вернитесь и еще раз присмотритесь к перечислению.

- Вы заметили, что элементы имеют разные значения?
- Свойство `Name` представляет собой строку, а строки также реализуют `IComparable`, поэтому мы можем вызвать метод `CompareTo` для их сравнения.
- Также повнимательнее присмотритесь к методу `CompareTo`: вы заметили, что он возвращает `-1`, если другая порода больше; `1`, если другая порода меньше, или `-Name.CompareTo(other.Name)` (обратите внимание на знак «минус»)? Сначала сортировка выполняется по породе (`Breed`), затем по кличке (`Name`), но сортировка как по `Breed`, так и по `Name` осуществляется в обратном порядке.

Результат:

```

A Dachshund named Franz
.....
A Dachshund named Brunhilda
.....
A Collie named Zippy
.....
A Collie named Petunia
.....
A Pug named Porkchop
.....
A Corgi named Carrie
.....
    
```

Когда метод `CompareTo` использует `<` и `>` для сравнения значений `Breed`, он использует значения `int` из перечисления `Breed`, так что `Collie` соответствует `3`, `Corgi` соответствует `-9`, `Dachshund` соответствует `7`, а `Pug` соответствует `0`.

КЛЮЧЕВЫЕ МОМЕНТЫ

- **Инициализаторы коллекций** задают содержимое `List<T>` или другой коллекции при создании. Список объектов, разделенных запятыми, указывается в угловых скобках.
- Инициализатор коллекции делает ваш код более **компактным**: он позволяет объединить создание списка с добавлением исходного набора элементов (хотя код быстрее работать не будет).
- **Метод `List.Sort`** сортирует содержимое коллекции, изменяя порядок содержащихся в ней элементов.
- **Интерфейс `IComparable<T>`** содержит один метод `CompareTo`, который используется `List.Sort` для определения порядка сортируемых объектов.
- **Перегруженный метод** представляет собой метод, который может вызываться разными способами с разными конфигурациями параметров. В окне IDE IntelliSense можно просмотреть разные перегруженные версии метода.
- Метод `Sort` имеет перегруженную версию, получающую объект `IComparer<T>`, который затем используется для сортировки.
- `IComparable.CompareTo` и `IComparer.Compare` **сравнивают пары объектов**. Они возвращают `-1`, если первый объект меньше второго; `1`, если первый объект больше второго; или `0`, если они равны.
- **Класс `String` реализует `IComparable`**. Интерфейс `IComparer` или `IComparable` для класса, включающего строковый компонент, может вызвать метод `Compare` или `CompareTo` для определения порядка сортировки.
- Каждый объект содержит **метод `ToString`**, который преобразует его в строку. Метод `ToString` вызывается каждый раз, когда вы используете строковую интерполяцию или конкатенацию.
- Реализация `ToString` по умолчанию наследуется от `Object`. Она возвращает **полное имя класса**, которое состоит из пространства имен, точки и имени класса.
- **Переопределите метод `ToString`**, чтобы при интерполяции, конкатенации и многих других операциях использовалось нестандартное строковое представление класса.



Циклы `foreach` под увеличительным стеклом

Поближе познакомимся с циклами `foreach`. Откройте IDE, найдите переменную `List<Duck>` и используйте IntelliSense для просмотра информации о методе `GetEnumerator`. Начните вводить «`GetEnumerator`» и посмотрите, что произойдет:

```
ducks.GetEnumerator();
```

```
List<Duck>.Enumerator List<Duck>.GetEnumerator()
Returns an enumerator that iterates through the List<T>.
```

Создайте массив `Array[Duck]` и сделайте то же самое — массив тоже содержит массив `GetEnumerator`. Дело в том, что списки, массивы и другие коллекции реализуют интерфейс `IEnumerable<T>`.

Вы уже знаете, что суть интерфейсов — поддержка одной функциональности разными объектами. Если объект реализует интерфейс `IEnumerable<T>`, то этой функциональностью является *поддержка перебора в необобщенных коллекциях* — иначе говоря, он позволяет писать код, который перебирает содержимое этих коллекций. А конкретно это означает, что коллекция может использоваться в цикле `foreach`.

Как же это выглядит с точки зрения внутреннего устройства? Воспользуйтесь командой `Go To Definition/Declaration` с `List<Duck>`, чтобы увидеть реализуемые интерфейсы, как это делалось ранее. Затем сделайте то же самое для просмотра компонентов `IEnumerable<T>`. Что вы видите?

Интерфейс `IEnumerable<T>` содержит один компонент: метод `GetEnumerator`, который возвращает объект `Enumerator`. Объект `Enumerator` предоставляет механизмы перебора списка по порядку. Допустим, вы написали следующий цикл `foreach`:

```
foreach (Duck duck in ducks) {
    Console.WriteLine(duck);
}
```

А вот как будет работать цикл во внутренней реализации:

```
IEnumerator<Duck> enumerator = ducks.GetEnumerator();
while (enumerator.MoveNext()) {
    Duck duck = enumerator.Current;
    Console.WriteLine(duck);
}
if (enumerator is IDisposable disposable) disposable.Dispose();
```

Оба цикла выводят на консоль одни и те же объекты `Duck`. Запустите их и убедитесь в том, что выводимые результаты совпадают. (И пока не обращайте внимания на последнюю строку; интерфейс `IDisposable` будет рассматриваться в главе 10.)

Когда вы перебираете содержимое списка или массива (или любой другой коллекции), метод `MoveNext` возвращает `true`, если в коллекции присутствует следующий элемент, или `false`, если перебор достиг конца. Свойство `Current` всегда возвращает ссылку на текущий элемент. Сложите все вместе — и вы получите цикл `foreach`.

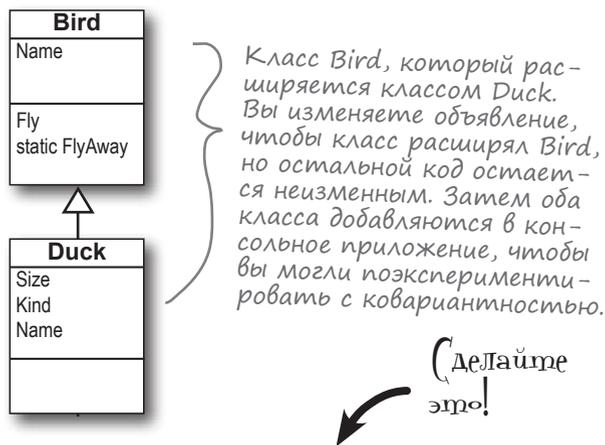
Если коллекция реализует `IEnumerable<T>`, это дает возможность использовать цикл, последовательно перебирающий ее содержимое.

Строго говоря, кода будет не-много больше, но и этот фрагмент дает общее представление о происходящем.

Повышающее приведение типа всего списка с использованием IEnumerable<T>

Помните, что любой объект можно повысить до его суперкласса? Если у вас имеется список List объектов, вы можете выполнить повышающее приведение типа для всего списка. Этот механизм называется **ковариантностью**, и все, что необходимо для его использования, — ссылка на интерфейс IEnumerable<T>.

Посмотрим, как это работает. Начнем с класса Duck, с которым вы уже работали в этой главе. Затем мы добавим класс Bird, который он расширяет. Класс Bird включает статический метод для перебора коллекции объектов Bird. Можно ли заставить его работать со списком List объектов Duck?



Так как все объекты Duck являются объектами Bird, ковариантность позволяет преобразовать коллекцию Duck в коллекцию Bird. Это может быть очень полезно, если вам приходится передавать List<Duck> методу, который принимает только List<Bird>.

- 1 **Создайте новый проект консольного приложения.** Добавьте базовый класс Bird (расширяемый классом Duck) и класс Penguin. Мы воспользуемся методом ToString, чтобы было лучше видно, какой класс что делает.

```
class Bird
{
    public string Name { get; set; }
    public virtual void Fly(string destination)
    {
        Console.WriteLine($"{this} is flying to {destination}");
    }

    public override string ToString()
    {
        return $"A bird named {Name}";
    }

    public static void FlyAway(List<Bird> flock, string destination)
    {
        foreach (Bird bird in flock)
        {
            bird.Fly(destination);
        }
    }
}
```

Ковариантность представляет собой механизм неявного преобразования ссылки на subclass в ссылку на суперкласс в C#. Термин «неявный» означает лишь то, что C# может определить, как выполнить преобразование, без явного приведения типа разработчиком.

Статический метод FlyAway работает с коллекцией объектов Bird. Но что, если вы хотите передать ему список List объектов Duck?

- 2 **Добавьте класс Duck в приложение.** Измените объявление, чтобы класс расширял Bird. Также необходимо добавить перечисление KindOfDuck, приведенное ранее в этой главе:

```
class Duck : Bird {
    public int Size { get; set; }
    public KindOfDuck Kind { get; set; }

    public override string ToString()
    {
        return $"A {Size} inch {Kind}";
    }
}
```

Мы изменили объявление, чтобы класс Duck расширял Bird. Остальной код Duck полностью совпадает с кодом предыдущего проекта.

```
enum KindOfDuck {
    Mallard,
    Muscovy,
    Loon,
}
```

Свойство Kind класса Duck использует перечисление KindOfDuck, поэтому его тоже необходимо добавить.

- 3 **Создайте коллекцию List<Duck>.** Добавьте следующий фрагмент в метод Main — это код, приведенный ранее в этой главе, и еще одна строка для повышающего приведения типа к List<Bird>:

```
List<Duck> ducks = new List<Duck>() {
    new Duck() { Kind = KindOfDuck.Mallard, Size = 17 },
    new Duck() { Kind = KindOfDuck.Muscovy, Size = 18 },
    new Duck() { Kind = KindOfDuck.Loan, Size = 14 },
    new Duck() { Kind = KindOfDuck.Muscovy, Size = 11 },
    new Duck() { Kind = KindOfDuck.Mallard, Size = 14 },
    new Duck() { Kind = KindOfDuck.Loan, Size = 13 },
};
```

Скопируйте инициализатор коллекции, который использовался ранее, для инициализации списка List объектов Duck.

```
Bird.FlyAway(ducks, "Minnesota");
```

```
CS1503 Argument 1: cannot convert from 'System.Collections.Generic.List<BirdCovariance.Duck>' to 'System.Collections.Generic.List<BirdCovariance.Bird>'
```

Код не компилируется! В сообщении об ошибке говорится о том, что коллекция Duck не может быть преобразована в коллекцию Bird. Попробуйте присвоить ducks списку List<Bird>:

```
List<Bird> upcastDucks = ducks;
```

Не работает. Мы получаем другую ошибку, в которой говорится о невозможности преобразования типа:

```
CS0029 Cannot implicitly convert type 'System.Collections.Generic.List<BirdCovariance.Duck>' to 'System.Collections.Generic.List<BirdCovariance.Bird>'
```

Разумно — происходящее в точности повторяет ситуацию с безопасным повышающим и понижающим преобразованием, о которой говорилось в главе 6: мы можем использовать присваивание для понижающего преобразования, но для безопасного повышающего преобразования необходимо использовать ключевое слово is. Как же выполнить безопасное повышение List<Duck> в List<Bird>?

- 4 **Используйте ковариантность для преобразования.** На помощь приходит ковариантность: вы можете воспользоваться присваиванием для повышающего преобразования List<Duck> в IEnumerable<Bird>. После получения IEnumerable<Bird> можно вызвать его метод ToList для преобразования в List<Bird>. Для этого в начало файла необходимо добавить директивы using System.Collections.Generic; и using System.Linq;

```
IEnumerable<Bird> upcastDucks = ducks;
Bird.FlyAway(upcastDucks.ToList(), "Minnesota");
```

Коллекция ссылок на Duck была преобразована в коллекцию ссылок на Bird.

Использование Dictionary для хранения ключей и значений

Список напоминает длинную страницу, заполненную именами. А теперь представьте, что вы хотите сопоставить каждому имени адрес. Или снабдить каждый автомобиль в вашей коллекции `garage` подробными техническими данными. Для этого вам потребуется другая разновидность коллекций .NET: **словарь** (dictionary). Эта структура позволяет взять конкретное значение — **ключ** — и связать его с данными — **значением**. Очень важный момент: каждый ключ встречается в словаре **только один раз**.

В реальных словарях ключом является определяемое слово. Оно используется для поиска значения, т. е. определения.

словарь, сущ.

Список слов в алфавитном порядке с объяснением их значений.

Определение является значением, т. е. данными, связанными с определенным ключом (в данном случае определяемое слово).

Словарь Dictionary в C# объявляется так:

```
Dictionary<TKey, TValue> dict = new Dictionary<TKey, TValue>();
```

Обобщенные типы для Dictionary. Тип `TKey` используется для ключей, а `TValue` — для значений. Таким образом, если в словаре хранятся слова и их определения, используется тип `Dictionary<string, string>`. А если вы хотите подсчитать количество вхождений каждого слова в книге, используйте тип `Dictionary<string, int>`.

Первый тип в угловых скобках всегда относится к ключу, а второй тип всегда относится к данным.

Рассмотрим пример практического использования словаря. Небольшое консольное приложение использует `Dictionary<string, string>` для отслеживания любимой еды нескольких друзей:

```
using System.Collections.Generic;
```

← Директива «using» необходима для использования словаря.

```
class Program
```

```
{
    static void Main(string[] args)
    {
```

```
        Dictionary<string, string> favoriteFoods = new Dictionary<string, string>();
        favoriteFoods["Alex"] = "hot dogs";
        favoriteFoods["A'ja"] = "pizza";
        favoriteFoods["Jules"] = "falafel";
        favoriteFoods["Naima"] = "spaghetti";
```

В словарь добавляются четыре пары «ключ/значение». В данном случае ключом является имя, а значением — любимая еда этого человека.

```
        string name;
        while ((name = Console.ReadLine()) != "")
        {
            if (favoriteFoods.ContainsKey(name))
                Console.WriteLine($"{name}'s favorite food is {favoriteFoods[name]}");
            else
                Console.WriteLine($"I don't know {name}'s favorite food");
        }
    }
```

Метод `ContainsKey` возвращает true, если словарь содержит значение с указанным ключом.

Так вы получаете значение, связанное с ключом.

Краткая сводка функциональности Dictionary

Словари во многом напоминают списки. Они гибки, позволяют вам работать с разными типами данных и имеют множество встроенных функций. Рассмотрим основные операции, которые могут выполняться со словарями.

★ Добавление элементов.

Для добавления элементов в словарь используется **индексатор** с угловыми скобками.

```
Dictionary<string, string> myDictionary = new Dictionary<string, string>();
myDictionary["some key"] = "some value";
```

Также можно добавить элемент в словарь **методом Add**:

```
Dictionary<string, string> myDictionary = new Dictionary<string, string>();
myDictionary.Add("some key", "some value");
```

★ Поиск значения по ключу.

Самым важным при работе со словарем является **поиск значений по индексатору** — собственно, они сохраняются в словаре именно для того, чтобы к ним можно было обращаться по уникальным ключам. Для словаря `Dictionary<string, string>` из нашего примера обращение к значению осуществляется по ключу `string`, возвращает он также строку.

```
string lookupValue = myDictionary["some key"];
```

★ Удаление элементов.

Как и при работе с объектами `List`, для удаления из словаря используется **метод Remove()**. Достаточно передать ему ключ, чтобы сам ключ и значение были удалены из словаря:

```
myDictionary.Remove("some key");
```

★ Получение списка ключей.

Для получения списка всех ключей в словаре используется свойство `Keys` в комбинации с циклом `foreach`. Это делается примерно так:

```
foreach (string key in myDictionary.Keys) { ... };
```

★ Подсчет пар в словаре.

Свойство Count возвращает число пар «ключ/значение», присутствующих в словаре:

```
int howMany = myDictionary.Count;
```

Ключи и значения могут иметь разные типы

Словари чрезвычайно гибки! В них можно хранить практически все что угодно — не только типы значений, но и *любые виды объектов*. В следующем примере словарь использует ключи типа `int`, а ссылки на объекты `Duck` являются значениями:

```
Dictionary<int, Duck> duckIds = new Dictionary<int, Duck>();
duckIds.Add(376, new Duck() { Kind = KindOfDuck.Mallard, Size = 15 });
```

Ключи словаря уникальны; каждый ключ встречается не более одного раза. Значения могут встречаться произвольное количество раз — двум ключам может соответствовать одно значение. Именно поэтому при поиске или удалении по ключу словарь знает, с какими данными должна выполняться операция.

Keys — свойство объекта `Dictionary`. В этом конкретном словаре используются строковые ключи, поэтому `Keys` представляет собой коллекцию строк.

На практике часто встречаются словари, которые связывают целые числа с объектами. Число становится уникальным идентификатором объекта.

Построение программы с использованием словаря



Вот небольшая программа, которая понравится бейсбольным фанатам команды «Нью-Йорк Янкиз». Как только важный игрок перестает выступать за команду, футболка с его номером перестает использоваться. **Создайте новое консольное приложение**, которое показывает, какие номера были у знаменитых игроков «Янкиз» и когда эти игроки ушли из большого спорта. Для хранения информации о бейсболистах, завершивших карьеру, будет использоваться следующий класс:

```
class RetiredPlayer
{
    public string Name { get; private set; }
    public int YearRetired { get; private set; }

    public RetiredPlayer(string player, int yearRetired)
    {
        Name = player;
        YearRetired = yearRetired;
    }
}
```

Йоги Берра играл под № 8 за «Нью-Йорк Янкиз», а Карл Рипкин-младший играл под тем же номером за «Балтимор Ориолз». В словаре могут встречаться одинаковые значения, но ключи должны быть уникальными. Попробуйте придумать способ хранения информации об игроках нескольких команд.

Ниже приведен класс Program с методом Main, который добавляет в словарь игроков, завершивших карьеру. Номер футболки может использоваться в качестве ключа словаря, потому что он **уникален**, — после того, как владелец этого номера уходит из спорта, команда *никогда не использует этот номер*. Это важный фактор, который должен учитываться при проектировании приложений, использующих словарь: вы никогда не должны оказаться в ситуации, когда вдруг выясняется, что ключи не настолько уникальны, как вы рассчитывали!

```
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        Dictionary<int, RetiredPlayer> retiredYankees = new Dictionary<int, RetiredPlayer>() {
            {3, new RetiredPlayer("Babe Ruth", 1948)},
            {4, new RetiredPlayer("Lou Gehrig", 1939)},
            {5, new RetiredPlayer("Joe DiMaggio", 1952)},
            {7, new RetiredPlayer("Mickey Mantle", 1969)},
            {8, new RetiredPlayer("Yogi Berra", 1972)},
            {10, new RetiredPlayer("Phil Rizzuto", 1985)},
            {23, new RetiredPlayer("Don Mattingly", 1997)},
            {42, new RetiredPlayer("Jackie Robinson", 1993)},
            {44, new RetiredPlayer("Reggie Jackson", 1993)},
        };

        foreach (int jerseyNumber in retiredYankees.Keys)
        {
            RetiredPlayer player = retiredYankees[jerseyNumber];
            Console.WriteLine($"{player.Name} #{jerseyNumber} retired in {player.
YearRetired}");
        }
    }
}
```

Используйте инициализатор коллекции для заполнения словаря объектами JerseyNumber.

Используйте цикл foreach для перебора по ключам и вывода строки для каждого игрока в коллекции.

ДРУГИЕ разновидности коллекций...

List и Dictionary — две самые популярные разновидности коллекций, входящие в .NET. Они чрезвычайно гибки — доступ к их данным осуществляется в произвольном порядке. Но иногда коллекция используется для представления сущностей реального мира, к которым необходимо обращаться в конкретном порядке. Для ограничения того, как ваш код обращается к данным коллекции, обычно используется **очередь** (Queue) или **стек** (Stack). Как и List<T>, они относятся к обобщенным коллекциям, а их главным преимуществом является обработка данных в определенном порядке.

← Также существуют другие разновидности коллекций, но чаще всего вам придется иметь дело именно с этими.

Используйте очередь (Queue), чтобы первый сохраненный объект обрабатывался первым, как в случае:

- ★ автомобилей, движущихся по улице с односторонним движением;
- ★ людей, стоящих в очереди;
- ★ клиентов, ожидающих обслуживания по телефону;
- ★ других ситуаций, когда первой обслуживается первая поступившая заявка.

↑
Очередь работает по принципу «первым вошел, первым вышел». То есть объект, первым помещенный в очередь, будет первым извлечен из очереди для обработки.

Используйте стек (Stack), когда первым всегда должен обрабатываться последний из сохраненных объектов, как в случае:

- ★ мебели, загруженной в кузов грузовика;
- ★ стопки книг, из которых вы хотите всегда сначала читать верхнюю;
- ★ людей, выходящих из самолета;
- ★ пирамиды из акробатов. Первым спрыгивать вниз должен тот, кто находится на самом верху. Только представьте, что произойдет, если кто-то уйдет из пирамиды снизу!

↑
Стек работает по принципу «последним вошел, первым вышел». То есть первый объект, помещенный в стек, последним покинет его.

Обобщенные коллекции .NET реализуют IEnumerable

Практически в каждом крупном проекте, над которым вы будете работать, используется та или иная разновидность обобщенной коллекции, потому что программы должны где-то хранить данные. Группы одинаковых объектов в реальном мире практически всегда можно объединить в категории, которые в той или иной степени соответствуют какой-то из коллекций. Независимо, какой из типов коллекций вы используете — List, Dictionary, Stack или Queue, вы всегда сможете использовать с ними цикл foreach, потому что все они реализуют IEnumerable<T>.

→ Цикл foreach позволяет осуществлять перебор в очереди и стеке, так как они реализуют интерфейс IEnumerable!

Очередь похожа на список, в котором объекты добавляются в конец, а чтение осуществляется от начала. Стек же предоставляет доступ только к тому объекту, который был в него добавлен последним.

Очередь работает по принципу FIFO — «первым вошел, первым вышел»

Очередь в целом похожа на список и отличается от списка прежде всего тем, что вы не можете добавлять и удалять элементы с произвольным индексом. Объект, помещаемый в очередь (**Enqueue**), добавляется в конец. При извлечении объекта из очереди (**Dequeue**) используется первый объект в начале очереди. В этом случае объект удаляется из очереди, а остальные объекты сдвигаются на одну позицию.

←
После первого вызова **Dequeue** первый элемент очереди удаляется и возвращается методом, а второй элемент сдвигается на его место.

// Создание очереди с добавлением четырех элементов

```
Queue<string> myQueue = new Queue<string>();  
myQueue.Enqueue("first in line");  
myQueue.Enqueue("second in line");  
myQueue.Enqueue("third in line");  
myQueue.Enqueue("last in line");
```

} Здесь метод **Enqueue** вызывается для добавления четырех элементов в очередь. При извлечении элементы будут возвращаться в том порядке, в каком они были добавлены.

// Метод **Peek** "подсматривает" первый элемент очереди и возвращает его без удаления

```
Console.WriteLine($"Peek() returned:\n{myQueue.Peek()}"); ①
```

// Метод **Dequeue** извлекает следующий элемент от НАЧАЛА очереди

```
Console.WriteLine(  
    $"The first Dequeue() returned:\n{myQueue.Dequeue()}"); ②
```

```
Console.WriteLine(  
    $"The second Dequeue() returned:\n{myQueue.Dequeue()}"); ③
```

// **Clear** удаляет все элементы из очереди

```
Console.WriteLine($"Count before Clear():\n{myQueue.Count}"); ④
```

```
myQueue.Clear();
```

```
Console.WriteLine($"Count after Clear():\n{myQueue.Count}"); ⑤
```



Объектам в очереди приходится ожидать своей очереди. Первый элемент, добавленный в очередь, станет первым элементом, который из нее выйдет.

Результат

```
Peek() returned:  
① first in line  
The first Dequeue() returned:  
② first in line  
The second Dequeue() returned:  
③ second in line  
Count before Clear():  
④ 2  
Count after Clear():  
⑤ 0
```

Стек работает по принципу LIFO — «последним вошел, первым вышел»

Стек очень похож на очередь — с одним принципиальным отличием. Пользователь **вносит** (Push) элементы в стек, а когда их потребуется получить, данные **извлекаются** (Pop) из стека. При извлечении элемента из стека вы получаете самый последний элемент, который был в него занесен. Стек напоминает стопку тарелок, журналов и т. д., — вы можете положить следующую тарелку на вершину, но ее необходимо снять, чтобы получить доступ к нижним тарелкам.

```
// Создание стека с добавлением четырех строк
Stack<string> myStack = new Stack<string>();
myStack.Push("first in line");
myStack.Push("second in line");
myStack.Push("third in line");
myStack.Push("last in line");
```

Стек создается так же, как и любая другая коллекция.

При занесении в стек новый элемент сдвигает другие элементы на одну позицию и остается на вершине.

```
// Peek со стеком работает так же, как с очередью
Console.WriteLine($"Peek() returned: \n{myStack.Peek()}"); ①
```

```
// Pop извлекает следующий элемент от КОНЦА стека
Console.WriteLine(
    $"The first Pop() returned: \n{myStack.Pop()}"); ②
Console.WriteLine(
    $"The second Pop() returned: \n{myStack.Pop()}"); ③
```

Извлекая элемент из стека, вы получаете элемент, который был добавлен последним.

```
Console.WriteLine($"Count before Clear(): \n{myStack.Count}"); ④
myStack.Clear();
Console.WriteLine($"Count after Clear(): \n{myStack.Count}"); ⑤
```

Результат

```
① Peek() returned:
last in line
The first Pop() returned:
② last in line
The second Pop() returned:
③ third in line
Count before Clear():
④ 2
Count after Clear():
⑤ 0
```

Последний элемент, добавленный в стек, станет первым элементом, который из него выйдет.





Погодите, я не поняла. А что можно сделать со стеком и очередью такого, чего нельзя сделать с List? Они всего лишь позволяют сделать код на пару строк короче, но я не имею доступа к средним элементам. Ведь то же самое можно без особых трудностей сделать со списком! Стоит ли ограничивать свои возможности ради минимальных удобств?

Вам не придется ни в чем себя ограничивать при работе со стеком или очередью.

Скопировать объект Queue в объект List очень легко. Так же легко, как скопировать объект List в объект Queue, Queue в объект Stack... более того, вы можете создать объекты List, Queue и Stack из любого другого объекта, реализующего интерфейс IEnumerable<T>. Достаточно воспользоваться перегруженным конструктором, который позволяет передавать копируемую коллекцию в параметре. Таким образом, в вашем распоряжении появляются гибкие и удобные средства для представления данных в виде коллекции, которая максимально соответствует вашим потребностям. (Но не забывайте, что при копировании создается новый объект, который будет занимать место в куче.)

Создадим стек с четырьмя элементами — в данном случае стек строк.

```
Stack<string> myStack = new Stack<string>();
myStack.Push("first in line");
myStack.Push("second in line");
myStack.Push("third in line");
myStack.Push("last in line");
```

Стек можно легко преобразовать в очередь, затем скопировать в список, а потом снова скопировать список в стек.

```
Queue<string> myQueue = new Queue<string>(myStack);
List<string> myList = new List<string>(myQueue);
Stack<string> anotherStack = new Stack<string>(myList);
```

```
Console.WriteLine($"myQueue has {myQueue.Count} items
myList has {myList.Count} items
anotherStack has {anotherStack.Count} items");
```

Все четыре элемента были скопированы в новые коллекции.

Результат

```
myQueue has 4 items
myList has 4 items
anotherStack has 4 items
```

...для обращения ко всем элементам очереди или стека достаточно воспользоваться циклом foreach!



Упражнение

Напишите программу, которая поможет хозяину кафе накормить лесорубов лепешками. Начните с очереди объектов Lumberjack (Лесоруб). Каждый объект Lumberjack содержит стек значений из перечисления Flapjack (Лепешка). Чтобы вам было проще начать, мы привели некоторые подробности. Сможете ли вы создать консольное приложение, которое выдает соответствующий результат?

Начните с класса Lumberjack и перечисления Flapjack

Класс Lumberjack содержит открытое свойство Name, значение которого задается в конструкторе, и приватное поле Stack<Flapjack> с именем flapjackStack, инициализируемое пустым стеком.

Метод TakeFlapjack получает единственный аргумент Flapjack и заносит его в стек. Метод EatFlapjacks извлекает лепешку из стека и выводит на консоль данные о лесорубе.

Lumberjack
Name private flapjackStack
TakeFlapjack EatFlapjacks

```
enum Flapjack {
    Crispy,
    Soggy,
    Brownd,
    Banana,
}
```

Добавьте метод Main

Метод Main запрашивает у пользователя имя первого лесоруба и количество лепешек. Если пользователь вводит допустимое число, программа вызывает метод TakeFlapjack заданное количество раз, каждый раз передавая случайный объект Flapjack, и добавляет объект Lumberjack в очередь. Программа продолжает запрашивать данные, пока пользователь не введет пустую строку, после чего в цикле while извлекает из очереди каждый объект Lumberjack и вызывает его метод EatFlapjacks для вывода данных на консоль.

Метод Main выводит эти строки и получает входные данные. Он создает каждый объект Lumberjack, задает его имя, выдает ему несколько лепешек из случайных категорий и добавляет в очередь.

Когда пользователь завершает ввод данных лесорубов, метод Main в цикле while извлекает из очереди каждого лесоруба и вызывает его метод EatFlapjacks. Остальные строки результата выводятся объектами Lumberjack.

Объект Lumberjack выводит эту строку, когда лесоруб начинает есть лепешки.

Этот лесоруб получил четыре лепешки. При вызове метода EatFlapjacks он извлек из стека четыре значения из перечисления Flapjack.

```
First lumberjack's name: Erik
Number of flapjacks: 4
Next lumberjack's name (blank to end): Hildur
Number of flapjacks: 6
Next lumberjack's name (blank to end): Jan
Number of flapjacks: 3
Next lumberjack's name (blank to end): Betty
Number of flapjacks: 4
Next lumberjack's name (blank to end):
Erik is eating flapjacks
Erik ate a soggy flapjack
Erik ate a brownd flapjack
Erik ate a brownd flapjack
Erik ate a soggy flapjack
Hildur is eating flapjacks
Hildur ate a brownd flapjack
Hildur ate a brownd flapjack
Hildur ate a crispy flapjack
Hildur ate a crispy flapjack
Hildur ate a soggy flapjack
Hildur ate a brownd flapjack
Jan is eating flapjacks
Jan ate a banana flapjack
Jan ate a crispy flapjack
Jan ate a soggy flapjack
Betty is eating flapjacks
Betty ate a soggy flapjack
Betty ate a brownd flapjack
Betty ate a brownd flapjack
Betty ate a crispy flapjack
```





Ниже приведен код класса Lumberjack и метода Main. Не забудьте включить строку using System.Collections.Generic; в начало класса.

```
class Lumberjack
{
    private Stack<Flapjack> flapjackStack = new Stack<Flapjack>();
    public string Name { get; private set; }

    public Lumberjack(string name)
    {
        Name = name;
    }

    public void TakeFlapjack(Flapjack flapjack)
    {
        flapjackStack.Push(flapjack);
    }

    public void EatFlapjacks() {
        Console.WriteLine($"{Name} is eating flapjacks");
        while (flapjackStack.Count > 0)
        {
            Console.WriteLine(
                $"{Name} ate a {flapjackStack.Pop().ToString().ToLower()} flapjack");
        }
    }
}
```

←
Стек со значениями из перечисления Flapjack. Он заполняется вызовами TakeFlapjack со случайными лепешками и опустошается при вызове метода EatFlapjacks.

←
Метод TakeFlapjack просто заносит объект Flapjack в стек.



```
class Program
{
    static void Main(string[] args)
    {
        Random random = new Random();
        Queue<Lumberjack> lumberjacks = new Queue<Lumberjack>();

        string name;
        Console.Write("First lumberjack's name: ");
        while ((name = Console.ReadLine()) != "") {
            Console.Write("Number of flapjacks: ");
            if (int.TryParse(Console.ReadLine(), out int number))
            {
                Lumberjack lumberjack = new Lumberjack(name);
                for (int i = 0; i < number; i++)
                {
                    lumberjack.TakeFlapjack((Flapjack)random.Next(0, 4));
                }
                lumberjacks.Enqueue(lumberjack);
            }
            Console.Write("Next lumberjack's name (blank to end): ");
        }

        while (lumberjacks.Count > 0)
        {
            Lumberjack next = lumberjacks.Dequeue();
            next.EatFlapjacks();
        }
    }
}
```

↑
Метод Main сохраняет свою ссылку Lumberjack в очереди.

←
Создает каждый объект Lumberjack, вызывает его метод TakeFlapjack со случайными лепешками, после чего помещает ссылку в очередь.

←
Когда пользователь завершит раздачу лепешек, метод Main в цикле while извлекает каждую ссылку Lumberjack из очереди и вызывает для нее метод EatFlapjacks.

В: Что произойдет, если я попытаюсь получить из словаря объект с несуществующим ключом?

О: Если передать словарю несуществующий ключ, будет выдано исключение. Например, если включить в консольное приложение следующий фрагмент:

```
Dictionary<string, string> dict =  
    new Dictionary<string, string>();  
string s = dict["This key doesn't exist"];
```

вы получите исключение «System.Collections.Generic.KeyNotFoundException: 'ключ 'This key doesn't exist' отсутствует в словаре». Для удобства в исключение входит ключ, а конкретнее, строка, которую возвращает метод ToString ключа. Это очень полезно, если вы пытаетесь отладить в программе проблему, которая много тысяч раз обращается к словарю.

В: Можно ли избежать этого исключения — например, если я заранее не знаю, существует ли ключ в словаре?

О: Да, избежать исключения KeyNotFoundException можно двумя способами. Во-первых, вы можете воспользоваться методом Dictionary.ContainsKey. Методу передается ключ, который вы хотите использовать со словарем; он возвращает true только в том случае, если ключ существует. Во-вторых, можно воспользоваться методом Dictionary.TryGetValue:

```
if (dict.TryGetValue("Key", out string value))  
{  
    // что-то происходит  
}
```

Этот код полностью эквивалентен следующему:

```
if (dict.ContainsKey("Key"))  
{  
    string value = dict["Key"];  
    // do something  
}
```

КЛЮЧЕВЫЕ МОМЕНТЫ

- Списки, массивы и другие коллекции реализуют интерфейс **IEnumerable<T>**, который поддерживает перебор необобщенных коллекций.
- Цикл **foreach** работает с любыми классами, реализующими **IEnumerable<T>**. Интерфейс включает метод, возвращающий объект **Enumerator**, который позволяет циклу по порядку перебрать содержимое.
- **Ковариантность** представляет собой механизм **C#** для неявного преобразования ссылки на subclass в ссылку на его суперкласс.
- Термином «неявный» обозначается способность **C#** автоматически определить, как должно выполняться преобразование, без выполнения явного приведения типа.
- Ковариантность особенно полезна при передаче коллекции объектов методу, который работает только с классом, от которого они наследуют. Например, ковариантность позволяет использовать **обычное присваивание для повышающего приведения типа List<Subclass>** в **IEnumerable<Superclass>**.
- Словарь **Dictionary<TKey, TValue>** — коллекция для хранения пар «ключ/значение», предоставляющая средства получения значения, ассоциированного с заданным ключом.
- Ключи и значения словарей могут относиться к **разным типам**. Каждый **ключ должен быть уникальным** в словаре, но значения могут повторяться.
- Класс **Dictionary** содержит **свойство Keys**, которое возвращает последовательность ключей с возможностью перебора.
- **Queue<T>** — коллекция, работающая по принципу «первым вошел, первым вышел»; содержит методы для постановки элемента в конец очереди и извлечения элемента от начала очереди.
- **Stack<T>** — коллекция, работающая по принципу «последним вошел, первым вышел»; содержит методы для занесения элемента на вершину стека и извлечения элемента с вершины стека.
- Классы **Stack<T>** и **Queue<T>** реализуют интерфейс **IEnumerable<T>** и могут быть легко преобразованы в **List** и другие разновидности коллекций.

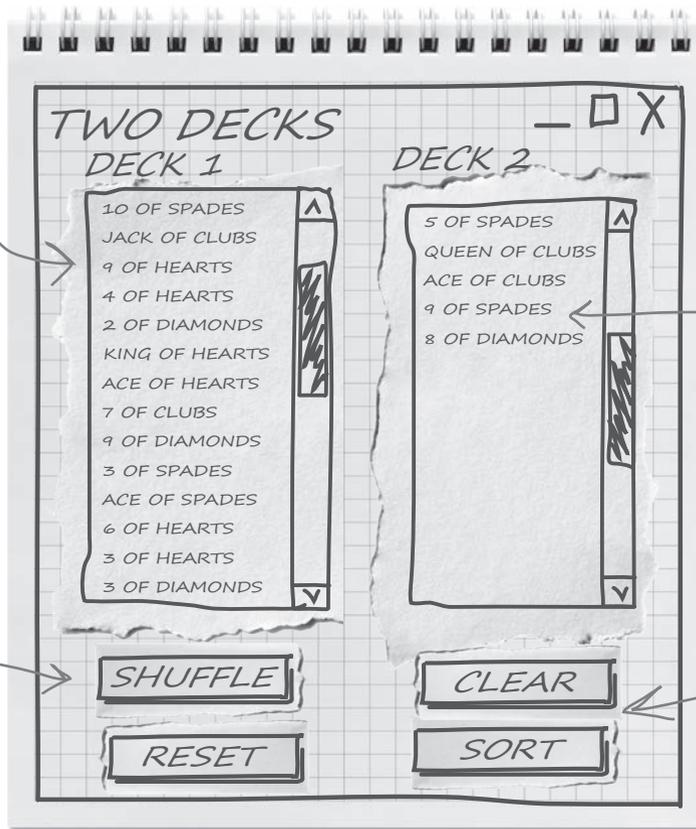


Упражнение: две колоды

В следующем упражнении мы построим приложение для перемещения карт между двумя колодами. Кнопки левой колоды позволяют перетасовать колоду и вернуть ее к исходному состоянию с 52 картами, а кнопки правой колоды удаляют из колоды все карты и сортируют ее.

При запуске приложения в левом поле содержится полная колода карт. Правое поле пусто.

Кнопка Shuffle тасует колоду 1, а кнопка Reset возвращает ее к исходному состоянию с 52 отсортированными картами.



Двойной щелчок на карте в колоде переводит ее в другую колоду. Таким образом, двойной щелчок на 9 пик удалит ее из колоды 2 и добавит в колоду 1.

Кнопка Clear удаляет все карты из колоды 2, а кнопка Sort сортирует карты, чтобы они следовали по порядку.

Одна из самых важных идей, которые мы неоднократно подчеркивали в книге: написание кода C# является профессиональным навыком, а лучший способ совершенствования этого навыка — **практика**. Мы хотим предоставить вам как можно больше возможностей для практики!

Именно поэтому мы создали **дополнительные проекты Windows WPF и macOS ASP.NET Core Blazor** для некоторых глав в оставшейся части книги. Мы также включили эти проекты в конец нескольких оставшихся глав. Загрузите PDF-файл с описанием проекта — мы считаем, что вам стоит заняться им перед тем, как переходить к следующей главе, потому что это поможет вам усвоить некоторые важные концепции, способствующие закреплению материала оставшихся глав книги.

Перейдите к репозиторию **GitHub** данной книги и загрузите PDF-файл проекта:

<https://github.com/head-first-csharp/fourth-edition>

Лабораторный курс Unity № 4

Пользовательские интерфейсы

В предыдущей лабораторной работе Unity вы начали строить игру. Мы использовали заготовку для создания экземпляров `GameObject`, которые появлялись в случайных точках трехмерного пространства игры и летали по кругу. В этой лабораторной работе мы продолжим с того места, на котором остановились в предыдущей главе; в ней вы сможете применить то, что узнали об интерфейсах `C#`, и многое другое.

До настоящего момента наша программа была интересной визуальной моделью. В этой лабораторной работе Unity мы **завершим построение игры**. При запуске счет игрока равен 0. Бильярдные шары появляются в случайных точках и летают по экрану. Когда игрок щелкает на шаре, счет увеличивается на 1 и шар исчезает. Далее появляются новые шары; если на экране появятся 15 шаров одновременно, игра завершается. Чтобы игра нормально работала, необходимо дать возможность игроку запустить ее; начать игру заново после ее завершения; а также выводить счет при щелчках на шарах. Для этого мы добавим **пользовательский интерфейс**, который будет выводить текущий счет в углу экрана, а также кнопку для начала новой игры.

Вывод текущего счета

Модель с летающими шарами выглядит довольно интересно, пришло время преобразовать ее в игру. **Добавьте новое поле** в класс `GameController` для отслеживания текущего счета — вы можете добавить его под полем `OneBallPrefab`:

```
public int Score = 0;
```

Затем **добавьте в класс `GameController` метод с именем `ClickedOnBall`**. Метод будет вызываться каждый раз, когда игрок щелкает на шаре:

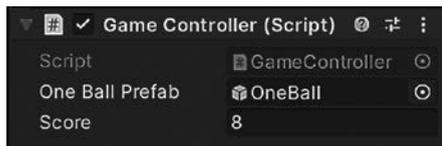
```
public void ClickedOnBall()
{
    Score++;
}
```

Unity предоставляет возможность объектам `GameObject` легко реагировать на щелчки мышью и другие способы ввода. Если вы добавите в сценарий метод `OnMouseDown`, Unity будет вызывать этот метод каждый раз, когда вы щелкаете на объекте `GameObject`, к которому он был присоединен. **Добавьте следующий метод к классу `OneBallBehavior`**:

```
void OnMouseDown()
{
    GameController controller = Camera.main.GetComponent<GameController>();
    controller.ClickedOnBall();
    Destroy(gameObject);
}
```

Первая строка `OnMouseDown` получает экземпляр класса `GameController`, а вторая строка вызывает метод `ClickedOnBall`, который увеличивает значение поля `Score`.

Запустите игру. Щелкните на строке `Main Camera` в иерархии и понаблюдайте за компонентом `GameController (Script)` в окне `Inspector`. Щелкните на вращающихся шарах — они исчезают из сцены, а текущий счет увеличивается.



Часть Задаваемые Вопросы

В: Почему мы используем метод `Instantiate` вместо ключевого слова `new`?

О: `Instantiate` и `Destroy` — *специальные методы, уникальные для Unity*. Вы не встретите их в других проектах `C#`. Метод `Instantiate` несколько отличается от ключевого слова `C# new`, потому что он создает новый экземпляр заготовки, а не класса. Unity создает новые экземпляры объектов, но также выполняет ряд других операций (например, включение объекта в цикл обновления). Когда сценарий объекта `GameObject` вызывает `Destroy(gameObject)`, он тем самым приказывает Unity уничтожить себя. Метод `Destroy` приказывает Unity уничтожить объект `GameObject` — но только после завершения цикла обновления.

В: Я так и не понял, как работает первая строка метода `OnMouseDown`. Что здесь происходит?

О: Разобьем команду на части. Первая часть выглядит вполне знакомо: она объявляет переменную с именем `controller` типа `GameController` — класса, определенного вами в сценарии, присоединенном к главной камере. Во второй части требуется вызвать метод для объекта `GameController`, присоединенного к главной камере. Соответственно, мы используем `Camera.main` для получения объекта главной камеры и `GetComponent<GameController>()` для получения экземпляра `GameController`, присоединенного к нему.

Включение двух режимов в игру

Запустите свою любимую игру. Вы немедленно оказываетесь в гуще событий? Наверное, нет, — скорее всего, сначала откроется начальное меню. Некоторые игры позволяют игроку приостановить действие, чтобы взглянуть на карту. Многие игры позволяют переключаться между перемещением игрока и работой с инвентарем или в случае гибели игрока выводят анимацию, которую невозможно прервать. Все это примеры разных **игровых режимов**.

Добавим два режима в игру с бильярдными шарами:

- * **Режим № 1:** игра работает. Шары добавляются в сцену; при щелчке шар исчезает, а счет увеличивается.
- * **Режим № 2:** игра завершена. Шары перестают добавляться в сцену, при щелчках ничего не происходит, а на экране появляется надпись «Game over».



На этом снимке экрана показана игра в рабочем режиме. Шары добавляются в сцену, а игрок может щелкать на них, чтобы получать очки.



Когда на экране появится последний шар, игра переходит в режим завершения. На экране появляется кнопка Play Again, и новые шары перестают появляться.

В игру будут добавлены два режима. Основной «игровой» режим уже реализован, так что остается добавить режим завершения игры.

Чтобы добавить в игру два режима, выполните следующие действия:

- ① **Включите в метод `GameController.AddABall` поддержку текущего игрового режима**
Новый усовершенствованный метод `AddABall` проверяет, не завершена ли игра, и создает новый экземпляр заготовки `OneBall` только в том случае, если игра продолжается.
- ② **Обработчик `OneBallBehaviour.OnMouseDown` должен работать только в игровом режиме.**
Если игра закончена, она должна перестать реагировать на щелчки. На экране должны остаться только ранее добавленные шары, пока игра не будет перезапущена.
- ③ **Метод `GameController.AddABall` должен завершить игру, если на экране окажется слишком много шаров.**
`AddABall` также увеличивает счетчик `NumberOfBalls`, который повышается на 1 при каждом добавлении шара. Если значение достигает `MaximumBalls`, переменной `GameOver` присваивается `true` для завершения игры.

В этой лабораторной работе мы строим игру по частям и вносим пошаговые изменения. Код каждой части можно загрузить из репозитория GitHub книги: <https://github.com/head-first-csharp/fourth-edition>.

Добавление игрового режима

Внесите изменения в классы `GameController` и `OneBallBehaviour`, чтобы добавить новые режимы в игру. Для отслеживания завершения игры будет добавлено логическое поле.

1 Включите в метод `GameController.AddABall` поддержку игрового режима.

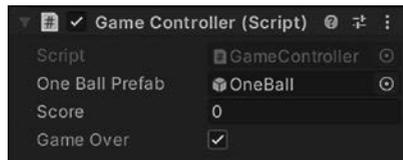
Объект `GameController` должен знать, в каком режиме находится игра. Для отслеживания информации, доступной объекту, используются поля. Так как игра может находиться в двух режимах — игровом и завершённом, для хранения режима хватит логического поля. Добавьте поле **GameOver** в класс `GameController`:

```
public bool GameOver = false;
```

Новые шары должны добавляться в сцену только в том случае, если игра работает. Измените метод `AddABall` и добавьте команду `if`, чтобы метод `Instantiate` вызывался только в том случае, если `GameOver` не содержит `true`:

```
public void AddABall()
{
    if (!GameOver)
    {
        Instantiate(OneBallPrefab);
    }
}
```

Протестируйте внесенные изменения. Запустите игру и щелкните на объекте **Main Camera** в окне **Hierarchy**.



← Установите флажок `Game Over` во время выполнения игры, чтобы переключить состояние поля объекта `GameController`. Если установить его во время игры, Unity сбросит значение при ее остановке.

Чтобы задать значение поля `GameOver`, снимите флажок в компоненте `Script`. Игра перестает добавлять шары, пока флажок не будет снова установлен.

2 Обработчик `OneBallBehaviour.OnMouseDown` должен работать только в игровом режиме.

Метод `OnMouseDown` уже вызывает метод `ClickedOnBall` класса `GameController`. Теперь измените метод **OnMouseDown** в `OneBallBehaviour`, чтобы он также использовал поле `GameOver` класса `GameController`:

```
void OnMouseDown()
{
    GameController controller = Camera.main.GetComponent<GameController>();
    if (!controller.GameOver)
    {
        controller.ClickedOnBall();
        Destroy(gameObject);
    }
}
```

Снова запустите игру и убедитесь в том, что шары исчезают, а счет увеличивается только в том случае, если игра не закончена.

3

Метод `GameController.AddABall` должен завершить игру, если на экране окажется слишком много шаров.

Игра должна каким-то образом отслеживать количество шаров в сцене. Для этого мы **добавим два поля** в класс `GameController` для хранения текущего и максимального количества шаров:

```
public int NumberOfBalls = 0;
public int MaximumBalls = 15;
```

Каждый раз, когда игрок щелкает на шаре, сценарий `OneBallBehaviour` шара вызывает `GameController.ClickedOnBall` для инкрементирования (увеличения на 1) счета. Также должно декрементироваться (уменьшаться на 1) значение `NumberOfBalls`:

```
public void ClickedOnBall()
{
    Score++;
    NumberOfBalls--;
}
```

Измените метод `AddBall`, чтобы шары добавлялись только во время игры, а при слишком большом количестве шаров в сцене игра заканчивалась:

```
public void AddABall()
{
    if (!GameOver)
    {
        Instantiate(OneBallPrefab);
        NumberOfBalls++;
        if (NumberOfBalls >= MaximumBalls)
        {
            GameOver = true;
        }
    }
}
```

Поле `GameOver` содержит `true`, если игра закончена, и `false` во время игры. В поле `NumberOfBalls` хранится текущее количество шаров в сцене. Когда оно достигает значения `MaximumBalls`, `GameController` присваивает `GameOver` значение `true`.

Снова протестируйте игру: запустите ее и щелкните на объекте `Main Camera` в окне `Hierarchy`. Игра должна работать как обычно, но как только поле `NumberOfBalls` достигнет значения `MaximumBalls`, метод `AddABall` присваивает полю `GameOver` значение `true` и игра завершается.



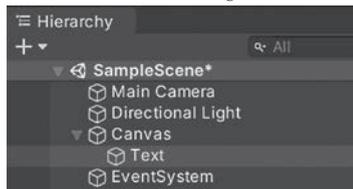
Когда это произойдет, щелчки на шарах ни к чему не приводят, потому что метод `OneBallBehaviour.OnMouseDown` проверяет поле `GameOver` и увеличивает счет/уничтожает шар только в том случае, если `GameOver` содержит `false`.

Ваша игра должна отслеживать текущий игровой режим. Поля хорошо подходят для этой цели.

Добавление пользовательского интерфейса к игре

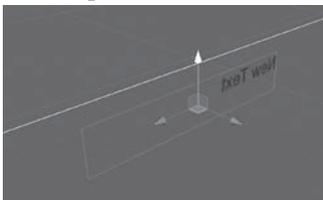
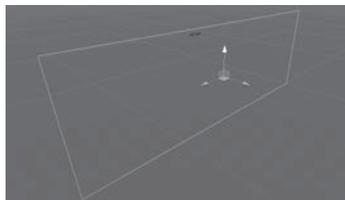
Почти в каждой игре, которую только можно представить, — от Pac-Man до Super Mario Brothers, от Grand Theft Auto 5 до Minecraft — присутствует **пользовательский интерфейс** (UI). Некоторые игры — скажем, Pac-Man — ограничиваются очень простым пользовательским интерфейсом, который выводит на экран текущий счет, рекорд, количество оставшихся жизней и номер уровня. Во многих играх реализован хитроумный пользовательский интерфейс, встроенный в игровую механику (например, колесо оружия, которое позволяет игроку быстро переключаться между разными видами оружия). Добавим пользовательский интерфейс в нашу игру.

Выберите команду UI>>Text из меню **GameObject**, чтобы добавить в пользовательский интерфейс игры объект **GameObject 2D Text**. В окне Hierarchy появляется объект **Canvas**, а под ним — объект **Text**:



Когда вы добавляете объект **Text** в сцену, Unity автоматически добавляет объекты **GameObject Canvas** и **Text**. Щелкните на кнопке с треугольником (▼) рядом с объектом **Canvas**, чтобы раскрыть или свернуть его, — объект **GameObject Text** появляется и исчезает, потому что он **вложен** в **Canvas**.

Сделайте двойной щелчок на объекте **Canvas** в окне Hierarchy, чтобы выделить его. Объект представляет собой 2D-прямоугольник. Щелкните на его манипуляторе **Move** и перетащите в сцене. Не двигается! Добавленный объект **Canvas** всегда отображается, масштабируется по размерам экрана и располагается перед всеми остальными объектами в игре.

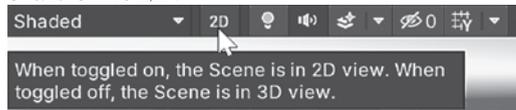


Затем сделайте двойной щелчок на объекте **Text**, чтобы выделить его. Он увеличивается в редакторе, но текст по умолчанию («New Text») записан в обратном направлении, потому что главная камера направлена на **Canvas** сзади.

Вы обратили внимание на объект **EventSystem** в окне Hierarchy? Unity автоматически добавляет его при создании пользовательского интерфейса. **EventSystem** управляет мышью, клавиатурой и другими источниками ввода и отправляет информацию объектам **GameObject** — все это происходит автоматически, так что вам не придется напрямую работать с ним.

Использование 2D-представления для работы с Canvas

Кнопка **2D** в верхней части окна Scene включает и отключает представление 2D:



Включите 2D-представление — редактор отображает 2D-элементы на сцене. Сделайте двойной щелчок на объекте **Text** в окне Hierarchy, чтобы приблизить камеру к тексту.



Используйте колесо мыши, чтобы увеличивать и уменьшать масштаб в 2D-представлении.

Canvas (холст) представляет собой двумерный объект **GameObject** для размещения частей пользовательского интерфейса игры. В нашей игре **Canvas** содержит два вложенных объекта: только что добавленный объект **GameObject Text**, который выводит счет в правом верхнем углу, и объект **GameObject Button** для запуска новой игры.

Кнопка **2D** используется для переключения между 2D- и 3D-представлениями. Снова щелкните на ней, чтобы вернуться к 3D-представлению.

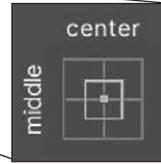
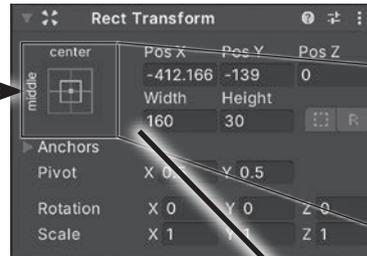
Настройка объекта Text для вывода счета в UI

Пользовательский интерфейс игры состоит из объектов GameObject Text и Button. Каждый из этих объектов GameObject **привязывается** к отдельной части пользовательского интерфейса. Например, объект GameObject Text для вывода счета будет отображаться в правом верхнем углу экрана (независимо от того, насколько большим или маленьким будет экран).

Объект Text привязывается к конкретной точке 2D Canvas.

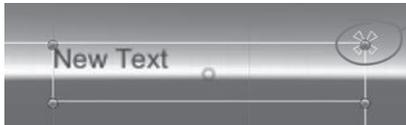
Щелкните на объекте Text в окне Hierarchy, чтобы выделить его, а затем просмотрите данные компонента Rect Transform. Объект Text должен располагаться в правом верхнем углу, поэтому **щелкните на поле Anchors** на панели Rect Transform.

Так как Text «живет» только внутри объекта 2D Canvas, он использует компонент **Rect Transform** (его позиция задается относительно прямоугольника Canvas). Щелкните на поле **Anchors**, чтобы вывести предварительные значения параметров привязки.



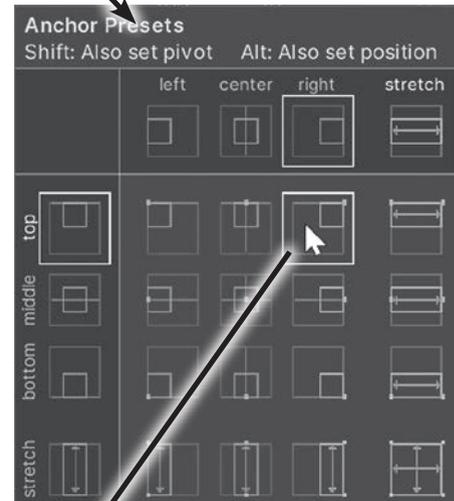
Удерживайте нажатыми клавиши Shift и Alt (Option на Mac), чтобы задать как ось вращения, так и позицию.

Окно Anchor Presets позволяет привязать объекты GameObject, составляющие пользовательский интерфейс, к различным частям Canvas. **Удерживайте нажатыми клавиши Alt и Shift** (Option+Shift на Mac) и выберите правую верхнюю заготовку привязки. Щелкните на той же кнопке, которую вы использовали для открытия окна Anchor Presets. Теперь объект Text оказывается в правом верхнем углу Canvas — сделайте на нем двойной щелчок, чтобы увеличить его.

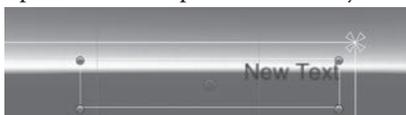


Ось вращения устанавливается в правом верхнем углу. Позиция Text определяется позицией якоря привязки относительно Canvas.

Добавим немного свободного места выше и справа от Text. Вернитесь к панели Rect Transform и **присвойте Pos X и Pos Y значение -10**, чтобы текст располагался на 10 единиц левее и на 10 единиц ниже правого верхнего угла. **Выберите в поле Alignment компонента Text выравнивание по правому краю** и при помощи поля в верхней части окна Inspector замените имя объекта GameObject на Score.



Новый объект Text должен отображаться в окне Hierarchy с именем Score. Текст выравнивается по правому краю, а между краем Text и краем Canvas существует небольшой отступ.



Кнопка для вызова метода, запускающего игру

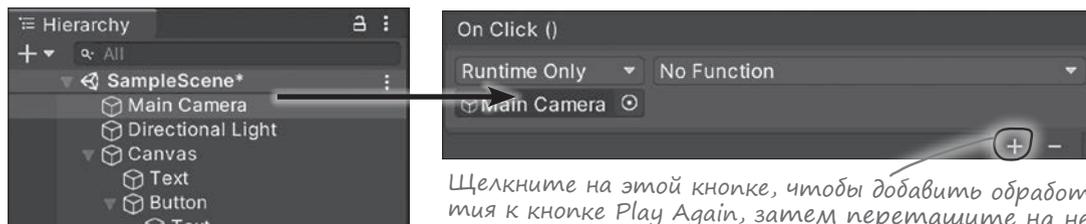
Когда игра находится в режиме завершения, на экране появляется кнопка Play Again; она вызывает метод для перезапуска игры. **Добавьте пустой метод StartGame** в класс GameController (код будет добавлен позднее):

```
public void StartGame()
{
    // Код этого метода будет добавлен позднее
}
```

Щелкните на объекте Canvas в окне Hierarchy, чтобы выделить его. **Выберите команду UI>>Button** в меню объекта GameObject, чтобы добавить кнопку Button. Так как объект Canvas уже выделен, редактор Unity добавит новый объект Button с привязкой его к центру Canvas. А вы заметили, что рядом с объектом Button в окне Hierarchy располагается кнопка с треугольником? Она открывает вложенный объект GameObject Text. Щелкните на нем и введите текст Play Again.

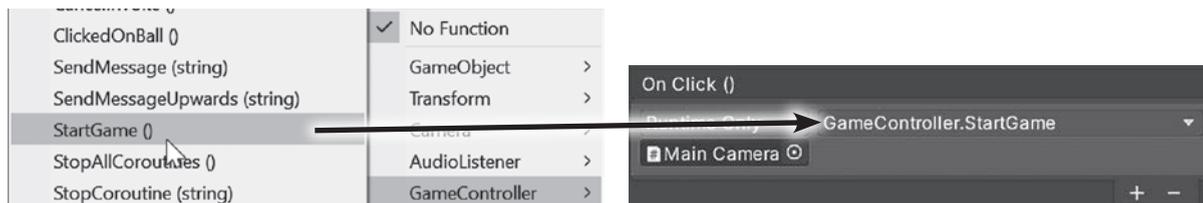


Итак, кнопка Button настроена, и теперь нужно заставить ее вызывать метод StartGame объекта GameController, присоединенного к главной камере Main Camera. Кнопка UI представляет собой **объект GameObject с компонентом Button**, и вы можете воспользоваться полем On Click() в окне Inspector для связывания ее с методом-обработчиком события. Щелкните на кнопке **+** в нижней части поля On Click(), чтобы добавить обработчик события, затем **перетащите Main Camera на поле None (Object)**.



Щелкните на этой кнопке, чтобы добавить обработчик события к кнопке Play Again, затем перетащите на нее объект Main Camera.

Теперь кнопка знает, какой объект GameObject следует использовать для обработчика события. Щелкните на раскрывающемся списке **No Function** и выберите команду **GameController>>StartGame**. Теперь при нажатии кнопки игроком будет вызываться метод StartGame для объекта GameController, связанного с Main Camera.



Кнопка Play Again и текущий счет

Пользовательский интерфейс вашей игры работает по следующей схеме:

- ★ При запуске игра переходит в режим завершения.
- ★ Кнопка Play Again запускает игру.
- ★ Объект Text в правом верхнем углу экрана используется для вывода текущего счета.

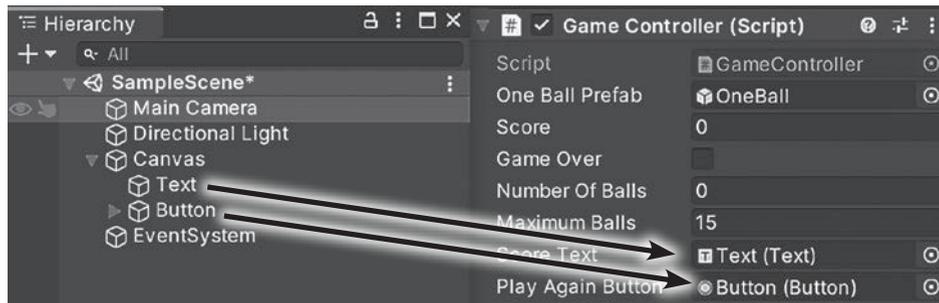
В коде будут использоваться классы Text и Button. Оба класса принадлежат пространству имен UnityEngine.UI, поэтому не забудьте **добавить директиву using** в начало класса GameController:

```
using UnityEngine.UI;
```

Теперь можно добавить поля Text и Button в класс GameController (непосредственно над полем OneBallPrefab):

```
public Text ScoreText;
public Button PlayAgainButton;
```

Щелкните на объекте Main Camera в окне Hierarchy. Перетащите объект GameObject Text из окна Hierarchy на поле Score Text компонента Script, после чего перетащите объект GameObject Button на поле Button.



Вернитесь к коду GameController и задайте значение по умолчанию для поля GameOver равным true:

```
public bool GameOver = true; ← Замените false на true.
```

Теперь вернитесь в Unity и проверьте компонент Script в окне Inspector.

Стоп, что-то не так!

Game Over

В редакторе Unity флажок Game Over снят — значение поля не изменилось. Установите его, чтобы игра запускаясь в режиме завершения:

Game Over

Игра запустится в режиме завершения, а игрок может щелкнуть на кнопке Play Again, чтобы начать игру.



Будьте осторожны!

Unity запоминает значения полей ваших сценариев.

Когда вы захотели изменить значение поля GameController.GameOver с false на true, внести изменения в код было недостаточно. Когда вы добавляете компонент Script, Unity отслеживает значения полей и не перезагружает значения по умолчанию, пока не будет выполнен сброс из контекстного меню (☰).

Завершение кода игры

Объект GameController, присоединенный к главной камере, отслеживает текущее состояние счета в поле Score. **Добавьте метод Update в класс GameController**, чтобы обновить текст счета в UI:

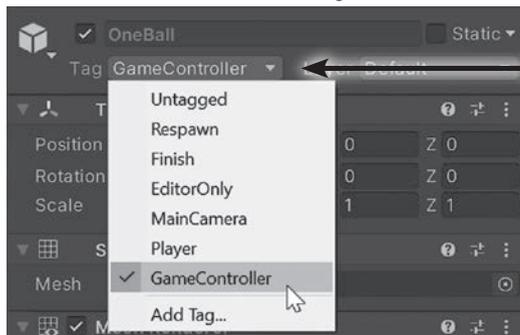
```
void Update()
{
    ScoreText.text = Score.ToString();
}
```

Затем **измените метод GameController.AddABall**, чтобы кнопка снова становилась доступной в конце игры:

```
if (NumberOfBalls >= MaximumBalls)
{
    GameOver = true;
    PlayAgainButton.gameObject.SetActive(true);
}
```

Каждый объект GameObject содержит свойство с именем gameObject для манипуляций с этим объектом. При помощи его метода SetActive можно делать кнопку Play Again видимой или невидимой.

Осталось сделать еще одно: наладить работу метода StartGame, чтобы он запускал игру. Метод должен решать несколько задач: уничтожить шары, которые в настоящее время летают в сцене, заблокировать кнопку Play Again, сбросить счет и количество шаров и переключиться в игровой режим. Вы уже знаете, как решить большинство из этих задач! Чтобы уничтожить шары, необходимо их найти. **Щелкните на заготовке OneBall в окне Project и задайте для нее тег:**



Тег — ключевое слово, которое можно присоединить к любому объекту GameObject; тег может использоваться в коде для идентификации или поиска объектов. Когда вы щелкаете на заготовке в окне Project и используете раскрывающийся список для назначения тега, этот тег будет назначен каждому созданному экземпляру заготовки.

У вас есть все необходимое для того, чтобы заполнить код метода StartGame. Он использует цикл foreach для нахождения и уничтожения всех шаров, оставшихся от предыдущей игры, скрывает кнопку, сбрасывает счет и количество шаров и изменяет режим игры:

```
public void StartGame()
{
    foreach (GameObject ball in GameObject.FindGameObjectsWithTag("GameController"))
    {
        Destroy(ball);
    }
    PlayAgainButton.gameObject.SetActive(false);
    Score = 0;
    NumberOfBalls = 0;
    GameOver = false;
}
```

Запустите игру. Она запускается в режиме завершения. Нажмите кнопку, чтобы начать игру. Счет увеличивается каждый раз, когда вы щелкаете на шаре. Как только будет создан 15-й экземпляр шара, игра завершается и кнопка Play Again появляется снова.



Упражнение

Пора потренироваться в программировании для Unity! Каждый объект `GameObject` содержит метод `transform.Translate`, перемещающий его на заданное расстояние от текущей позиции. Цель этого упражнения — изменить игру так, чтобы вместо `transform.RotateAround` для вращения шаров по оси `Y` сценарий `OneBallBehaviour` использовал `transform.Translate`, чтобы шары случайным образом перемещались по сцене.

- Удалите поля `XRotation`, `YRotation` и `ZRotation` из `OneBallBehaviour`. **Замените их полями** для хранения скорости по осям `X`, `Y` и `Z` с именами `XSpeed`, `YSpeed` и `ZSpeed`. Поля относятся к типу `float`, задавать их значения не обязательно.
- Замените весь код метода `Update`** следующей строкой с вызовом метода `transform.Translate`:

```
transform.Translate(Time.deltaTime * XSpeed,
                  Time.deltaTime * YSpeed, Time.deltaTime * ZSpeed);
```

Параметры представляют скорость перемещения шара по осям `X`, `Y` и `Z`. Таким образом, если значение `XSpeed` равно 1.75, после умножения на `Time.deltaTime` шар будет перемещаться по оси `X` со скоростью 1.75 единицы в секунду.

- Замените поле `DegreesPerSecond`** полем `Multiplier` со значением 0.75F — **суффикс F важен!** Используйте его для обновления поля `XSpeed` в методе `Update`, затем **добавьте две аналогичные строки** для полей `YSpeed` и `ZSpeed`:
`XSpeed += Multiplier - Random.value * Multiplier * 2;`

В этом упражнении очень важно **понять, как работает эта строка кода**. `Random.value` — статический метод, который возвращает случайное число с плавающей точкой от 0 до 1. Что делает эта строка кода с полем `XSpeed`?

.....
.....
.....

- Затем **добавьте метод с именем `ResetBall`** и вызовите его из метода `Start`.

Добавьте следующую строку в метод `ResetBall`:

```
XSpeed = Multiplier - Random.value * Multiplier * 2;
```

Что делает эта строка кода?

↑
*Прежде чем продолжать,
попробуйте определить, что
делают эти строки кода.*
↓

.....
.....

Добавьте в `ResetBall` еще две аналогичные строки для обновления `YSpeed` и `ZSpeed`. Затем переместите строку кода, обновляющую `transform.position`, **из метода `Start`** в метод `ResetBall`.

- Измените класс `OneBallBehaviour`, **добавьте в него поле `TooFar`** и присвойте ему значение 5. Измените метод `Update`, чтобы он проверял, не зашел ли шар слишком далеко. Для проверки дальности шара по оси `X` используется следующая команда:

```
Mathf.Abs(transform.position.x) > TooFar
```

Команда проверяет *абсолютное значение* позиции `X`; иначе говоря, она определяет, что `transform.position.x` больше `5F` или меньше `-5F`. Следующая команда `if` проверяет, что шар зашел слишком далеко по оси `X`, `Y` или `Z`:

```
if ((Mathf.Abs(transform.position.x) > TooFar)
    || (Mathf.Abs(transform.position.y) > TooFar)
    || (Mathf.Abs(transform.position.z) > TooFar)) {
```

Измените метод `OneBallBehaviour.Update`, чтобы в нем использовалась команда `if` для вызова `ResetBall`, если шар зашел слишком далеко.



Упражнение Решение

Ниже приведен код класса OneBallBehaviour после обновления по инструкциям из упражнения. Ключевое место в работе игры занимает то, что скорость каждого шара по осям X, Y и Z определяется его текущими значениями XSpeed, YSpeed и ZSpeed. Внося небольшие изменения в эти значения, вы заставляете шар случайно перемещаться по сцене.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class OneBallBehaviour : MonoBehaviour
{
    public float XSpeed;
    public float YSpeed;
    public float ZSpeed;
    public float Multiplier = 0.75F;
    public float TooFar = 5;

    static int BallCount = 0;
    public int BallNumber;

    // Start вызывается перед первым обновлением кадра
    void Start()
    {
        BallCount++;
        BallNumber = BallCount;

        ResetBall();
    }

    // Update вызывается один раз на кадр
    void Update()
    {
        transform.Translate(Time.deltaTime * XSpeed,
                            Time.deltaTime * YSpeed, Time.deltaTime * ZSpeed);

        XSpeed += Multiplier - Random.value * Multiplier * 2;
        YSpeed += Multiplier - Random.value * Multiplier * 2;
        ZSpeed += Multiplier - Random.value * Multiplier * 2;

        if ((Mathf.Abs(transform.position.x) > TooFar)
            || (Mathf.Abs(transform.position.y) > TooFar)
            || (Mathf.Abs(transform.position.z) > TooFar))
        {
            ResetBall();
        }
    }
}
```

Вы добавили эти поля в класс OneBallBehaviour. Не забудьте добавить суффикс F в 0.75F, в противном случае программа строиться не будет.

← При создании экземпляра шара его метод Start вызывает ResetBall для определения случайной позиции и скорости.

← Метод Update сначала перемещает шар, затем обновляет скорость и, наконец, проверяет, не вышел ли шар за границы допустимой области. Если вы выполняете эти операции в другом порядке, это нормально.



Упражнение
Решение

```
void ResetBall()
{
    XSpeed = Random.value * Multiplier;
    YSpeed = Random.value * Multiplier;
    ZSpeed = Random.value * Multiplier;

    transform.position = new Vector3(3 - Random.value * 6,
        3 - Random.value * 6, 3 - Random.value * 6);
}

void OnMouseDown()
{
    GameController controller = Camera.main.GetComponent<GameController>();
    if (!controller.GameOver)
    {
        controller.ClickedOnBall();
        Destroy(gameObject);
    }
}
```

Сброс параметров шара происходит при создании экземпляра или при выходе за границы допустимой области; при этом шару присваивается случайная скорость и позиция. Если вы сначала назначаете позицию, это нормально.

Ниже приводятся наши ответы на вопросы — а ваши ответы похожи на них?

```
XSpeed += Multiplier - Random.value * Multiplier * 2;
```

Что эта строка кода делает с полем XSpeed?

*Random.value * Multiplier * 2* получает случайное число от 0 до 1.5. Вычитание его из *Multiplier* дает случайное число в диапазоне от -0.75 до 0.75. Прибавление этого значения к XSpeed приводит к некоторому ускорению или замедлению шара на каждый кадр.

В результате увеличения или уменьшения скорости шара по всем трем осям каждый шар движется по неустойчивой случайной траектории.

```
XSpeed = Multiplier - Random.value * Multiplier * 2;
```

Что делает эта строка кода?

Она присваивает полю XSpeed случайное значение от -0.75 до 0.75. В результате некоторые шары движутся по оси X вперед, другие движутся назад, с разными скоростями.

А вы заметили, что в класс GameController никаких изменений вносить не пришлось? Дело в том, что мы не изменяли ничего из того, что GameController делает, — например, управление пользовательским интерфейсом или игровой режим. Если вы можете обновить код, изменяя один класс без модификации других, это может указывать на то, что система классов была хорошо спроектирована.

Проявите фантазию!

А вы можете предложить другие улучшения игры? Несколько наших идей:

- ★ Игра слишком проста? Слишком сложна? Попробуйте изменить параметры, которые вы передаете `InvokeRepeating` в методе `GameController.Start`. Попробуйте преобразовать их в поля. Также поэкспериментируйте со значением `MaximumBalls`. Небольшие изменения этих значений оказывают значительное влияние на игровой процесс.
- ★ Мы предоставили готовые текстурные карты для бильярдных шаров. Попробуйте добавить другие шары с другим поведением. Используйте масштабирование, чтобы шары различались в размерах, и измените их параметры, чтобы они двигались быстрее или медленнее либо по другим траекториям.
- ★ Сможете ли вы реализовать шар типа «падающая звезда», который очень быстро пролетает в одном направлении и дает много очков, если игрок успешно щелкнет на нем? А как насчет шара типа «внезапная смерть», щелчок на котором немедленно завершает игру?
- ★ Измените метод `GameController.ClickedOnBall`, чтобы он получал параметр со счетом (вместо увеличения поля `Score`) и прибавлял к нему переданное вами значение. Попробуйте назначить разным шарам разное количество очков.

Если вы измените поля в сценарии `OneBallBehaviour`, не забудьте сбросить компонент `Script` заготовки `OneBall!` В противном случае он будет «помнить» старые значения.

Чем больше опыта у вас будет в написании кода `C#`, тем проще пойдет дело. Эксперименты с кодом игры — отличная возможность получить такой опыт!

КЛЮЧЕВЫЕ МОМЕНТЫ

- Игры Unity отображают **пользовательский интерфейс (UI)** с элементами и графикой на двумерной плоскости, располагающейся перед 3D-сценой игры.
- Unity предоставляет набор **2D-объектов `GameObject`**, предназначенных специально для построения пользовательских интерфейсов.
- **Кнопка 2D** в верхней части окна Scene включает и выключает 2D-представление, упрощающее размещение компонентов пользовательского интерфейса.
- Когда вы добавляете в Unity **компонент `Script`**, он продолжает отслеживать значения полей и не перезагружает значения по умолчанию вплоть до сброса из контекстного меню.
- Кнопка **Button** может вызвать любой метод в сценарии, присоединенном к объекту `GameObject`.
- В окне Inspector можно **изменять значения полей** в сценариях объектов `GameObject`. Если вы измените их во время выполнения, они вернуться к сохраненным значениям при остановке игры.
- Метод **`transform.Translate`** перемещает объект `GameObject` на заданное расстояние от текущей позиции.
- **Tag** — ключевое слово, которое можно присоединить к любому объекту `GameObject`. Теги используются в коде для идентификации или поиска объектов.
- Метод **`GameObject.FindGameObjectsWithTag`** возвращает коллекцию объектов `GameObject` с заданным тегом.

9 LINQ и лямбда-выражения

Контроль над данными

И если взять первые пять слов из этой статьи и последние пять из этой и потом добавить к ним заголовок в обратном порядке, ты получишь секретное сообщение от инопланетян!



Этим миром правят данные... И нам нужно знать, как в нем жить. Прошли те времена, когда можно было программировать днями и даже неделями, не имея дела с огромными объемами данных. В наши дни данные стали *сутью любой программы*. LINQ — технология C# и .NET, которая позволяет не только **обращаться к данным** в коллекциях .NET на интуитивно понятном уровне, но и **группировать данные** и выполнять **слияние данных из разных источников**. **Модульные тесты** помогут убедиться в том, что ваш код работает так, как предполагалось. А когда вы освоитесь с задачей разбиения данных на блоки, с которыми удобно работать, вы можете воспользоваться **лямбда-выражениями**, провести рефакторинг кода C# и сделать его еще более выразительным.

Джимми — фанат Капитана Великолепного...

Знакомьтесь: Джимми — один из самых завзятых поклонников комиксов, графических романов и атрибутики о Капитане Великолепном. Он знает о Капитане все, у него есть фигурки из всех фильмов и у него есть коллекция комиксов, которую можно назвать только... Великолепной.

Смотри: это кружка из ограниченной серии со второй ежегодной **Великолепной конвенции**, с подписями Художников исходных комиксов!



Джимми ухи-трился где-то отыскать редкую фигурку Капитана Великолепного 2005 года в исходной упаковке.

Верно, это настоящий великолепно-мобиль из сериала, который шел с сентября по ноябрь 1973 года. Как Джимми раздобыл его?



...но его коллекция разбросана по всему дому

Джимми — настоящий энтузиаст, но ему не хватает собранности. Он пытается хранить информацию о жемчужинах своей коллекции, но сам он с этой задачей не справляется. Поможем Джимми построить приложение для хранения информации о комиксах?



LINQ работает со значениями и объектами.

Мы будем использовать LINQ для обращения с запросами к числовым коллекциям для знакомства с основными идеями и синтаксисом. Возможно, вы думаете: «Но как это поможет в управлении комиксами?» Это отличный вопрос, о котором стоит помнить в первой части этой главы. Позднее аналогичные вопросы LINQ будут использоваться для управления коллекциями объектов Comic.



Обложка легендарного выпуска «Смерть Объекта» с автографами авторов.



Использование LINQ для управления коллекциями

В этой главе вы узнаете о **LINQ (Language-Integrated Query)**. LINQ объединяет исключительно полезные классы и методы с мощной функциональностью, встроенной непосредственно в язык C#, для работы с последовательностями данных, такими как коллекция комиксов Джимми.

Воспользуемся Visual Studio, чтобы приступить к исследованию LINQ. **Создайте проект консольного приложения (.NET Core)** и присвойте ему имя *LinqTest*. Добавьте приведенный ниже код, а когда вы доберетесь до последней строки, **введите точку** и просмотрите окно IntelliSense:

```
using System;
```

```
namespace LinqTest
```

```
{
```

```
    using System.Collections.Generic;
```

```
    using System.Linq;
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            List<int> numbers = new List<int>();
```

```
            for (int i = 1; i <= 99; i++)
```

```
                numbers.Add(i);
```

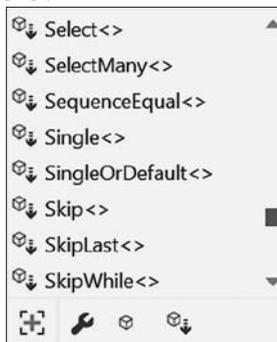
```
            IEnumerable<int> firstAndLastFive = numbers.
```

```
        }
    }
}
```

Директива `using System.Linq;` дополняет ваши коллекции новыми методами для применения к ним разнообразных запросов `on them`.

Введите «numbers» и нажмите . (точка), чтобы вызвать окно IntelliSense. Методов стало гораздо больше, чем прежде!

Вы уже работали с массивами и списками в течение некоторого времени, но эти методы вам еще не встречались. Попробуйте удалить директиву `using System.Linq;` в начале класса, снова откройте окно IntelliSense — новые методы исчезли! Они отображаются только при наличии директивы `using`.



Воспользуемся некоторыми из новых методов для завершения консольного приложения:

```
        IEnumerable<int> firstAndLastFive = numbers.Take(5).Concat(numbers.TakeLast(5));
```

```
        foreach (int i in firstAndLastFive)
```

```
        {
```

```
            Console.Write($"{i} ");
```

```
        }
    }
}
```

Запустите приложение. На консоль выводится строка текста:

```
1 2 3 4 5 95 96 97 98 99
```

Что же произошло?

LINQ (or... — LINQ (Language-Integrated Query) — комбинация функциональности C# и классов .NET для работы с последовательностями данных.



Запрос LINQ под увеличительным стеклом

Присмотримся к тому, как в программе используются методы LINQ `Take`, `TakeLast` и `Concat`.

numbers

Исходная коллекция `List<int>`, созданная в цикле `for`.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27			
28	29	30	31	32	33	34	35	36	37	38	39			
40	41	42	43	44	45	46	47	48	49	50	51			
52	53	54	55	56	57	58	59	60	61	62	63			
64	65	66	67	68	69	70	71	72	73	74	75			
76	77	78	79	80	81	82	83	84	85	86	87			
88	89	90	91	92	93	94	95	96	97	98	99			

numbers.Take(5)

Метод `Take` получает первые элементы последовательности.

1	2	3	4	5
---	---	---	---	---

numbers.TakeLast(5)

Метод `TakeLast` получает последние элементы последовательности.

96	97	98	99	100
----	----	----	----	-----

numbers.Take(5).Concat(numbers.TakeLast(5))

Метод `Concat` объединяет две последовательности.

1	2	3	4	5	96	97	98	99	100
---	---	---	---	---	----	----	----	----	-----

Цепочки вызовов методов LINQ

При включении в код директивы `using System.Linq`; методы LINQ добавляются в списки. Они также добавляются к массивам, очередям, стекам... собственно, они добавляются ко всем объектам, расширяющим `IEnumerable<T>`. Так как почти все методы LINQ возвращают `IEnumerable<T>`, вы можете получить результат метода LINQ и вызвать для него другой метод LINQ без использования переменной для хранения промежуточных результатов. Этот синтаксис, называемый сцеплением вызовов, позволяет писать очень компактный код.

Например, для хранения результатов `Take` и `TakeLast` можно было бы воспользоваться переменными:

```
IEnumerable<int> firstFive = numbers.Take(5);
IEnumerable<int> lastFive = numbers.TakeLast(5);
IEnumerable<int> firstAndLastFive = firstFive.Concat(lastFive);
```

Но сцепление вызовов позволяет объединить все вызовы в одну строку кода, с вызовом `Concat` непосредственно для результата `numbers.Take(5)`. Оба фрагмента делают одно и то же. Учтите, что компактный код не всегда лучше подробного! Иногда разбиение цепочки вызовов на строки упрощает ее понимание. Вы сами должны решить, какой вариант будет лучше читаться в каждом конкретном проекте.

LINQ работает с любыми реализациями IEnumerable<T>

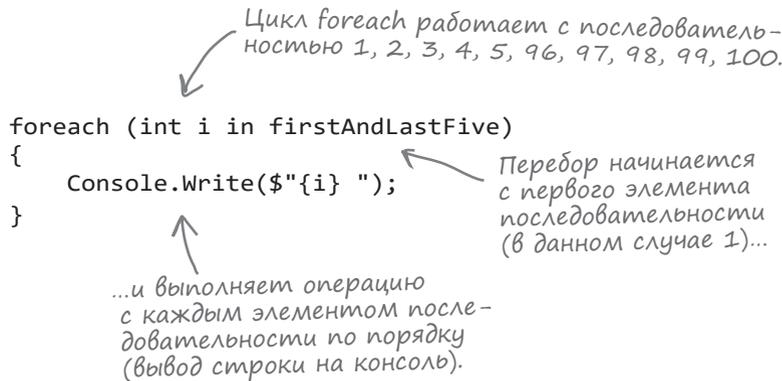
Когда вы добавляете в свой код директиву using System.Linq;, коллекция List<int> внезапно «усиливается» — в ней появляется группа методов LINQ. То же самое можно сделать с любым *классом, реализующим IEnumerable<T>*.

Если класс реализует IEnumerable<T>, любой экземпляр этого класса является **последовательностью** (sequence):

- ★ Список чисел от 1 до 99 был последовательностью.
- ★ Когда вы вызывали метод Take, он возвращал ссылку на последовательность, содержащую первые пять элементов.
- ★ Когда вы вызывали метод TakeLast, он возвращал другую последовательность из пяти элементов.
- ★ А когда вы использовали метод Concat для объединения двух последовательностей из пяти элементов, метод создал новую последовательность из 10 элементов и вернул ссылку на новую последовательность.

Методы LINQ перебирают последовательности

Вы уже знаете, что циклы foreach работают с объектами IEnumerable. Подумайте, что делает цикл foreach:



Когда метод работает с каждым элементом последовательности по порядку, это называется **перебором** последовательности. Методы LINQ работают именно по такому принципу.

Объекты, реализующие интерфейс IEnumerable, могут быть задействованы в переборе. Эта функциональность обеспечивается объектами, реализующими интерфейс IEnumerable.

пе-ре-би-рать, глагол.
Упомянуть элементы последовательности один за одним.

Если у вас имеется объект, реализующий интерфейс IEnumerable, вы имеете дело с последовательностью, которая может использоваться с LINQ. Поочередное выполнение операции с элементами этой последовательности называется перебором этой последовательности.

Enumerable.Range(8, 5); → 8 9 10 11 12

А если потребуется найти первые 30 выпусков из коллекции Джимми, начиная с выпуска 118? LINQ предоставляет удобные методы для подобных задач. Статический метод Enumerable.Range генерирует последовательность целых чисел. Вызов Enumerable.Range(8, 5) возвращает последовательность из 5 чисел, начинающуюся с 8: 8, 9, 10, 11, 12.

IEnumerable<int> Enumerable.Range(int start, int count)
Generates a sequence of integral numbers within a specified range.

Exceptions:
ArgumentOutOfRangeException

↑ Вы потренируетесь в использовании метода Enumerable.Range в следующем упражнении с карандашом и бумагой.

Возьми в руку карандаш



Ниже перечислены несколько методов LINQ, которые добавляются к последовательностям при включении директивы using System.Linq;. Имена методов интуитивно понятны — сможете ли вы определить, что они делают? Соедините каждый вызов метода с его результатом.

Enumerable.Range(1, 5) .Sum()	9
Enumerable.Range(1, 6) .Average()	17
new int[] { 3, 7, 9, 1, 10, 2, -3 } .Min()	104
new int[] { 8, 6, 7, 5, 3, 0, 9 } .Max()	15
Enumerable.Range(10, 3721) .Count()	3.5
Enumerable.Range(5, 100) .Last()	10
new List<int>() { 3, 8, 7, 6, 9, 6, 2 } .Skip(4) .Sum()	-3
Enumerable.Range(10, 731) .Reverse() .Last()	3721



Возьми в руку карандаш

Решение

Ниже перечислены некоторые методы LINQ, которые добавляются к последовательностям при включении директивы `using System.Linq`; Имена методов интуитивно понятны — сможете ли вы определить, что они делают? **Соедините** каждый вызов метода с его результатом.

Метод `Sum` суммирует все значения последовательности и возвращает результат.

```
Enumerable.Range(1, 5)
    .Sum()
```

```
Enumerable.Range(1, 6)
    .Average()
```

```
new int[] { 3, 7, 9, 1, 10, 2, -3 }
    .Min()
```

```
new int[] { 8, 6, 7, 5, 3, 0, 9 }
    .Max()
```

```
Enumerable.Range(10, 3721)
    .Count()
```

```
Enumerable.Range(5, 100)
    .Last()
```

```
new List<int>() { 3, 8, 7, 6, 9, 6, 2 }
    .Skip(4)
    .Sum()
```

```
Enumerable.Range(10, 731)
    .Reverse()
    .Last()
```

9
Вызов `Skip(4)` пропускает первые 4 элемента последовательности, возвращая {6, 9, 2}. `Sum` суммирует эти элементы: $6+9+2=17$.

17
`Range(5, 100)` возвращает {100, 101, 102, 103, 104}, а `List()` получает последнее число этой последовательности.

104
15
`Range(10, 731)` возвращает последовательность из 731 числа, начиная с 10. Вызов `Reverse` переставляет элементы в обратном порядке, так что последним элементом инвертированной последовательности становится 10.

3.5

10

-3

3721

По именам методов LINQ в этом упражнении нетрудно догадаться, что они делают. Некоторые методы LINQ (такие, как `Sum`, `Min`, `Max`, `Count`, `First` и `Last`) возвращают одно значение. Метод `Sum` суммирует все значения в последовательности. Метод `Average` вычисляет их среднее значение. Методы `Min` и `Max` возвращают наименьшее и наибольшее значение в последовательности. Методы `First` и `Last` возвращают первое и последнее значение.

Другие методы LINQ, такие как `Take`, `TakeLast`, `Concat`, `Reverse` (переставляет элементы последовательности в обратном порядке) и `Skip` (пропускает начальные элементы последовательности), возвращают другую последовательность.

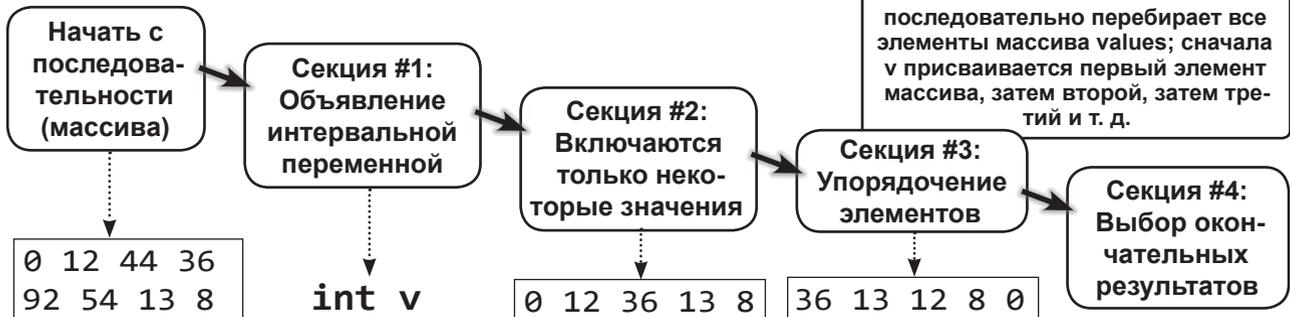
Синтаксис запросов LINQ

Методы LINQ, упоминавшиеся до настоящего момента, вряд ли способны ответить на все вопросы о данных, которые у вас могли возникнуть, — или на вопросы, которые возникают у Джимми по поводу его коллекции комиксов.

И здесь в игру вступает **синтаксис декларативных запросов LINQ**. В нем используются специальные ключевые слова — `where`, `select`, `groupby`, `join` и т. д. — для построения запросов непосредственно в коде.

Запросы LINQ строятся из секций

Построим запрос, который находит в массиве числа, меньшие 37, и выстраивает их по возрастанию. Для этого используются четыре **секции**, которые указывают, к какому объекту должен быть обращен запрос, по какому критерию отбираются элементы, как отсортировать результаты и как должны быть возвращены результаты.



Запросы LINQ работают с **последовательностями**, т. е. объектами, реализующими `IEnumerable<T>`. Запросы LINQ начинаются с секции `from`:

`from` (переменная) `in` (последовательность)

Она сообщает запросу, к какой последовательности применяется запрос, и назначает имя, которое будет использоваться для представления каждого элемента. Секция `from` отчасти напоминает первую строку цикла `foreach`: она объявляет переменную, которая должна использоваться для перебора последовательности и которой присваивается каждый элемент последовательности. Таким образом, команда

`from v in values`

последовательно перебирает все элементы массива `values`; сначала `v` присваивается первый элемент массива, затем второй, затем третий и т. д.

```
int[] values = new int[] {0, 12, 44, 36, 92, 54, 13, 8};
```

```
IEnumerable<int> result =
```

```
    from v in values
```

```
    where v < 37
```

```
    orderby -v
```

```
    select v;
```

Запрос LINQ состоит из четырех секций: `from`, `where`, `orderby` и `select`.

Секция `from` присваивает значение переменной, называемой **интервальной**, которая будет представлять каждое значение при переборе массива. При первой итерации переменная содержит 0, затем 12, затем 44 и т. д.

Секция `where` содержит условие, по которому запрос определяет значения, включаемые в результат, — в данном случае включаются любые значения, меньшие 37.

Секция `orderby` содержит выражение, используемое для сортировки результатов, — в данном случае выражение `-v` сортирует данные по убыванию (от больших к меньшим).

Запрос завершается **секцией `select`** с выражением, которое определяет, что должно быть включено в результаты.

```
// Вывод результатов в цикле foreach
```

```
foreach(int i in result)
    Console.Write($"{i} ");
```

Результат: 36 13 12 8 0

Часть Задаваемые Вопросы

В: Значит, размещение директивы `using` в начале файла «по волшебству» добавляет методы LINQ в каждую реализацию `IEnumerable`?

О: По сути, да. Чтобы использовать методы LINQ (а также запросы LINQ), необходимо включить директиву `using System.Linq`; в начало файла. Как было показано в последней главе, для использования `IEnumerable<T>` (например, в возвращаемом значении) также должна присутствовать директива `using System.Collections.Generic`;

(Очевидно, никакого *волшебства* здесь нет. LINQ использует возможность C# — так называемые **методы расширения**, о которых вы узнаете в главе 11. А пока достаточно знать, что при добавлении директивы `using` LINQ может использоваться с любой ссылкой на `IEnumerable<T>`.)

В: Мне все еще непонятно, что такое «сцепленные вызовы». Как они работают и почему ими следует пользоваться?

О: Сцепление вызовов — очень распространенный способ вызова нескольких методов подряд. Так как многие методы LINQ возвращают последовательность, реализующую `IEnumerable<T>`, для результата можно вызвать другой метод LINQ. Впрочем, сцепление вызовов не является отличительной особенностью LINQ. Вы можете использовать его в своих классах.

В: А вы можете привести пример класса, в котором используется сцепление вызовов?

О: Конечно. Вот класс с двумя методами, построенными для сцепления:

Методы Add и Subtract классов AddSubtract возвращают экземпляры AddSubtract. Идеально подходит для сцепления!

```
class AddSubtract
{
    public int Value { get; set; }
    public AddSubtract Add(int i) {
        Console.WriteLine($"Value: {Value}, adding {i}");
        return new AddSubtract() { Value = Value + i };
    }
    public AddSubtract Subtract(int i) {
        Console.WriteLine(
            $"Value: {Value}, subtracting {i}");
        return new AddSubtract() { Value = Value - i };
    }
}
```

Методы класса можно вызывать так:

```
AddSubtract a = new AddSubtract() { Value = 5 }
    .Add(5)
    .Subtract(3)
    .Add(9)
    .Subtract(12);
Console.WriteLine($"Result: {a.Value}");
```

Попробуйте добавить класс AddSubtract в новое консольное приложение, а затем добавьте этот код в метод Main.



Все это, конечно, хорошо. Но как это поможет мне в управлении огромной коллекцией комиксов?

LINQ работает не только с числами, но и с объектами.

Когда Джимми смотрит на стопки неупорядоченных комиксов, он видит бумагу, краску и хаос. Когда на них смотрим мы, разработчики, мы видим нечто другое: **большой объем данных**, которые нужно упорядочить. Как упорядочить данные комиксов в C#? Так же, как мы упорядочивали игральные карты, пчел или позиции меню в забегаловке Джо: мы создаем класс, а затем используем коллекцию для работы с этим классом. Таким образом, чтобы помочь Джимми, нужно написать класс `Comic`, а потом код, который наведет порядок в его коллекции. А LINQ нам в этом поможет!

LINQ работает с объектами

← (Делайте это!

Джимми хочет знать, сколько стоят самые редкие комиксы в его коллекции, поэтому он нанял профессионального оценщика. Оказывается, отдельные позиции ценятся очень дорого! Воспользуемся коллекциями для управления этими данными.

1 Создайте новое консольное приложение и добавьте класс Comic.

Используйте два автоматических свойства для названия и номера выпуска:

```
using System.Collections.Generic;
class Comic {
    public string Name { get; set; }
    public int Issue { get; set; }
```

```
    public override string ToString() => $"{Name} (Issue #{Issue})";
```

Оператор => нам еще не встречался! А вы сможете понять по контексту, что он делает? Вы знаете, как работают методы ToString, — получается, что => каким-то образом заставляет ToString вернуть интерполированную строку справа от оператора.

2 Добавьте список List с каталогом комиксов Джимми.

Добавьте статическое поле Catalog в класс Comic. Оно возвращает последовательность самых ценных комиксов:

```
public static readonly IEnumerable<Comic> Catalog =
    new List<Comic> {
        new Comic { Name = "Johnny America vs. the Pinko", Issue = 6 },
        new Comic { Name = "Rock and Roll (limited edition)", Issue = 19 },
        new Comic { Name = "Woman's Work", Issue = 36 },
        new Comic { Name = "Hippie Madness (misprinted)", Issue = 57 },
        new Comic { Name = "Revenge of the New Wave Freak (damaged)", Issue = 68 },
        new Comic { Name = "Black Monday", Issue = 74 },
        new Comic { Name = "Tribal Tattoo Madness", Issue = 83 },
        new Comic { Name = "The Death of the Object", Issue = 97 },
    };
```

Мы опустили круглые скобки () в инициализаторах коллекции и объектов после <Comic>, потому что они необязательны.

3 Используйте словарь для управления ценами.

Добавьте статическое поле Comic.Prices — это словарь Dictionary<int, decimal> для получения цены каждого комикса по номеру выпуска (используйте синтаксис инициализатора коллекций для словарей, который был представлен в главе 8). Обратите внимание: интерфейс IReadOnlyDictionary используется для инкапсуляции — этот интерфейс включает только методы для чтения значений (чтобы предотвратить случайное изменение цен):

```
public static readonly IReadOnlyDictionary<int, decimal> Prices =
    new Dictionary<int, decimal> {
        { 6, 3600M },
        { 19, 500M },
        { 36, 650M },
        { 57, 13525M },
        { 68, 250M },
        { 74, 75M },
        { 83, 25.75M },
        { 97, 35.25M },
    };
```

Интерфейс IReadOnlyDictionary используется словом для того, чтобы предотвратить случайное изменение цен в поле Price.

Редкое издание выпуска №57 (с опечаткой) стоит \$13 525. Ничего себе!

Для хранения цен на комиксы использовался словарь. Также можно было добавить в класс свойство с именем Price. Мы решили хранить информацию о комиксах и ценах по отдельности. Такой вариант был выбран из-за того, что цены постоянно изменяются, а названия и номера выпусков остаются неизменными. Как вы относитесь к этому решению?

Использование запроса LINQ в приложении для Джимми

Ранее синтаксис декларативных запросов LINQ использовался для создания запроса, состоящего из четырех секций: секции `from` для создания интервальной переменной; секции `where` для включения только чисел, меньших 37; секции `orderby` для сортировки результата по убыванию; и секции `select` для определения того, какие элементы должны включаться в итоговую последовательность.

Добавим в метод `Main` запрос LINQ, который работает точно так же, не считая того, что он использует объекты `Comic` вместо значений `int`, чтобы на консоль выводился список комиксов с ценой >500 в обратном порядке. Начнем с двух объявлений `using`, чтобы иметь возможность использовать методы `IEnumerable<T>` и LINQ. Запрос будет возвращать `IEnumerable<Comic>`, а затем использовать цикл `foreach` для перебора объектов и вывода результатов.

4 Измените метод Main для использования запросов LINQ.

Ниже приведен весь класс `Program`, включая директивы `using`, которые необходимо добавить в начало класса:

```
using System.Collections.Generic;
using System.Linq;
```

Пространство имен `System.Collections.Generic` необходимо для использования интерфейса `IEnumerable<T>`, а `System.Linq` — для добавления методов LINQ ко всем объектам, реализующим этот интерфейс.

```
class Program
{
    static void Main(string[] args)
    {
        IEnumerable<Comic> mostExpensive =
            from comic in Comic.Catalog
            where Comic.Prices[comic.Issue] > 500
            orderby -Comic.Prices[comic.Issue]
            select comic;

        foreach (Comic comic in mostExpensive)
        {
            Console.WriteLine($"{comic} is worth {Comic.Prices[comic.Issue]:c}");
        }
    }
}
```

Обратите внимание на интервальную переменную «`comic`». Эта переменная типа `Comic` объявляется в секции `from` и используется в секциях `where` и `orderby`.

Секция `select` определяет данные, возвращаемые запросом. Так как выводится переменная `Comic`, результат запроса представляет собой `IEnumerable<Comic>`.

В предыдущих главах было показано, что «`:c`» форматирует число как локальную денежную сумму, поэтому в Великобритании вместо \$ будет выводиться знак фунта £.

Результат:

```
Hippie Madness (misprinted) is worth $13,525.00
Johnny America vs. the Pinko is worth $3,600.00
Woman's Work is worth $650.00
```

5 Использование ключевого слова `descending` для удобочитаемости секции `orderby`

В секции `orderby` используется знак «минус» для изменения знака цен на комиксы перед сортировкой, в результате чего запрос сортирует их по убыванию. Впрочем, минус легко пропустить, когда вы читаете код и пытаетесь разобраться в том, как он работает. К счастью, того же результата можно добиться другим способом. Удалите знак «минус» и добавьте ключевое слово `descending` в конец секции:

```
orderby Comic.Prices[comic.Issue] descending
```

Ключевое слово `descending` обеспечивает сортировку в обратном порядке.



Анатомия запроса

Чтобы разобраться в том, как работают запросы LINQ, внесем пару незначительных изменений в запрос:

- ★ Знак «минус» в секции `orderby` слишком легко пропустить. Используйте ключевое слово `descending`, добавленное на шаге 5.

Изменение секции `select` приводит к тому, что запрос возвращает последовательность строк.

- ★ Только что написанная секция `select` выбирала комик, поэтому результатом запроса была последовательность ссылок на объекты `Comic`. Заменяем его интерполированной строкой, в которой используется интервальная переменная `comic`, — теперь результат запроса представляет собой последовательность строк.

Ниже приведен обновленный запрос LINQ. Каждая секция запроса создает последовательность, которая передается следующей секции, — мы разместили под каждой секцией таблицу, в которой отображается ее результат.

```
IEnumerable<string> mostExpensiveComicDescriptions =  
from comic in Comic.Catalog
```

{ Name = "Johnny America vs. the Pinko", Issue = 6 }
{ Name = "Rock and Roll (limited edition)", Issue = 19 }
{ Name = "Woman's Work", Issue = 36 }
{ Name = "Hippie Madness (misprinted)", Issue = 57 }
{ Name = "Revenge of the New Wave Freak (damaged)", Issue = 68 }
{ Name = "Black Monday", Issue = 74 }
{ Name = "Tribal Tattoo Madness", Issue = 83 }
{ Name = "The Death of the Object", Issue = 97 }

Секция `from` перебирает содержимое `Comic.Catalog`, извлекает из него каждое значение и присваивает интервальной переменной «`comic`». Результатом секции `from` является последовательность ссылок на объекты `Comic`.

```
where Comic.Prices[comic.Issue] > 500
```

{ Name = "Johnny America vs. the Pinko", Issue = 6 }
{ Name = "Woman's Work", Issue = 36 }
{ Name = "Hippie Madness (misprinted)", Issue = 57 }

Секция `where` начинается с результатов секции `from`, для чего «`comic`» присваивается каждому значению и используется для проверки условия, которое получает цену из словаря `Comic.Prices` и включает в результат только комиксы с ценой более 500.

```
orderby Comic.Prices[comic.Issue] descending
```

{ Name = "Hippie Madness (misprinted)", Issue = 57 }
{ Name = "Johnny America vs. the Pinko", Issue = 6 }
{ Name = "Woman's Work", Issue = 36 }

Секция `orderby` начинается с результатов секции `where` и сортирует их по убыванию цены.

```
select $"{comic} is worth {Comic.Prices[comic.Issue]:c}";
```

"Hippie Madness (misprinted) is worth \$13,525.00"
"Johnny America vs. the Pinko is worth \$3,600.00"
"Woman's Work is worth \$650.00"

Секция `select` перебирает результаты секции `orderby`, используя интервальную переменную «`comic`» со строковой интерполяцией для возвращения последовательности строк.

Ключевое слово var позволяет C# определить тип переменной за вас

Вы только что видели, что незначительное изменение в секции `select` привело к изменению типа возвращаемой последовательности. Когда секция содержала `select comic;`, использовался возвращаемый тип `IEnumerable<Comic>`. Когда секция была приведена к виду `select $"{comic} is worth {Comic.Prices[comic.Issue]:c}";`, возвращаемый тип изменился на `IEnumerable<string>`. При работе с LINQ это происходит сплошь и рядом — вы постоянно настраиваете свои запросы. Не всегда очевидно, какой именно тип они возвращают. Иногда необходимость возвращаться и обновлять все объявления начинает раздражать.

К счастью, C# предоставляет очень полезный инструмент, который помогает сохранить объявления переменных простыми и удобочитаемыми. Любые изменения переменных можно заменить **ключевым словом var**. Таким образом, любые из следующих объявлений:

```
IEnumerable<int> numbers = Enumerable.Range(1, 10);
string s = $"The count is {numbers.Count()}";
IEnumerable<Comic> comics = new List<Comic>();
IReadOnlyDictionary<int, decimal> prices = Comic.Prices;
```

можно заменить следующими объявлениями, которые делают то же:

```
var numbers = Enumerable.Range(1, 10);
var s = $"The count is {numbers.Count()}";
var comics = new List<Comic>();
var prices = Comic.Prices;
```

Используя ключевое слово `var`, вы приказываете C# использовать переменную с **неявно определяемым типом**. Термин «неявный» уже встречался вам в главе 8, когда мы рассматривали ковариантность. Он означает, что C# определяет типы самостоятельно.

И вам не придется изменять свой код. Просто замените типы на `var`, и все будет работать.

C ключевым словом `var` C# определяет тип переменной автоматически

Опробуйте внесенные изменения. Закомментируйте первую строку только что написанного запроса LINQ, затем замените `IEnumerable<Count>` на `var`:

```
// IEnumerable<Comic> mostExpensive =
var mostExpensive =
    from comic in Comic.Catalog
    where Comic.Prices[comic.Issue] > 500
    orderby -Comic.Prices[comic.Issue]
    select comic;
```

Когда вы используете `var` в объявлении переменной, IDE определяет ее тип на основании того, как переменная используется в коде.

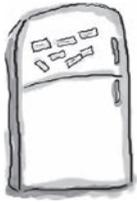
Когда вы временно блокируете секцию `orderby` в этом запросе, `mostExpensive` преобразуется к типу `IEnumerable<T>`.

Наведите указатель мыши на имя переменной в цикле `foreach`, чтобы увидеть ее тип:

```
foreach (Comic comic in mostExpensive)
{
    Console.WriteLine($"{comic}");
}
```

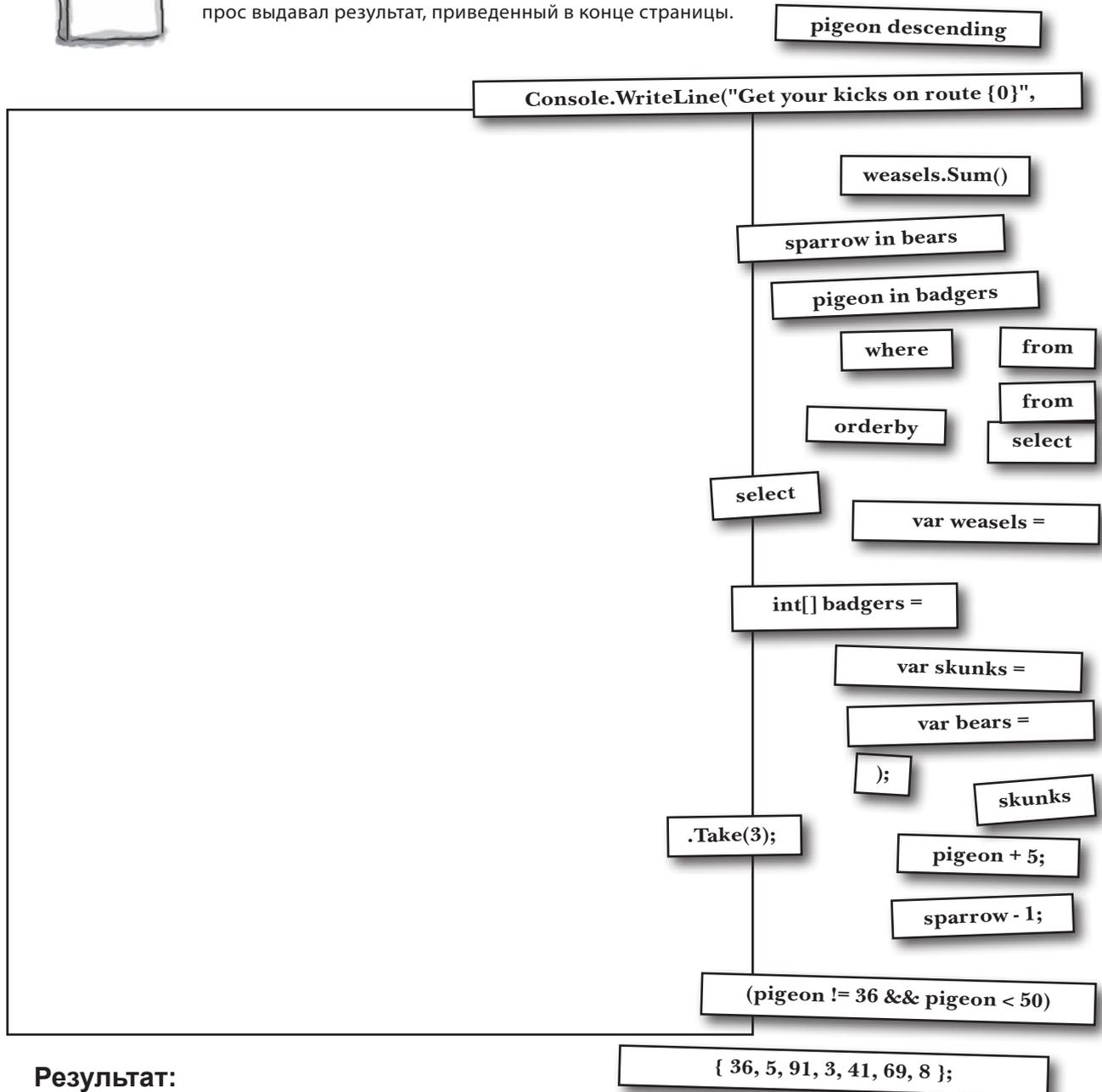
(local variable) IOrderedEnumerable<Comic> mostExpensive

IDE определила тип переменной `mostExpensive` — и этот тип нам еще не встречался. Помните, как в главе 7 упоминалось о том, что интерфейсы могут расширять другие интерфейсы? Интерфейс `IOrderedEnumerable` является частью LINQ — он используется для представления *отсортированной* последовательности, и он расширяет интерфейс `IEnumerable<T>`. Попробуйте закомментировать секцию `orderby` и навести указатель мыши на переменную `mostExpensive` — вы увидите, что она преобразовалась к типу `IEnumerable<Comic>`. Дело в том, что C# анализирует код, чтобы **вычислить тип любой переменной, объявленной с ключевым словом var**.



Развлечения с Магнитами

На холодильнике из магнитов был выложен запрос LINQ с ключевым словом `var`, но кто-то хлопнул дверью и часть магнитов упала на пол! Расставьте магниты так, чтобы запрос выдавал результат, приведенный в конце страницы.



Результат:
Get your kicks on route 66



Развлечения с МаГнитами

Решение

Разместите магниты так, чтобы программа выдавала результат, приведенный в нижней части страницы.

LINQ начинает с некоторой разновидностью последовательности, коллекции или массива — в данном случае массива целых чисел.

```
int[] badgers = { 36, 5, 91, 3, 41, 69, 8 };
```

Мы намеренно выбрали невразумительные имена. Обороты типа «from pigeon in badgers» смахивают на головоломку, и понять такой код достаточно сложно. Если принять интервальной переменной, это упростит чтение кода.

```
var skunks =
    from pigeon in badgers
    where (pigeon != 36 && pigeon < 50)
    orderby pigeon descending
    select pigeon + 5;
```

После этой команды skunks содержит четыре числа: 46, 13, 10 и 8.

Эта команда LINQ извлекает из массива все числа, меньшие 50 и не равные 36. Каждое число увеличивается на 5, числа сортируются по убыванию и помещаются в новый объект, который присваивается ссылке skunks.

```
var bears =
    skunks .Take(3);
```

После этой команды bears содержит три числа: 46, 13 и 10.

Здесь первые три числа из skunks помещаются в новую последовательность с именем bears.

```
var weasels =
    from sparrow in bears
    select sparrow - 1;
```

После этой команды weasels содержит три числа: 45, 12 и 9.

Команда просто уменьшает каждое число в bears на 1 и помещает их в weasels.

```
Console.WriteLine("Get your kicks on route {0}",
    weasels.Sum() );
```

$45 + 12 + 9 = 66$

Числа в weasels в сумме дают 66.

Результат:
Get your kicks on route 66



И вы серьезно говорите, что я могу заменить любой тип в любом объявлении переменной на `var` и мой код все равно будет работать? Не может быть, чтобы все было так просто.

Да, вы можете использовать `var` в объявлениях переменных.

Да, все может быть так просто. Многие разработчики C# почти всегда объявляют локальные переменные с `var` и включают тип только тогда, когда это упрощает понимание кода. Если переменная объявляется и инициализируется в одной команде, вы можете использовать `var`.

Однако для применения `var` существует ряд важных ограничений. Например:

- С `var` можно объявить только одну переменную за раз.
- Объявляемая переменная не может использоваться в объявлении.
- Переменная не может объявляться равной `null`.

Если вы создаете переменную с именем `var`, в дальнейшем вы не сможете использовать это ключевое слово:

- Ключевое слово `var` определенно не может использоваться в объявлениях полей или свойств — только в локальных переменных внутри метода.
- Придерживаясь этих базовых правил, вы сможете использовать `var` практически везде.

Таким образом, когда вы использовали эти объявления в главе 4:

```
int hours = 24;
short RPM = 33;
long radius = 3;
char initial = 'S';
int balance = 345667 - 567;
```

Или эти в главе 6:

```
SwordDamage swordDamage = new SwordDamage(RollDice(3));
ArrowDamage arrowDamage = new ArrowDamage(RollDice(1));
```

Или эти в главе 8:

```
List<Card> cards = new List<Card>();
```

Также можно было использовать объявления:

```
var hours = 24;
var RPM = 33;
var radius = 3;
var initial = 'S';
var balance = 345667 - 567;
```

Или эти:

```
var swordDamage = new SwordDamage(RollDice(3));
var arrowDamage = new ArrowDamage(RollDice(1));
```

Или эти:

```
var cards = new List<Card>();
```

...и ваш код работал бы точно так же.

Однако `var` не может использоваться для объявления полей или свойств:

```
class Program
{
    static var random = new Random(); // Произойдет ошибка компиляции.

    static void Main(string[] args)
    {
```

Часть Задаваемые Вопросы

В: Как работает секция from?

О: У нее много общего с первой строкой цикла foreach. Один из аспектов, немного усложняющих понимание запросов LINQ, заключается в том, что в них выполняется не одна операция. Запрос выполняется снова и снова для каждого элемента в коллекции — иначе говоря, он перебирает последовательность. Таким образом, секция from решает две задачи: она сообщает LINQ, какая коллекция должна использоваться для запроса, и выбирает имя, которое должно использоваться для каждого элемента коллекции в запросе.

Создание нового имени для каждого элемента в секции from очень похоже на то, как это делается в цикле foreach. Первая строка цикла foreach выглядит так:

```
foreach (int i in values)
```

Цикл foreach временно создает переменную с именем i, которой последовательно присваивается каждый элемент коллекции значений. Теперь взгляните на секцию from запроса LINQ той же коллекции:

```
from i in values
```

Эта секция делает практически то же самое. Она создает интервальную переменную с именем i и последовательно присваивает ей каждый элемент коллекции values. Цикл foreach выполняет один блок для каждого элемента коллекции, а запрос LINQ применяет один критерий из секции where к каждому элементу коллекции, чтобы определить, нужно ли включить его в результаты.

В: Вы взяли запрос LINQ, который возвращал последовательность ссылок на Comic, и заставили его возвращать строки. Как именно это было сделано?

О: Мы изменили секцию select. Секция select включает выражение, которое применяется к каждому элементу последовательности, и это выражение определяет тип результата. Таким образом, если запрос производит последовательность значений или ссылок на объекты, вы можете воспользоваться строковой интерполяцией в секции select для преобразования каждого элемента последовательности в строку. Запрос в решении завершался конструкцией select comic, поэтому он возвращал последовательность ссылок на Comic. В коде из раздела «Анатомия запроса» она была заменена конструкцией select \$"{comic} is worth {Comic.Prices[comic].Issue}:c"; — и это заставило запрос вернуть последовательность строк вместо последовательности Comic.

В: Как LINQ решает, какие данные должны быть включены в результаты?

О: Для этого нужна секция select. Каждый запрос LINQ возвращает последовательность, и все элементы последовательности относятся к одному типу. Секция select сообщает LINQ, что должна содержать эта последовательность. Когда вы обращаетесь с запросом к массиву или списку одного типа, например массиву int или List<string>, вполне очевидно, что должно попасть в секцию select. А если вы выбираете данные из списка объектов Comic? Можно сделать то, что сделал Джимми, и выбрать весь класс. Также можно изменить последнюю строку запроса на select comic.Name, чтобы приказать запросу вернуть последовательность строк. А можно использовать select comic.Issue, чтобы запрос вернул последовательность int.

В: Я понимаю, как использовать var в коде, но как работает это ключевое слово?

О: var — ключевое слово, которое приказывает компилятору автоматически определить тип переменной на стадии компиляции. Компилятор C# определяет тип локальной переменной, которая используется LINQ для запроса. При построении решения компилятор заменяет var типом, соответствующим данным, с которыми вы работаете.

Таким образом, при компиляции следующей строки:

```
var result = from v in values
```

компилятор заменит var типом

```
IEnumerable<int>
```

Вы всегда можете проверить фактический тип переменной, наведя на нее указатель мыши в IDE.

Секция from в запросе LINQ решает две задачи: она сообщает LINQ, какая коллекция должна использоваться для запроса, и выбирает имя, которое должно использоваться для каждого элемента коллекции в запросе.

В: В запросах LINQ используются ключевые слова, которые мне еще не встречались, — `from`, `where`, `orderby`, `select`... Похоже на совершенно новый язык. Почему LINQ так отличается от остального кода C#?

О: Потому что LINQ используется для другой цели. Большая часть синтаксиса C# создавалась для выполнения одной небольшой операции или вычисления за раз. Запуск цикла, определение переменной, выполнение математической операции, вызов метода... все это отдельные операции. Пример:

```
var under10 =
    from number in sequenceOfNumbers
    where number < 10
    select number;
```

Запрос выглядит относительно просто — здесь не так много всего, верно? Но в действительности это довольно сложный фрагмент кода.

Подумайте, что должно произойти, чтобы программа выбрала все числа из переменной `sequenceOfNumbers` (которая должна содержать ссылку на объект, реализующий `IEnumerable<T>`), меньшие 10. Сначала необходимо в цикле перебрать весь массив. Затем каждое число сравнивается с 10. Полученные результаты необходимо собрать вместе, чтобы они могли использоваться в остальном коде.

Вот почему LINQ выделяется на общем фоне: потому что запросы позволяют вместить значительный объем поведения в малом — но легко читаемом! — объеме кода C#.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Если класс реализует `IEnumerable<T>`, любой экземпляр этого класса является **последовательностью**.
- При включении `using System.Linq`; в начало кода **методы LINQ** могут использоваться с любой ссылкой на последовательность.
- Когда метод последовательно получает все элементы последовательности, это называется **перебором** последовательности. Механизм перебора лежит в основе всех методов LINQ.
- **Метод Take** получает первые элементы последовательности. Метод `TakeLast` получает последние элементы последовательности. Метод `Concat` объединяет две последовательности воедино.
- **Метод Average** возвращает среднее значение для последовательности чисел. **Методы Min и Max** возвращают наименьшее и наибольшее значение в последовательности.
- **Методы First и Last** возвращают первый или последний элемент последовательности. **Метод Skip** пропускает начальные элементы последовательности и возвращает остальные.
- Многие методы LINQ возвращают последовательность, что позволяет использовать **сцепление вызовов**, т. е. вызов очередного метода LINQ непосредственно для результата без сохранения его в промежуточной переменной.
- **Интерфейс IReadOnlyDictionary** полезен для инкапсуляции. Вы можете присвоить ему любой словарь, чтобы создать ссылку, которая не позволяет обновлять словарь.
- **Синтаксис декларативных запросов LINQ** использует специальные ключевые слова (`where`, `select`, `groupby`, `join` и т. д.) для построения запросов непосредственно в коде.
- Запросы LINQ начинаются с **секции from**, в которой назначается переменная, представляющая все значения в процессе перебора последовательности.
- Переменная, объявленная в секции `from`, называется **интервальной переменной**. Она может использоваться в запросе.
- **Секция where** содержит условие, которое используется запросом для определения значений, включаемых в результат.
- Секция `orderby` содержит выражение, используемое для сортировки результатов. Также в ней можно использовать ключевое слово `descending` для перехода к сортировке по убыванию.
- Запрос завершается **секцией select** с выражением, которое указывает, что должно быть включено в результаты.
- Ключевое слово `var` используется для объявления переменной с неявным типом; это означает, что компилятор C# определяет тип переменной самостоятельно.
- Ключевое слово `var` может использоваться **вместо типа переменной** в каждом объявлении с инициализацией переменных.
- В **секцию select** можно включить выражение C#. Это выражение применяется к каждому элементу результата и определяет тип последовательности, возвращаемой запросом.

Гибкость LINQ

Возможности LINQ отнюдь не ограничиваются извлечением нескольких элементов из коллекции. Вы можете изменить элементы перед тем, как возвращать их. После того как вы сгенерируете набор последовательностей результатов, LINQ предоставляет набор методов для работы с ними. Кратко рассмотрим некоторые возможности LINQ, которые уже встречались вам ранее.

Ключевое слово `var` может использоваться для объявления массива с неявным типом. Просто используйте `new[]` с инициализатором коллекции, а компилятор C# определит тип массива за вас. При указании смешанного набора типов необходимо указать тип массива:

```
var mixed = new object[] { 1, "x", new Random() };
```

★ Измените каждый элемент, возвращаемый массивом.

Этот код присоединяет строку в конец каждого элемента в массиве строк. Сам массив при этом не изменяется — вместо этого создается новая последовательность измененных строк.

```
var sandwiches = new[] { "ham and cheese", "salami with mayo",
    "turkey and swiss", "chicken cutlet" };
```

```
var sandwichesOnRye =
    from sandwich in sandwiches
    select $"{sandwich} on rye";
```

Секцию «select» можно рассматривать как механизм внесения одних и тех же изменений в каждый элемент последовательности — в данном случае в конец добавляется «on rye».

```
foreach (var sandwich in sandwichesOnRye)
    Console.WriteLine(sandwich);
```

Теперь в конец всех возвращаемых объектов добавляется строка «on rye».

Результат:

```
ham and cheese on rye
salami with mayo on rye
turkey and swiss on rye
chicken cutlet on rye
```

★ Выполните вычисления с последовательностями.

Методы LINQ могут использоваться для получения статистики о числовой последовательности.

```
var random = new Random();
var numbers = new List<int>();
int length = random.Next(50, 150);
for (int i = 0; i < length; i++)
    numbers.Add(random.Next(100));
```

Статический метод `String.Join` выполняет конкатенацию всех элементов последовательности в строку, а также задает разделитель для разделения элементов.

```
Console.WriteLine($"Stats for these {numbers.Count()} numbers:
The first 5 numbers: {String.Join(", ", numbers.Take(5))}
The last 5 numbers: {String.Join(", ", numbers.TakeLast(5))}
The first is {numbers.First()} and the last is {numbers.Last()}
The smallest is {numbers.Min()}, and the biggest is {numbers.Max()}
The sum is {numbers.Sum()}
The average is {numbers.Average():F2}");
```

Результат тестового запуска. Длина последовательности и входящие в нее числа будут случайными при каждом выполнении.

```
Stats for these 61 numbers:
The first 5 numbers: 85, 30, 58, 70, 60
The last 5 numbers: 40, 83, 75, 26, 75
The first is 85 and the last is 75
The smallest is 2, and the biggest is 99
The sum is 3444
The average is 56.46
```

Запросы LINQ выполняются только при обращении к результатам

При включении в код запросов LINQ используется **отложенное вычисление**. Это означает, что запрос LINQ на самом деле не выполняет перебор до того момента, когда в вашем коде будет выполнена команда, *использующая результаты запроса*. Возможно, это звучит немного странно, но когда вы увидите отложенное вычисление в действии, картина начнет проясняться. **Создайте новое консольное приложение** и добавьте следующий код:

Сделайте это!

```
class PrintWhenGetting
{
    private int instanceNumber;
    public int InstanceNumber
    {
        set { instanceNumber = value; }
        get
        {
            Console.WriteLine($"Getting #{instanceNumber}");
            return instanceNumber;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        var listOfObjects = new List<PrintWhenGetting>();
        for (int i = 1; i < 5; i++)
            listOfObjects.Add(new PrintWhenGetting() { InstanceNumber = i });

        Console.WriteLine("Set up the query");
        var result =
            from o in listOfObjects
            select o.InstanceNumber;

        Console.WriteLine("Run the foreach");
        foreach (var number in result)
            Console.WriteLine($"Writing #{number}");
    }
}
```

Console.WriteLine в get-методе не вызывается до момента непосредственного выполнения цикла foreach. Отложенное вычисление выглядит так:

```
Set up the query
Run the foreach
Getting #1
Writing #1
Getting #2
Writing #2
Getting #3
Writing #3
Getting #4
Writing #4
```

Странная ошибка компилятора? Убедитесь в том, что вы добавили в код две директивы using!

Вызов ToList или другого метода LINQ, который обращается к каждому элементу последовательно, обеспечивает немедленное выполнение запроса.

Запустите приложение. Обратите внимание: строка Console.WriteLine, которая выводит сообщение "Set up the query", выполняется до get-метода. Дело в том, что запрос LINQ не будет выполнен до цикла foreach.

Чтобы запрос был выполнен **немедленно**, можно вызвать метод LINQ, который должен перебрать весь список, — например, метод ToList, который преобразует его в List<T>. Добавьте следующую строку и измените foreach для использования нового списка List:

```
var immediate = result.ToList();

Console.WriteLine("Run the foreach");
foreach (var number in immediate)
    Console.WriteLine($"Writing #{number}");
```

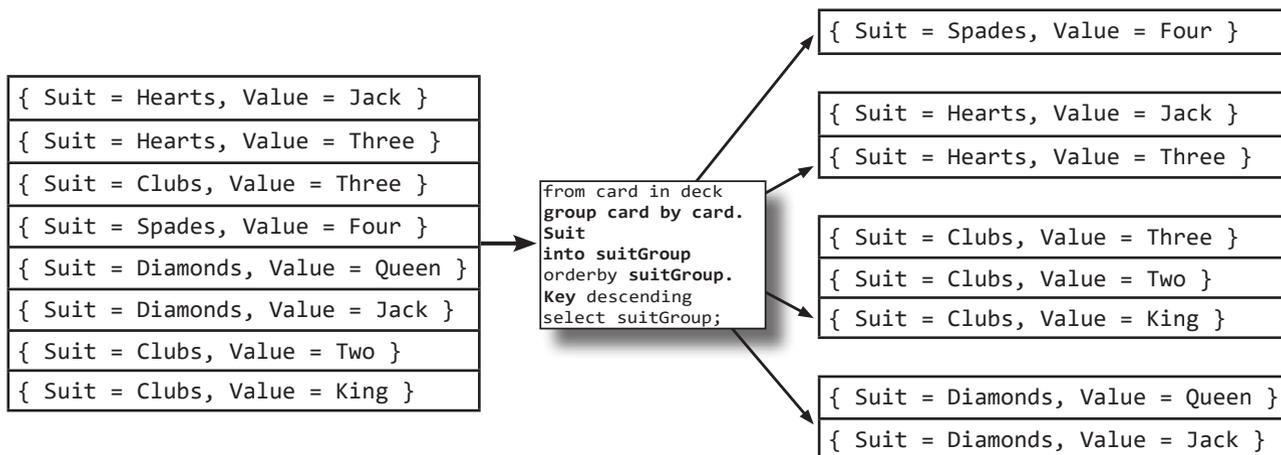
```
Set up the query
Getting #1
Getting #2
Getting #3
Getting #4
Run the foreach
Writing #1
Writing #2
Writing #3
Writing #4
```

Снова запустите приложение. На этот раз get-методы будут вызваны до того, как цикл foreach начнет выполняться, — и это разумно, потому что метод ToList должен обратиться к каждому элементу последовательности для преобразования в List. Такие методы, как Sum, Min и Max, тоже должны обратиться к каждому элементу последовательности, поэтому их применение также заставляет LINQ немедленно выполнить запрос.

далее ▶

Использование запросов *group* для разделения последовательности на группы

Иногда данные требуется разбить на группы. Допустим, Джимми хочет сгруппировать свои комиксы по десятилетиям, в которых они были опубликованы. А может быть, он хочет разбить их по ценам (дешевые — в одной коллекции, дорогие — в другой). Такая группировка может пригодиться во многих случаях. И тогда вам стоит воспользоваться **запросом LINQ *group***.



1 Создайте новое консольное приложение, добавьте классы и перечисления.

Создайте **новое консольное приложение .NET Core** с именем *CardLinq*. Затем перейдите на панель Solution Explorer, щелкните правой кнопкой мыши на имени проекта и выберите команду Add>>Existing Items (или Add>>Existing Files на Mac). Перейдите в папку, в которой был сохранен проект Two Decks из главы 8. Добавьте файлы с перечислениями **Suit** и **Value**, затем классы **Deck**, **Card** и **CardComparerByValue**.

Не забудьте **изменить пространство имен в каждом добавленном файле** и привести его в соответствие с пространством имен в Program.cs, чтобы метод Main мог получить доступ к добавленным классам.

Класс Deck был создан в проекте «Two Decks», который можно загрузить по ссылке в конце главы 8.

2 Обеспечьте возможность сортировки класса Card при помощи интерфейса **IComparable<T>**.

Мы будем использовать секцию LINQ *orderby* для сортировки групп, поэтому класс Card должен поддерживать сортировку. К счастью, данная возможность работает точно так же, как метод List.Sort, о котором вы узнали в главе 7. Измените класс Card, чтобы он **расширял интерфейс IComparable**:

```

class Card : IComparable<Card>
{
    public int CompareTo(Card other)
    {
        return new CardComparerByValue().Compare(this, other);
    }
}

```

// Остальной код класса остается без изменений

Мы также будем использовать методы LINQ *Min* и *Max* для нахождения наибольшей и наименьшей карты в каждой группе, а они тоже используют интерфейс *IComparable*.

3 Измените метод `Deck.Shuffle` для поддержки сцепления вызовов.

Класс `Shuffle` тасует колоду. Все, что нужно сделать для поддержки сцепления вызовов, — изменить его так, чтобы он возвращал ссылку на экземпляр `Deck` для только что перетасованной колоды:

```
public Deck Shuffle()
{
    // Остальной код класса остается без изменений
    return this;
}
```

Если метод `Shuffle` возвращает ссылку на тот же объект `Deck`, который был перетасован, вы можете вызвать его и присоединить дополнительные вызовы методов к результату.

4 Использование запроса LINQ с секцией `group...by` для группировки карт по мастям.

Метод `Main` получает 16 случайных карт. Для этого он сначала тасует колоду, а затем при помощи метода LINQ `Take` получает первые 16 карт. Затем запрос LINQ с секцией `group...by` используется для разделения колоды на меньшие последовательности, по одной для каждой масти среди 16 карт:

```
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        var deck = new Deck()
            .Shuffle()
            .Take(16);

        var grouped =
            from card in deck
            group card by card.Suit into suitGroup
            orderby suitGroup.Key descending
            select suitGroup;

        foreach (var group in grouped)
        {
            Console.WriteLine($"Group: {group.Key}
                Count: {group.Count()}
                Minimum: {group.Min()}
                Maximum: {group.Max()}");
        }
    }
}
```

Используйте методы LINQ `Count`, `Min` и `Max` для получения информации о каждой группе, возвращаемой запросом.

Теперь, когда метод `Shuffle` поддерживает сцепление вызовов, вызов метода LINQ `Take` можно присоединить к вызову `Shuffle`.

У каждой группы имеет-ся свойство `Key`, которое возвращает ее ключ — в данном случае масть.

Секция `group...by` в запросе LINQ разделяет последовательность на группы:

```
group card by card.Suit
into suitGroup
```

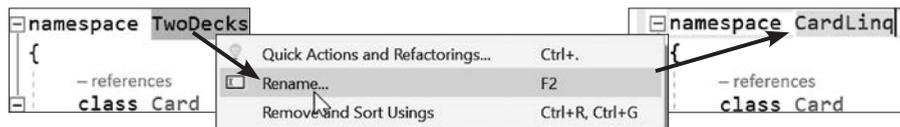
Ключевое слово `group` сообщает, какая последовательность содержит группируемые элементы; ключевое слово `by` задает критерий, используемый для определения групп; ключевое слово `into` объявляет новую переменную, которая используется другими секциями для ссылок на группы. Результат запроса `group` представляет собой последовательность последовательностей. Каждая группа представляется последовательностью, реализующей интерфейс `IGrouping<Suits, Card>` представляет группу карт (`Card`), которая использует масть (`Suits`) в качестве ключа группировки.

Наведите указатель мыши на переменную «`grouped`», чтобы увидеть ее тип.

(local variable) `IOrderedEnumerable<IGrouping<Suits, Card>> grouped`

Подсказки для IDE: Время менять имена!

Если вам потребуется изменить имя переменной, поля, свойства, пространства имен или класса, вы можете воспользоваться удобным средством рефакторинга, встроенным в Visual Studio. Просто щелкните на имени правой кнопкой мыши и выберите в меню команду «`Rename...`». IDE выделяет имя, отредактируйте его — IDE **автоматически переименует все его вхождения в коде**.



Используйте команду `Rename` для переименования переменных, полей, свойств, классов и пространств имен (и не только!). При переименовании одного вхождения IDE изменяет имя везде, где оно встречается в коде.



Анатомия запроса group

Давайте повнимательнее разберемся в том, как работают запросы group.

```
var grouped =
    from card in deck
```

Эта случайная выборка по случайности начинается с двух карт одной масти (червы), за которыми идет карта пиковой масти и две бубновые карты.

{ Suit = Hearts, Value = Jack }
{ Suit = Hearts, Value = Three }
{ Suit = Clubs, Value = Three }
{ Suit = Spades, Value = Four }
{ Suit = Diamonds, Value = Queen }
{ Suit = Diamonds, Value = Jack }
{ Suit = Clubs, Value = Two }
{ Suit = Clubs, Value = King }

Секция from работает точно так же, как в других использованных нами запросах LINQ. Она присваивает диапозонной переменной «card» каждую карту в последовательности — в данном случае в объекте Deck, который вы перетасовали и взяли из него несколько карт.

```
group card by card.Suit into suitGroup
```

Секция group...by перебирает последовательность, создавая новые группы с обнаружением каждого нового ключа. Таким образом, группы следуют в том же порядке, в каком вхождения мастей впервые встречаются в случайной выборке.

{ Suit = Hearts, Value = Jack }
{ Suit = Hearts, Value = Three }
{ Suit = Clubs, Value = Three }
{ Suit = Clubs, Value = Two }
{ Suit = Clubs, Value = King }
{ Suit = Spades, Value = Four }
{ Suit = Diamonds, Value = Queen }
{ Suit = Diamonds, Value = Jack }

Секция group...by разбивает карты на группы. Она включает выражение «by card.Suit» — оно сообщает, что ключом каждой группы является масть карты. Оно также объявляет новую переменную с именем suitGroup, которая будет использоваться остальными секциями для работы с группами.

Секция group...by группирует карты по card.Suit, так что ключом каждой группы является масть. Это означает, что все карты в каждой группе имеют одинаковую масть и все карты этой масти принадлежат данной группе. Секция orderby сортирует группы по ключу, в результате чего они располагаются в порядке их вхождения в перечень Suits (в обратном порядке): Spades, Hearts, Clubs и Diamonds.

```
orderby suitGroup.Key descending
select suitGroup;
```

{ Suit = Spades, Value = Four }
{ Suit = Hearts, Value = Jack }
{ Suit = Hearts, Value = Three }
{ Suit = Clubs, Value = Three }
{ Suit = Clubs, Value = Two }
{ Suit = Clubs, Value = King }
{ Suit = Diamonds, Value = Queen }
{ Suit = Diamonds, Value = Jack }

Секция group...by создает последовательность групп, реализующую интерфейс IGrouping. IGrouping расширяет IEnumerable и добавляет ровно один компонент: свойство с именем Key. Таким образом, каждая группа представляет собой последовательность других последовательностей — в данном случае это группа последовательностей Card, у которой ключом является масть карты (из перечисления Suits). Полный тип каждой группы имеет вид IGrouping<Suits, Card>, что означает, что это последовательность последовательностей Card, каждая из которых использует значение Suits в качестве ключа.

Использование запросов join для слияния данных из двух последовательностей

Каждый опытный коллекционер знает, что критические отзывы сильно влияют на цены. Джими следит за рейтингами на двух крупнейших агрегаторах отзывов на комиксы, MuddyCritic и Rotten Tornadoes. Теперь он хочет сопоставить их со своей коллекцией. Как ему это сделать?

На помощь приходит LINQ! Ключевое слово `join` позволяет объединить данные из двух последовательностей одним запросом. Для этого элементы одной последовательности сравниваются с соответствующими элементами второй последовательности. (LINQ хватается сообразительности на то, чтобы делать это эффективно, — каждая пара элементов сравнивается только в том случае, если это действительно необходимо.) Окончательный результат объединяет пары с совпадением.

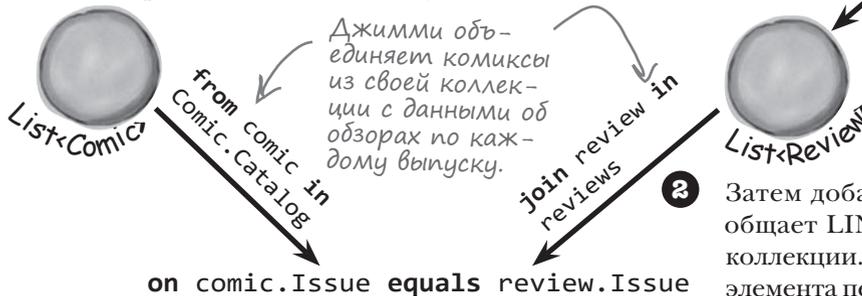
- 1 Запрос начинается с обычной секции `from`. Но вместо того чтобы указывать критерии отбора данных для включения в результаты, вы добавляете следующую конструкцию:

```
join имя in коллекция
```

Секция `join` приказывает LINQ перебрать обе последовательности, чтобы сопоставить пары, включающие по одному элементу каждой последовательности. *Имя* присваивается элементу, который будет извлекаться из объединенной коллекции при каждой итерации. Это имя используется в секции `where`.

Данные Джими хранятся в коллекции объектов `Review` с именем «reviews».

```
class Review
{
    public int Issue { get; set; }
    public Critics Critic { get; set; }
    public double Score { get; set; }
}
```



```
on comic.Issue equals review.Issue
```

- 3 Далее в запросе LINQ следуют секции `where` и `orderby`, как обычно. Запрос может завершаться обычной секцией `select`, но чаще всего возвращаются результаты, которые извлекают некие данные из одной коллекции, и другие данные из другой коллекции.



- 2 Затем добавляется секция `on`, которая сообщает LINQ, как следует объединять две коллекции. За ключевым словом следует имя элемента первой коллекции, ключевое слово `equals` и имя элемента второй коллекции, по которому они должны связываться.

{ Name = "Woman's Work", Issue = 36, Critic = MuddyCritic, Score = 37.6 }
{ Name = "Black Monday", Issue = 74, Critic = RottenTornadoes, Score = 22.8 }
{ Name = "Black Monday", Issue = 74, Critic = MuddyCritic, Score = 84.2 }
{ Name = "The Death of the Object", Issue = 97, Critic = MuddyCritic, Score = 98.1 }

Результатом является последовательность объектов, которые включают свойства `Name` и `Issue` из `Comic`, а также свойства `Critic` и `Score` из `Review`. Результат не может быть последовательностью объектов `Comic`, но также не может быть и последовательностью объектов `Review`, потому что ни один класс не содержит все эти свойства. Результатом является другая разновидность типа: *анонимный тип*.

Использование ключевого слова `new` для создания анонимных типов

Вы использовали ключевое слово `new` для создания экземпляров классов еще с главы 3. Каждый раз, когда вы используете его, вы включаете в программу новый тип (так что команда `new Guy()`; создает экземпляр типа `Guy`). Также ключевое слово `new` может использоваться без типа для создания **анонимного типа**. Это абсолютно допустимый тип, который содержит только свойства для чтения, но не имеет имени. Наш запрос возвращает тип, который объединяет комиксы Джимми с обзорами. Этот тип является анонимным. Вы можете добавить свойства в анонимный тип при помощи инициализатора объекта. Вот как это выглядит:

```
public class Program
{
    public static void Main()
    {
        var whatAmI = new { Color = "Blue", Flavor = "Tasty", Height = 37 };
        Console.WriteLine(whatAmI);
    }
}
```

Попробуйте вставить код в новое приложение и выполнить его. Вы получите следующий результат:

```
{ Color = Blue, Flavor = Tasty, Height = 37 }
```

Теперь наведите указатель мыши на `whatAmI` в IDE и взгляните на окно IntelliSense:

`whatAmI`



IDE точно знает, что это за тип: это объектный тип с двумя строковыми свойствами и свойством `int`. Просто у этого типа нет имени, поэтому он называется анонимным типом.

Переменная `whatAmI` относится к ссылочному типу, как любая другая ссылка на объект. Она указывает на объект, находящийся в куче, и может использоваться для обращения к компонентам этого объекта, в данном случае к двум его свойствам:

```
Console.WriteLine($"My color is {whatAmI.Color} and I'm {whatAmI.Flavor}");
```

Кроме того факта, что у анонимных типов нет имен, они не отличаются от всех остальных типов.



Теперь я понимаю, почему мы изучали ключевое слово `var`. Оно нужно для объявления анонимных типов.

Верно! Ключевое слово `var` используется для объявления анонимных типов.

Собственно, это одно из самых важных применений ключевого слова `var`.

а-НО-НИМ-НЫЙ, прил. неидентифицируемый по имени.

Приведенный ранее запрос LINQ, объединяющий комиксы Джимми с обзорами, возвращает анонимный тип. Этот запрос будет добавлен в приложение позднее в этой главе.

Возьми в руку карандаш



Джо, Боб и Алиса — ведущие мировые игроки в Go Fish. Код LINQ объединяет два массива с анонимными типами для генерирования списка их выигрышей. Прочитайте код и запишите результат, который будет выведен на консоль.

```
var players = new[]
{
    new { Name = "Joe", YearsPlayed = 7, GlobalRank = 21 },
    new { Name = "Bob", YearsPlayed = 5, GlobalRank = 13 },
    new { Name = "Alice", YearsPlayed = 11, GlobalRank = 17 },
};
```

```
var playerWins = new[]
{
    new { Name = "Joe", Round = 1, Winnings = 1.5M },
    new { Name = "Alice", Round = 2, Winnings = 2M },
    new { Name = "Bob", Round = 3, Winnings = .75M },
    new { Name = "Alice", Round = 4, Winnings = 1.3M },
    new { Name = "Alice", Round = 5, Winnings = .7M },
    new { Name = "Joe", Round = 6, Winnings = 1M },
};
```

← Мы используем `var` и `new[]` для создания массивов с анонимными типами.

```
var playerStats =
    from player in players
    join win in playerWins
    on player.Name equals win.Name
    orderby player.Name
    select new
    {
        Name = player.Name,
        YearsPlayed = player.YearsPlayed,
        GlobalRank = player.GlobalRank,
        Round = win.Round,
        Winnings = win.Winnings,
    };
```

```
foreach (var stat in playerStats)
    Console.WriteLine(stat);
```

```
{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 2, Winnings = 2 }
```

```
{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 4, Winnings = 1.3 }
```

```
.....
.....
.....
.....
```

Этот код выводит шесть строк на консоль. Мы заполнили первые две строки, чтобы вам было проще. Обратите внимание: обе строки содержат одинаковые имена («Alice»). Запрос `join` находит все совпадения между ключевыми свойствами обеих последовательностей. Если совпадений несколько, результаты будут включать по одному элементу для каждого совпадения. Если во входной последовательности присутствует ключ, для которого нет совпадения в другой последовательности, он не включается в результаты.



Возьми в руку карандаш



Решение

Джо, Боб и Алиса — ведущие мировые игроки в Go Fish. Код LINQ объединяет два массива с анонимными типами для генерирования списка их выигрышей. Прочитайте код и запишите результат, который будет выведен на консоль.

```
{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 2, Winnings = 2 }
.....
{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 4, Winnings = 1.3 }
.....
{ Name = Alice, YearsPlayed = 11, GlobalRank = 17, Round = 5, Winnings = 0.7 }
.....
{ Name = Bob, YearsPlayed = 5, GlobalRank = 13, Round = 3, Winnings = 0.75 }
.....
{ Name = Joe, YearsPlayed = 7, GlobalRank = 21, Round = 1, Winnings = 1.5 }
.....
{ Name = Joe, YearsPlayed = 7, GlobalRank = 21, Round = 6, Winnings = 1 }
```

Часто задаваемые вопросы

В: Можете еще раз объяснить, как работает var?

О: Да, разумеется. Ключевое слово var решает неочевидную проблему, которая возникает при использовании LINQ. Обычно при вызове метода или выполнении команды абсолютно ясно, с какими типами вы работаете. Например, если у вас есть метод, который возвращает строку, можно сохранить результаты в строковой переменной или в поле.

Однако с LINQ все не так просто. Когда вы строите команду LINQ, она может вернуть анонимный тип, который *не определен нигде в программе*. Да, вы знаете, что это будет некая последовательность. Но что это за последовательность? Неизвестно — потому что объекты, содержащиеся в последовательности, полностью зависят от содержимого запроса LINQ. Для примера возьмем следующий запрос из кода, написанного ранее для Джимми. Изначально был написан следующий запрос:

```
IEnumerable<Comic> mostExpensive =
    from comic in Comic.Catalog
    where Comic.Prices[comic.Issue] > 500
    orderby -Comic.Prices[comic.Issue]
    select comic;
```

Но потом первая строка была изменена для использования ключевого слова var:

```
var mostExpensive =
```

И это удобно. Например, если изменить последнюю строку и привести ее к следующему виду:

```
select new {
    Name = comic.Name,
    IssueNumber = $"#{comic.Issue}"
};
```

обновленный запрос вернет другой (но абсолютно допустимый!) тип — анонимный тип с двумя компонентами: строкой Name и строкой IssueNumber. Но в нашей программе нет определения класса для этого типа! Вам не нужно запускать программу, чтобы увидеть, как определяется этот тип, но переменная mostExpensive должна быть объявлена с *каким-то* типом.

Вот почему C# предоставляет ключевое слово var, которое сообщает компилятору: «Да, мы знаем, что это допустимый тип, но пока не можем точно сказать, что это за тип. Может, ты это как-нибудь сам определишь и не будешь нас беспокоить? Большое спасибо».

Подсказки для IDE: средства рефакторинга

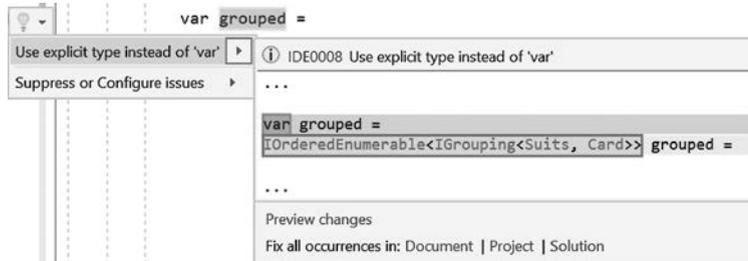
Переключение переменных между неявными и явными типами

При работе с запросами с группировкой часто используется ключевое слово `var` — не только потому, что это удобно, но и потому, что тип, возвращаемый запросом, может быть довольно громоздким.

```
var grouped =
    from card in deck
    group card by card.Suit into suitGroup
    orderby suitGroup.Key descending
    select suitGroup;
```

→ (local variable) IOrderedEnumerable<IGrouping<Suits, Card>> grouped

Но иногда наш код проще понять при использовании **явного типа**. К счастью, IDE упрощает переключение между неявным типом (`var`) и явным типом переменной. Откройте меню Quick Actions и выберите «**Use explicit type instead of 'var'**», чтобы преобразовать `var` в явный тип.

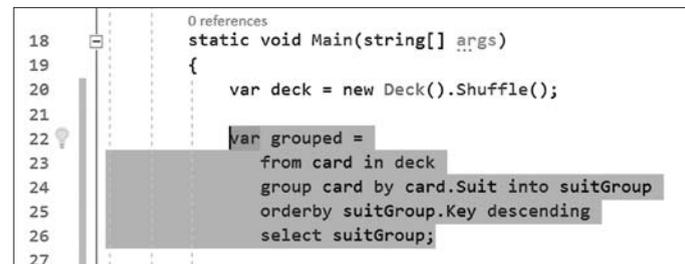


← Меню Quick Actions может использоваться для замены неявного типа «`var`» явным типом — в данном случае `IOrderedEnumerable<IGrouping<Suits, Card>>` (согласитесь, вводить такое имя никому не захочется!).

Вы также можете выбрать «**Use implicit type**» из меню Quick Actions, чтобы вернуть переменную к объявлению с `var`.

Выделение методов

Часто ваш код будет проще читаться, если вы возьмете большой метод и разобьете его на несколько меньших. Вот почему одним из самых распространенных приемов рефакторинга кода является **выделение методов**, т. е. извлечение блока кода из большого метода и перемещение его в отдельный метод. IDE



предоставляет в ваше распоряжение полезные инструменты, упрощающие решение этой задачи.

Для начала выделите блок кода. Затем выберите команду **Refactor>>Extract Method** из меню Edit (в Windows) или **Extract Method** (на Mac) из меню Quick Actions.

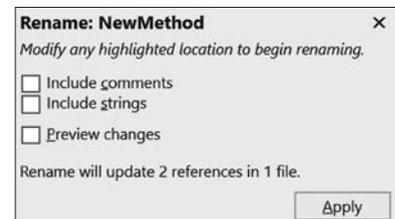
Как только вы это сделаете, IDE переместит выделенный код в новый метод с именем `NewMethod` и возвращаемым типом,

соответствующим типу, возвращаемому кодом. Затем IDE немедленно активизирует функцию переименования, чтобы вы могли начать ввод нового имени метода.

На этом снимке выделен весь запрос LINQ из проекта группировки карт, приведенного ранее в этой главе. После выделения метода он выглядит так:

```
IOrderedEnumerable<IGrouping<Suits, Card>> grouped = NewMethod(deck);
```

При этом остается новая переменная `grouped` с явной типизацией — IDE определила, что эта переменная используется позднее в коде. Это еще один пример того, как IDE помогает писать более чистый код.



Часть Задаваемые Вопросы

В: Можете чуть подробнее рассказать, как работает `join`?

О: `join` работает с двумя последовательностями. Допустим, вы печатаете футболки для игроков, используя коллекцию с именем `players`. Коллекция содержит объекты со свойством `Name` и свойством `Numbers`. А если для игроков с двузначными номерами потребуется специальный дизайн футболок? Из коллекции можно извлечь игроков с номерами, большими 10:

```
var doubleDigitPlayers =  
    from player in players  
    where player.Number > 10  
    select player;
```

А если необходимо еще получить размеры футболок? Если у вас имеется последовательность с именем `jerseys`, элементы которой содержат свойство `Number` и свойство `Size`, запрос `join` хорошо подойдет для объединения данных:

```
var doubleDigitShirtSizes =  
    from player in players  
    where player.Number > 10  
    join shirt in jerseys  
    on player.Number equals shirt.Number  
    select shirt;
```

В: Запрос возвращает набор объектов. А если я хочу связать каждого игрока с его размером футболки, а номер меня не интересует?

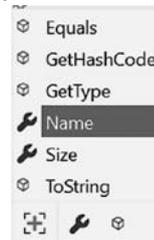
О: Для таких ситуаций нужны **анонимные типы** — вы можете сконструировать анонимный тип, который содержит только те данные, которые вам нужны. Также анонимные типы позволяют выбрать данные из разных коллекций, которые вы объединяете.

Таким образом, вы можете выбрать имя и размер футболки игрока, и ничего более:

```
var doubleDigitShirtSizes =  
    from player in players  
    where player.Number > 10  
    join shirt in jerseys  
    on player.Number equals shirt.Number  
    select new {  
        player.Name,  
        shirt.Size  
    };
```

IDE достаточно умна, чтобы понять, какие результаты будет создавать запрос. Если вы создадите цикл для перебора результатов, то при вводе имени переменной IDE откроет список `IntelliSense`:

```
foreach (var size in doubleDigitShirtSizes)  
    size.
```



Обратите внимание: список содержит свойства `Name` и `Size`. Если вы добавите новые элементы в секцию `select`, они также появятся в списке, потому что запрос создаст новый анонимный тип с другими компонентами.

В: Как написать метод, возвращающий анонимный тип?

О: Никак. Методы не могут возвращать анонимные типы. `C#` не дает такой возможности. Также невозможно объявить поле или свойство с анонимным типом или использовать анонимный тип для параметра метода или конструктора, именно поэтому вы не сможете использовать ключевое слово `var` ни в одной из этих ситуаций.

И если задуматься, все это вполне логично. Каждый раз, когда вы используете `var` в объявлении переменной, вы всегда должны привести значение, которое используется компилятором `C#` или IDE для определения типа переменной. Если вы объявляете поле или параметр метода, указать это значение невозможно, а следовательно, `C#` не сможет определить тип. (Да, для свойства можно задать значение, но это не одно и то же — формально значение задается только перед вызовом конструктора.)

Ключевое слово `var` может использоваться только при объявлении переменных. Оно не может использоваться с полем или свойством или для написания метода, который возвращает анонимный тип или получает его в параметре.



Упражнение

Используйте то, что вы узнали от LINQ, для **построения нового консольного приложения с именем *JimmyLinq***, которое упорядочивает коллекцию комиксов Джимми. Начните с **добавления перечисления Critics** с двумя компонентами, *MuddyCritic* и *RottenTornadoes*, и перечисления *PriceRange* с двумя компонентами *Cheap* и *Expensive*. Затем **добавьте класс *Review*** с тремя автоматическими свойствами: `int Issue`, `Critics Critic` и `double Score`.

Вам понадобятся данные, поэтому добавьте в класс *Comic* статическое поле, возвращающее последовательность обзоров:

```
public static readonly IEnumerable<Review> Reviews = new[] {
    new Review() { Issue = 36, Critic = Critics.MuddyCritic, Score = 37.6 },
    new Review() { Issue = 74, Critic = Critics.RottenTornadoes, Score = 22.8 },
    new Review() { Issue = 74, Critic = Critics.MuddyCritic, Score = 84.2 },
    new Review() { Issue = 83, Critic = Critics.RottenTornadoes, Score = 89.4 },
    new Review() { Issue = 97, Critic = Critics.MuddyCritic, Score = 98.1 },
};
```

Метод *Main* и два метода, которые из него вызываются:

```
static void Main(string[] args) {
    var done = false;
    while (!done) {
        Console.WriteLine(
            "\nPress G to group comics by price, R to get reviews, any other key to quit\n");
        switch (Console.ReadKey(true).KeyChar.ToString().ToUpper()) {
            case "G":
                done = GroupComicsByPrice();
                break;
            case "R":
                done = GetReviews();
                break;
            default:
                done = true;
                break;
        }
    }
}
```

Присмотритесь к циклу while. Он использует команду switch для определения вызываемого метода. Метод возвращает true, присваивая <<done>> значение false, а цикл while выполняет очередную итерацию. Если пользователь нажимает любую другую клавишу, кроме G и R, то <<done>> присваивается true, а цикл завершается.

Циклы foreach в методе *GroupComicsByPrice* являются вложенными: один цикл выполняется внутри другого. Внешний цикл выводит информацию о каждой группе, а внутренний перебирает группу.

```
private static bool GroupComicsByPrice() {
    var groups = ComicAnalyzer.GroupComicsByPrice(Comic.Catalog, Comic.Prices);
    foreach (var group in groups) {
        Console.WriteLine($"{group.Key} comics:");
        foreach (var comic in group)
            Console.WriteLine($"#{comic.Issue} {comic.Name}: {Comic.Prices[comic.Issue]:c}");
    }
    return false;
}

private static bool GetReviews() {
    var reviews = ComicAnalyzer.GetReviews(Comic.Catalog, Comic.Reviews);
    foreach (var review in reviews)
        Console.WriteLine(review);
    return false;
}
```

*Методы *GroupComicsByPrice* и *GetReviews* вызывают методы статического класса *ComicAnalyzer* (который вскоре будет написан), которые выполняют запросы LINQ.*

Ваша задача — создать **статический класс с именем *ComicAnalyzer*** с тремя статическими методами (два из которых объявлены открытыми):

- Приватный статический метод с именем *CalculatePriceRange* получает ссылку на *Comic* и возвращает *PriceRange.Cheap*, если цена ниже 100, или *PriceRange.Expensive* в противном случае.
- *GroupComicByPrice* упорядочивает комиксы по цене, а затем группирует их по *CalculatePriceRange(comic)* и возвращает последовательность групп комиксов (`IEnumerable<IGrouping<PriceRange, Comic>>`).
- *GetReviews* упорядочивает комиксы по номеру выпуска, а затем выполняет объединение (см. ранее в этой главе) и возвращает последовательность строк следующего вида: *MuddyCritic rated #74 'Black Monday' 84.20*



Упражнение
Решение

Используйте то, что вы узнали от LINQ, для построения нового консольного приложения с именем **JimmyLinq**, которое упорядочивает коллекцию комиксов Джимми. Начните с добавления перечисления **Critics** с двумя компонентами, **MuddyCritic** и **RottenTornadoes**, и перечисления **PriceRange** с двумя компонентами, **Cheap** и **Expensive**. Затем добавьте класс **Review** с тремя автоматическими свойствами: **int Issue**, **Critics Critic** и **double Score**.

Добавьте перечисления Critics и PriceRange:

```
enum Critics {
    MuddyCritic,
    RottenTornadoes,
}

enum PriceRange {
    Cheap,
    Expensive,
}
```

Затем добавьте класс Review:

```
class Review {
    public int Issue { get; set; }
    public Critics Critic { get; set; }
    public double Score { get; set; }
}
```

Когда это будет сделано, добавьте статический класс **ComicAnalyzer** с приватным методом **PriceRange** и открытыми методами **GroupComicByPrice** и **GetReviews**:

```
using System.Collections.Generic;
using System.Linq;
```

Не забудьте про директиву using.

```
static class ComicAnalyzer
```

```
{
    private static PriceRange CalculatePriceRange(Comic comic)
    {
        if (Comic.Prices[comic.Issue] < 100)
            return PriceRange.Cheap;
        else
            return PriceRange.Expensive;
    }
}
```

А вы сможете найти ошибку? Она весьма хитроумна...

Мы намеренно допустили ошибку в методе CalculatePriceRange. Проследите за тем, чтобы код вашего метода совпадал с кодом этого решения.

Средства рефакторинга, представленные ранее в этой главе, упростят правильное определение возвращаемого типа метода GroupComicsByPrice.

```
public static IEnumerable<IGrouping<PriceRange, Comic>> GroupComicsByPrice(
    IEnumerable<Comic> comics, IReadOnlyDictionary<int, decimal> prices)
{
    IEnumerable<IGrouping<PriceRange, Comic>> grouped =
        from comic in comics
        orderby prices[comic.Issue]
        group comic by CalculatePriceRange(comic) into priceGroup
        select priceGroup;

    return grouped;
}
```

Мы предложили вам упорядочить комиксы по цене, а затем сгруппировать их. В результате каждая группа будет отсортирована по цене, потому что группы создаются в порядке перебора последовательности секцией group...by.

```
public static IEnumerable<string> GetReviews(
    IEnumerable<Comic> comics, IEnumerable<Review> reviews)
{
    var joined =
        from comic in comics
        orderby comic.Issue
        join review in reviews on comic.Issue equals review.Issue
        select $"{review.Critic} rated #{comic.Issue} '{comic.Name}' {review.Score:0.00}";

    return joined;
}
```

Очень похоже на запрос join, описанный ранее в этой главе.

А вы столкнулись с ошибкой компилятора «непоследовательные уровни доступа», которая сообщает, что возвращаемый тип обладает меньшей доступностью, чем метод? Это происходит тогда, когда класс помечен как открытый, но содержит компоненты с внутренней доступностью (что происходит, если модификатор доступа не указан). Проследите за тем, чтобы классы или перечисления не были помечены как открытые.



Спасибо за помощь с коллекцией! Уж теперь я стану самым главным фанатом.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Секция `group...by` приказывает LINQ сгруппировать результаты — при использовании этой секции LINQ создает последовательность последовательностей групп.
- Каждая группа содержит элементы с общим элементом, который называется **ключом** группы. Используйте ключевое слово `by` для определения ключа для группы. Каждая последовательность группы содержит компонент `Key`, являющийся ключом группы.
- Запросы `join` используют секцию `on...equals`, чтобы сообщить LINQ, как следует связывать пары элементов.
- Секция `join` используется для объединения двух коллекций в одном запросе. При этом LINQ сравнивает каждый элемент первой коллекции с каждым элементом второй коллекции и включает в результат пары с совпадениями.
- При выполнении запроса `join` вы обычно хотите получить набор результатов, включающий некие данные из одной коллекции и другие данные из другой коллекции. Секция `select` позволяет построить нужный результат из обеих коллекций.
- Используйте `select new` для построения специализированных результатов запросов LINQ с анонимным типом, включающим в итоговую последовательность только нужные вам свойства.
- Запросы LINQ используют **отложенное выполнение**. Это означает, что они не выполняются до того момента, когда результаты запроса будут использоваться в команде.
- Ключевое слово `new` создает новый экземпляр **анонимного типа**, или объект с правильно определенным типом, не имеющим имени. Компоненты, указанные в команде `new`, становятся автоматическими свойствами анонимного типа.
- Используйте команду **Rename** в Visual Studio для удобного переименования всех вхождений имени переменной, поля, свойства, класса или пространства имен.
- Используйте меню Quick Actions в Visual Studio для преобразования объявления `var` к **явному типу** или возврата к `var` (т. е. неявному типу).
- Один из самых распространенных приемов рефакторинга — **выделение методов**. Функция `Extract Method` в Visual Studio позволяет очень легко преобразовать блок кода в отдельный метод.
- **Ключевое слово `var` может использоваться** только при объявлении переменной. Вы не сможете написать метод, который возвращает анонимный тип или получает его в параметре, а также использовать `var` с полем или свойством.

Модульные тесты помогают понять, как работает код

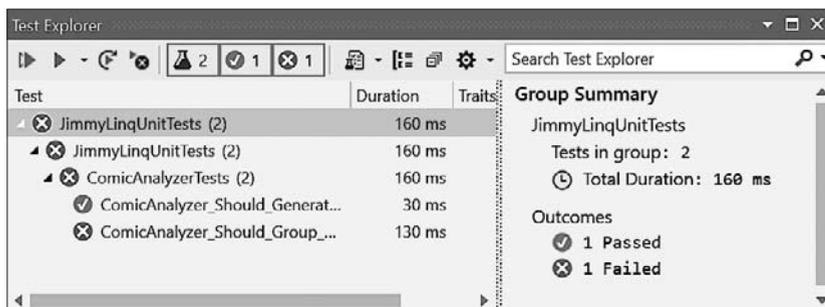
Мы намеренно оставили ошибку в приведенном коде... но является ли эта ошибка *единственной*? Написать код, который делает не совсем то, что предполагалось, очень просто. К счастью, существуют различные способы поиска ошибок для их исправления. **Модульные тесты** (unit tests) представляют собой *автоматизированные* тесты, которые проверяют, что ваш код работает именно так, как предполагалось. Каждый модульный тест реализуется в виде метода, проверяющего работоспособность конкретной части кода. Если метод выполняется без выдачи исключения, тест считается пройденным. Если же при выполнении выдается исключение, тест не проходит. Для многих крупных программ создаются **наборы** текстов, покрывающие большую часть кода.

Visual Studio содержит встроенные средства тестирования, которые помогают писать тесты и следить за тем, какие из них проходят или не проходят. Для создания модульных тестов в этой книге будет использоваться **MSTest** — фреймворк модульного тестирования (т. е. набор классов, предоставляющих средства для написания модульных тестов), разработанный компанией Microsoft.

В Visual Studio также поддерживаются модульные тесты, написанные в NUnit и xUnit — двух популярных фреймворках модульного тестирования для кода C# и .NET, расположенных с открытым кодом.

Окно Test Explorer в Visual Studio для Windows

Откройте окно Test Explorer командой *View>>Test Explorer* из главного меню. В левой части окна отображаются модульные тесты, а в правой — результаты их последнего выполнения. Кнопки панели инструментов предназначены для запуска всех тестов, запуска одного теста и повторения последнего запуска.



Когда вы добавляете тесты в решение, вы можете **запустить** тесты кнопкой **Run All Tests**. **Отладка** модульных тестов в Windows осуществляется командой **Tests>>Debug**, а на Mac — кнопкой **Debug All Tests** в окне инструментов **Unit Tests**.

Окно Unit Tests в Visual Studio для Mac

Откройте окно Unit Tests командой меню *View>>Tool Windows>>Unit Tests*. Окно содержит кнопки для запуска или отладки тестов. Когда вы запускаете модульные тесты, IDE выводит результаты в окне Test Results (обычно в нижней части окна IDE).

Трассировка стека помогает найти конкретную строку, в которой произошел сбой теста.

```
at JimmyLinqUnitTests.ComicAnalyzerTests.ComicAnalyzer_Should_Group_Comics() in /Users/andrewstellman/Projects/RefactoredJimr
```

В главе 3 вы узнали о прототипах — ранних версиях игр, в которые **LINQ и лямбда-выражения** можно играть, тестировать, анализировать и совершенствовать. Вы узнали, что эта концепция применима к любым проектам, не только к играм. То же самое можно сказать о тестировании. Иногда идея тестирования может показаться немного абстрактной. Изучение процесса тестирования разработчиками игр поможет вам привыкнуть к этой идее и сделает концепцию тестирования более понятной.



Тестирование

Разработка игр... и не только

Как только у вас появится работоспособный прототип игры, можно подумать о **тестировании видеоигры**. Если вы представите свою игру пользователям и получите от них обратную связь, это может в корне изменить ситуацию: в одном случае вы получаете игру, которая всем нравится, а в другом — поделку, которая раздражает новых пользователей, оставляя у них неудовлетворительные впечатления. Если вы когда-либо играли в игру, в которой вам было совершенно непонятно, что от вас нужно, или встречались головоломки, которые требовали сверхчеловеческих способностей, тогда вы знаете, что происходит в играх, которые не прошли достаточного **игрового тестирования**.

Существует несколько подходов к тестированию игр, которые стоит рассмотреть, когда вы начнете проектировать и создавать игры:

- **Индивидуальное игровое тестирование:** предложите знакомым вам людям поиграть в игру (желательно под вашим наблюдением). В самом неформальном варианте вы просто даете игру своему другу и обсуждаете его впечатления прямо по ходу игры. Если тестирующий будет постоянно говорить, что (по его мнению) от него хочет игра и что он думает об игровом процессе, это поможет вам спроектировать игру, которая нравится другим людям и будет доступной для них. Не предоставляйте тестирующим слишком подробных инструкций и обращайтесь особое внимание, если у них возникнут проблемы. Это поможет вам оценить, понятна ли игроку суть игры, и осознать, что некоторые включенные в игру механики окажутся неочевидными для пользователя. Запишите все мнения, высказанные тестирующими, чтобы вы могли исправить все выявленные недостатки.

Получение обратной связи может осуществляться как неформально, так и в более формальной обстановке: вы составляете список задач, которые предлагается выполнить пользователю, и **анкету** для получения обратной связи по игре. Зачастую обратную связь стоит запрашивать при добавлении новых возможностей, и вам стоит обращаться к людям с просьбой об игровом тестировании на самой ранней стадии разработки, чтобы проблемы с игрой обнаруживались на том этапе, когда их будет проще всего исправить.

- **Бета-версии:** когда вы будете готовы представить свой проект более широкой аудитории, предложите большей группе людей поиграть в нее, прежде чем публиковать игру для всех желающих. Бета-версии прекрасно подходят для выявления потенциальных проблем с нагрузкой и быстродействием в игре. Обычно обратная связь от бета-тестирования анализируется через журналы. В журнале можно зарегистрировать время, потраченное пользователями на выполнение различных операций в игре, и по собранным данным выявить проблемы с выделением ресурсов в игре. Иногда при этом также выявляются области, недостаточно понятные для пользователей. Как правило, тестирующие подают заявки на участие в бета-тестировании, так что вы можете позднее расспросить их об игровом опыте и получить содействие в диагностике проблем, выявленных в журнальных данных.
- **Структурированные тесты проверки качества:** в большинстве игр существуют специализированные тесты, которые выполняются как часть процесса разработки. Обычно эти тесты базируются на представлении о том, как должна работать игра, и они могут быть автоматизированными или ручными. Такое тестирование проводится для того, чтобы убедиться в том, что продукт работает так, как ожидалось. Идея проведения тестов проверки качества заключается в том, чтобы обнаружить как можно больше ошибок до того, как с ними столкнется пользователь. Ошибки регистрируются с четкими пошаговыми инструкциями, чтобы их можно было воспроизвести и исправить. Затем они обрабатываются в порядке убывания их влияния на впечатления игрока от игры и исправляются в соответствии с этим приоритетом. Если игра «падает» каждый раз, когда пользователь входит в комнату, вероятно, эту ошибку стоит исправить до некорректной прорисовки оружия в этой комнате.

Многие команды разработки пытаются по возможности автоматизировать процесс тестирования и запускать тесты перед каждым сохранением изменений в кодовой базе. При таком подходе они могут быть уверены в том, что при исправлении или добавлении новой функциональности не будут случайно внесены новые ошибки.

Добавление проекта модульного теста в приложение Джимми

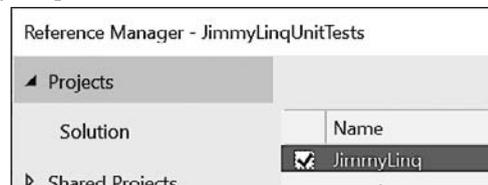
1 Добавьте новый проект MSTest (.NET Core).

Щелкните правой кнопкой мыши на имени решения в Solution Explorer, затем выберите команду **Add>>New Project...** из меню. Проследите за тем, чтобы был выбран вариант **MSTest Test Project (.NET Core)**: в Windows воспользуйтесь полем «Search for Templates» для поиска MSTest; в macOS выберите вариант Tests из категории «Web and Console», чтобы увидеть шаблон проекта. Присвойте проекту имя *JimmyLinqUnitTests*.

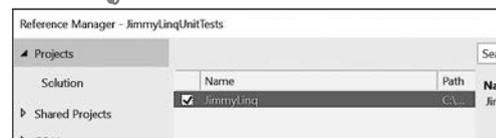
2 Добавьте зависимость от существующего проекта.

Мы будем строить модульные тесты для класса *ComicAnalyzer*. Если одно решение включает два проекта, эти проекты *независимы* — по умолчанию классы одного проекта не могут использовать классы другого проекта, поэтому необходимо настроить зависимость, чтобы модульные тесты могли пользоваться классом *ComicAnalyzer*.

Раскройте проект *JimmyLinqUnitTests* в окне Solution Explorer, затем щелкните правой кнопкой мыши на строке Dependencies и выберите из меню команду **Add Reference....** Выберите проект *JimmyLinq*, созданный для этого упражнения.

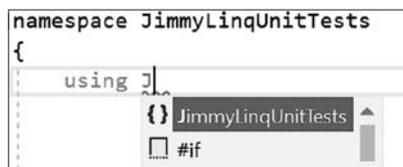


В Windows и macOS окно Reference Manager в Visual Studio выглядят по-разному, но работает одинаково.



3 Объявите класс ComicAnalyzer открытым.

Когда среда Visual Studio добавила проект с модульными тестами, она создала класс с именем *UnitTest1*. Отредактируйте файл *UnitTest1.cs* и попробуйте добавить директиву `using JimmyLinq`; в пространстве имен:



Вот почему мы предложили вам удалить модификатор доступа «public» из всех классов и перечислений проекта JimmyLinq — и вы сможете исследовать модификатор доступа «internal» в Visual Studio.

Хм-м, что-то не так: IDE не позволяет добавить директиву. Дело в том, что проект *JimmyLinq* не содержит открытых классов, перечислений или других компонентов. Попробуйте изменить перечисление *Critics*, чтобы объявить его открытым (**public enum Critics**), затем вернитесь и попробуйте добавить директиву `using`. Теперь все получается! IDE видит, что пространство имен *JimmyLinq* содержит открытые компоненты, и добавляет его в окно. Теперь измените объявление *ComicAnalyzer* и добавьте модификатор `public`: **public static class ComicAnalyzer**.

И снова что-то не так. Компилятор выдал ошибки «непоследовательные уровни доступа»?

```
Inconsistent accessibility: return type 'IEnumerable<IGrouping<PriceRange, Comic>>' is less accessible than method 'ComicAnalyzer.GroupComicsByPrice(IEnumerable<Comic>, IReadOnlyDictionary<int, decimal>)'
```

Проблема в том, что класс *ComicAnalyzer* объявлен открытым, но он предоставляет компоненты, не имеющие модификаторов доступа, из-за чего они интерпретируются как внутренние (*internal*) — и остаются невидимыми для других проектов в решении. **Добавьте модификатор доступа public ко всем классам и перечислениям проекта JimmyLinq.** Теперь решение снова строится нормально.

Первый модульный тест

IDE добавила в проект MSTest класс с именем UnitTest1. Переименуйте класс (и файл) и присвойте ему имя ComicAnalyzerTests. Класс содержит тестовый метод с именем TestMethod1. Присвойте методу содержательное имя: ComicAnalyzer_Should_Group_Comics. Код класса модульного теста выглядит так:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace JimmyLinqUnitTests
{
    using JimmyLinq;
    using System.Collections.Generic;
    using System.Linq;

    [TestClass]
    public class ComicAnalyzerTests
    {
        IEnumerable<Comic> testComics = new[]
        {
            new Comic() { Issue = 1, Name = "Issue 1"},
            new Comic() { Issue = 2, Name = "Issue 2"},
            new Comic() { Issue = 3, Name = "Issue 3"},
        };

        [TestMethod]
        public void ComicAnalyzer_Should_Group_Comics()
        {
            var prices = new Dictionary<int, decimal>()
            {
                { 1, 20M },
                { 2, 10M },
                { 3, 1000M },
            };

            var groups = ComicAnalyzer.GroupComicsByPrice(testComics, prices);

            Assert.AreEqual(2, groups.Count());
            Assert.AreEqual(PriceRange.Cheap, groups.First().Key);
            Assert.AreEqual(2, groups.First().First().Issue);
            Assert.AreEqual("Issue 2", groups.First().First().Name);
        }
    }
}
```

Visual Studio добавляет эту директиву using над объявлением пространства имен.

Переименуйте тестовый класс (и убедитесь в том, что файл также был переименован).

Присвойте содержательное имя тестовому методу.

Группы сортируются по возрастанию цены, так что первым элементом первой группы должен быть выпуск № 2.

Когда вы запускаете модульные тесты, IDE ищет любой класс, перед которым стоит метка [TestClass]. Эта метка называется **атрибутом**. Класс теста включает тестовые методы, которые должны быть помечены атрибутом [TestMethod].

Модульные тесты MSTest используют **класс Assert**, который содержит статические методы, проверяющие, что поведение вашего кода соответствует ожиданиям. Данный модульный тест использует метод **Assert.AreEqual**. Этот метод получает два параметра, ожидаемый результат (то, что, по вашему мнению, должен сделать код) и фактический результат (то, что он делает), и если они не равны — выдает исключение.

Для этого теста создаются очень ограниченные тестовые данные: последовательность из трех комиксов и словарь с тремя ценами. Затем тест вызывает **GroupComicsByPrice** и при помощи **Assert.AreEqual** проверяет, что результат соответствует нашим ожиданиям.

Взгляните повнимательнее на ожидаемые результаты. Тестовые данные состоят из 3 комиксов: 2 с ценой ниже 100 и 1 с ценой выше 100. Следовательно, для них должны быть созданы две группы: сначала группа с 2 дешевыми, затем группа с 1 дорогим комиксом.

Запустите тест командой меню **Test>>Run All Tests** (Windows) или **Run>>Run Unit Tests** (Mac). IDE открывает окно Test Explorer (Windows) или панель Test Results (Mac) с результатами теста:

Test method JimmyLinqUnitTests.ComicAnalyzerTests.ComicAnalyzer_Should_Group_Comics threw exception: System.Collections.Generic.KeyNotFoundException: The given key '2' was not present in the dictionary.

Модульный тест **не прошел**. Найдите значок **✘ 1** в Windows или сообщение **Failed: 1** в нижней части окна IDE в Visual Studio для Mac — так вы определите количество непрошедших модульных тестов.

А вы ожидали, что модульный тест не пройдет? Сможете ли вы определить, что не так с этим тестом?

По следу



Задача модульного тестирования — поиск мест, в которых ваш код работает не так, как ожидалось, и определение причин. Шерлок Холмс однажды сказал: «Худшая ошибка — строить теорию, не собрав данные».

Начнем с тестовых утверждений

Хорошей отправной точкой всегда становятся тестовые утверждения (assertions), включенные в тест, потому что они сообщают, какой конкретный код тестируется и какое поведение от него ожидается. Тестовые утверждения из нашего модульного теста:

```
Assert.AreEqual(2, groups.Count());
Assert.AreEqual(PriceRange.Cheap, groups.First().Key);
Assert.AreEqual(2, groups.First().First().Issue);
Assert.AreEqual("Issue 2", groups.First().First().Name);
```

Если посмотреть на тестовые данные, поставляемые методу GroupComicByPrice, эти тестовые утверждения выглядят правильно. Они действительно должны возвращать две группы. Первая группа должна иметь ключ PriceRange.Cheap. Группы сортируются по возрастанию цены, так что первый комикс в первой группе должен иметь свойства Issue = 2 и Name = "Issue 2" — именно это проверяют наши тестовые утверждения. Таким образом, если возникает проблема, она кроется не здесь — эти тестовые утверждения выглядят правильными.

Обратимся к трассировке стека

Вы уже видели много исключений. К каждому исключению прилагается **трассировка стека** — список всех вызовов методов, сделанных программой непосредственно в той строке, в которой это исключение произошло. Для методов указывается, в какой строке кода был вызван этот метод, из какой строки был вызван предыдущий, и так далее вплоть до метода Main. **Откройте трассировку стека** для непрошедшего модульного теста:

- Windows: откройте окно Test Explorer (View>>Test Explorer), щелкните на тесте и прокрутите результаты до Test Detail Summary.
- Mac: перейдите на панель Test Results, раскройте тест, а затем раскройте в его описании раздел Stack Trace.

Трассировка стека начинается примерно так (на Mac будут использоваться полные имена классов вида JimmyLinq.ComicAnalyzer):

```
at Dictionary`2.get_Item(TKey key)
at CalculatePriceRange(Comic comic) in ComicAnalyzer.cs:line 11
at <<c.<GroupComicsByPrice>b__1_1(Comic comic) in ComicAnalyzer.cs:line 22
```

Щелкните (Windows) или сделайте двойной щелчок (Mac) на трассировке стека, чтобы перейти к коду.

Трассировка стека поначалу выглядит немного странно, но когда вы привыкнете, она может предоставить много полезной информации. Теперь вы знаете, что тест не прошел из-за того, что где-то внутри CalculatePriceRange было выдано исключение, связанное с ключами словаря.

Используйте отладчик для сбора информации

Установите **точку прерывания в первой строке** метода CalculatePriceRange: `if (Comic.Prices[comic.Issue] < 100)`

Затем **проведите отладку модульных тестов**: в Windows выберите команду **Test>>Debug All Tests**, на Mac откройте панель Unit Tests (View>>Tests) и щелкните на кнопке Debug All Tests в верхней части. Наведите указатель мыши на comic.Issue — значение равно 2. Минутку! В словаре Comic.Prices **нет элемента** с ключом 2. **Неудивительно, что в программе произошло исключение!**

Теперь становится ясно, **как исправить ошибку**:

- Добавьте в метод CalculatePriceRange второй параметр:

```
private static PriceRange CalculatePriceRange(Comic comic, IReadOnlyDictionary<int, decimal> prices)
```
- Измените первую строку, чтобы в ней использовался новый параметр: `if (prices[comic.Issue] < 100)`
- Измените запрос LINQ: `group comic by CalculatePriceRange(comic, prices) into priceGroup`

Снова запустите тест. На этот раз он проходит успешно!



Passed: 1 Failed: 0

Написание модульного теста для метода GetReviews

В модульном тесте для метода GroupComicsByPrice использовался статический метод MSTest Assert.AreEqual для сравнения ожидаемых значений с фактическими. Метод GetReviews *возвращает последовательность строк*, а не отдельное значение. Теоретически *можно* использовать Assert.AreEqual для сравнения отдельных элементов этой последовательности, как это делалось в двух последних тестовых утверждениях, используя методы LINQ (такие, как First) для получения конкретных элементов... но такое решение получится **ОЧЕНЬ** длинным.

К счастью, в MSTest предусмотрен более эффективный способ сравнения коллекций: **класс CollectionAssert** содержит статические методы для сравнения ожидаемых результатов-коллекций с фактическими. Таким образом, если у вас имеется коллекция с ожидаемыми результатами и коллекция с фактическими результатами, их можно сравнить так:

```
CollectionAssert.AreEqual(expectedResults, actualResults);
```

Если ожидаемый результат не совпадает с фактическим, тест не проходит. **Добавьте следующий тест** в метод ComicAnalyzer.GetReviews:

```
[TestMethod]
public void ComicAnalyzer_Should_Generate_A_List_Of_Reviews()
{
    var testReviews = new[]
    {
        new Review() { Issue = 1, Critic = Critics.MuddyCritic, Score = 14.5},
        new Review() { Issue = 1, Critic = Critics.RottenTornadoes, Score = 59.93},
        new Review() { Issue = 2, Critic = Critics.MuddyCritic, Score = 40.3},
        new Review() { Issue = 2, Critic = Critics.RottenTornadoes, Score = 95.11},
    };

    var expectedResults = new[]
    {
        "MuddyCritic rated #1 'Issue 1' 14.50",
        "RottenTornadoes rated #1 'Issue 1' 59.93",
        "MuddyCritic rated #2 'Issue 2' 40.30",
        "RottenTornadoes rated #2 'Issue 2' 95.11",
    };

    var actualResults = ComicAnalyzer.GetReviews(testComics, testReviews).ToList();
    CollectionAssert.AreEqual(expectedResults, actualResults);
}
```

Снова запустите тесты. На этот раз два модульных теста должны пройти.



Что произойдет, если передать ComicAnalyzer.GetReviews последовательность с дубликатами обзоров? А если передать обзор с отрицательной оценкой?

Написание модульных тестов для обработки граничных случаев и аномальных данных

В реальном мире данные не идеальны. Например, мы нигде точно не определяли, как должны выглядеть данные обзоров. Вы видели оценки от 0 до 100. Вы посчитали, что допустимы только значения из этого диапазона? В реальном мире на сайтах с обзорами это определено так. А если вы получите обзоры с аномальными оценками — например, отрицательными, или очень большими, или нулевыми? Если вы получите более одной оценки от конкретного рецензента для конкретного выпуска? Даже если все эти аномалии *не должны* встречаться, они могут встретиться.

Код должен быть **защищенным от ошибок**; это означает, что он должен успешно обрабатывать проблемы, сбои и особенно некорректные входные данные. Построим модульный тест, который передает GetReviews некорректные данные и убеждается в том, что они не нарушают работоспособность кода:

```
[TestMethod]
public void ComicAnalyzer_Should_Handle_Weird_Review_Scores()
{
    var testReviews = new[]
    {
        new Review() { Issue = 1, Critic = Critics.MuddyCritic, Score = -12.1212},
        new Review() { Issue = 1, Critic = Critics.RottenTornadoes, Score = 391691234.48931},
        new Review() { Issue = 2, Critic = Critics.RottenTornadoes, Score = 0},
        new Review() { Issue = 2, Critic = Critics.MuddyCritic, Score = 40.3},
        new Review() { Issue = 2, Critic = Critics.MuddyCritic, Score = 40.3},
        new Review() { Issue = 2, Critic = Critics.MuddyCritic, Score = 40.3},
        new Review() { Issue = 2, Critic = Critics.MuddyCritic, Score = 40.3},
    };

    var expectedResult = new[]
    {
        "MuddyCritic rated #1 'Issue 1' -12.12",
        "RottenTornadoes rated #1 'Issue 1' 391691234.49",
        "RottenTornadoes rated #2 'Issue 2' 0.00",
        "MuddyCritic rated #2 'Issue 2' 40.30",
        "MuddyCritic rated #2 'Issue 2' 40.30",
        "MuddyCritic rated #2 'Issue 2' 40.30",
        "MuddyCritic rated #2 'Issue 2' 40.30",
    };

    var actualResults = ComicAnalyzer.GetReviews(testComics, testReviews).ToList();
    CollectionAssert.AreEqual(expectedResults, actualResults);
}
```

Справится ли наш код с отрицательными числами? Очень большими числами? Нулями? Такие случаи прекрасно подходят для модульных тестов.

А если одни и те же данные встречаются несколько раз? Вроде бы очевидно, что код должен успешно обработать их, но это не означает, что они действительно будут обработаны.

Метод GetReviews возвращает по-следовательность строк и округляет оценку до двух знаков в дробной части.

Всегда пишите модульные тесты для граничных случаев и аномальных данных — считайте, что это попросту обязательно. Модульное тестирование должно раскинуть свою сеть для выявления ошибок как можно шире, и такие тесты очень эффективно работают в этом отношении.

Очень важно добавить модульные тесты для обработки граничных случаев и аномальных данных. Они помогут выявить проблемы в коде, которые можно было бы упустить в других ситуациях.

В: Почему тесты называются *модульными*?

О: «Модульный тест» — общий термин, используемый во многих языках, не только в C#. Он происходит от представления о том, что код делится на отдельные структурные единицы, называемые модулями. В разных языках используются разные единицы; в C# базовой единицей считается класс. С этой точки зрения термин «модульный тест» выглядит логично: вы пишете тесты для отдельных модулей, или в нашем случае — для отдельных классов.

В: Я создал два проекта в одном решении. Как это работает?

О: Когда вы делаете новый проект C# в Visual Studio, среда создает решение и добавляет в него проект. Все решения, описанные до настоящего момента в книге, состояли из одного проекта — до проекта с модульными тестами. На самом деле решение может содержать много проектов. Мы использовали отдельный проект для того, чтобы отделить модульные тесты от тестируемого кода. Также можно добавить несколько консольных приложений, проектов ASP.NET или WPF — решение может содержать комбинацию разнотипных проектов.

В: Если решение содержит несколько проектов консольных приложений, WPF или ASP.NET, как IDE определяет, какой проект должен запускаться?

О: Взгляните на окно Solution Explorer (или панель Solution в VS для Mac) — одно из имен проектов выделено жирным шрифтом. IDE называет его **стартовым проектом**. Вы можете щелкнуть правой кнопкой мыши на любом проекте в решении и сообщить IDE, что этот проект должен использоваться в качестве стартового. В этом случае при следующем нажатии кнопки Run на панели инструментов будет запущен выбранный вами проект.

В: Еще раз — как работает модификатор доступа `internal`?

О: Когда вы помечаете класс или интерфейс модификатором `internal`, это означает, что к ним можно обращаться только из кода этого проекта. Если модификатор доступа вообще не указан, для класса или интерфейса по умолчанию используется модификатор `internal`. Именно по этой причине вам приходилось следить за тем, чтобы ваши классы имели пометку `public`, в противном случае модульные тесты их не увидят. Также будьте осторожны с модификаторами доступа: хотя при отсутствии модификатора классы и интерфейсы по умолчанию используют `internal`, для компонентов классов — методов, полей и свойств — по умолчанию используется модификатор `private`.

В: Если мои модульные тесты хранятся в отдельном проекте, как они могут обратиться к приватным методам тестируемого кода?

О: Они не могут. Модульные тесты обращаются к части, видимой для остального кода (для ваших классов C# это открытые методы и поля), и используют их, чтобы убедиться в работоспособности модуля. Модульные тесты обычно должны работать по принципу «черного ящика», это означает, что они проверяют только видимые им методы (в отличие от тестов, работающих по принципу «белого ящика», которые «видят» внутреннюю реализацию тестируемого модуля).

В: Все мои тесты должны проходить? А если какие-то тесты не проходят, это критично?

О: Да, все тесты должны проходить. Если какие-то тесты не проходят, это совершенно недопустимо. Взгляните на происходящее так: если какая-то часть вашего кода не работает, это нормально? Конечно нет! Таким образом, если ваш тест не проходит, значит, ошибка присутствует либо в коде, либо в тесте. В любом случае ошибку следует исправить, чтобы тест проходил.



Кажется, написание тестов требует немалой лишней работы. Разве не быстрее просто написать код и обойтись без модульных тестов?

На самом деле при написании модульных тестов ваши проекты идут быстрее.

Серьезно! Тот факт, что для написания *большого* объема кода требуется меньше времени, выглядит противоестественно, но если вы привыкнете писать модульные тесты, ваши проекты пойдут заметно быстрее, потому что ошибки будут обнаруживаться и исправляться на ранней стадии. В первых восьми с половиной главах этой книги вы написали большой объем кода, а это означает, что вам почти наверняка приходилось отслеживать и исправлять ошибки в своем коде. Когда вы исправляли эти ошибки, не приходилось ли вам исправлять другой код в своем проекте? Когда в коде проявляется неожиданная ошибка, вам часто приходится бросать то, чем вы занимаетесь, чтобы найти и исправить ошибку. Подобные переключения — потеря хода мысли, необходимость прерываться и браться за другую задачу — могут серьезно замедлить ход работы. Модульные тесты помогают находить подобные ошибки на ранней стадии, прежде чем им представится возможность прервать вашу работу.

Все еще размышляете над тем, когда следует заниматься написанием модульных тестов? В конце главы мы приводим проект, загружаемый по ссылке; он поможет вам получить ответ на этот вопрос.

Оператор `=>` и создание лямбда-выражений

В самом начале главы один вопрос остался без ответа. Помните загадочную строку, которую мы предложили добавить в класс `Comic`? Приведем ее еще раз:

```
public override string ToString() => $"{Name} (Issue #{Issue})";
```

Вы не раз использовали этот метод `ToString` в главе — значит, он работает. А что бы вы сделали, если бы мы предложили переписать этот метод в том виде, в каком вы записывали методы до этого? Вы бы написали что-нибудь такое:

```
public override string ToString() {
    return $"{Name} (Issue #{Issue})";
}
```

И это было бы правильно. Что же происходит? Что делает оператор `=>`?

Оператор `=>`, который вы использовали в методе `ToString`, называется **лямбда-оператором**. Он используется для определения **лямбда-выражений**, или *анонимных функций*, определяемых в одной команде. Лямбда-выражения выглядят так:

(входные-параметры) => выражение;

Лямбда-выражение состоит из двух частей:

- ★ **Входные-параметры** — список параметров, аналогичных тем, которые используются при объявлении метода. Если параметр только один, круглые скобки можно опустить.
- ★ **Выражение** — произвольное выражение `C#`; это может быть интерполируемая строка, команда с оператором, вызов метода — практически все, что может содержать команда.

Лямбда-выражения на первый взгляд выглядят странно, но на самом деле это всего лишь другой способ использования *уже известных вам выражений C#*, которыми вы пользовались в книге, — по сути, это тот же метод `Comic.ToString`, который работает одинаково независимо от того, используется лямбда-выражение или нет.



Если метод `ToString` работает одинаково независимо от того, используется лямбда-выражение или нет, получается, что мы провели **рефакторинг?**

Да! Лямбда-выражения могут использоваться для рефакторинга многих методов и свойств.

В этой книге вы написали много методов, состоящих всего из одной команды. Большинство из них можно подвергнуть рефакторингу и использовать вместо исходной реализации лямбда-выражения. Во многих случаях это упростит чтение и понимание кода. Лямбда-выражения расширяют вашу свободу выбора — вы можете сами решить, когда их использование улучшает ваш код.



Тест-драйв лямбда-выражений

Опробуем на практике лямбда-выражения — новый способ записи методов, включая те, которые возвращают значения или получают параметры.

- 1 **Создайте новое консольное приложение.** Добавьте класс Program с методом Main:

```
class Program
{
    static Random random = new Random();
    static double GetRandomDouble(int max)
    {
        return max * random.NextDouble();
    }
    static void PrintValue(double d)
    {
        Console.WriteLine($"The value is {d:0.0000}");
    }
    static void Main(string[] args)
    {
        var value = Program.GetRandomDouble(100);
        Program.PrintValue(value);
    }
}
```



Выполните его несколько раз. Каждый раз будет выводиться новое случайное число — например, The value is 37.8709

- 2 **Проведите рефакторинг методов GetRandomDouble и PrintValue с использованием оператора =>:**

```
static double GetRandomDouble(int max) => max * random.NextDouble();
static void PrintValue(double d) => Console.WriteLine($"The value is {d:0.0000}");
```

Снова запустите программу — она должна вывести новое случайное число, как и в предыдущем варианте.

Прежде чем продолжать рефакторинг, **наведите указатель мыши на поле random** и ознакомьтесь с содержимым окна IntelliSense:

```
static Random random = new Random();
```

(field) static Random Program.random

Это не настоящий рефакторинг, потому что мы изменили поведение кода, а не только его структуру.

Каждое из этих лямбда-выражений получает параметр — как и метод, который они заменяют.

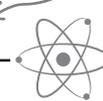
- 3 **Измените поле random для использования лямбда-выражения:**

```
static Random random => new Random();
```

Программа все еще работает так же. Снова наведите указатель мыши на поле random:

```
static Random random => new Random();
```

Random Program.random {get;}



МОЗГОВОЙ ШТУРМ

Как вы думаете, какая из версий кода проще читается?

Один момент — random уже не является полем. При преобразовании в лямбда-выражении поле превратилось в свойство! Дело в том, что **лямбда-выражения всегда работают как методы**. Пока значение random оставалось полем, его экземпляр создавался при выполнении конструктора класса. Когда вы заменяете = на => и преобразуете реализацию в лямбда-выражение, random становится методом, а это означает, что *новый экземпляр Random создается при каждом обращении к свойству*.

Рефакторинг клоунов с использованием лямбда-выражений

Сделайте это!

В главе 7 мы создали интерфейс IClown с двумя компонентами:

```
interface IClown
{
    string FunnyThingIHave { get; }
    void Honk();
}
```

Интерфейс IClown из главы 7 содержит два компонента: одно свойство и один метод.

Подсказки для IDE: реализация интерфейсов

Когда класс реализует интерфейс, команда меню Quick Actions «Implement Interface» приказывает IDE добавить все недостающие компоненты интерфейса.

Класс был изменен для использования этого интерфейса:

```
class TallGuy {
    public string Name;
    public int Height;

    public void TalkAboutYourself() {
        Console.WriteLine($"My name is {Name} and I'm {Height} inches tall.");
    }
}
```

Продедаем то же самое снова — но на этот раз с лямбда-выражениями. **Создайте проект нового консольного приложения**, добавьте интерфейс IClown и TallGuy. Измените TallGuy для реализации IClown:

```
class TallGuy : IClown {
```

Теперь откройте меню Quick Actions и выберите команду «**Implement interface**». IDE заполняет все компоненты интерфейса, чтобы они выдавали исключение NotImplementedException, как это делалось при использовании Generate Method.

```
    public string FunnyThingIHave => throw new NotImplementedException();
    public void Honk()
    {
        throw new NotImplementedException();
    }
```

Когда вы приказали IDE реализовать интерфейс IClown, среда воспользовалась оператором => для создания лямбда-выражения, реализующего свойство.

Проведем рефакторинг методов, чтобы они делали то же самое, но с использованием лямбда-выражений:

```
    public string FunnyThingIHave => "big red shoes";
    public void Honk() => Console.WriteLine("Honk honk!");
```

Среда IDE создала тело метода при добавлении компонентов интерфейса, но вы можете заменить его лямбда-выражением.

Добавим метод Main, использованный в главе 7:

```
TallGuy tallGuy = new TallGuy() { Height = 76, Name = "Jimmy" };
tallGuy.TalkAboutYourself();
Console.WriteLine($"The tall guy has {tallGuy.FunnyThingIHave}");
tallGuy.Honk();
```

Запустите приложение. Класс TallGuy работает так же, как в главе 7, но после рефакторинга методов с лямбда-выражениями код стал более компактным.

Мы считаем, что новый улучшенный класс TallGuy проще читается. А вы как думаете?

У таинственного метода ToString в начале главы также использовалось лямбда-выражение в качестве тела.





Ниже приведен класс NectarCollector из проекта системы управления ульем (глава 6) и класс ScaryScary (глава 7). Ваша задача — **провести рефакторинг некоторых компонентов класса с использованием лямбда-оператора (=>)**. Запишите полученные методы.

```
class NectarCollector : Bee
{
    public const float NECTAR_COLLECTED_PER_SHIFT = 33.25f;
    public override float CostPerShift { get { return 1.95f; } }
    public NectarCollector() : base("Nectar Collector") { }

    protected override void DoJob()
    {
        HoneyVault.CollectNectar(NECTAR_COLLECTED_PER_SHIFT);
    }
}
```

Проведите рефакторинг свойства CostPerShift в форме лямбда-выражения:

.....

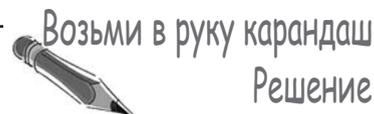
```
class ScaryScary : FunnyFunny, IScaryClown {
    private int scaryThingCount;
    public ScaryScary(string funnyThing, int scaryThingCount) : base(funnyThing)
    {
        this.scaryThingCount = scaryThingCount;
    }
    public string ScaryThingIHave { get { return $"{scaryThingCount} spiders"; } }
    public void ScareLittleChildren()
    {
        Console.WriteLine($"Boo! Gotcha! Look at my {ScaryThingIHave}");
    }
}
```

Проведите рефакторинг свойства ScaryThingProperty в форме лямбда-выражения:

.....

Проведите рефакторинг метода ScareLittleChildren в форме лямбда-выражения:

.....



Ниже приведен класс NectarCollector из проекта системы управления ульем (глава 6) и класс ScaryScary (глава 7). Ваша задача — **провести рефакторинг некоторых компонентов класса с использованием лямбда-оператора (=>)**. Запишите полученные методы.

Проведите рефакторинг свойства CostPerShift в форме лямбда-выражения:

```
public float CostPerShift { get => 1.95f; }
```

Проведите рефакторинг свойства ScaryThingProperty в форме лямбда-выражения:

```
public string ScaryThingIHave { get => $"{scaryThingCount} spiders"; }
```

Проведите рефакторинг метода ScareLittleChildren в форме лямбда-выражения:

```
public void ScareLittleChildren() => Console.WriteLine($"Boo! Gotcha! Look at my {ScaryThingIHave}");
```

Часто задаваемые вопросы

В: Вернитесь к тест-драйву, где мы изменяли поле `random`. Вы сказали, что это «не настоящий рефакторинг», — можно пояснить, что вы имели в виду?

О: Конечно. При рефакторинге кода вы изменяете его *структуру* без изменения *поведения*. Когда мы преобразовали методы `PrintValue` и `GetRandomDouble` в лямбда-выражения, они работали абсолютно так же — изменилась структура, но изменение никак не повлияло на поведение.

Но когда в объявлении поля `random` знак равенства (=) был преобразован в лямбда-оператор (=>), изменилось поведение. Поле напоминает переменную — вы один раз объявляете его, а потом используете повторно. Таким образом, когда `random` было полем, новый экземпляр `Random` создавался при запуске программы и ссылка на этот экземпляр сохранялась в поле `random`.

Но при использовании лямбда-выражения всегда создается метод. Таким образом, когда вы заменили поле `random` следующим объявлением:

```
static Random random => new Random();
```

компилятор C# уже не видит поле. Вместо него он видит свойство. И это логично — в главе 5 вы узнали, что к свойствам вы обращаетесь как к полям, но в действительности они являются методами.

А чтобы убедиться в этом, достаточно навести указатель мыши на поле:

```
static Random random => new Random();
```

IDE сообщает, что `random` теперь является свойством, показывая, что у него теперь есть `get`-метод `{ get; }`.

Оператор => может использоваться для создания свойства с get-методом, выполняющим лямбда-выражение.

Использование оператора ?: для принятия решений в лямбда-выражениях

А если вы хотите, чтобы лямбда-выражение делало... нечто большее? Было бы здорово, если бы они могли принимать решения... и в этом вам поможет **условный оператор** (который иногда называется **тернарным оператором**). Он работает примерно так:

условие ? следствие : альтернатива;

Выражение выглядит немного странно, поэтому рассмотрим пример. Прежде всего оператор ?: работает не только с лямбда-выражениями — они могут использоваться где угодно. Возьмем команду `if` из класса `AbilityScoreCalculator` в главе 4:

```
if (added < Minimum)
    Score = Minimum;
else
    Score = added;
```

Если условие истинно, выражение возвращает значение `Minimum`.
 В противном случае возвращается `added`.
 Выражение ?: проверяет условие (`added < Minimum`).

Команду можно переработать с использованием оператора ?:, результат выглядит примерно так:

`Score = (added < Minimum) ? Minimum : added;`

Обратите внимание на то, что `Score` присваивается результат выражения ?: . Выражение ?: **возвращает значение**: оно проверяет условие (`added < Minimum`), а затем возвращает либо *следствие* (`Minimum`), либо *альтернативу* (`added`).

Если у вас имеется метод, который работает по логике `if/else`, вы можете воспользоваться оператором ?: для рефакторинга его в лямбда-выражение. Для примера возьмем метод из класса `PaintballGun` из главы 5:

```
public void Reload()
{
    if (balls > MAGAZINE_SIZE)
        BallsLoaded = MAGAZINE_SIZE;
    else
        BallsLoaded = balls;
}
```

Эта условная команда «`if`» в случае истинности условия (`balls > MAGAZINE_SIZE`) выполняет основную ветвь (`BallsLoaded = MAGAZINE_SIZE`), а в случае ложности выполняет ветвь `else` (`BallsLoaded = balls`).

Мы преобразовали «`if`» в компонент, тело которого определяется выражением (в операторе ?: как следствие, так и альтернатива возвращают значение), и использовали его для задания свойства `BallsLoaded`.

Перепишем этот код в виде более лаконичного лямбда-выражения:

```
public void Reload() => BallsLoaded = balls > MAGAZINE_SIZE ? MAGAZINE_SIZE : balls;
```

Обратите внимание на небольшое изменение — в версии `if/else` свойство `BallsLoaded` задавалось в обеих ветвях. Мы воспользовались условным оператором, который сравнивает `MAGAZINE_SIZE`, возвращает правильное значение и использует его для задания свойства `BallsLoaded`.



Простое правило поможет запомнить, как работает условный оператор.

Многим людям трудно запомнить порядок ? и : в тернарном операторе ?: . К счастью, простое мнемоническое правило поможет запомнить структуру оператора.

Оператор словно задает вопрос, а вопрос всегда задается до получения ответа. Спросите себя:

условие истинно ? да : нет

И тогда становится очевидно, что ? предшествует : в выражении.

Забавный факт: мы узнали об этом из превосходной документации Microsoft с описанием оператора ?: : <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/conditional-operator>

Лямбда-выражения и LINQ

Добавьте этот небольшой запрос LINQ в любое приложение C#, затем наведите указатель мыши на ключевое слово `select` в коде:

```
var array = new[] { 1, 2, 3, 4 };
var result = from i in array select i * 2;
```

Выглядит как объявление метода — такое же, какое отображается при наведении указателя мыши на любой другой метод.

 (extension) `IEnumerable<int> IEnumerable<int>.Select<int, int>(Func<int, int> selector)`
 Projects each element of a sequence into a new form.
 Returns:
 An `IEnumerable<out T>` whose elements are the result of invoking the transform function on each element of source.
 Exceptions:
`ArgumentNullException`

При наведении указателя мыши на метод IDE открывает окно с подсказкой. Присмотритесь к первой строке, в которой выводится объявление метода:

```
IEnumerable<int> IEnumerable<int>.Select<int, int>(Func<int, int> selector)
```

Из объявления метода можно узнать кое-что полезное:

- ★ Метод `IEnumerable<int> Select` возвращает `IEnumerable<int>`.
- ★ Он получает один параметр типа `Func<int, int>`.

Метод `IEnumerable<int> Select` получает один параметр типа `Func<int, int>`; это означает, что вы можете использовать лямбда-выражение, которое получает параметр `int` и возвращает `int`.

Использование лямбда-выражений с методами, получающими параметр Func

Если метод получает параметр `Func<int, int>`, его можно **вызвать с лямбда-выражением**, которое получает параметр `int` и возвращает значение `int`. Таким образом, запрос `select` можно переписать в следующем виде:

```
var array = new[] { 1, 2, 3, 4 };
var result = array.Select(i => i * 2);
```

Убедитесь в этом сами в консольном приложении. Добавьте команду `foreach` для вывода результата:

```
foreach (var i in result) Console.WriteLine(i);
```

При выводе результатов переработанного запроса вы получите последовательность `{2, 4, 6, 8}` — точно такой же результат, как с синтаксисом запроса LINQ до его рефакторинга.



Каждый раз, когда вы видите **Func** в методе LINQ, это означает, что здесь может использоваться лямбда-выражение.

О `Func` намного подробнее рассказано в главе, посвященной событиям и делегатам, доступной в виде PDF-файла на странице GitHub. А пока скажем, что при использовании параметра метода LINQ с типом `Func<TSource, TResult>` метод можно вызвать с передачей ему лямбда-выражения, которое получает параметр типа `TSource` и возвращает тип `TResult`.

Запросы LINQ могут записываться в виде сцепленных вызовов методов LINQ

Возьмите запрос LINQ, который был приведен ранее, и **добавьте его в приложение**, чтобы исследовать другие методы LINQ:

```
int[] values = new int[] { 0, 12, 44, 36, 92, 54, 13, 8 };
IEnumerable<int> result =
    from v in values
    where v < 37
    orderby -v
    select v;
```

Последовательность `int` содержит метод `OrderBy` с лямбда-выражением, которое получает `int` и возвращает `int`. Он работает так же, как и методы сравнения, описанные в главе 8.

Метод LINQ `OrderBy` сортирует последовательность

Наведите указатель мыши на ключевое слово **orderby** и обратите внимание на его параметр:

 (extension) `IOrderedEnumerable<int>` `IEnumerable<int>.OrderBy<int, int>(Func<int, int> keySelector)`
Sorts the elements of a sequence in ascending order according to a key.

Returns:

An `IOrderedEnumerable<out TElement>` whose elements are sorted according to a key.

Когда вы используете секцию `orderby` в запросе LINQ, она вызывает метод LINQ `OrderBy`, который сортирует последовательность. В данном случае ему можно передать лямбда-выражение с параметром `int`, которое **возвращает ключ сортировки** или любое значение (которое должно реализовать `IComparer`), которое может использоваться для сортировки результатов.

Метод LINQ `Where` использует лямбда-выражение, которое получает элемент последовательности и возвращает `true`, если элемент включается в результат, или `false`, если его следует пропустить.

Метод LINQ `Where` выделяет подмножество последовательности

Наведите указатель мыши на ключевое слово **where** в запросе LINQ:

 (extension) `IEnumerable<int>` `IEnumerable<int>.Where<int>(Func<int, bool> predicate)`
Filters a sequence of values based on a predicate.

Returns:

An `IEnumerable<out T>` that contains elements from the input sequence that satisfy the condition.

Секция `where` в запросе LINQ вызывает метод LINQ `Where`, который использует лямбда-выражение и возвращает `bool`. **Метод `Where` вызывает лямбда-выражение для каждого элемента в последовательности.** Если лямбда-выражение возвращает `true`, то элемент включается в результат. Если возвращается `false`, то элемент исключается.



Упражнение по работе с лямбда-выражениями! А вы сможете провести рефакторинг этого запроса LINQ и преобразовать его в запрос сцепленных вызовов методов LINQ? Начните с `result`, присоедините методы `Where` и `OrderBy` для получения той же последовательности.

```
IEnumerable<int> result =
    from v in values
    where v < 37
    orderby -v
    select v;
```



Код под увеличительным стеклом

Запросы LINQ можно переписать в виде серии сцепленных вызовов методов LINQ, и многие из этих методов могут использовать лямбда-выражения для определения генерируемых последовательностей.

Решение для мини-упражнения — запрос LINQ преобразуется к следующему виду:

```
var result = values.Where(v => v < 37).OrderBy(v => -v);
```

Разберемся, как же запрос LINQ преобразуется в сцепленные вызовы методов:



IEnumerable<int> result =	— Используем var для объявления переменной —>	var result =
from v in values	Начинаем с последовательности values	values
where v < 37	— Вызываем Where с лямбда-выражением, включающим значения < 37 —>	.Where(v => v < 37)
orderby -v	— Вызываем OrderBy с лямбда-выражением, меняющим знак значения —>	.OrderBy(v => -v);
select v;	Метод .Select не нужен, потому что секция select не изменяет значения	

Используйте метод OrderByDescending в тех ситуациях, когда бы вы использовали ключевое слово descending в запросе LINQ

Помните, как вы использовали ключевое слово descending для изменения секции orderby в запросе? Существует эквивалентный метод LINQ OrderByDescending, который делает то же самое:

```
var result = values.Where(v => v < 37).OrderByDescending(v => v);
```

Обратите внимание на использование лямбда-выражения `v => v` — это лямбда-выражение всегда возвращает переданное ему значение (оно иногда называется тождественным отображением). Таким образом, `OrderByDescending(v => v)` меняет порядок элементов на противоположный.

Используйте метод GroupBy для создания запросов с группировкой из сцепленных методов

Следующий запрос с группировкой был приведен ранее в этой главе:

```
var grouped =
    from card in deck
    group card by card.Suit into suitGroup
    orderby suitGroup.Key descending
    select suitGroup;
```

Наведите указатель мыши на ключевое слово `group`. Вы увидите, что оно вызывает метод LINQ `GroupBy`, возвращающий тот же тип, который был приведен ранее в этой главе. Вы можете воспользоваться лямбда-выражением для группировки результата по масти карты: `card => card.Suit`

Затем другое лямбда-выражение используется для упорядочения групп по ключу: `group => group.Key`.

Запрос LINQ, преобразованный посредством рефакторинга в сцепленные вызовы методов `GroupBy` и `OrderByDescending`:

```
var grouped =
    deck.GroupBy(card => card.Suit)
    .OrderByDescending(group => group.Key);
```

Попробуйте вернуться к приложению, в котором вы использовали этот запрос, и заменить его сцепленными вызовами методов. Вы получите точно такой же результат. Сами решите, какая версия кода более понятна и проще читается.

Какой вариант использовать: синтаксис декларативных запросов LINQ или сцепленные вызовы? Оба варианта приводят к одному результату. Иногда первый вариант упрощает чтение кода, иногда — второй; вы должны уметь пользоваться обеими формами записи.

(extension) IEnumerable<IGrouping<Suits, Card>> IEnumerable<Card>.GroupBy<Card, Suits>(Func<Card, Suits> keySelector)
Groups the elements of a sequence according to a specified key selector function.

Использование оператора `=>` для создания выражений `switch`

Команды `switch` использовались, начиная с главы 6, для проверки значения переменной по нескольким возможным вариантам. Это полезный инструмент... но вы заметили, что его возможности ограничены? Например, попробуйте добавить альтернативу `case` для сравнения с переменной:

```
case myVariable:
```

Компилятор C# выдает ошибку: «*Ожидается константа*». Дело в том, что в командах `switch`, которыми вы пользовались, могут использоваться только значения-константы — например, литералы и переменные, определенные с ключевым словом `const`.

Но все меняется с оператором `=>`, который позволяет создавать **выражения `switch`**. Они имеют много общего с командами `switch`, которыми вы пользовались, но при этом являются выражениями, которые возвращают значение. Выражение `switch` начинается с проверяемого значения и ключевого слова `switch`, за которым следует серия *ветвей `switch`* в фигурных скобках, разделенных запятыми. Каждая ветвь использует оператор `=>` для сравнения проверки значения по выражению. Если первая ветвь не подходит, происходит переход к следующей; в итоге возвращается значение для подходящей ветви.

```
var returnValue = valueToCheck switch
{
    pattern1 => returnValue1,
    pattern2 => returnValue2,
    ...
    _ => defaultReturnValue,
}
```

← Выражение `switch` начинается с проверяемого значения, за которым следует ключевое слово `switch`.

← Тело выражения `switch` представляет собой серию ветвей, которые используют оператор `=>` для проверки `valueToCheck` и возвращают значение при совпадении.

Допустим, вы работаете над карточной игрой, которая должна присваивать определенное количество очков в зависимости от масти; за пики начисляется 6 очков, за червы — 4 очка, а все остальные карты приносят 2 очка. Можно написать команду `switch` следующего вида:

```
var score = 0;
switch (card.Suit)
{
    case Suits.Spades:
        score = 6;
        break;
    case Suits.Hearts:
        score = 4;
        break;
    default:
        score = 2;
        break;
}
```

← В каждой секции `case` этой команды `switch` присваивается значение переменной `score`. Такая команда становится отличным кандидатом для преобразования в выражение `switch`.

Выражения `switch` должны быть исчерпывающими, т. е. их условия должны покрывать все возможные значения. Шаблон `_` подходит для любых значений, которые не совпали с остальными ветвями.

Единственная цель этой команды `switch` — присваивание значения переменной `score`. Очень многие команды `switch` работают по этой схеме. Воспользуемся оператором `=>` для создания выражения `switch`, который делает то же самое:

```
var score = card.Suit switch
{
    Suits.Spades => 6,
    Suits.Hearts => 4,
    _ => 2,
};
```

← Это выражение `switch` проверяет значение `card.Suit` — если оно равно `Suits.Spades`, то выражение возвращает 6; если оно равно `Suits.Hearts`, то возвращается 4; во всех остальных случаях возвращается 2.

Возьми в руку карандаш



В этом консольном приложении используются классы `Suit`, `Value` и `Deck`, которые встречались вам в этой главе. Приложение выводит шесть строк на консоль. Ваша задача — **записать результат выполнения программы**. Когда это будет сделано, добавьте программу в консольное приложение и проверьте свой ответ.

```
class Program
{
    static string Output(Suits suit, int number) =>
        $"Suit is {suit} and number is {number}";

    static void Main(string[] args)
    {
        var deck = new Deck();
        var processedCards = deck
            .Take(3)
            .Concat(deck.TakeLast(3))
            .OrderByDescending(card => card)
            .Select(card => card.Value switch
            {
                Values.King => Output(card.Suit, 7),
                Values.Ace => $"It's an ace! {card.Suit}",
                Values.Jack => Output((Suits)card.Suit - 1, 9),
                Values.Two => Output(card.Suit, 18),
                _ => card.ToString(),
            }) ;

        foreach(var output in processedCards)
        {
            Console.WriteLine(output);
        }
    }
}
```

← Лямбда-выражение получает два параметра, `Suit` и `int`, и возвращает интерполированную строку.

← Эти методы LINQ ничем не отличаются от тех, которые были представлены в начале главы.

← Мы можем использовать `OrderByDescending`, потому что класс `Card` реализует `IComparable<Card>`.

← Метод `Select` использует команду `switch` для проверки номинала карты и генерирует соответствующую строку.

Запишите результат выполнения программы. **Мы не приводим решение** — чтобы проверить свой ответ, включите код в консольное приложение и выполните его.

.....

.....

.....

.....

.....

.....

Это **серьезное** испытание! Здесь происходит очень много всего — лямбда-выражения, выражение `switch`, методы LINQ, приведение типа перечислений, сцепленные вызовы методов и т. д. Не жалейте времени и разберитесь в том, как работает код, потом запишите свой ответ и только **после этого** запустите программу. Если ваш ответ не совпал с результатом выполнения программы, у вас появляется отличная возможность разобраться, почему программа работает не так, как вы ожидали.



Упражнение

Используйте все, что вы узнали о лямбда-выражениях, командах `switch` и методах LINQ для рефакторинга класса `ComicAnalyzer` и метода `Main`. При помощи модульных тестов убедитесь в том, что ваш код все еще работает так, как вы ожидали.

Замените запросы LINQ в `ComicAnalyzer`

`ComicAnalyzer` содержит два запроса LINQ:

- Метод `GroupComicsByPrice` содержит запрос LINQ, использующий ключевое слово `group` для группировки комиксов по цене.
- Метод `GetReviews` содержит запрос LINQ, использующий ключевое слово `join` для объединения последовательности объектов `Comic` со словарем цен на выпуски.

Измените запросы LINQ в этих методах, чтобы в них использовались методы LINQ `OrderBy`, `GroupBy`, `Select` и `Join`. Небольшая загвоздка: мы еще не рассказали вам о методе `Join`! Зато мы привели примеры, показывающие, как использовать IDE для исследования методов LINQ. Метод `Join` чуть сложнее других — но мы поможем вам разбить его на части. Он получает четыре параметра:

`sequence.Join(последовательность для объединения, лямбда-выражение для части 'on' объединения, лямбда-выражение для части 'equals' объединения, лямбда-выражение, которое получает два параметра и возвращает результат 'select');`

Этому лямбда-выражению передается каждая пара элементов из двух объединяемых последовательностей.

Внимательно присмотритесь к частям «`on`» и «`equals`» запроса LINQ, соответствующим первым двум лямбда-выражениям. Метод `Join` будет последним в цепочке. Подсказка: лямбда-выражение последнего параметра начинается так: (`comic, review`) =>

Когда оба модульных теста будут проходить успешно, рефакторинг класса `ComicAnalyzer` можно считать завершенным.

Замените команду `switch` в методе `Main` выражением `switch`

Метод `Main` содержит команду `switch`, которая вызывает приватные методы и присваивает их возвращаемые значения переменной `done`. Замените ее выражением `switch` с тремя ветвями. Чтобы протестировать выражение, запустите приложение — если при нажатии правильной клавиши вы будете получать правильный результат, значит, все готово.

Это упражнение научит вас использовать модульные тесты для безопасного рефакторинга вашего кода



За сценой

Рефакторинг может быть хлопотным, даже раздражающим занятием. Вы берете работоспособный код и вносите в него изменения, чтобы улучшить его структуру, удобочитаемость и возможности повторного использования. Когда вы вносите изменения в свой код, очень легко допустить ошибку, из-за которой он работать перестанет, причем иногда внесенные ошибки оказываются на редкость коварными и трудноловимыми. Здесь на помощь приходят модульные тесты. Одна из самых важных областей для применения модульных тестов разработчиками — повышение безопасности рефакторинга. Вот как это работает:

- Прежде чем браться за рефакторинг, напишите тесты, которые подтверждают, что ваш код работает, — как тесты, добавленные ранее в этой главе для проверки класса `ComicAnalyzer`.
- При рефактинге класса просто запустите тесты для класса после внесения изменений. Это существенно сокращает цикл обратной связи для разработки — можно выполнить нормальную отладку, но этот способ работает намного быстрее, потому что код выполняется непосредственно в классе (вместо запуска программы и использования ее интерфейса для выполнения кода, использующего класс).
- При рефактинге метода можно начать даже с запуска конкретного теста или тестов, выполняющих этот метод. Если этот тест работает, можно перейти к полному набору тестов, который проверяет, что больше ничего не было сломано.
- Если тест не проходит, не впадайте в уныние: на самом деле это хорошая новость! Вы узнали, что что-то сломалось, и теперь можете исправить ошибку.



Упражнение
Решение

Используйте все, что вы узнали о лямбда-выражениях, командах switch и методах LINQ, для рефакторинга класса ComicAnalyzer и метода Main. При помощи модульных тестов убедитесь в том, что ваш код все еще работает так, как вы ожидали.

Ниже приведены методы GroupComicsByPrice и GetReviews из класса ComicAnalyzer после рефакторинга:

```
public static IEnumerable<IGrouping<PriceRange, Comic>> GroupComicsByPrice(
    IEnumerable<Comic> comics, IReadOnlyDictionary<int, decimal> prices)
{
    var grouped =
        comics
        .OrderBy(comic => prices[comic.Issue])
        .GroupBy(comic => CalculatePriceRange(comic, prices));

    return grouped;
}
```

Сравните лямбда-выражения OrderBy и GroupBy с секциями orderby и group...by в запросе LINQ. Они почти идентичны.

```
public static IEnumerable<string> GetReviews(
    IEnumerable<Comic> comics, IEnumerable<Review> reviews)
{
    var joined =
        comics
        .OrderBy(comic => comic.Issue)
        .Join(
            reviews,
            comic => comic.Issue,
            review => review.Issue,
            (comic, review) =>
                $"{review.Critic} rated #{comic.Issue} '{comic.Name}' {review.
                Score:0.00}");

    return joined;
}
```

Запрос join начинается с «join reviews», так что первым аргументом, передаваемым методу Join, станет reviews.

Сравните два средних аргумента, передаваемых методу Join, с частями «on» и «equals» запроса join: on comic.Issue equals review.Issue.

Последнее лямбда-выражение вызывается для каждой пары «комикс/обзор» с совпадением из двух объединенных последовательностей и возвращает строку, которая включается в результат.

Ниже приведен метод Main с выражением switch, заменившим команду switch, после рефакторинга:

```
static void Main(string[] args)
{
    var done = false;
    while (!done)
    {
        Console.WriteLine(
            "\nPress G to group comics by price, R to get reviews, any other key to quit\n");
        done = Console.ReadKey(true).KeyChar.ToString().ToUpper() switch
        {
            "G" => GroupComicsByPrice(),
            "R" => GetReviews(),
            _ => true,
        };
    }
}
```

Выражение switch намного компактнее эквивалентной команды switch. Не все команды switch можно преобразовать в выражения switch — в данном случае это возможно, потому что во всех вариантах case присваивается значение одной и той же переменной (done).

Исследование класса Enumerable

Мы уже знаем, что последовательности работают с циклами `foreach` и LINQ. Но какие механизмы обеспечивают работу последовательностей? Начнем с класса **Enumerable**, а конкретно с его трех статических методов: `Range`, `Empty` и `Repeat`. Метод `Enumerable.Range` уже встречался вам ранее в этой главе. IDE поможет нам понять, как работают два других метода. Введите `Enumerable.` и наведите указатель мыши на `Range`, `Empty` и `Repeat` в окне IntelliSense для просмотра объявлений и комментариев.

<pre>Enumerable.</pre> <ul style="list-style-type: none"> ★ Range ★ Empty ★ Repeat 	<pre>IEnumerable<TResult> Enumerable.Empty<TResult>() Returns an empty IEnumerable<out T> that has the specified type argument. ★ IntelliCode suggestion based on this context</pre>
	<pre>IEnumerable<TResult> Enumerable.Repeat<TResult>(TResult element, int count) Generates a sequence that contains one repeated value. ★ IntelliCode suggestion based on this context</pre>

Enumerable.Empty создает пустую последовательность любого типа

Иногда требуется передать пустую последовательность методу, получающему `IEnumerable<T>` (например, в модульном тесте). В таких случаях удобен метод **Enumerable.Empty**:

```
var emptyInts = Enumerable.Empty<int>(); // пустая последовательность int
var emptyComics = Enumerable.Empty<Comic>(); // пустая последовательность ссылок на Comic
```

Enumerable.Repeat возвращает значение, повторенное заданное количество раз

Предположим, вам нужна последовательность из 100 значений цифры «3», или 12 строк «yes», или 83 идентичных анонимных объектов. Вы не поверите, насколько часто такое происходит! Для этого можно воспользоваться методом **Enumerable.Repeat** — он возвращает последовательность повторяющихся значений:

```
var oneHundredThrees = Enumerable.Repeat(3, 100);
var twelveYesStrings = Enumerable.Repeat("yes", 12);
var eightyThreeObjects = Enumerable.Repeat(
    new { cost = 12.94M, sign = "ONE WAY", isTall = false }, 83);
```

Как же работает IEnumerable<T>?

Мы уже довольно давно пользуемся `IEnumerable<T>`, но так и не ответили на вопрос, что же *представляет собой* последовательность с поддержкой перебора. Самый эффективный способ понять что-то — попытаться сделать это своими руками, поэтому в завершение этой главы мы построим несколько последовательностей с нуля.



Если бы вам пришлось проектировать интерфейс `IEnumerable<T>` самостоятельно, какие компоненты вы бы включили в него?

Ручное создание последовательности с поддержкой перебора

Допустим, у вас есть перечисление с видами спорта:

```
enum Sport { Football, Baseball, Basketball, Hockey, Boxing, Rugby, Fencing }
```

Очевидно, можно создать новый список `List<Sport>` и воспользоваться инициализатором коллекции для его заполнения. Но чтобы разобраться в том, как работают последовательности, мы сделаем это вручную. Создайте новый класс с именем `ManualSportSequence`, реализующий интерфейс `IEnumerable<Sport>`. Он содержит два компонента, возвращающие `IEnumerator`:

```
class ManualSportSequence : IEnumerable<Sport> {  
    public IEnumerator<Sport> GetEnumerator() {  
        return new ManualSportEnumerator();  
    }  
  
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() {  
        return GetEnumerator();  
    }  
}
```

Когда мы выбрали команду меню Quick Actions «Implement interface», IDE использовала полные имена классов для `IEnumerator` и `IEnumerable`.

Хорошо, так что же собой представляет `IEnumerator`? Это интерфейс, который позволяет перебрать содержимое последовательности, перемещаясь от одного элемента к другому. Свойство `Current` возвращает текущий элемент в процессе перебора. Метод `MoveNext` перемещается к следующему элементу последовательности, возвращая `false`, если других элементов не осталось. После вызова `MoveNext` свойство `Current` возвращает следующий элемент. Наконец, метод `Reset` возвращается к началу последовательности. С этими методами можно построить последовательность с поддержкой перебора.

IEnumerator<T>
Current
MoveNext Reset Dispose

Реализуем `IEnumerator<Sport>`:

```
using System.Collections.Generic;  
  
class ManualSportEnumerator : IEnumerator<Sport> {  
    int current = -1;  
    public Sport Current { get { return (Sport)current; } }  
    public void Dispose() { return; } // О методе Dispose будет рассказано в главе 10.  
    object System.Collections.IEnumerator.Current { get { return Current; } }  
    public bool MoveNext() {  
        var maxEnumValue = Enum.GetValues(typeof(Sport)).Length;  
        if ((int)current >= maxEnumValue - 1)  
            return false;  
        current++;  
        return true;  
    }  
    public void Reset() { current = 0; }  
}
```

Также необходимо реализовать интерфейс `IDisposable`, о котором вы узнаете в следующей главе. Он содержит всего один метод `Dispose`.

Наша ручная реализация `IEnumerator` преобразует тип `int` в перечисление. Статический метод `Enum.GetValues` используется для получения общего количества элементов в перечислении, а `int` — для хранения индекса текущего значения.

И это все, что необходимо для создания собственной реализации `IEnumerable`. Попробуйте ее на практике. **Создайте новое консольное приложение**, добавьте `ManualSportSequence` и `ManualSportEnumerator`, а затем переберите содержимое последовательности в цикле `foreach`:

```
var sports = new ManualSportSequence();  
foreach (var sport in sports)  
    Console.WriteLine(sport);
```

Использование `yield return` для создания собственной последовательности

C# предоставляет намного более удобный способ создания последовательностей с поддержкой перебора: команду `yield return`. Команда `yield return` — своего рода универсальный автоматический создатель последовательностей. Лучше всего понять ее на примере. Воспользуемся **многопроектным решением** (а заодно потренируемся в работе с ним).

Добавьте в решение новый проект консольного приложения — подобно тому, как вы добавляли проект MSTest ранее в этой главе, но на этот раз вместо типа проекта MSTest выберите тот же тип проекта (Console App), который использовался для большинства проектов в книге. Затем щелкните правой кнопкой мыши на проекте под решением и **выберите команду «Set as startup project»**. Теперь при запуске отладчика в IDE будет запускаться новый проект. Также вы можете щелкнуть правой кнопкой мыши на любом проекте в решении и выбрать команду его запуска или отладки.

Код нового консольного приложения:

```
static IEnumerable<string> SimpleEnumerable() {
    yield return "apples";
    yield return "oranges";
    yield return "bananas";
    yield return "unicorns";
}

static void Main(string[] args) {
    foreach (var s in SimpleEnumerable()) Console.WriteLine(s);
}
```

Этот метод возвращает `IEnumerable<string>`, поэтому каждая команда `yield return` возвращает строковое значение.

Запустите приложение. Оно выводит четыре строки: `apples`, `oranges`, `bananas` и `unicorns`. Как это работает?

Исследование `yield return` в отладчике

Установите точку прерывания в первой строке метода `Main` и запустите отладчик. Затем используйте команду **Step Into** (F11 / ⇧ ⌘ I), чтобы отладить код строку за строкой:

- ★ Выполняйте код в пошаговом режиме, пока не достигнете первой строки метода `SimpleEnumerable`.
- ★ Сделайте следующий шаг. Происходит то же, что при выполнении команды `return`, — управление возвращается команде, из которой был вызван метод: в данном случае команде `foreach`, которая вызывает `Console.WriteLine` для вывода `apples`.
- ★ Сделайте еще два шага. Приложение возвращается методу `SimpleEnumerable`, но **первая команда метода пропускается** и управление передается во вторую строку:

```
11 static IEnumerable<string> SimpleEnumerable()
12 {
13     yield return "apples";
14     yield return "oranges";
15     yield return "bananas";
16     yield return "unicorns";
17 }
```

Каждый раз, когда цикл `foreach` получает элемент из последовательности, возвращаемой методом `SimpleEnumerable`, он передает управление методу сразу же после выполнения последней команды `yield return`.

- ★ Продолжайте пошаговое выполнение. Приложение возвращается к циклу `foreach`, затем переходит к **третьей строке** метода, снова к циклу `foreach` и, наконец, к четвертой строке метода.

Таким образом, `yield return` заставляет метод **вернуть последовательность с поддержкой перебора**. Для этого при каждом вызове возвращается следующий элемент последовательности, а также отслеживается точка, из которой произошел возврат, чтобы продолжить работу с того момента, на котором она была прервана.

Использование yield return для рефакторинга ManualSportSequence

Вы можете создать собственную реализацию IEnumerable<T>, используя yield return для реализации метода GetEnumerator. Например, ниже приведен класс BetterSportSequence, который делает точно то же самое, что и метод ManualSportSequence. Эта версия получается намного компактнее, потому что она использует yield return в своей реализации GetEnumerator:

```
using System.Collections.Generic;
class BetterSportSequence : IEnumerable<Sport> {
    public IEnumerator<Sport> GetEnumerator() {
        int maxEnumValue = Enum.GetValues(typeof(Sport)).Length - 1;
        for (int i = 0; i <= maxEnumValue; i++) {
            yield return (Sport)i;
        }
    }
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
}
```

Команда yield return может использоваться для реализации метода GetEnumerator в IEnumerable<T> и создания собственной последовательности с поддержкой перебора.

Добавьте в решение новый проект консольного приложения. Добавьте новый класс betterSportSequence, измените метод Main для создания его экземпляра и перебора последовательности.

Добавление индексатора для BetterSportSequence

Вы уже видели, что yield return может использоваться в методе для создания IEnumerator<T>. Также с ее помощью можно создать класс, реализующий IEnumerable<T>. Одно из преимуществ создания отдельного класса для последовательности связано с возможностью добавления **индексатора**. Вы уже использовали индексаторы: каждый раз, когда вы включаете в программу квадратные скобки [] для извлечения объекта из списка, массива или словаря (например, myList[3] или myDictionary["Steve"]), вы используете индексатор. В действительности индексатор является обычным методом. Он очень похож на свойство, если не считать того, что он получает один именованный параметр.

В IDE существует *особенно полезный фрагмент кода*, помогающий в добавлении индексаторов. Введите indexer и два раза нажмите клавишу табуляции; IDE автоматически добавит заготовку индексатора в программу.

Индексатор для класса SportCollection выглядит так:

```
public Sport this[int index] {
    get => (Sport)index;
}
```

Вызов индексатора [3] возвращает значение Hockey:

```
var sequence = new BetterSportSequence();
Console.WriteLine(sequence[3]);
```

Повнимательнее присмотритесь к происходящему при использовании фрагмента для создания индексатора — он позволяет задать тип. Вы можете определить индексатор, который получает разные типы, включая строки и даже объекты. И хотя индексатор имеет только get-метод, он также может включать set-метод (вроде тех, которые используются для присваивания значений элементов List).



Будьте
осмотрены!

Последовательности не являются коллекциями

Попробуйте создать класс, реализующий ICollection<int>, и воспользуйтесь меню Quick Actions для реализации его компонентов. Вы увидите, что коллекция должна реализовать не только методы IEnumerable<T>, но и дополнительные свойства (включая Count) и методы (включая Add и Clear). Отсюда видно, что коллекция работает не так, как последовательность с поддержкой перебора.



Упражнение

Создайте класс с поддержкой перебора, который в процессе перебора возвращает последовательность `int`, состоящую из степеней 2 (начиная с 0 и заканчивая наибольшей степенью 2, которая может быть сохранена в `int`).

Используйте `yield return` для создания последовательности степеней 2

Создайте класс с именем `PowersOfTwo`, реализующий `IEnumerable<int>`. Он должен содержать цикл `for`, который начинается с 0 и использует `yield return` для возвращения последовательности, содержащей все степени 2.

Приложение должно выводить на консоль следующую последовательность: 1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536 131072 262144 524288 1048576 2097152 4194304 8388608 16777216 33554432 67108864 134217728 268435456 536870912 1073741824

Верните желаемую последовательность значений

Вы можете использовать методы статического класса `System.Math` для решения следующих задач:

- Вычисление конкретной степени 2: `Math.Pow(power, 2)`
- Определение наибольшей степени 2, которая может быть сохранена в `int`: `Math.Round(Math.Log(int.MaxValue, 2))`

Часто задаваемые вопросы

В: Кажется, я понимаю, как работает команда `yield return`, но можете ли вы объяснить, как она передает управление в середину метода?

О: Когда вы используете `yield return` для создания последовательности с поддержкой перебора, происходит нечто такое, чего вы еще не видели в `C#`. Обычно когда метод выполняет команду `return`, ваша программа выполняет команду сразу же после той, в которой был вызван метод. То же самое происходит при переборе последовательности, созданной `yield return`, — с одним отличием: в ходе перебора запоминается последняя команда `yield return`, выполненная в методе. Затем при перемещении к следующему элементу последовательности, вместо того чтобы начинать от начала метода, ваша программа выполняет следующую команду за последним выполненным вызовом `yield return`. Это позволяет вам построить метод, который возвращает `IEnumerable<T>` простой серией команд `yield return`.

В: Когда я добавил класс, реализующий `IEnumerable<T>`, мне пришлось добавить метод `MoveNext` и свойство `Current`. При использовании `yield return` как я реализовал этот интерфейс без обязательной реализации этих двух компонентов?

О: Когда компилятор встречает метод с командой `yield return`, который возвращает `IEnumerable<T>`, он автоматически добавляет метод `MoveNext` и свойство `Current`. При выполнении первая обнаруженная команда `yield return` возвращает первое значение цикла `foreach`. Когда цикл `foreach` продолжается (вызовом метода `MoveNext`), выполнение возобновляется с командой `yield return`. Его метод `MoveNext` возвращает `false`, когда текущая позиция перебора выходит за последний элемент в коллекции. Следить за всем этим на бумаге может быть непросто, зато все существенно упрощается, если вы загрузите программу в отладчике, вот почему мы начали с пошагового выполнения простой последовательности, использующей `yield return`.



Упражнение
Решение

Создайте класс с поддержкой перебора, который в процессе перебора возвращает последовательность `int`, состоящую из степеней 2 (начиная с 0 и заканчивая наибольшей степенью 2, которая может быть сохранена в `int`).

```
class PowersOfTwo : IEnumerable<int> {
    public IEnumerator<int> GetEnumerator() {
        var maxPower = Math.Round(Math.Log(int.MaxValue, 2));
        for (int power = 0; power < maxPower; power++)
            yield return (int)Math.Pow(2, power);
    }

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}

class Program {
    static void Main(string[] args) {
        foreach (int i in new PowersOfTwo())
            Console.WriteLine($"{i}");
    }
}
```

Не забудьте включить директивы:

```
using System;
using System.Linq;
using System.Collections;
using System.Collections.Generic;
```

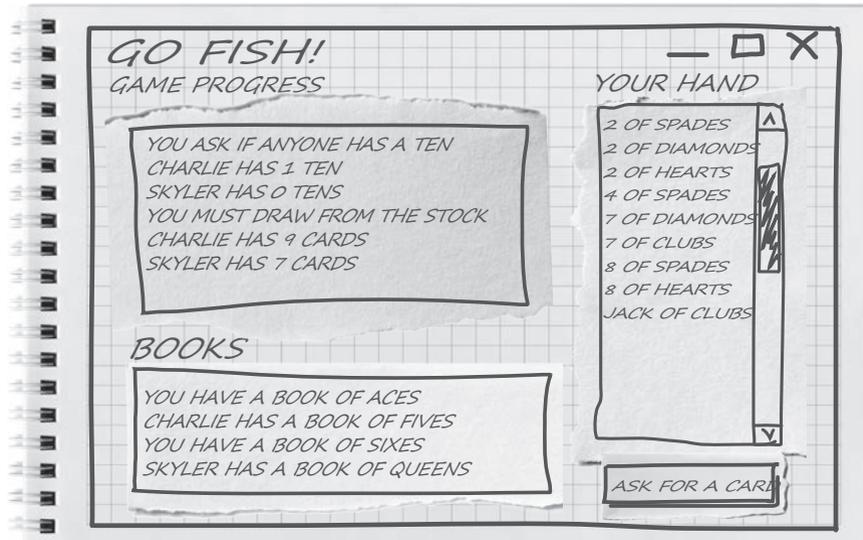
КЛЮЧЕВЫЕ МОМЕНТЫ

- **Модульные тесты** представляют собой автоматизированные тесты, которые позволяют убедиться в том, что код работает так, как ожидалось, и провести **безопасный рефакторинг** вашего кода.
- **MSTest** — **фреймворк модульного тестирования**, т. е. набор классов, предоставляющих инструменты для написания модульных тестов. Visual Studio содержит инструменты для выполнения и просмотра результатов модульных тестов.
- Модульные тесты используют **тестовые утверждения** (assertions) для проверки конкретных аспектов поведения.
- **Ключевое слово** `internal` предоставляет классам одного проекта доступ к другому проекту в многопроектной сборке.
- Добавление модульных тестов для граничных случаев и аномальных данных делает ваш код **более устойчивым к ошибкам**.
- Лямбда-оператор `=>` определяет **лямбда-выражения**, т. е. анонимные функции, определяемые в одной команде, которые выглядят так: (*входные-параметры*) `=>` *выражение*;
- Если класс реализует интерфейс, команда **«Implement Interface»** в меню Quick Actions приказывает IDE добавить все недостающие компоненты интерфейса.
- Секции `orderby` и `where` в запросах LINQ можно переписать с использованием методов LINQ **OrderBy** и **Where**.
- Оператор `=>` может использоваться для **преобразования поля в свойство** с `get`-методом, выполняющим лямбда-выражение.
- **Оператор ?:** (условный или тернарный оператор) позволяет создать выражение, в котором реализуется логика `if/else`.
- Методы LINQ, которые получают параметр **Func<T1, T2>**, могут вызываться с лямбда-выражением, которое получает параметр `T1` и возвращает значение `T2`.
- Оператор `=>` может использоваться для создания **выражений switch** — аналогов команд `switch`, которые возвращают значение.
- Класс `Enumerable` содержит статические **методы Range, Empty и Repeat**, упрощающие создание последовательностей с поддержкой перебора.
- Команды `yield return` создают методы, возвращающие последовательности с поддержкой перебора.
- Когда метод выполняет команду `yield return`, команда возвращает следующее значение в последовательности. При следующем вызове метода **выполнение возобновляется** со следующей команды после последней выполненной команды `yield return`.



Упражнение: Go Fish

В следующем упражнении мы построим карточную игру Go Fish, в которой вы играете против компьютерных игроков. Модульное тестирование очень важно в этом проекте, потому что мы будем практиковать разработку через тестирование — методологию, при которой модульные тесты пишутся ранее того кода, который они тестируют.



Перейдите к репозиторию GitHub данной книги и загрузите PDF-файл проекта:
<https://github.com/head-first-csharp/fourth-edition>

Прибереги последний байт для меня

Ладно, что нужно
купить? Проволочная
сетка... Текила...
Мармелад... Бинты...
Да, я все записываю.



Иногда настойчивость окупается. Пока что все ваши программы жили недолго. Они запускались, некоторое время работали и закрывались. Но этого недостаточно, когда имеешь дело с важной информацией. Вы должны уметь **сохранять свою работу**. В этой главе мы поговорим о том, как **записать данные в файл**, а затем о том, как **прочитать эту информацию**. Вы познакомитесь с **потоками данных**, узнаете о сохранении объектов в файлах с использованием **сериализации**, а также освоите работу с **шестнадцатеричными и двоичными** данными и кодировку Юникод.

Для чтения и записи данных в .NET используются потоки данных

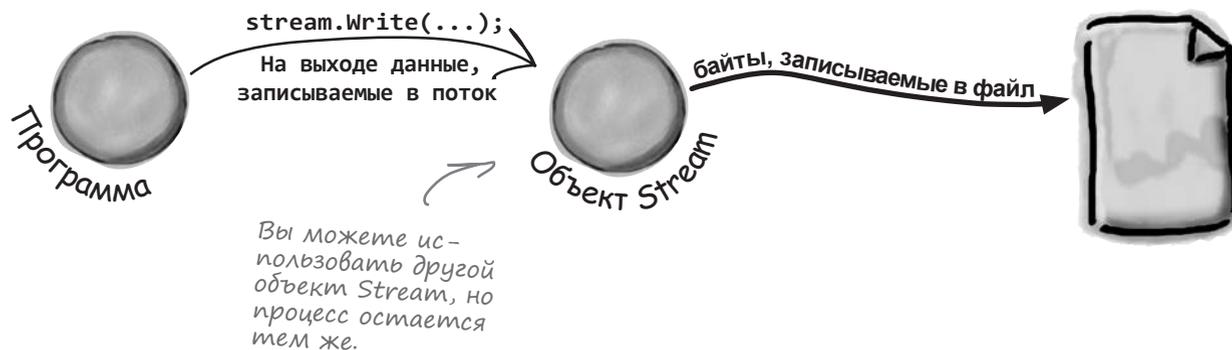
Поток данных, или просто поток (stream), — механизм передачи/получения данных от программы в .NET Framework. Каждый раз, когда программа читает или записывает файл, соединяется с другим компьютером по сети или выполняет какие-либо действия, связанные с **отправкой или получением** байтов, вы используете потоки. В одних случаях потоки используются напрямую, в других — косвенно. Даже когда вы используете классы, которые не предоставляют прямого доступа к потокам, в их внутренней реализации почти всегда задействованы потоки.

Каждый раз, когда вам потребуется прочитать данные из файла или записать данные в файл, вы используете объект Stream.

Допустим, у вас имеется простое приложение, которое должно читать данные из файла. В простейшем варианте можно воспользоваться объектом Stream.

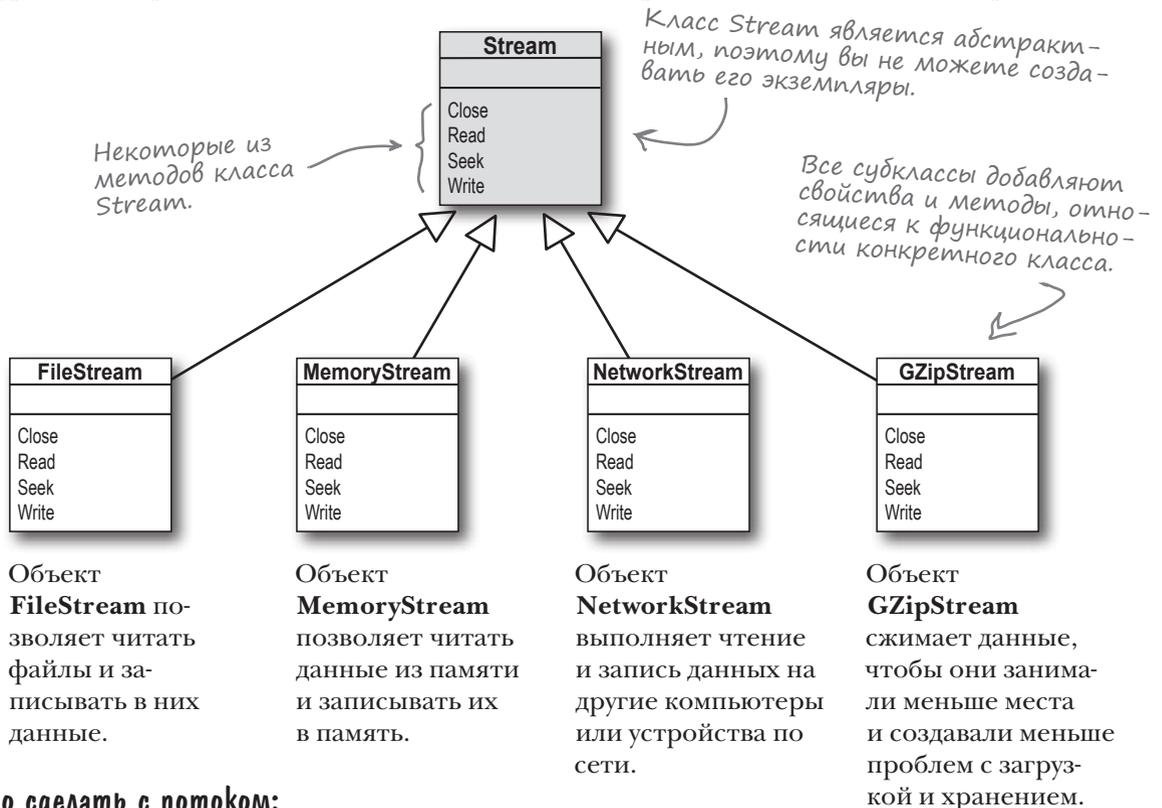


А если вашему приложению потребуется записать данные в файл, оно может воспользоваться другим объектом Stream.



Различные потоки для разных данных

Каждый поток является subclassом **абстрактного класса Stream**, и существует множество subclassов Stream, предназначенных для различных операций. В данной главе мы сосредоточимся на чтении и записи в обычные файлы, но все, что вы узнаете о потоках в этой главе, также применимо к сжатым и зашифрованным файлам и даже к сетевым потокам, которые вообще не используют файлы.



Что можно сделать с потоком:

- 1 **Записывать данные в поток.**
Запись данных в поток осуществляется методом `Write()`.
- 2 **Читать данные из потока.**
Метод `Read()` используется для чтения данных из файла, сети или из памяти (и вообще из любого хранилища, использующего поток). Вы можете читать данные из очень больших файлов — настолько огромных, что они не помещаются в памяти.
- 3 **Изменять текущую позицию в потоке.**
Большинство потоков поддерживают метод `Seek()`, который ищет позицию в потоке для чтения или вставки данных в конкретном месте. Тем не менее не все классы Stream поддерживают `Seek`, что логично, так как в некоторых источниках потоковых данных возврат в принципе невозможен.

Потоки позволяют читать и записывать данные. Выбирайте тип потока, подходящий для тех данных, с которыми вы работаете.

Объект FileStream читает и записывает байты в файл

Если вашей программе потребуется записать несколько строк текста в файл, должно произойти следующее:

1 Вы создаете объект FileStream и приказываете ему записать данные в файл.

2 Объект FileStream присоединяется к файлу.

3 Потоки записывают в файлы байтовые данные, поэтому строка, которая должна быть записана, преобразуется в массив байтов.

Эта процедура называется кодированием; мы поговорим о ней чуть позже...

Eureka! →

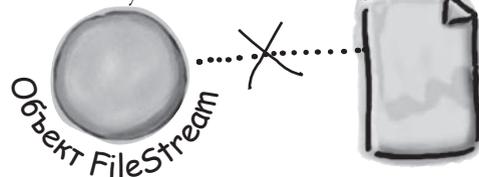


4 Вызывается метод Write(), которому передается массив байтов.

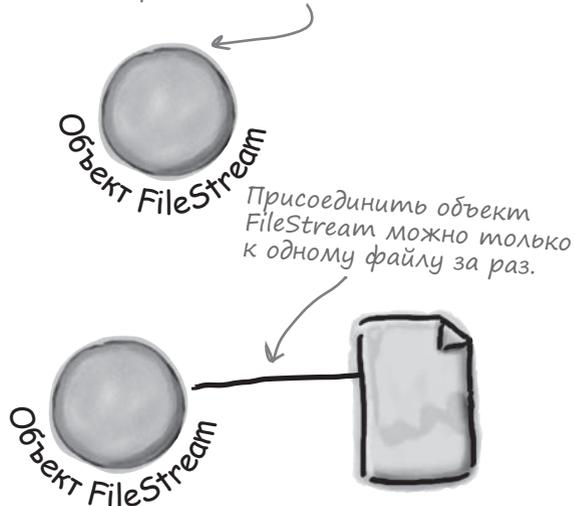


5 Поток закрывается, чтобы другие программы могли получить доступ к файлу.

Не забывайте закрывать потоки, это очень важно. Файл останется заблокированным, и другие программы не смогут работать с ним, пока вы не закроете свой поток.



В любой программе, использующей потоки, должна присутствовать директива <code>using System.IO;</code>



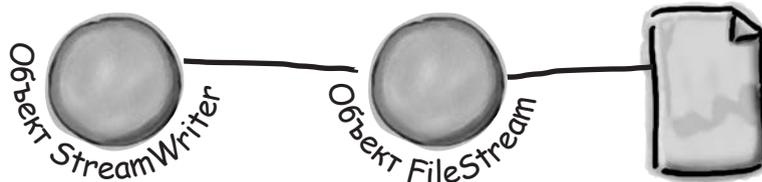
Запись текста в файл за три простых шага

В C# существует удобный класс **StreamWriter**, который упрощает всю эту процедуру. Вам нужно только создать объект `StreamWriter` и присвоить ему имя. Он *автоматически* создаст объект `FileStream` и откроет файл. После этого вам остается только воспользоваться методами `Write()` и `WriteLine()` для записи в файл любых данных на ваше усмотрение.

Класс
StreamWriter
автоматически
создает объект
FileStream
и управляет им.

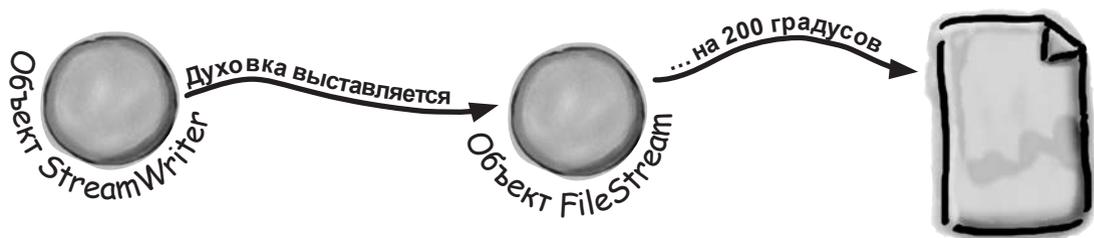
- Для открытия и создания файлов используйте конструктор класса StreamWriter.**
Имя файла можно передать конструктору `StreamWriter`. В таком случае файл открывается автоматически. В классе `StreamWriter` существует также перегруженный конструктор, позволяющий задать *режим добавления данных*: со значением `true` текст добавляется в конец существующего файла, а со значением `false` существующий файл удаляется и создается новый файл с тем же именем.

```
var writer = new StreamWriter("toaster oven.txt", true);
```



- Для записи в файл используйте методы Write() и WriteLine()**
Эти методы работают так же, как их аналоги из класса `Console`: `Write()` записывает текст, а `WriteLine()` добавляет к тексту символ новой строки.

```
writer.WriteLine($"The {appliance} is set to {temp} degrees.");
```



- Для освобождения файла используйте метод Close.**
Если оставить поток открытым и соединенным с файлом, то файл окажется заблокированным и ни одна программа не сможет работать с ним. Не забывайте всегда закрывать свои файлы!

```
writer.Close();
```

Дьявольский план Пройдохи

Жители Объектвиля долгое время боялись Пройдохи, главного противника Капитана Великолепного. И вот он решил воспользоваться объектом StreamWriter, чтобы воплотить в жизнь свой зловещий план. Посмотрим на него поближе. Создайте проект консольного приложения и добавьте этот код в метод Main, начиная с объявления using, потому что класс StreamWriter находится в пространстве имен System.IO:

```
using System.IO;

class Program
{
    static void Main(string[] args)
    {
        StreamWriter sw = new StreamWriter("secret_plan.txt");

        sw.WriteLine("How I'll defeat Captain Amazing");
        sw.WriteLine("Another genius secret plan by The Swindler");
        sw.WriteLine("I'll unleash my army of clones upon the citizens of Objectville.");

        string location = "the mall";
        for (int number = 1; number <= 5; number++)
        {
            sw.WriteLine("Clone #{0} attacks {1}", number, location);
            location = (location == "the mall") ? "downtown" : "the mall";
        }
        sw.Close();
    }
}
```

Эта строка создает объект StreamWriter и сообщает ему, где находится файл.

StreamWriter принадлежит пространству имен System.IO.

Методы Write и WriteLine класса StreamWriter работают так же, как аналогичные методы класса Console: Write записывает текст, а WriteLine записывает текст с символом новой строки. Оба класса поддерживают фигурные скобки: sw.WriteLine("Clone #{0} attacks {1}", number, location);

Когда вы включаете {0} в текст, этот заполнитель заменяется первым параметром после строки; {1} заменяется вторым параметром, {2} — третьим, и т. д.

Попробуйте определить, что происходит с переменной location и тернарным оператором ?.

Очень важно вызвать Close при завершении работы с StreamWriter — тем самым вы разрываете связь с файлом и освобождаете другие ресурсы, с которыми работает экземпляр StreamWriter. Если не закрыть поток, то может оказаться, что часть текста не была записана (а возможно, весь текст!).

Так как вы не включили полный путь в имя файла, выходной файл находится в одной папке с двоичным — если приложение выполняется из Visual Studio, проверьте папку bin\Debug\netcoreapp3.1 в папке решения.

Если вы используете другую версию .NET, то подкаталог внутри Debug может выглядеть иначе.

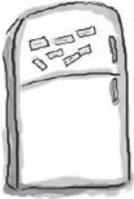
Пройдоха — главный враг Капитана Великолепного; призрачный сверхзлодей, стремящийся к захвату Объектвиля.

Результат, записанный в файл secret_plan.txt:

```
How I'll defeat Captain Amazing
Another genius secret plan by The Swindler
I'll unleash my army of clones upon the citizens of Objectville.
Clone #1 attacks the mall
Clone #2 attacks downtown
Clone #3 attacks the mall
Clone #4 attacks downtown
Clone #5 attacks the mall
```

Результат:





Развлечения с Магнитами

Какая неприятность! Из магнитов на холодильнике был аккуратно выложен код класса Flobbo, но кто-то хлопнул дверью и магниты упали на пол. Сможете ли вы расставить их так, чтобы метод Main выводил приведенный ниже результат?

```
static void Main(string[] args) {
    Flobbo f = new Flobbo("blue yellow");
    StreamWriter sw = f.Snobbo();
    f.Blobbo(f.Blobbo(f.Blobbo(sw), sw), sw);
}
```

Предполагается, что в начале всех файлов с кодом располагается директива System.IO;

```
public bool Blobbo
    (bool Already, StreamWriter sw)
{
```

```
public bool Blobbo(StreamWriter sw) {
```

```
    if (Already) {
```

```
        private string zap;
        public Flobbo(string zap) {
            this.zap = zap;
        }
    }
```

```
        public StreamWriter Snobbo() {
```

```
        }
        else
        {
```

```
            sw.WriteLine(zap);
            zap = "green purple";
            return false;
        }
```

```
        return new
            StreamWriter("macaw.txt");
    }
```

```
class Flobbo
{
```

```
    sw.WriteLine(zap);
    sw.Close();
    return false;
}
```

```
    sw.WriteLine(zap);
    zap = "red orange";
    return true;
}
```

```
    }
}
}
```

Результат работы приложения, записываемый в файл с именем macaw.txt.

Результат
blue yellow
green purple
red orange

Мы добавили дополнительное упражнение.

С методом Blobbo происходит что-то странное. На первых двух магнитах приведены два разных объявления? Мы определили Blobbo как **перегруженный метод** — существуют две разные версии этого метода, каждая со своим набором параметров (как и перегруженные методы, использовавшиеся в предыдущих главах).



Развлечения с магнитами. Решение

Вам было предложено сконструировать класс Flobbo из магнитов, чтобы получить нужный результат.

```
static void Main(string[] args) {
    Flobbo f = new Flobbo("blue yellow");
    StreamWriter sw = f.Snobbo();
    f.Blobbo(f.Blobbo(f.Blobbo(sw), sw), sw);
}
```

Добавьте код в консольное приложение. Результат будет записан в файл *macaw.txt* в той же папке, где находится двоичный файл, — подкаталоге папки *bin\Debug* внутри папки проекта.

Предполагается, что в начале всех файлов с кодом располагается директива **System.IO**;

Результат

```
blue yellow
green purple
red orange
```

```
class Flobbo
{
```

```
    private string zap;
    public Flobbo(string zap) {
        this.zap = zap;
    }
```

```
    public StreamWriter Snobbo() {
        return new
            StreamWriter("macaw.txt");
    }
```

```
    public bool Blobbo(StreamWriter sw) {
        sw.WriteLine(zap);
        zap = "green purple";
        return false;
    }
```

```
    public bool Blobbo
        (bool Already, StreamWriter sw)
    {
```

```
        if (Already) {
            sw.WriteLine(zap);
            sw.Close();
            return false;
        }
```

```
        else {
            sw.WriteLine(zap);
            zap = "red orange";
            return true;
        }
    }
```

```
    }
}
```

Определение перегруженных методов

В главе 8 вы узнали, что метод *Random.Next* является перегруженным, — он существует в трех версиях с разными наборами параметров. Метод *Blobbo* тоже перегружен — у него два объявления с разными параметрами:

```
public bool Blobbo(StreamWriter sw)
```

и

```
public bool Blobbo(bool Already, StreamWriter sw)
```

Два перегруженных метода *Blobbo* полностью независимы друг от друга. Они обладают разным поведением (как и перегруженные версии *Random.Next*). Если добавить эти два метода в класс, IDE будет отображать их как перегруженные методы (опять-таки по аналогии с *Random.Next*).

После завершения работы не забудьте закрыть файлы. Подумайте и попытайтесь ответить, почему метод вызывается после записи всего текста.

Просто напомним: в этих упражнениях мы намеренно выбрали невразумительные имена переменных и методов, потому что с содержательными именами головоломки оказались бы слишком простыми! Не используйте такие имена в своем коде, хорошо?

Использование StreamReader для чтения файла

Прочитаем секретные планы Пройдохи при помощи StreamReader. Этот класс имеет много общего со StreamWriter, кроме того, что вместо записи файла вы создаете объект StreamReader и передаете ему имя файла, из которого будут читаться данные, в конструкторе. Метод ReadLine возвращает строку, содержащую следующую строку текста из файла. Можно написать цикл, который читает данные по строкам, пока поле EndOfStream не станет равным true, т. е. пока не кончатся новые строки для чтения. Добавьте консольное приложение, использующее StreamReader для чтения одного файла, и StreamWriter для записи другого файла:

```
using System.IO;
```

```
class Program
```

```
{
```

```
    static void Main(string[] args)
```

```
    {
```

```
        var folder = Environment.GetFolderPath(Environment.SpecialFolder.Personal);
```

```
        var reader = new StreamReader($"{folder}{Path.DirectorySeparatorChar}secret_plan.txt");
        var writer = new StreamWriter($"{folder}{Path.DirectorySeparatorChar}emailToCaptainA.txt");
```

```
        writer.WriteLine("To: CaptainAmazing@objectville.net");
        writer.WriteLine("From: Commissioner@objectville.net");
        writer.WriteLine("Subject: Can you save the day... again?");
        writer.WriteLine();
        writer.WriteLine("We've discovered the Swindler's terrible plan:");
```

```
        while (!reader.EndOfStream)
```

```
        {
```

```
            var lineFromThePlan = reader.ReadLine();
            writer.WriteLine($"The plan -> {lineFromThePlan}");
```

```
        }
```

```
        writer.WriteLine();
        writer.WriteLine("Can you help us?");
```

```
        writer.Close();
        reader.Close();
```

```
    }
```

```
}
```

Класс StreamReader читает символы из потоков, но сам потоком не является.

Когда вы передаете имя файла его конструктору, он создает поток за вас и закрывает его при вызове метода Close. Также он содержит перегруженный конструктор, получающий ссылку на Stream.

Возвращает путь к папке Documents пользователя в Windows или к домашнему каталогу пользователя в macOS. Убедитесь в том, что файл secret_plan.txt был скопирован в эту папку! За информацией о других папках, доступных для поиска, обращайтесь к описанию перечисления SpecialFolder.

Передайте имя файла, который вы собираетесь прочитать, конструктору StreamReader.

Свойство EndOfStream содержит true, если все данные в файле были прочитаны.

Этот цикл читает строку из StreamReader и записывает ее в StreamWriter.

```
To: CaptainAmazing@objectville.net
From: Commissioner@objectville.net
Subject: Can you save the day... again?
```

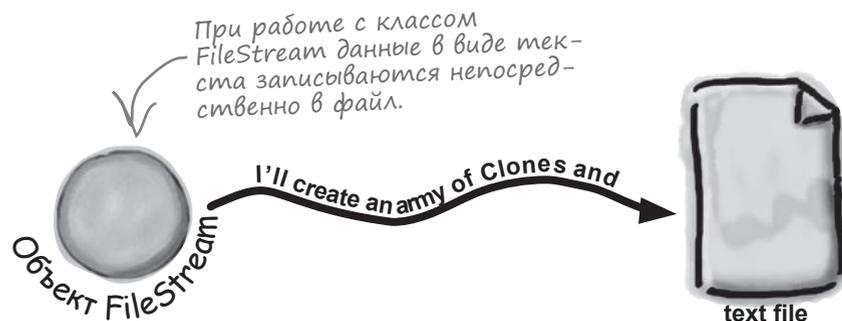
```
We've discovered the Swindler's terrible plan:
The plan -> How I'll defeat Captain Amazing
The plan -> Another genius secret plan by The Swindler
The plan -> I'll unleash my army of clones upon the citizens of Objectville.
The plan -> Clone #1 attacks the mall
The plan -> Clone #2 attacks downtown
The plan -> Clone #3 attacks the mall
The plan -> Clone #4 attacks downtown
The plan -> Clone #5 attacks the mall
```

```
Can you help us?
```

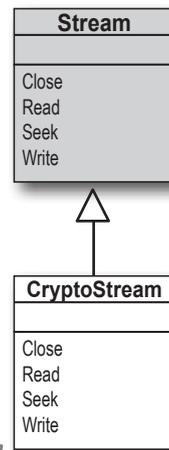
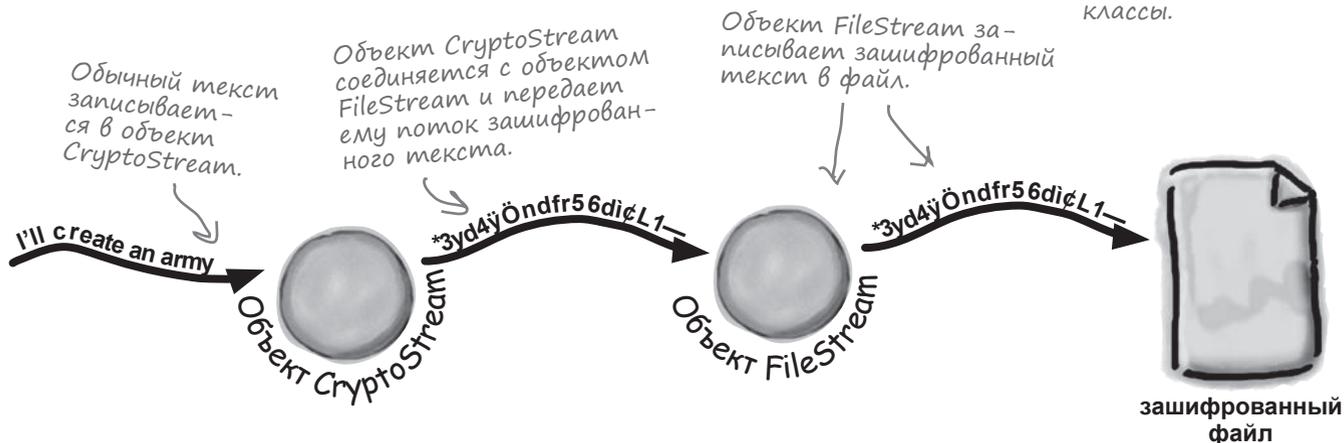
Результат

Данные могут проходить через несколько потоков

У потоков .NET есть одно важное преимущество: они позволяют передать данные приемнику через несколько потоков. К числу многочисленных типов данных потоков .NET Core принадлежит класс `CryptoStream`. Он позволяет зашифровать данные перед тем, как проделать с ними все остальные операции. Таким образом, вместо записи простого текста в обычный текстовый файл:



Пройдоха может объединить эти потоки **в цепочку** и передать текст через объект `CryptoStream`, прежде чем записывать вывод в `FileStream`:



Класс `CryptoStream` наследует от абстрактного класса `Stream`, как и все прочие потоковые классы.

Потоки можно ОБЪЕДИНИТЬ В ЦЕПОЧКУ. Один поток записывает свои данные в другой, который, в свою очередь, записывает свои данные в третий поток... концом цепочки часто является сетевой ресурсе или файловый поток.

У бассейна

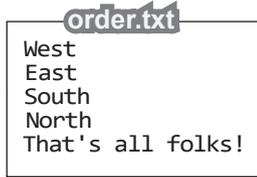


Ваша **задача** — взять фрагменты кода из бассейна и разместить их в пустых строках классов Pineapple, Pizza и Party. Любой фрагмент можно использовать несколько раз, использовать все фрагменты не обязательно. Ваша **цель** — добиться того, чтобы программа записала файл с именем order.txt с пятью строками, приведенными ниже.

```
class Pineapple {
    const _____d = "delivery.txt";
    public _____
        { North, South, East, West, Flamingo }
    public static void Main(string[] args) {
        _____o = new _____("order.txt");
        var pz = new _____(new _____(d, true));
        pz._____( Fargo.Flamingo);
        for ( _____w = 3; w >= 0; w--) {
            var i = new _____(new _____(d, false));
            i.Idaho(( Fargo)w);
            Party p = new _____(new _____(d));
            p.HowMuch(o);
        }
        o._____("That's all folks!");
        o._____( );
    }
}
```

```
class Pizza {
```

Код записывает эти строки в файл order.txt.



Дополнительный вопрос



Возьми в руку карандаш

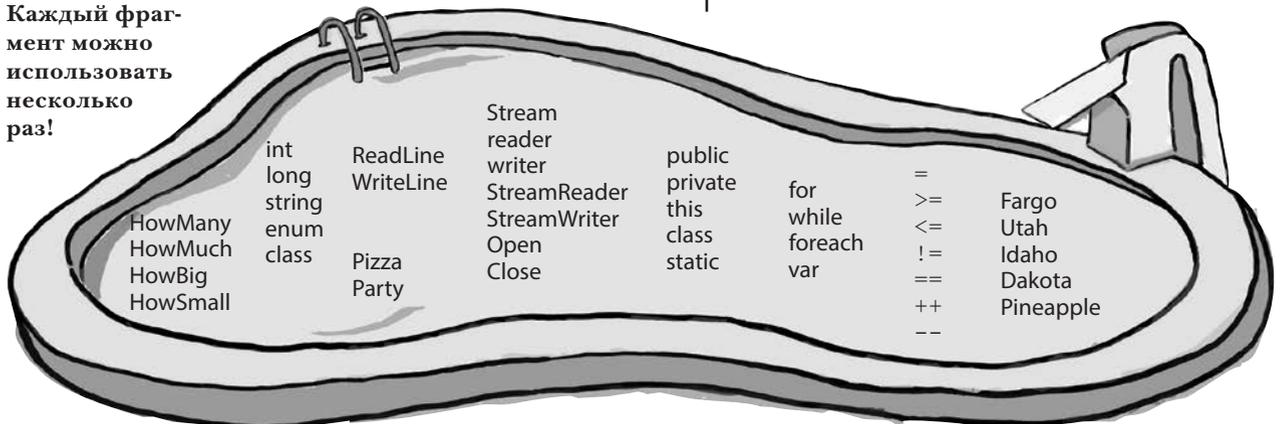
Какой текст приложение запишет в файл delivery.txt?

.....

```
private _____writer;
public Pizza(_____ ) {
    this.writer = writer;
}
public void Idaho(_____ Fargo f) {
    writer._____(f);
    writer._____( );
}
}

class Party {
    private _____reader;
    public Party(_____ ) {
        this.reader = reader;
    }
    public void HowMuch(_____ q) {
        q._____(reader._____( ));
        reader._____( );
    }
}
}
```

Каждый фрагмент можно использовать несколько раз!





У бассейна. Решение

Это перечисление используется для вывода результата. В главе 8 вы узнали, что метод ToString перечисления возвращает эквивалентную строку, так что вызов Fargo.North.ToString() вернет строку «North».

Точка входа программы. Здесь создается объект StreamWriter, который передается классу Party. Затем элементы перечисления Fargo в цикле передаются методу Pizza.Idaho для вывода.

```
class Pineapple {
    const string d = "delivery.txt";
    public enum Fargo
        { North, South, East, West, Flamingo }
    public static void Main(string[] args) {
        var o = new StreamWriter("order.txt");
        var pz = new Pizza(new StreamWriter(d, true));
        pz.Idaho(Fargo.Flamingo);
        for (int w = 3; w >= 0; w--) {
            var i = new Pizza(new StreamWriter(d, false));
            i.Idaho((Fargo)w);
            Party p = new Party(new StreamReader(d));
            p.HowMuch(o);
        }
        o.WriteLine("That's all folks!");
        o.Close();
    }
}

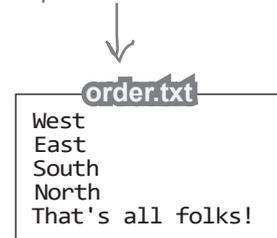
class Pizza {
    private StreamWriter writer;
    public Pizza(StreamWriter writer) {
        this.writer = writer;
    }
    public void Idaho(Pineapple.Fargo f) {
        writer.WriteLine(f);
        writer.Close();
    }
}

class Party {
    private StreamReader reader;
    public Party(StreamReader reader) {
        this.reader = reader;
    }
    public void HowMuch(StreamWriter q) {
        q.WriteLine(reader.ReadLine());
        reader.Close();
    }
}
```

Класс Pizza хранит объект StreamWriter в приватном поле, а его метод Idaho записывает в файл элементы перечисления Fargo; для этого используются их методы ToString(), автоматически вызываемые методом WriteLine().

В классе Party имеется поле StreamReader; метод HowMuch() читает из StreamReader строку и записывает ее в StreamWriter.

Результат, который записывается приложением в файл order.txt.



MM Возьми в руку карандаш
Решение

Какой текст приложение запишет в файл delivery.txt?

..... North

В: Объясните, для чего нужны {0} и {1} при вызове методов StreamWriter Write и WriteLine?

О: При выводе строк в файл часто оказывается, что вместе с текстом нужно вывести содержимое множества переменных. И тогда приходится писать громоздкие конструкции следующего вида:

```
writer.WriteLine("My name is " + name +
    "and my age is " + age);
```

Объединять строки оператором + утомительно, к тому же возрастает риск ошибок. Проще воспользоваться **составным форматированием**, т. е. **форматной строкой с заполнителями вида {0}, {1}, {2} и т. д.**, а также переменными для замены заполнителей:

```
writer.WriteLine(
    "My name is {0} and my age is {1}", name, age);
```

Возможно, у вас возникла мысль — все это очень похоже на строковую интерполяцию? И вы абсолютно правы! Часто строковая интерполяция читается проще, в других случаях вариант с форматной строкой более нагляден. Как и при строковой интерполяции, **форматные строки поддерживают форматирование**. Например, {1:0.00} означает, что второй аргумент должен форматироваться как число с двумя знаками в дробной части, а {3:c} приказывает отформатировать четвертый аргумент как денежную сумму в местной валюте.

Да, и еще — форматные строки также работают с Console.WriteLine и Console.WriteLine!

В: Что это за поле Path.DirectorySeparatorChar, которое использовалось в консольном приложении с StringReader?

О: Мы писали этот код так, чтобы он работал и в Windows, и в macOS. Для этого пришлось воспользоваться некоторыми средствами .NET Core, которые помогают добиться этой цели. В Windows в качестве разделителя путей используется обратная косая черта (C:\Windows), а в macOS — обычная (/Users).

Path.DirectorySeparatorChar — поле, доступное только для чтения, в котором хранится разделитель пути для текущей операционной системы: \ для Windows или / для macOS и Linux.

Мы также использовали метод Environment.GetFolderPath, который возвращает путь к одной из специальных папок текущего пользователя, в данном случае к папке Documents в Windows или домашнему каталогу в macOS.

В: А что там говорилось про преобразование строки в массив байтов? Как это работает?

О: Наверняка вы слышали, что файлы на диске представлены в виде битов и байтов. Другими словами, при записи файла на диск операционная система воспринимает его как набор байтов. Объекты StreamReader и StreamWriter преобразуют эти байты в понятные вам символы, т. е. выполняют *кодирование* и *декодирование*. Помните, в главе 4 упоминалось, что переменная типа byte хранит значения от 0 до 255? Все файлы на жестком диске представляют собой длинные последовательности чисел из этого диапазона. Программа, читающая и записывающая эти файлы, должна интерпретировать байты как осмысленные данные. Например, при открытии файла в приложении Блокнот каждый байт преобразуется в символ: например, E соответствует 69, а a — 97 (впрочем, все зависит от кодировки... но об этом чуть позже). Соответственно, при сохранении введенного в Блокноте текста символы преобразуются обратно в байты. Это преобразование необходимо и для записи строки в поток.

В: Если я использую StreamWriter для записи файла, зачем мне знать, что он создает за меня объект FileStream?

О: Если вы ограничиваетесь записью и чтением строк из текстового файла, для этого действительно достаточно объектов StreamReader и StreamWriter. Но как только вы займетесь более сложными задачами, вам придется работать с другими потоками. Записывать в файл числа, массивы, коллекции и объекты StreamWriter не умеет. Впрочем, эта тема будет подробно рассмотрена чуть позже.

В: Почему вы постоянно говорите, что потоки нужно закрывать после окончания работы?

О: Сообщал ли вам когда-нибудь текстовый редактор, что он не может открыть файл, потому что «файл используется другим приложением»? Windows блокирует открытые файлы и не позволяет открывать в других приложениях. Ваши программы не являются исключением — Windows блокирует файлы, открытые вашими приложениями. Если вы не вызовете метод Close, файл может остаться заблокированным вплоть до завершения программы.

Классы Console и StreamWriter могут использовать составное форматирование, при котором заполнители заменяются значениями параметров, передаваемых при вызове Write или WriteLine.

Работа с файлами и каталогами с использованием статических классов File и Directory

Класс File, как и класс StreamWriter, незаметно для разработчика создает потоки для работы с файлами. Методы этого класса дают возможность выполнять большинство операций без предварительного создания объекта FileStream. Класс Directory предназначен для работы с каталогами, содержащими множество файлов.

Что можно сделать со статическим классом File:

- 1 **Проверить, существует ли файл.**
Для проверки существования файла используется метод File.Exists. Он возвращает true, если файл существует, или false в противном случае.
- 2 **Читать из файла и записывать данные в файл.**
Метод File.OpenRead() читает данные из файла, а методы File.Create() и File.OpenWrite() записывают данные в файл.
- 3 **Присоединять текст к файлу.**
Метод File.AppendAllText() присоединяет текст к существующему файлу; если файл не найден на момент выполнения метода, он автоматически создается.
- 4 **Получать информацию о файле.**
Методы File.GetLastAccessTime() и File.GetLastWriteTime() возвращают время и дату последнего обращения к файлу и его последнего изменения.

Что можно сделать со статическим классом Directory:

- 1 **Создать новую папку**
Метод Directory.CreateDirectory() создает каталоги. От вас потребуется лишь указать путь; метод сделает все остальное.
- 2 **Получить список файлов в папке**
Метод Directory.GetFiles() создает массив с информацией о файлах в каталоге. Укажите путь к каталогу, метод сделает все остальное.
- 3 **Удалить каталог.**
Потребовалось удалить каталог? Вызовите метод Directory.Delete().

FileInfo работает как File

Если вы собираетесь часто и помногу работать с файлами, имеет смысл создать экземпляр класса FileInfo вместо использования статических методов класса File.

Класс FileInfo делает все то же, что и класс File, кроме того, что для работы с файлами потребуется создать его экземпляр. Вы можете создать новый экземпляр FileInfo и обратиться к его методу Exists или методу OpenRead точно так же, как это делалось с File.

Между двумя классами есть принципиальное различие: класс File быстрее работает при небольшом количестве операций с файлом, в то время как класс FileInfo лучше подходит для многочисленных операций.

Класс File является статическим, т. е. представляет собой простой набор методов для работы с файлами. Класс FileInfo поддерживает создание экземпляров; после того, как это будет сделано, вы получаете доступ к тому же набору методов, что и при работе с классом File.



Возьми в руку карандаш

Для работы с файлами и папками в .NET существуют два класса с многочисленными статическими методами и интуитивно понятными именами. Класс `File` содержит методы для работы с файлами, а класс `Directory` дает возможность работать с каталогами. Напишите, как, с вашей точки зрения, работают представленные слева строки кода, а затем ответьте на два дополнительных вопроса в конце.

Код	Что делает
<code>if (!Directory.Exists(@"C:\SYP")) { Directory.CreateDirectory(@"C:\SYP"); }</code>	
<code>if (Directory.Exists(@"C:\SYP\Bonk")) { Directory.Delete(@"C:\SYP\Bonk"); }</code>	
<code>Directory.CreateDirectory(@"C:\SYP\Bonk");</code>	
<code>Directory.SetCreationTime(@"C:\SYP\Bonk", new DateTime(1996, 09, 23));</code>	
<code>string[] files = Directory.GetFiles(@"C:\SYP\ "*.log", SearchOption.AllDirectories);</code>	
<code>File.WriteAllText(@"C:\SYP\Bonk\weirdo.txt", @"This is the first line and this is the second line and this is the last line");</code>	
<code>File.Encrypt(@"C:\SYP\Bonk\weirdo.txt");</code> <i>Этот метод вы еще не видели — попробуйте догадаться, что он делает.</i>	
<code>File.Copy(@"C:\SYP\Bonk\weirdo.txt", @"C:\SYP\copy.txt");</code>	
<code>DateTime myTime = Directory.GetCreationTime(@"C:\SYP\Bonk");</code>	
<code>File.SetLastWriteTime(@"C:\SYP\copy.txt", myTime);</code>	
<code>File.Delete(@"C:\SYP\Bonk\weirdo.txt");</code>	

Почему мы поставили @ перед каждой строкой, передаваемой в аргументах этих методов?

Все имена файлов в этих примерах начинаются с C:\ для работы в Windows. Что произойдет при попытке выполнить этот код в macOS или Linux?

.....
.....

.....
.....



Возьми в руку карандаш Решение

Для работы с файлами и папками в .NET существуют два класса с многочисленными статическими методами и интуитивно понятными именами. Класс `File` содержит методы для работы с файлами, а класс `Directory` дает возможность работать с каталогами. Вам было предложено записать, что делает каждый блок кода.

Код	Что делает
<pre>if (!Directory.Exists(@"C:\SYP")) { Directory.CreateDirectory(@"C:\SYP"); }</pre>	Проверяет, существует ли папка <code>C:\SYP</code> , и если не существует — создает ее.
<pre>if (Directory.Exists(@"C:\SYP\Bonk")) { Directory.Delete(@"C:\SYP\Bonk"); }</pre>	Проверяет, существует ли папка <code>C:\SYP\Bonk</code> , и если существует — удаляет ее.
<pre>Directory.CreateDirectory(@"C:\SYP\Bonk");</pre>	Создает каталог <code>C:\SYP\Bonk</code> .
<pre>Directory.SetCreationTime(@"C:\SYP\Bonk", new DateTime(1996, 09, 23));</pre>	Устанавливает время создания для папки <code>C:\SYP\Bonk</code> на 23 сентября 1996 г.
<pre>string[] files = Directory.GetFiles(@"C:\SYP\", "*.log", SearchOption.AllDirectories);</pre>	Получает список всех файлов из <code>C:\SYP</code> , соответствующих шаблону <code>*.log</code> , включая все подходящие файлы в любом подкаталоге.
<pre>File.WriteAllText(@"C:\SYP\Bonk\weirdo.txt", @"This is the first line and this is the second line and this is the last line");</pre>	Создает файл с именем «weirdo.txt» (если он не существует) в папке <code>C:\SYP\Bonk</code> и записывает в него три строки текста.
<pre>File.Encrypt(@"C:\SYP\Bonk\weirdo.txt");</pre>	Использует встроенные средства шифрования Windows для шифрования файла «weirdo.txt» с использованием реквизитов текущей учетной записи.
<pre>File.Copy(@"C:\SYP\Bonk\weirdo.txt", @"C:\SYP\copy.txt");</pre>	Копирует файл <code>C:\SYP\Bonk\weirdo.txt</code> в <code>C:\SYP\Copy.txt</code> .
<pre>DateTime myTime = Directory.GetCreationTime(@"C:\SYP\Bonk");</pre>	Объявляет переменную <code>myTime</code> и присваивает ей время создания папки <code>C:\SYP\Bonk</code> .
<pre>File.SetLastWriteTime(@"C:\SYP\copy.txt", myTime);</pre>	Изменяет время последней записи в файл <code>copy.txt</code> из <code>C:\SYP</code> , чтобы оно было равно времени, хранящемуся в переменной <code>myTime</code> .
<pre>File.Delete(@"C:\SYP\Bonk\weirdo.txt");</pre>	Удаляет файл <code>C:\SYP\Bonk\weirdo.txt</code> .

↑ Альтернатива для `CryptoStream`

Почему мы поставили @ перед каждой строкой, передаваемой в аргументах этих методов?

Все имена файлов в этих примерах начинаются с `C:\` для работы в Windows. Что произойдет при попытке выполнить этот код в macOS или Linux?

.....
@ не позволяет интерпретировать символы \
.....
в строке как часть служебных последовательностей.
.....

.....
Программа создаст файл с именем, начинающимся
.....
с «C:\», в одной папке с двоичным файлом.
.....

Интерфейс IDisposable обеспечивает корректное закрытие объектов

Многие классы .NET реализуют исключительно полезный интерфейс IDisposable. Этот интерфейс содержит всего **один метод** Dispose. Если класс реализует IDisposable, он тем самым сообщает, что есть важные операции, которые требуется сделать для корректного завершения работы, — обычно из-за того, что **выделенные ресурсы** не освобождаются самостоятельно. Метод Dispose сообщает объекту, как освободить эти ресурсы.

Исследование IDisposable в IDE

Воспользуемся функцией IDE Go To Definition (или «Go to Declaration» на Mac) для просмотра определения интерфейса IDisposable. Введите IDisposable в произвольном месте внутри класса, щелкните на этой строке правой кнопкой мыши и выберите в меню команду Go To Definition. Откроется новая вкладка с кодом. Вот что вы увидите:

Многие классы выделяют такие важные ресурсы, как память, файлы и другие объекты. Это означает, что класс захватил данный ресурс и не вернет его системе, пока не получит сигнал, что работа закончена.

```
namespace System
{
    /// <summary>
    /// Предоставляет механизм освобождения неуправляемых ресурсов.
    /// </summary>
    public interface IDisposable
    {
        /// <summary>
        /// Выполняет операции, определяемые приложением, при
        /// освобождении или сбросе неуправляемых ресурсов.
        /// </summary>
        void Dispose();
    }
}
```

↑
Любой класс, реализующий интерфейс IDisposable, должен немедленно освободить любые за- действованные ресурсы при вызове его метода Dispose. Это почти всегда становится последним действием перед завершением работы с объектом.

ВЫДЕ-ЛЯТЬ, глагол.
Раздавать ресурсы или обязанности для определенных целей.

Предотвращение ошибок файловой системы командами `using`

На протяжении главы вам твердили о необходимости закрывать потоки. Ведь самые распространенные ошибки, с которыми сталкиваются программисты при работе с файлами, возникают как раз из-за того, что потоки не были закрыты как положено. К счастью, в C# имеется замечательный инструмент, позволяющий избежать подобной ситуации, — это интерфейс `IDisposable` с его методом `Dispose`. Если заключить код работы с потоком в команду `using`, поток начнет закрываться автоматически. Вам нужно только объявить ссылку на поток в команде `using`, а затем указать блок кода (в фигурных скобках), в котором используется эта ссылка. В результате C# автоматически вызывает метод `Dispose` сразу же после завершения блока.

← В данном случае речь идет вовсе не о тех командах (директивах) `using`, которые располагаются в начале части кода.

За командой `using` всегда следует объявление объекта...

...а затем блок кода в фигурных скобках.

```
using (var sw = new StreamWriter("secret_plan.txt")) {  
    sw.WriteLine("How I'll defeat Captain Amazing");  
    sw.WriteLine("Another genius secret plan");  
    sw.WriteLine("by The Swindler");  
}
```

↑ После выполнения последней команды в блоке для используемого объекта вызывается метод `Dispose`.

← В данном случае на используемый объект указывает ссылка `sw`, объявленная внутри команды `using`. Поэтому будет выполнен метод `Dispose()` класса `Stream...` который и закроет поток.

Команда `using` объявляет переменную `sw`, содержащую ссылку на новый объект `StreamWriter`, после которой идет блок кода. После того как будут выполнены все команды в блоке, блок `using` автоматически вызовет `sw.Dispose`.

По одной команде `using` на объект

Операторы `using` можно размещать друг за другом, при этом не потребуются дополнительный набор фигурных скобок или дополнительные отступы:

```
using (var reader = new StreamReader("secret_plan.txt"))  
using (var writer = new StreamWriter("email.txt"))  
{  
    // команды, использующие StreamReader и StreamWriter  
}
```

Каждый поток содержит метод `Dispose`, который закрывает этот поток. Поток, объявленный в команде `using`, будет закрыт гарантированно! И это важно, потому что некоторые потоки записывают последнюю часть своих данных только в момент закрытия.

Если вы объявили объект в блоке `using`, метод `Dispose` будет вызван для этого объекта автоматически.

Потоки MemoryStream и хранение данных в памяти

До настоящего момента мы использовали потоки для чтения и записи файлов. А если вы хотите прочитать данные из файла, а потом... как-то их обработать? Используйте класс **MemoryStream**, который отслеживает все переданные ему данные и сохраняет их в памяти. Например, можно создать новый объект **MemoryStream** и передать его в аргументе конструктора **StreamWriter**, после чего любые данные, записываемые с помощью **StreamWriter**, будут отправляться **MemoryStream**. Для получения этих данных можно воспользоваться методом **MemoryStream.ToArray**, который возвращает все данные, переданные **MemoryStream**, в виде байтового массива.

Преобразование байтовых массивов в строку методом `Encoding.UTF8.GetString`

Одна из самых распространенных операций, выполняемых с байтовыми массивами, — преобразование их в строки. Например, если у вас имеется байтовый массив с именем `bytes`, один из способов его преобразования в строку выглядит так:

```
var converted = Encoding.UTF8.GetString(bytes);
```

Ниже приведено маленькое консольное приложение, которое использует составное форматирование для записи числа в **MemoryStream**, преобразует его в байтовый массив, а затем в строку. Есть только одна проблема... *она не работает!*

Создайте новое консольное приложение и включите в него приведенный ниже код. Вам удастся обнаружить ошибку и исправить ее?

```
using System;
using System.IO;
using System.Text;
```

```
class Program
{
    static void Main(string[] args)
    {
        using (var ms = new MemoryStream())
        using (var sw = new StreamWriter(ms))
        {
            sw.WriteLine("The value is {0:0.00}", 123.45678);
            Console.WriteLine(Encoding.UTF8.GetString(ms.ToArray()));
        }
    }
}
```

← (Делайте это!

Это приложение не работает! Оно должно выводить строку текста на консоль, но оно вообще ничего не выводит. Мы объясним, что не так, но сначала попробуйте обнаружить ошибку самостоятельно.

Подсказка: вы сможете определить, когда потоки закрываются?

Метод `MemoryStream.ToArray` возвращает все потоковые данные в виде байтового массива. Метод `GetString` преобразует байтовый массив в строку.

В: Напомните, почему мы ставим @ в начало строк с именами файлов в упражнении «Возьми в руку карандаш»?

О: Потому, что в противном случае символы \S в «C:\SYP» будут интерпретированы как недопустимая служебная последовательность, и программа выдаст исключение. Когда вы включаете в свою программу строковый литерал, компилятор преобразует служебные последовательности (такие, как \n и \r) в специальные символы. Имена файлов Windows могут содержать символы \, но в строках C# этот символ обычно начинает служебную последовательность. Если поставить @ перед строкой, тем самым вы приказаете C# не интерпретировать служебные последовательности. Кроме того, @ перед строкой призывает C# включать разрывы строк в текст, так что если вы нажмете Enter в процессе ввода, этот разрыв строки будет включен в вывод.

В: Для чего нужны служебные последовательности?

О: Служебная последовательность позволяет включать специальные символы в строки. Например, \n представляет символ новой строки, \t — табуляцию, а \r — возврат курсора (или только его половину в Windows — в текстовых файлах Windows строка должна завершаться комбинацией \r\n; в macOS и Linux строки завершаются \n). Если вам понадобится включить кавычки в строку, используйте последовательность \" — она не будет интерпретирована как завершение строки. Если же вы захотите включить символ \ в строку, но так, чтобы C# не интерпретировал его как начало служебной последовательности, используйте удвоенную обратную косую черту: \\.

Мы дали вам возможность обнаружить ошибку самостоятельно. А теперь посмотрите, как исправили ее мы.



II о следу

Шерлок Холмс однажды сказал: «Факты! Факты! Факты! Когда под рукой нет глины, из чего лепить кирпичи?» Начнем с места преступления: кода, который не работает. Мы извлечем из него всю возможную информацию и найдем улики.

А сколько из этих улик нашли вы?

- * Мы создаем экземпляр StreamWriter, который передает данные новому экземпляру MemoryStream.
- * StreamWriter записывает строку текста в MemoryStream.
- * Содержимое MemoryStream копируется в массив и преобразуется в строку.
- * Все это происходит в блоке using, так что потоки определенно закрываются.

Если вы обнаружили все улики, поздравляем — ваши навыки расследования в коде, бесспорно, совершенствуются! Но как и в каждой настоящей тайне, всегда остается последняя улика — какой-то факт, который вы узнали ранее и который играет ключевую роль в раскрытии преступления и поиске настоящего виновника.

Мы использовали блок using, поэтому мы точно знаем, что потоки закрываются. Но когда они закрываются? И тут обнаруживается ключ ко всей тайне, первостепенная улика, о которой мы узнали еще до преступления: **некоторые потоки не записывают все свои данные, пока не будут закрыты.**

Объекты StreamWriter и MemoryStream объявляются в одном блоке using, поэтому оба метода Dispose вызываются *после выполнения последней строки блока*. Что это значит? То, что метод MemoryStream.ToArray вызывается **до** закрытия StreamWriter.

Проблему можно решить добавлением **вложенного** блока using, чтобы сначала объект StreamWriter закрывался, а потом вызывался метод ToArray:

```
using System;
using System.IO;
using System.Text;
```

```
class Program
{
    static void Main(string[] args)
    {
        using (var ms = new MemoryStream())
        {
            using (var sw = new StreamWriter(ms))
            {
                sw.WriteLine("The value is {0:0.00}", 123.45678);
            }
            Console.WriteLine(Encoding.UTF8.GetString(ms.ToArray()));
        }
    }
}
```

Объект MemoryStream объявляется во внешнем блоке using, так что он может оставаться открытым даже после закрытия StreamWriter.

Внутренний блок using гарантирует, что объект StreamWriter будет закрыт после вызова метода MemoryStream.ToArray.

Объекты потоков часто хранят в памяти **буферизованные** данные, ожидающие записи на диск. Когда поток записывает все оставшиеся данные, это называется **сбросом на диск**. Чтобы сбросить на диск все буферизованные данные без закрытия потока, **вызовите метод Flush**.



Упражнение

В главе 8 мы создали класс Deck для хранения последовательности объектов Card, с методами для сброса колоды до 52 карт по порядку, тасования карт для размещения их в случайном порядке, а также сортировки карт для возвращения их к исходному порядку. Теперь мы добавим метод для записи карт в файл, а также конструктор, позволяющий инициализировать новую колоду картами, прочитанными из файла.

Посмотрите классы Deck и Card, написанные в главе 8

Мы создали класс Deck посредством расширения обобщенной коллекции объектов Card. Это позволило нам использовать некоторые полезные компоненты, унаследованные Deck от Collection<Card>.

- ★ Свойство Count возвращает количество карт в колоде.
- ★ Метод Add добавляет карту на верх колоды.
- ★ Метод RemoveAt удаляет карту с заданным индексом из колоды.
- ★ Метод Clear удаляет все карты из колоды.

Эта реализация станет хорошей отправной точкой для добавления метода Reset, который очищает колоду, а потом добавляет 52 карты по порядку (от туза до короля в каждой масти). Метод Deal удаляет карту с верха колоды и возвращает ее, метод Shuffle переставляет карты в случайном порядке, а метод Sort снова выстраивает их по порядку.

Добавьте метод для записи всех карт из колоды в файл

Класс Card содержит свойство Name, которое возвращает строку вида «Three of Clubs» или «Ace of Hearts». Добавьте метод с именем WriteCards, который получает строку с именем файла в параметре и записывает имя каждой карты в файл, таким образом, если сбросить колоду, а потом вызвать WriteCards, в файл будут записаны 52 строки, по одной для каждой карты.

Добавьте перегруженный конструктор Deck, который читает данные колоды карт из файла

Добавьте второй конструктор в класс Deck. Ниже приведено описание того, что он должен делать:

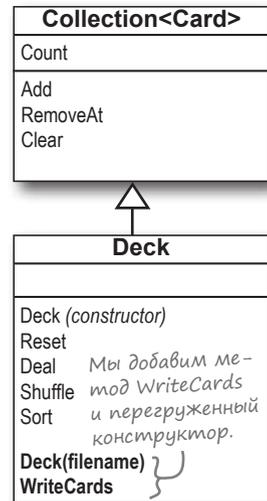
```
public Deck(string filename)
{
    // Создать новый объект StreamReader для чтения файла.
    // Для каждой строки в файле выполнить следующие четыре операции:
    // Использовать метод String.Split: var cardParts = nextCard.Split(new char[] { ' ' });
    // Использовать выражение switch для получения масти каждой карты: var suit = cardParts[2] switch {
    // Использовать выражение switch для получения номинала каждой карты: var value = cardParts[0] switch {
    // Добавить карту в колоду.
}
```

В главе 9 вы узнали, что выражение switch должно быть исчерпывающим, поэтому добавьте вариант по умолчанию, который **выдает исключение InvalidDataException** при обнаружении неопознанной масти или номинала, — тем самым будет гарантирована правильность всех карт.

Следующий метод Main может использоваться для тестирования приложения. Он создает колоду из 10 случайных карт, записывает ее в файл, а затем читает этот файл во вторую колоду и выводит описание каждой карты на консоль.

```
static void Main(string[] args) {
    var filename = "deckofcards.txt";
    Deck deck = new Deck();
    deck.Shuffle();
    for (int i = deck.Count - 1; i > 10; i--)
        deck.RemoveAt(i);
    deck.WriteCards(filename);

    Deck cardsToRead = new Deck(filename);
    foreach (var card in cardsToRead)
        Console.WriteLine(card.Name);
}
```



Метод String.Split позволяет задать массив символов-разделителей (в данном случае пробел), использует их для разбиения строки на части и возвращает массив, содержащий все части.



Упражнение
Решение

Ниже приведены два метода, добавленные в класс Deck. Метод WriteCards использует StreamWriter для записи карт в файл, а перегруженный конструктор Deck использует StreamReader для чтения карт из файла. Так как вы работаете с StreamWriter и StreamReader, не забудьте добавить using System.IO; в начало файла.

```

public void WriteCards(string filename)
{
    using (var writer = new StreamWriter(filename))
    {
        for (int i = 0; i < Count; i++)
        {
            writer.WriteLine(this[i].Name);
        }
    }
}

public Deck(string filename)
{
    using (var reader = new StreamReader(filename))
    {
        while (!reader.EndOfStream)
        {
            var nextCard = reader.ReadLine();
            var cardParts = nextCard.Split(new char[] { ' ' });
            var value = cardParts[0] switch
            {
                "Ace" => Values.Ace,
                "Two" => Values.Two,
                "Three" => Values.Three,
                "Four" => Values.Four,
                "Five" => Values.Five,
                "Six" => Values.Six,
                "Seven" => Values.Seven,
                "Eight" => Values.Eight,
                "Nine" => Values.Nine,
                "Ten" => Values.Ten,
                "Jack" => Values.Jack,
                "Queen" => Values.Queen,
                "King" => Values.King,
                _ => throw new InvalidDataException($"Unrecognized card value: {cardParts[0]}");
            };
            var suit = cardParts[2] switch
            {
                "Spades" => Suits.Spades,
                "Clubs" => Suits.Clubs,
                "Hearts" => Suits.Hearts,
                "Diamonds" => Suits.Diamonds,
                _ => throw new InvalidDataException($"Unrecognized card suit: {cardParts[2]}");
            };
            Add(new Card(value, suit));
        }
    }
}

```

Эта строка приказывает C# разбить строку nextCard по пробелам. Таким образом, строка «Six of Diamonds» будет преобразована в массив {"Six", "of", "Diamonds"}

Выражение switch проверяет, совпадает ли первое слово в строке с заданными значениями. Если совпадение будет найдено, переменной «value» присваивается соответствующий элемент из перечисления Value.

Секция по умолчанию выражения switch выдает исключение, если файл содержит недействительную карту.

То же самое делается с третьим словом в строке, которое преобразуется в элемент перечисления Suit.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Каждый раз, когда программа читает данные из файла или записывает данные в файл, вы используете объект **Stream**. Класс **Stream** является абстрактным и имеет ряд специализированных subclasses.
- Класс **FileStream** предназначен для чтения и записи данных в файлы. Класс **MemoryStream** читает и записывает данные в память.
- Для записи данных в поток используется **метод Write**, а для чтения данных — **метод Read**.
- Класс **StreamWriter** предоставляет простые средства для записи данных в файл. **StreamWriter** автоматически создает объект **FileStream** и управляет им.
- Класс **StreamReader** читает символы из потока, но сам потоком не является. Он создает поток за вас, читает из него данные и закрывает его при вызове метода **Close**.
- Методы **Write** и **WriteLine** классов **StreamWriter** и **Console** поддерживают **составное форматирование**: они получают форматную строку с заполнителями вида **{0}**, **{1}**, **{2}** и т. д., с поддержкой расширенного форматирования вида **{1:0.00}** и **{3:c}**.
- **Path.DirectorySeparatorChar** — поле, доступное только для чтения, в котором хранится разделитель пути для текущей операционной системы: **** для **Windows** или **/** для **macOS** и **Linux**.
- **Метод Environment.GetFolderPath** возвращает путь кодовой из специальных папок для текущего пользователя (например, к папке **Documents** пользователя в **Windows** или домашнему каталогу пользователя в **macOS**).
- **Класс File** содержит статические методы, включая **Exists** (проверяет, существует ли файл), **OpenRead** и **OpenWrite** (получают потоки для чтения и записи в файл), а также **AppendAllText** (записывает текст в файл одной командой).
- **Класс Directory** содержит статические методы, включая **CreateDirectory** (создает папки), **GetFiles** (получает список файлов) и **Delete** (удаляет папки).
- **Класс FileInfo** похож на класс **File**, но вместо вызова статических методов создается экземпляр этого класса.
- Не забывайте **всегда закрывать поток** после завершения работы с ним. Некоторые потоки записывают остаток своих данных на диск только при закрытии или при вызове метода **Flush**.
- **Интерфейс IDisposable** обеспечивает гарантированное закрытие объектов. Он содержит один метод **Dispose**, который предоставляет механизм освобождения неуправляемых ресурсов.
- Используйте **команду using** для создания экземпляра класса, реализующего **IDisposable**. За командой **using** следует блок кода; объект, созданный в команде **using**, уничтожается в конце блока.
- Используйте несколько команд **using** подряд для объявления объектов, которые должны быть уничтожены в конце одного блока.

В Windows и macOS используются разные завершители строк

Если вы работаете в **Windows**, откройте **Блокнот**. Если вы работаете в **macOS**, откройте **TextEdit**. Создайте файл из двух строк: первая строка содержит символы **L1**, а вторая — символы **L2**.

В **Windows** файл содержит следующие шесть байтов: **76 49 13 10 76 50**.

В **macOS** файл содержит следующие пять байтов: **76 49 10 76 50**.

Видите различия? Первая и вторая строки кодируются одними и теми же байтами: **L** — **76**, **1** — **49**, а **2** — **50**. Разрыв строки кодируется иначе: в **Windows** он кодируется двумя байтами **13** и **10**, а в **macOS** — одним байтом **10**. В этом проявляются различия между завершителями строк в стиле **Windows** и в стиле **Unix** (**macOS** является разновидностью **Unix**). Если вам нужно написать код, который работает в разных операционных системах и записывает файлы с завершителями строк, вы можете воспользоваться статическим свойством **Environment.NewLine**, которое возвращает **"\r\n"** в **Windows** или **"\n"** в **macOS** или **Linux**.

Столько кода для чтения всего одной карты?
Не многовато ли? А что делать с объектами с **множеством значений и полей**? Неужели мне придется писать оператор switch для каждого?



Существует более простой способ сохранения объектов в файлах — сериализация (serialization).

Сериализацией называется запись полного состояния объекта в файл или строку. Обратный процесс — чтение состояния объекта из файла или строки — называется **десериализацией**. Таким образом, вместо скрупулезной записи в файл каждого поля и значения можно сохранить объект посредством сериализации его в поток. Сериализация приводит к **деструктуризации** объекта, т. е. превращению его в набор байтов. С другой стороны, при десериализации объекта данные читаются из файла и используются для создания объекта.

Ладно, чтобы не было недоразумений: существует метод `Enum.Parse`, который преобразует строку «Spades» в элемент перечисления `Suits.Spades`. У него даже есть парный метод `Enum.TryParse`, работающий как метод `int.TryParse`, который встречался вам в книге. Впрочем, в рассматриваемом случае лучше использовать сериализацию, в чем вы скоро убедитесь...

Что происходит с объектом при сериализации?

Трансформация объекта при копировании его из кучи и сохранении в файле выглядит таинственно, но на самом деле там все очень просто.

1 Объект в куче



Созданный вами экземпляр объекта обладает **состоянием**. При этом объект «знает» только то, чем экземпляр одного класса отличается от другого экземпляра этого же класса.

2 Сериализованный объект



При сериализации объекта в С# **полностью сохраняется состояние объекта**, что позволяет воссоздать идентичный экземпляр (объект) в куче.

Этот объект имеет два байтовых поля, width и height.



Экземпляры переменных width и height были сохранены в файле file.dat вместе с дополнительной информацией, необходимой для дальнейшего восстановления объекта (например, информацией о типе самого объекта и каждого из его полей).

Объект снова в куче



3 А потом...

Позднее — может быть, через много дней и в другой программе — вы можете вернуться к файлу и провести **десериализацию** объекта. Исходный класс будет восстановлен из файла в точно таком состоянии, в каком он был сериализован, со всеми полями и значениями.

Но что именно СЧИТАЕТСЯ состоянием объекта? Что необходимо сохранить?

Вы уже знаете, что объект хранит информацию в полях и свойствах. Следовательно, при сериализации нужно сохранить в файле каждое из этих значений.

С более сложными объектами сериализация перестает быть тривиальным делом. Переменные разных типов — `char`, `int`, `double` и др. — можно записать в файл без изменений. А как быть, если в составе объекта присутствует переменная-экземпляр, содержащая ссылку на объект? А если пять переменных-экземпляров, содержащих ссылки на объект? А если эти ссылки, в свою очередь, указывают на другие ссылки?

Подумайте об этом. Какая часть объекта является потенциально уникальной? Что именно необходимо восстановить, чтобы получить объект, идентичный сохраненному? Так или иначе, но в файл нужно записать все, что хранится в куче.

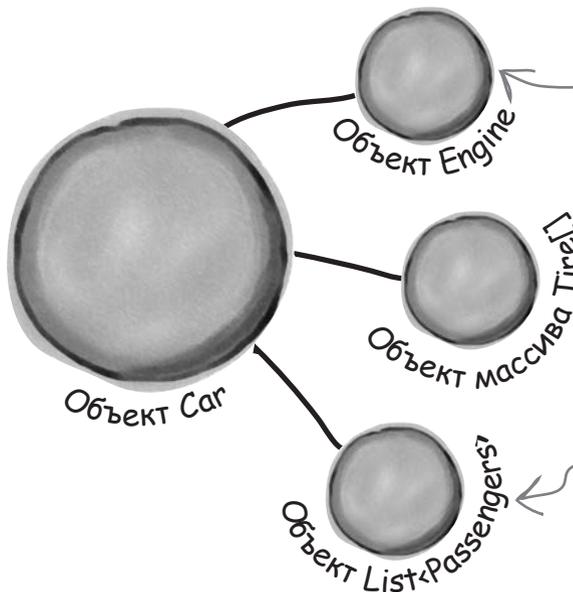
«Напряги мозги» — версия «Мозгового штурма» для самых умных. Потратьте несколько минут и хорошенько поразмыслите над задачей.



НАПРЯГИ МОЗГИ

Каким образом следует сохранить объект `Car`, чтобы потом его можно было восстановить в исходном состоянии? Предположим, в автомобиле едут три пассажира, он имеет трехлитровый двигатель и всесезонные шины... разве вся эта информация не является частью состояния объекта `Car`? И что с ней делать?

Объект `Car` содержит ссылку на объект `Engine` (Двигатель), массив объектов `Tire` (Шина) и перечисленные `List<>` с объектами `Passenger` (Пассажир). Это составные части его состояния. Что должно произойти с ними при сериализации?



Объект `Engine` является приватным. Нужно ли его сохранять наряду с другими?

Каждый из объектов `Passenger` содержит ссылки на другие объекты. Следует ли их сохранять?

При сериализации объекта также сериализуются все объекты, на которые он ссылается...

...а также все объекты, на которые ссылаются они, и все объекты, на которые ссылаются *эти объекты*, и т. д. Не беспокойтесь — это звучит сложно, но все происходит автоматически. С# начинает с объекта, который вы хотите сериализовать, и проверяет его поля в поисках связанных объектов. Найдя такой объект, он повторяет аналогичную операцию. В результате в файл будут записаны все объекты наряду со всей информацией, необходимой С# для полного восстановления при десериализации объекта.

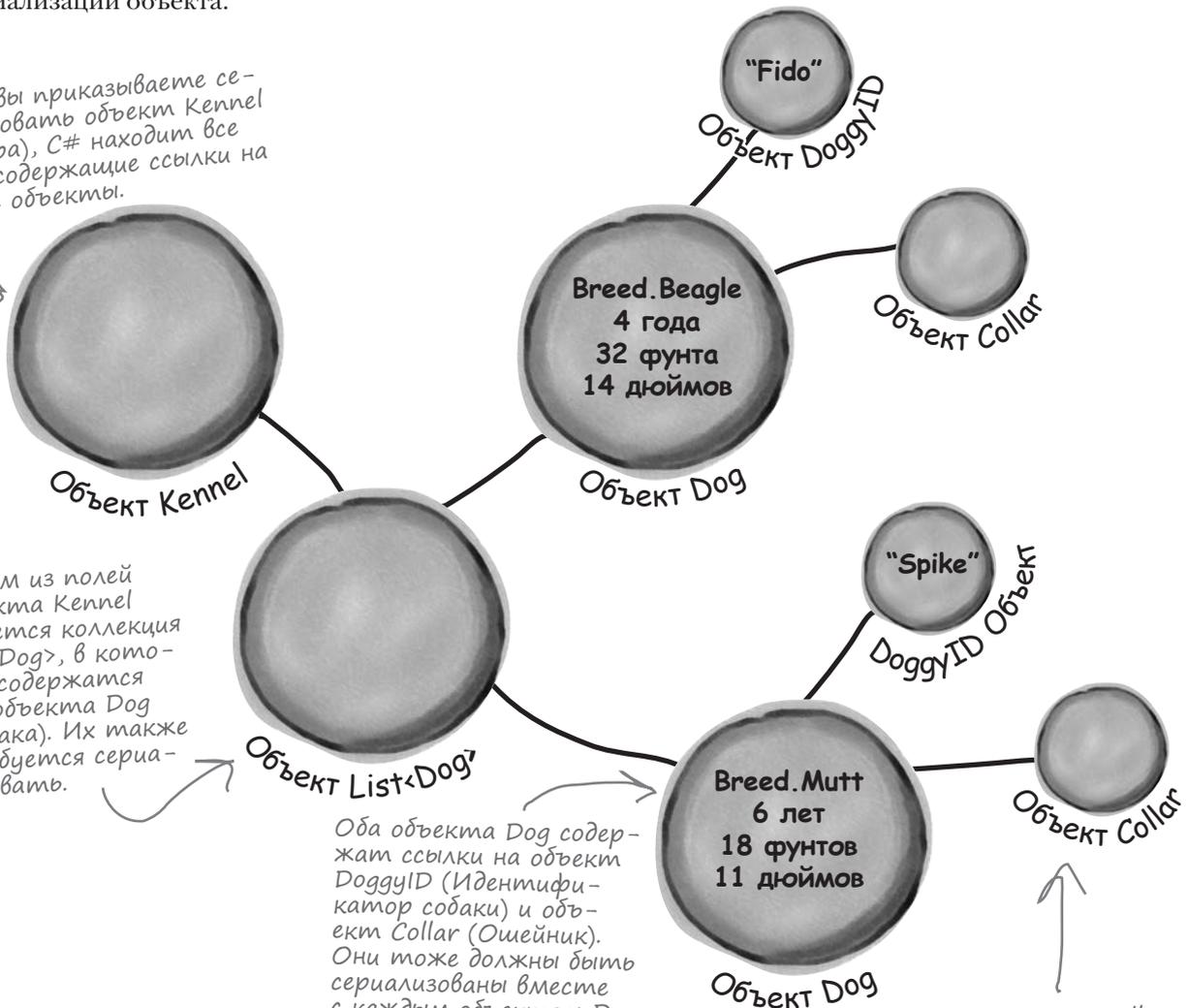
Группа объектов, связанных друг с другом по ссылкам, иногда называется «графом».

Когда вы приказываете сериализовать объект Kennel (Конура), С# находит все поля, содержащие ссылки на другие объекты.

Одним из полей объекта Kennel является коллекция List<Dog>, в которой содержатся два объекта Dog (Собака). Их также требуется сериализовать.

Оба объекта Dog содержат ссылки на объект DoggyID (Идентификатор собаки) и объект Collar (Ошейник). Они тоже должны быть сериализованы вместе с каждым объектом Dog.

Объекты DoggyID и Collar завершают цепочку ссылок — они не ссылаются на другие объекты.



Использование JsonSerializer для сериализации объектов

Ваши возможности не ограничены чтением и записью строк текста в файлы. Механизм **сериализации JSON** позволяет программам **копировать целые объекты** в строки (которые затем можно записать в файлы!), читать их из файла... и все это в нескольких строках кода! Посмотрим, как это работает. Начнем с **создания консольного приложения**.

← (Делайте это!)

1 Напишите несколько классов для графа объектов.

Добавьте перечисление HairColor и классы Guy, Outfit и HairStyle для нового консольного приложения:

```
class Guy {
    public string Name { get; set; }
    public HairStyle Hair { get; set; }
    public Outfit Clothes { get; set; }
    public override string ToString() => $"{Name} with {Hair} wearing {Clothes}";
}

class Outfit {
    public string Top { get; set; }
    public string Bottom { get; set; }
    public override string ToString() => $"{Top} and {Bottom}";
}

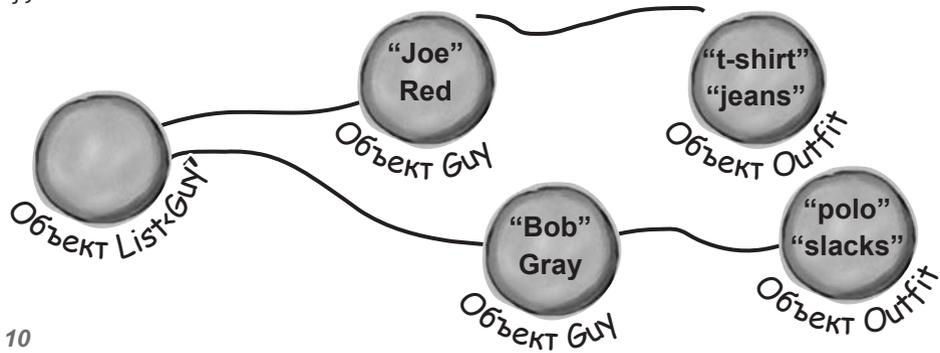
enum HairColor {
    Auburn, Black, Blonde, Blue, Brown, Gray, Platinum, Purple, Red, White
}

class HairStyle {
    public HairColor Color { get; set; }
    public float Length { get; set; }
    public override string ToString() => $"{Length:0.0} inch {Color} hair";
}
```

2 Создайте граф объектов для сериализации.

Теперь создайте небольшой граф объектов для сериализации: новый список List<Guy>, содержащий указатели на пару объектов Guy. Добавьте в метод Main следующий код, в котором инициализатор коллекции и инициализаторы объектов используются для построения графа объектов:

```
static void Main(string[] args) {
    var guys = new List<Guy>() {
        new Guy() { Name = "Bob", Clothes = new Outfit() { Top = "t-shirt", Bottom = "jeans" },
            Hair = new HairStyle() { Color = HairColor.Red, Length = 3.5f }
        },
        new Guy() { Name = "Joe", Clothes = new Outfit() { Top = "polo", Bottom = "slacks" },
            Hair = new HairStyle() { Color = HairColor.Gray, Length = 2.7f }
        }
    };
}
```



3 Используйте JsonSerializer для сериализации объектов в строку.

Начните с добавления директивы `using` в начало файла с кодом:

```
using System.Text.Json;
```

После этого **весь граф сериализуется** одной строкой кода:

```
var jsonString = JsonSerializer.Serialize(guys);
Console.WriteLine(jsonString);
```

Запустите приложение и внимательно рассмотрите то, что оно выводит на консоль:

```
[{"Name": "Bob", "Hair": {"Color": 8, "Length": 3.5}, "Clothes": {"Top": "t-shirt", "Bottom": "jeans"}}, {"Name": "Joe", "Hair": {"Color": 5, "Length": 2.7}, "Clothes": {"Top": "polo", "Bottom": "slacks"}}]
```

Так выглядит граф объектов, **сериализованный в JSON** — *формат обмена данными, удобный для восприятия человеком*. Иначе говоря, сложные объекты сохраняются в строковом виде, понятном для человека. Так как этот формат может читаться человеком, мы видим в нем все части графа: имена и предметы одежды («Bob», «t-shirt») кодируются в строках, а перечисления кодируются как их целочисленные значения.

4 Используйте JsonSerializer для десериализации JSON в граф объектов.

Итак, у нас есть строка с графом объектов, сериализованным в JSON; эту строку можно **десериализовать** (т. е. использовать ее для создания новых объектов). `JsonSerializer` также позволяет сделать это в одной строке кода. Добавьте следующий код в метод `Main`:

```
var copyOfGuys = JsonSerializer.Deserialize<List<Guy>>(jsonString);
foreach (var guy in copyOfGuys)
    Console.WriteLine("I deserialized this guy: {0}", guy);
```

Снова запустите свое приложение. Оно десериализует данные из строки JSON и выводит их на консоль:

```
I deserialized this guy: Bob with 3.5 inch Red hair wearing t-shirt and jeans
I deserialized this guy: Joe with 2.7 inch Gray hair wearing polo and slacks
```

JSON под увеличительным стеклом



Разберемся подробнее, как работает JSON. Вернитесь к приложению с графом объектов Guy и замените строку, которая сериализует граф в строку, следующим фрагментом:

```
var options = new JsonSerializerOptions() { WriteIndented = true };
var jsonString = JsonSerializer.Serialize(guys, options);
```

Этот код вызывает перегруженный метод `Json.Serializer.Serialize`, который получает объект `JsonSerializerOptions` с параметрами сериализации. В данном случае включается запись JSON в виде текста с отступами — другими словами, добавляются разрывы строки и пробелы, которые упрощают чтение JSON человеком.

Снова запустите программу. Вывод должен выглядеть так: 

А теперь разберемся, что же мы видим:

- ★ JSON начинается и заканчивается квадратными скобками `[]`. Так в JSON сериализуются списки. Список чисел выглядит так: `[1, 2, 3, 4]`.
- ★ Этот конкретный блок JSON представляет собой список с двумя объектами. Каждый объект начинается и заканчивается фигурными скобками `{}`; присмотревшись к JSON, вы увидите, что во второй строке располагается открывающая фигурная скобка `{`, в предпоследней — закрывающая `}`, а в середине блока — строка `,` за которой следует строка `{`. Так в JSON представляются два объекта — в данном случае два объекта Guy.
- ★ Каждый объект содержит набор разделенных двоеточиями ключей и значений, соответствующих свойствам сериализованного объекта. Например, `"Name": "Joe"` представляет свойство `Name` второго объекта Guy.
- ★ Свойство `Guy.Clothes` — это ссылка, указывающая на объект `Outfit`. Оно представляется вложенным объектом со значениями `Top` и `Bottom`.

```
[
  {
    "Name": "Bob",
    "Hair": {
      "Color": 8,
      "Length": 3.5
    },
    "Clothes": {
      "Top": "t-shirt",
      "Bottom": "jeans"
    }
  },
  {
    "Name": "Joe",
    "Hair": {
      "Color": 5,
      "Length": 2.7
    },
    "Clothes": {
      "Top": "polo",
      "Bottom": "slacks"
    }
  }
]
```

При использовании `JsonSerializer` для сериализации графа объектов в JSON генерируется (относительно) удобочитаемый текст, представляющий данные каждого объекта.

JSON включает только данные, но не конкретные типы C#

Просматривая данные JSON, вы видите понятные для человека версии данных ваших объектов: строки «Bob» и «slacks», числа 8 и 3.5 и даже списки и вложенные объекты. А чего мы *не видим* при просмотре данных JSON? В JSON **отсутствуют имена типов**. Загляните в файл JSON — вы не увидите в нем таких имен классов, как Guy, Outfit, HairColor или Hairstyle, и даже имен базовых типов int, string или double. Дело в том, что JSON содержит только данные, а JsonSerializer делает все возможное для десериализации данных в найденные свойства.

Чтобы убедиться в этом, добавьте в проект новый класс:

```
class Dude
{
    public string Name { get; set; }
    public Hairstyle Hair { get; set; }
}
```

Данные десериализуются из списка List объектов Guy в стек объектов Dude.

Добавьте следующий код в конец метода Main:

```
var dudes = JsonSerializer.Deserialize<Stack<Dude>>(jsonString);
while (dudes.Count > 0)
{
    var dude = dudes.Pop();
    Console.WriteLine($"Next dude: {dude.Name} with {dude.Hair} hair");
}
```

Снова запустите свой код. Так как JSON содержит только список объектов, JsonSerializer.Deserialize спокойно разместит их в стеке (или очереди, или массиве, или любой другой разновидности коллекций). Так как Dude содержит открытые свойства Name и Hair, соответствующие данным, десериализация заполняет все данные, которые может заполнить. Результат выглядит так:

```
Next dude: Joe with 2.7 inch Gray hair hair
Next dude: Bob with 3.5 inch Red hair hair
```

Возьми в руку карандаш



Воспользуемся JsonSerializer для анализа преобразования строк в JSON. Добавьте следующий код в консольное приложение и запишите, что каждая строка кода выведет на консоль. Последняя строка сериализует эмодзи со слонем.

```
Console.WriteLine(JsonSerializer.Serialize(3)); .....
Console.WriteLine(JsonSerializer.Serialize((long)-3)); .....
Console.WriteLine(JsonSerializer.Serialize((byte)0)); .....
Console.WriteLine(JsonSerializer.Serialize(float.MaxValue)); .....
Console.WriteLine(JsonSerializer.Serialize(float.MinValue)); .....
Console.WriteLine(JsonSerializer.Serialize(true)); .....
Console.WriteLine(JsonSerializer.Serialize("Elephant")); .....
Console.WriteLine(JsonSerializer.Serialize("Elephant".ToCharArray())); .....
Console.WriteLine(JsonSerializer.Serialize("🐘")); .....
```

Панель эмодзи использовалась в главе 1 для ввода эмодзи.

И последнее! Мы продемонстрировали базовый процесс реализации на примере JsonSerializer. Осталась еще пара подробностей, о которых вам необходимо знать.



Будьте
осторожны!

Класс JsonSerializer сериализует только открытые свойства (не поля), и ему необходим конструктор без параметров.

Помните класс SwordDamage из главы 5? Его свойство Damage имеет приватный set-метод:

```
public int Damage { get; private set; }
```

Также он имеет конструктор, получающий параметр int:

```
public SwordDamage(int startingRoll)
```

JsonSerializer использует set-методы объекта при десериализации, и если объект имеет приватный set-метод, присвоить значение не удастся.

JsonSerializer сериализует объекты SwordDamage без малейших проблем. При попытке десериализации JsonSerializer выдаст исключение — по крайней мере в приведенном примере. Если вы захотите сериализовать объекты, сохраняющие свое состояние в полях или приватных свойствах либо использующие конструкторы с параметрами, придется создать преобразователь. Эта тема более подробно рассматривается в документации по сериализации в .NET Core: <http://docs.microsoft.com/en-us/dotnet/standard/serialization>.

КЛЮЧЕВЫЕ МОМЕНТЫ

- **Сериализацией** называется запись полного состояния объекта в файл или строку. Обратный процесс — чтение состояния объекта из файла или строки — называется **десериализацией**.
- Группа объектов, связанных друг с другом по ссылке, иногда называется **графом**.
- При сериализации объекта сериализуется **весь граф** объектов, на которые он ссылается, чтобы все объекты могли быть сериализованы вместе.
- **Класс JsonSerializer** содержит статический метод `Serialize`, сериализующий граф объектов в JSON, и статический метод `Deserialize`, создающий экземпляр графа объектов на основании сериализованных данных JSON.
- Данные JSON могут читаться человеком (большая часть). Значения сериализуются в виде простого текста; строки заключаются в "кавычки", а другие литералы (например, числа и логические значения) кодируются без кавычек.
- В JSON **массивы** значений заключаются в квадратные скобки `[]`.
- В JSON **объекты** заключаются в фигурные скобки `{}`, а компоненты и их значения представляются в виде пар «ключ/значение», разделенных двоеточием.
- JSON **не сохраняет имена конкретных типов** (`string`, `int`, имена отдельных классов). Вместо этого «умные» классы (такие, как `JsonSerializer`) прилагают максимум усилий к тому, чтобы сопоставить данные с типом, в который они десериализуются.

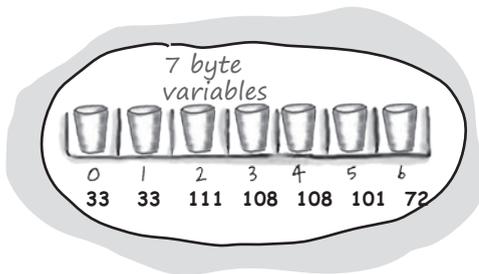
Следующий шаг: углубленный анализ данных

Ранее вы написали большой объем кода, в котором использовались типы значений: `int`, `bool`, `double`, а также создавались объекты, хранящие данные в полях. Пришло время разобраться в происходящем на более низком уровне. В оставшейся части этой главы вы увидите, какие реальные байты используются в C# и .NET для их представления.

Вот что нам предстоит сделать.



Мы исследуем, как строки C# кодируются в Юникоде — .NET использует Юникод для хранения символов и текста.



Мы запишем данные в двоичной форме, а затем прочитаем их и посмотрим, какие данные были записаны.

```
0000: 45 6c 65 6d 65 6e 74 61 Elementa
0005: 72 79 2c 20 6d 79 20 64 ry, my d
0010: 65 61 72 20 57 61 74 73 ear Wats
0015: 6f 6e 21 on!
```

Мы построим программу вывода шестнадцатеричного дампа, которая поможет в анализе битов и байтов в файлах.



Доступность

Разработка игр... и не только

Как бы вы играли в свою любимую игру, если бы вы плохо видели? Что, если двигательные нарушения или ограниченные двигательные возможности усложняют использование контроллера? Цель доступности — гарантировать, что построенные вами программы спроектированы таким образом, чтобы они могли использоваться людьми с ограниченными возможностями и нарушениями. Вы стремитесь к тому, чтобы сделать вашу игру доступной для всех, независимо от любых проблем со здоровьем.

Существует ряд подходов к тестированию игр, которые стоит продумать в начале проектирования и построения игр:

- **«Погодите, что? Есть слепые игроки?»** Да! А вы думали, что «видео» в видеоиграх означает, что эти игры недоступны для людей с нарушениями зрения? Потратьте несколько минут на поиск в YouTube по строке «blind games» и посмотрите ролики, показывающие, как люди с нарушениями зрения (включая полностью слепых) демонстрируют серьезные игровые навыки.
- Чтобы вы как разработчик могли сделать свою игру доступной, очень важно **понять игроков с нарушениями**. Что вы узнали из просмотра этих роликов?
- Наблюдая за слепыми игроками, мы узнали, что они используют **игровые звуки**, чтобы ориентироваться в происходящем. В файтинге разные приемы могут сопровождаться разными звуками. В платформенной игре враги,двигающиеся к игроку, могут издавать характерные звуки. Вы также узнали, что хотя некоторые игры предоставляют адекватные звуковые ориентиры, другие этого не делают — и лишь немногие игры спроектированы для того, чтобы полностью слепым игрокам хватало звуковой информации для игры.
- С другой стороны, у многих игроков встречаются **нарушения слуха**, поэтому важно не ограничиваться только звуковыми ориентирами. Попробуйте поиграть в вашу любимую игру с отключенным звуком. Не мешает ли это передаче важной информации игроку? Существуют ли визуальные ориентиры, сопровождающие аудио? Субтитры для диалогов? Убедитесь в том, что в вашу игру можно играть без звука.
- Один из двенадцати мужчин (и одна из двухсот женщин) страдают той или иной формой **дальтонизма**, т. е. неспособностью различать цвета (кстати, к их числу принадлежит и один из авторов этой книги!). Многие высокобюджетные игры включают режим для дальтоников, в котором выполняется нетривиальная регулировка цветов. Для улучшения доступности игр для дальтоников можно использовать цвета, заметно **контрастирующие** друг с другом.
- Многие геймеры страдают от **нарушений моторики**, начиная с травмы от повторяющихся деформаций до паралича. Игроки, которые не могут работать с традиционными устройствами ввода (такими, как мышь или клавиатура), могут пользоваться **вспомогательным оборудованием** (например, устройством для отслеживания направления взгляда или модифицированным геймпадом). Если вы хотите привлечь таких игроков к игре, реализуйте механизм привязки клавиатуры, чтобы игроки могли создать профиль, в котором специальные клавиши соответствуют разным элементам управления игры.
- Возможно, вы заметили, что многие игры начинаются с предупреждения о возможных приступах. Дело в том, что многие игроки с **эпилепсией** чувствительны к свету, и мерцание или вспышки могут стать причиной приступа. Хотя предупреждения важны, но можно пойти дальше. Для разработчика важно приложить сознательные усилия к тому, чтобы понять суть проблемы и избегать световых вспышек, мерцания и других визуальных эффектов, которые с наибольшей вероятностью приводят к приступам. Прочитайте редакционную статью рецензента видеоигр Кэти Вайс (Cathy Vice) о ее личном опыте **игрока с эпилепсией** (<https://indiegamerchick.com/2013/08/06/the-epilepsy-thing>).

Поддержка доступности часто добавляется «задним числом», но ваши игры — как, впрочем, и любые другие программы! — будут более качественными, если вы будете учитывать этот аспект с самого начала.

Строки C# кодируются в Юникоде

Вы использовали строки с того момента, когда ввели текст "Hello, world!" в начале главы 1. Так как строки интуитивно понятны, мы не анализировали их и не стали подробно описывать, как они работают. Но спросите себя... *что же представляет собой строка?*

Строка C# представляет собой **набор символов, доступный только для чтения**. Таким образом, если вы посмотрите на строку «Elephant», хранящуюся в памяти, вы увидите символы 'E', 'l', 'e', 'p', 'h', 'a', 'n' и 't'. А теперь задайтесь вопросом... *что же представляет собой символ?*

Символ представляет собой знак, представленный в Юникоде — отраслевом стандарте **кодирования** символов или преобразования их в байты, позволяющем хранить их в памяти, передавать по сети, включать в документы и выполнять практически любые возможные операции, — и вы всегда гарантированно получите правильные символы.

Этот аспект особенно важен, если задуматься над тем, сколько символов существует в разных алфавитах. Стандарт Юникод поддерживает свыше 150 **скриптов** (наборов символов для конкретных языков), включая не только латинский (с 26 буквами английского алфавита и такими модификациями, как é и ç), но и скрипты для многих языков из всех стран. Список поддерживаемых скриптов постоянно растет, так как консорциум Юникода каждый год расширяет его (текущий список доступен по адресу <http://www.unicode.org/standard/supported.html>).

В Юникоде поддерживается еще один важный набор символов: **эмодзи**. Все эмодзи, от подмигивающего смайлика (☺) до неизменно популярной кучи дерьма (💩), являются символами Юникода.



В игре с поиском пар из главы 1 символы Юникода обрабатывались точно так же, как и любые другие символы Юникода.



Будьте осторожны!

Символы Юникода могут препятствовать работе вспомогательных технологий

Доступность чрезвычайно важна. Мы считаем, что очень важно поднять тему доступности, потому что, как и во многих темах из разделов «Разработка игр... и не только», из нее можно извлечь урок, относящийся к разработке любых программ. Вероятно, вы видели в социальных сетях сообщения с эмодзи или другими «необычными» символами — курсивными, жирными или перевернутыми. На некоторых платформах для этого используются символы Юникода, и в этом варианте они могут создать серьезные проблемы для вспомогательных технологий.

Возьмем следующее сообщение в социальной сети: I'm 🤝 using 🤝 hand 🤝 claps 🤝 to 🤝 emphasize 🤝 points

На экране такое сообщение выглядит нормально. Однако Экранный диктор Windows или VoiceOver в macOS может прочитать сообщение, заменяя пробелы словами «аплодирующие руки»).

Возможно, вы видели «шрифты» следующего вида:

this is a MeSᵀαGE in a really рIєм FONT

Экранный диктор либо прочитает описание m как «малое m Fraktur полужирный», а для остальных символов — что-то вроде «буква n из скрипта», либо вообще проигнорирует их. Все это может оказаться совершенно бесполезным для читателей, использующих систему чтения Брайля. Выходит, возможности вспомогательных технологий ограничены? Вовсе нет — они справляются со своим делом. Все это реальные имена символов Юникода, а вспомогательные технологии точно описывают текст. Помните об этих примерах, читая следующую часть этой главы, — они помогут вам лучше понять, как работает Юникод.



Когда я сериализовал эмодзи со слоном в JSON, получилось что-то вроде `"\uD83D\uDC18"` — наверняка это как-то связано с Юникодом?

Каждому символу Юникода, включая эмодзи, ставится в соответствие уникальное число, называемое кодовым пунктом.

Числовое представление символа Юникода называется **кодовым пунктом**. Полный список всех символов Юникода доступен по адресу <https://www.unicode.org/Public/UNIDATA/UnicodeData.txt>.

Это очень большой текстовый файл, в котором каждый символ Юникода представлен отдельной строкой. Загрузите его, проведите поиск строки «ELEPHANT», и вы найдете строку следующего вида: 1F418;ELEPHANT. Число 1F418 представляет **шестнадцатеричное** значение. Шестнадцатеричные числа состоят из цифр от 0 до 9 и букв A–F, и со значениями из Юникода (и двоичными значениями вообще) в шестнадцатеричной форме часто бывает проще работать, чем в десятичной. Чтобы создать шестнадцатеричный литерал в C#, добавьте префикс 0x: 0x1F418.

1F418 является **кодовым пунктом UTF-8** для эмодзи со слоном. UTF-8 — самый распространенный способ **кодирования** символов в Юникоде (т. е. представления их в числовом виде). Эта кодировка переменной длины использует от 1 до 4 байт. В данном случае используются 3 байта: 0x01 (или 1), 0xF4 (или 244) и 0x18 (или 24).

Тем не менее это не те значения, которые будут выведены средствами сериализации JSON. Выводится более длинное шестнадцатеричное число: D83DDC18. Дело в том, что **тип char в C# использует кодировку UTF-16**, в которой кодовые пункты состоят из одного или двух 2-байтовых чисел. Кодовый пункт эмодзи со слоном в UTF-16 имеет вид 0xD83D 0xDC18. UTF-8 намного популярнее UTF-16, особенно во Всемирной паутине, так что при поиске кодовые пункты UTF-8 встретятся вам с гораздо большей вероятностью, чем UTF-16.

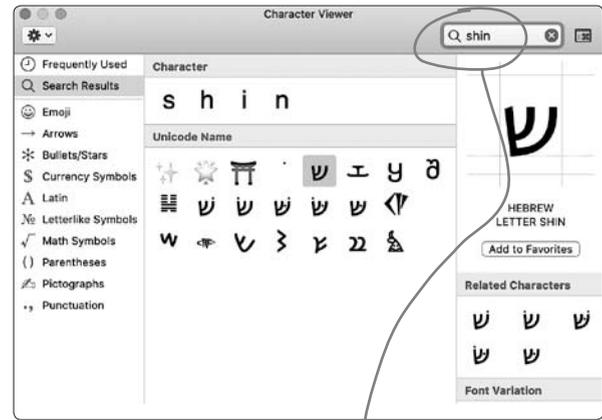
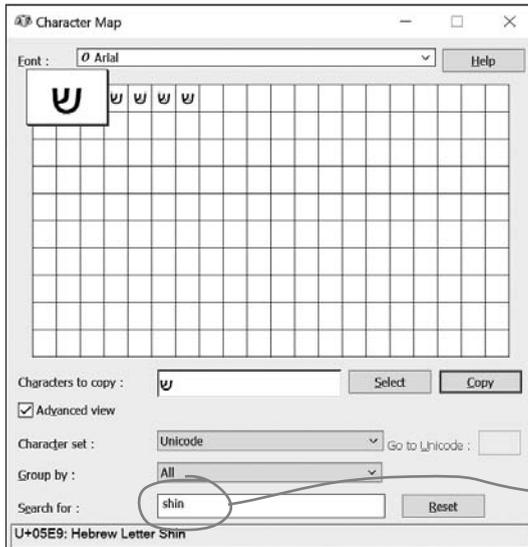
UTF-8 — кодировка переменной длины, используемая в большинстве веб-страниц и во многих системах. Она позволяет хранить символы с представлением их одним, двумя, тремя или большим количеством байтов. UTF-16 — кодировка фиксированной длины, которая всегда использует одно или два 2-байтовых числа. .NET хранит значения char в памяти в виде значений UTF-16.

Поддержка Юникода в Visual Studio

Воспользуемся Visual Studio и посмотрим, как IDE работает с символами Юникода. В главе 1 было показано, что в программах можно использовать эмодзи. На что еще способна IDE? Вернитесь к редактору кода и введите следующую команду:

```
Console.WriteLine("Hello ");
```

Если вы работаете в Windows, откройте приложение Таблица символов. На Mac нажмите **Ctrl+⌘**, чтобы открыть приложение Character Viewer. Найдите букву «шин» (ש) из иврита, скопируйте ее в буфер.



Как Таблица символов Windows, так и Character Viewer в macOS позволяют искать символы Юникода и копировать их в буфер.

Установите курсор в конец строки между пробелом и кавычкой и вставьте скопированную букву «шин». Хм-м, происходит что-то странное:

```
Console.WriteLine("Hello ש");
```

Вы заметили, что курсор оказался слева от вставленной буквы? Что ж, продолжим. Не щелкайте в IDE — оставьте курсор в его текущей позиции, затем переключитесь на приложение Таблица символов/Character Viewer, найдите и скопируйте букву иврита «ламед» (ל). Переключитесь обратно в IDE: убедитесь в том, что курсор все еще находится слева от «шин», и вставьте «ламед»:

```
Console.WriteLine("Hello של");
```

При вставке IDE добавляет новый символ слева от «шин». Теперь найдите буквы иврита «вав» (ו) и «мем» (מ). Вставьте их в IDE — символы вставляются слева от курсора:

```
Console.WriteLine("Hello מושל");
```

IDE знает, что *текст на иврите читается справа налево*, и ведет себя соответствующим образом. Щелкните, чтобы выделить текст рядом с началом команды, и медленно перетащите курсор вправо, чтобы выделить Hello и символы מושל. Внимательно наблюдайте за тем, что происходит, когда выделение достигает букв иврита. «Шин» пропускается, а символы выделяются справа налево — именно этого и ожидает читатель, знающий иврит.

.NET использует Юникод для хранения символов и текста

Два типа C# для хранения текста и символов — `string` и `char` — хранят свои данные в памяти в Юникоде. Когда эти данные записываются в файл в виде байтов, каждое из этих чисел записывается в файл. Попробуем разобраться в том, как именно данные Юникода записываются в файл. **Создайте новое консольное приложение.** Для исследования Юникода мы воспользуемся методами `File.WriteAllBytes` и `File.ReadAllBytes`.

1 Запишите обычную строку в файл и прочитайте ее.

Добавьте следующий код в метод `Main` — метод `File.WriteAllText` используется для записи строки «Eureka!» в файл с именем `eureka.txt` (а значит, потребуется директива `using System.IO`); затем создается новый массив байтов с именем `eurekaBytes`, в него читается содержимое файла и выводятся все прочитанные байты:

```
File.WriteAllText("eureka.txt", "Eureka!");  
byte[] eurekaBytes = File.ReadAllBytes("eureka.txt");  
foreach (byte b in eurekaBytes)  
    Console.Write("{0} ", b);  
Console.WriteLine(Encoding.UTF8.GetString(eurekaBytes));
```

Метод `ReadAllBytes` возвращает ссылку на новый массив байтов, содержащий все байты, прочитанные из файла.

Мы видим, что на выходе записываются следующие байты: 69 117 114 101 97 33. Последняя строка вызывает метод `Encoding.UTF8.GetString`, который преобразует массив байтов, содержащий символы в кодировке UTF-8, в строку. Теперь **откройте файл в Блокноте** (Windows) или `TextEdit` (Mac). Файл содержит текст «Eureka!».

2 Добавьте код записи байтов в виде шестнадцатеричных чисел.

При кодировании данных часто используется шестнадцатеричная система; опробуем эту возможность. Добавьте в конец метода `Main` следующий код, который записывает те же байты, используя `{0:x2}` для **форматирования каждого байта как шестнадцатеричного числа**:

```
foreach (byte b in eurekaBytes)  
    Console.Write("{0:x2} ", b);  
Console.WriteLine();
```

В шестнадцатеричной системе используются цифры от 0 до 9 и буквы от A до F для представления чисел с основанием 16, так что шестнадцатеричное число 6B соответствует десятичному 107.

Фрагмент приказывает `Write` вывести параметр 0 (первый после выводимой строки) в виде двухсимвольного шестнадцатеричного кода. Следовательно, те же 7 байт будут записаны в шестнадцатеричном формате вместо десятичного: 45 75 72 65 6b 61 21.

3 Измените первую строку, чтобы вместо «Eureka!» выводились буквы иврита «פֿיֿרֿ».

Недавно вы добавили в другую программу текст на иврите פֿיֿרֿ из Таблицы символов (Wi) или `Character Viewer` (Mac). **Закомментируйте первую строку метода `Main` и замените ее следующим кодом**, который выводит «פֿיֿרֿ» (вместо «Eureka!») в файл. Мы добавили дополнительный параметр `Encoding.Unicode`, чтобы данные записывались в UTF-16 (класс `Encoding` принадлежит пространству имен `System.Text`, поэтому в начало необходимо добавить директиву `using System.Text`):

```
File.WriteAllText("eureka.txt", "פֿיֿרֿ", Encoding.Unicode);
```

Снова запустите код и присмотритесь к выводу: ff fe e9 05 dc 05 d5 05 dd 05. Первые два символа «FF FE» в Юникоде сообщают о том, что далее идет строка, состоящая из 2-байтовых символов. Остальные байты содержат буквы иврита, но в обратном порядке байтов, так что U+05E9 соответствуют байты e9 05. Откройте файл в Блокноте или в `TextEdit` и проверьте результат.

4 Используйте JsonSerializer для анализа кодовых пунктов UTF-8 и UTF-16.

Когда вы сериализовали эмодзи со слонем, класс `JsonSerializer` сгенерировал последовательность `\uD83D\uDC18`, которая, как мы теперь знаем, является 4-байтовым кодовым пунктом UTF-16 в шестнадцатеричном виде. Теперь опробуем его с буквой иврита «шин». Включите директиву `using System.Text.Json;` в начало приложения, после чего добавьте следующую строку:

```
Console.WriteLine(JsonSerializer.Serialize("ש"));
```

Снова запустите приложение. На этот раз выводится значение из двух шестнадцатеричных байтов, «`\u05E9`» — кодовый пункт буквы иврита «шин» в UTF-16. Также он является кодовым пунктом той же буквы в UTF-8.

Но постоит: мы узнали, что кодовый пункт эмодзи со слонем в UTF-8 равен `0x1F418` и он *отличается* от кодового пункта UTF-16 (`0xD83D 0xDC18`). Что происходит?

Оказывается, большинство символов с 2-байтовыми кодовыми пунктами UTF-8 имеют такие же кодовые пункты в UTF-16. Но когда мы добираемся до значений UTF-8, требующих 3 и более байт (которые включают уже знакомые эмодзи, использовавшиеся в книге), они начинают различаться. Таким образом, хотя буква иврита «шин» представлена кодовым пунктом UTF-8 и UTF-16, эмодзи со слонем имеет кодовый пункт `0x1F418` в UTF-8 и `0xD8ED 0xDC18` в UTF-16.

Служебные последовательности `\u` и включение Юникода в строки

Когда мы сериализовали эмодзи со слонем, класс `JsonSerializer` сгенерировал для эмодзи 4-байтовый кодовый пункт UTF-16 `\uB83B\uDC18` в шестнадцатеричной форме. Это объясняется тем, что строки JSON и C# используют служебные последовательности UTF-16 — и как выясняется, те же служебные последовательности используются JSON.

Символы с 2-байтовыми кодовыми пунктами (такие, как 🐘) представляются служебной последовательностью `\u`, за которой следует шестнадцатеричный кодовый пункт (`\u05E9`); символы с 4-байтовыми кодовыми пунктами (такие, как 🐘) представляются последовательностью `\u` и двумя старшими байтами, за которыми следует `\u` и младшие 2 байта (`\uD83D \uDC18`).

В C# также поддерживается другая служебная последовательность: `\U` (с U в ВЕРХНЕМ РЕГИСТРЕ), за которой следуют 8 шестнадцатеричных байт, позволяет встроить кодовый пункт UTF-32, длина которого всегда равна 4 байтам. Это еще одна кодировка Юникода, и она по-настоящему полезна, потому что вы можете преобразовать UTF-8 в UTF-32 простым дополнением шестнадцатеричного числа нулями, — таким образом, кодовый пункт `<` в UTF-32 равен `\U000005E9`, а для 🐘 он равен `\U0001F418`.

5 Используйте служебные последовательности Юникода для кодирования 🐘.

Добавьте следующие строки в метод `Main`, чтобы записать эмодзи со слонем в два файла с использованием служебных последовательностей UTF-16 и UTF-32:

```
File.WriteAllText("elephant1.txt", "\uD83D\uDC18");
File.WriteAllText("elephant2.txt", "\U0001F418");
```

Снова запустите свое приложение, затем откройте оба файла в Блокноте или в `TextEdit`. В файле должен содержаться правильный символ.

Вы использовали служебные последовательности UTF-16 и UTF-32 для создания эмодзи, но метод `WriteAllText` записывает файл в UTF-8. Метод `Encoding.UTF8.GetString`, использованный на шаге 1, преобразует массив байтов с данными, закодированными в UTF-8, в строку.

C# может использовать массивы байтов для перемещения данных

Так как все данные в конечном итоге кодируются **байтами**, есть смысл представлять себе файл как один **большой массив байтов**... а вы уже знаете, как читать и записывать массивы байтов.

Этот код создает массив байтов, открывает входной поток и читает текст "Hello!" в байты с 0-го по 6-й массива.



```
byte[] greeting;
greeting = File.ReadAllBytes(filename);
```



Статический метод класса Array переставляет байты в обратном порядке. Мы используем его только для того, чтобы показать, что изменения в массиве байтов точно записываются в файл.

Эти числа являются значениями символов строки "Hello!" в Юникоде.

```
Array.Reverse(greeting);
File.WriteAllBytes(filename, greeting);
```

Когда программа записывает массив байтов в файл, символы текста тоже переставляются в обратном порядке.



Байты переставлены в обратном порядке.

Перестановка байтов "Hello!" работает только из-за того, что длина каждого символа составляет один байт. А вы сможете разобраться, почему он не работает для 01010?

Объект StreamWriter тоже кодирует данные, но он специализируется на тексте — по умолчанию используется UTF-8.

Использование BinaryWriter для записи двоичных данных

Данные различных типов — все эти строки, char, int и float — перед записью в файл можно преобразовать в массивы байтов, но это крайне монотонная и утомительная работа. Поэтому в .NET появился очень полезный класс **BinaryWriter**, автоматически преобразующий данные и записывающий их в файл. Вам нужно только создать объект FileStream и передать его конструктору класса BinaryWriter (они принадлежат пространству имен System.IO, поэтому необходимо добавить директиву using System.IO;). После этого вызывается метод записи данных. Давайте потренируемся в использовании класса BinaryWriter для записи двоичных данных в файл.

Делайте это!

1 Создайте консольное приложение и данные для записи в файл:

```
int intValue = 48769414;
string stringValue = "Hello!";
byte[] byteArray = { 47, 129, 0, 116 };
float floatValue = 491.695F;
char charValue = 'E';
```

Если вы используете File.Create, он создает новый файл — а если файл существует, он будет уничтожен, и на его месте будет создан новый. Метод File.OpenWrite открывает имеющийся файл и начинает записывать новую информацию поверх старой.

2 Чтобы использовать BinaryWriter, сначала необходимо открыть новый поток методом File.Create:

```
using (var output = File.Create("binarydata.dat"))
using (var writer = new BinaryWriter(output))
{
```

3 Вызовите метод Write. При каждом вызове этого метода новые байты будут добавляться в конец файла с закодированной версией данных, переданных в параметре:

```
writer.Write(intValue);
writer.Write(stringValue);
writer.Write(byteArray);
writer.Write(floatValue);
writer.Write(charValue);
}
```

Каждый вызов Write преобразует в байты одно значение, которое передается объекту FileStream. Передать можно любой тип значения, и он будет закодирован автоматически.

Объект FileStream записывает байты в конец файла.

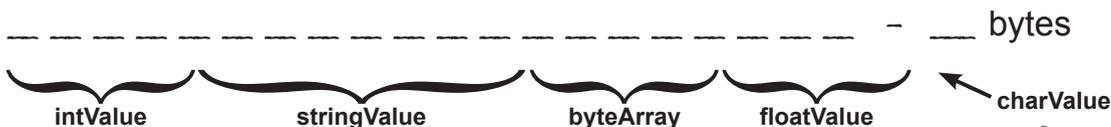
Возьми в руку карандаш

4 Используем написанный ранее код для чтения только что записанных данных:

```
byte[] dataWritten = File.ReadAllBytes("binarydata.dat");
foreach (byte b in dataWritten)
    Console.WriteLine("{0:x2} ", b);
Console.WriteLine(" - {0} bytes", dataWritten.Length);
```

Запишите результат в пустых полях внизу. Вы сможете **определить, какие байты соответствуют** каждому из пяти вызовов writer.Write(...)? Мы поместили фигурной скобкой группу байтов, соответствующую каждому вызову; это поможет вам определить, какие байты в файле соответствуют данным, записанным приложением.

Подсказка: строки могут иметь разную длину, поэтому каждая строка начинается с числа, которое сообщает .NET ее длину. BinaryWriter использует UTF-8 для кодирования строк, а в UTF-8 все символы "Hello!" имеют кодовые пункты UTF, состоящие из 1 байта. Загрузите файл Unicode.txt на сайте unicode.org (URL-адрес приводился ранее) и воспользуйтесь им для нахождения кодовых пунктов каждого символа.



Использование BinaryReader для чтения данных

Класс BinaryReader работает аналогично классу BinaryWriter. Вы создаете поток, присоединяете к нему объект BinaryReader и вызываете его методы... Но BinaryReader **не знает, что за данные содержатся в файле!** И знать не может. Значение типа float 491.695F преобразовалось в d8 f5 43 45. Но эти же байты соответствуют вполне допустимому значению типа int — 1 140 185 334, а следовательно, необходимо явно сообщить BinaryReader, данные какого типа читаются из файла. Добавьте в программу следующий код, который будет читать только что записанные данные.

Не верьте нам на слово. Замените строку, в которой читается float, вызовом метода ReadInt32 (тип floatRead нужно будет заменить на int). И вы сами увидите, какие данные читаются из файла.

- 1 Начнем с создания объектов FileStream и BinaryReader:

```
using (var input = File.OpenRead("binarydata.dat"))
using (var reader = new BinaryReader(input))
{
```

- 2 Сообщите BinaryReader, данные какого типа должны читаться при вызове разных методов:

```
int intRead = reader.ReadInt32();
string stringRead = reader.ReadString();
byte[] byteArrayRead = reader.ReadBytes(4);
float floatRead = reader.ReadSingle();
char charRead = reader.ReadChar();
```

Для каждого типа значения в BinaryReader определен отдельный метод, возвращающий данные правильного типа. Большинству из них параметры не нужны, хотя метод ReadBytes() использует параметр, сообщающий объекту BinaryReader количество читаемых байтов.

- 3 Данные, прочитанные из файла, выводятся на консоль:

```
Console.WriteLine("int: {0} string: {1} bytes: ", intRead, stringRead);
foreach (byte b in byteArrayRead)
    Console.WriteLine("{0} ", b);
Console.WriteLine(" float: {0} char: {1} ", floatRead, charRead);
}
```

Результат, который выводится на консоль:

int: 48769414 string: Hello! bytes: 47 129 0 116 float: 491.695 char: E

Возьми в руку карандаш
Решение



Первый байт в строке равен 6; это длина строки. Чтобы найти каждый из символов «Hello!», используйте Таблицу символов — слово начинается с U+0048 и заканчивается U+0021.

В калькуляторах для Windows и Mac предусмотрены режимы для программистов, преобразующие эти байты из шестнадцатеричной системы в десятичную, чтобы вы могли преобразовать их обратно в значения из массива.

Переменная типа char содержит символ Юникода, а 'E' занимает всего 1 байт — он кодируется в формате U+0045.

Часть Задаваемые Вопросы

В: Ранее в этой главе, когда я записывал слово «Eureka!» в файл, а потом читал его обратно, каждый символ представлялся одним байтом. Почему же для записи букв на иврите в слове **עִירָא** требуется по 2 байта? И что это за символы «FF FE» в начале файла?

О: Здесь проявляются различия между двумя тесно связанными кодировками Юникод. Латинским буквам (включая буквы английского алфавита), цифрам, знакам препинания, а также некоторым стандартным символам (фигурным скобкам, амперсандам и другим знакам, которые вы видите на клавиатуре) соответствуют небольшие значения в Юникоде — от 0 до 127. Они соответствуют очень старой кодировке ASCII, которая появилась еще в 1960-х годах, а кодировка UTF-8 разрабатывалась с целью сохранения обратной совместимости с ASCII. Файл, состоящий только из этих символов Юникода, содержит только эти байты, и ничего более.

Все усложняется, когда на сцену выходят символы с большими значениями. В одном байте могут храниться только значения от 0 до 255. В двух байтах уже можно хранить числа от 0 до 65 536 — в шестнадцатеричной системе счисления это FFFF. Файл должен иметь возможность сообщить программе, которая будет открывать файл, о присутствии символов с высокими значениями; для этого в начало файла помещается зарезервированная последовательность символов: FF FE. Это так называемая метка порядка байтов. Обнаружив ее, программа узнает, что все символы кодируются 2 байтами (таким образом, E кодируется с начальным нулем в виде 00 45).

В: Почему эти символы называются меткой порядка байтов?

О: Вернитесь к коду, который записывал буквы **עִירָא** в файл, а затем выводил записанные байты. Вы увидите, что байты в файле следуют в обратном порядке. Например, кодовый пункт **א** был записан в файл в виде E9 05. Этот порядок называется прямым — он означает, что менее значимый байт записывается первым. Вернитесь к коду, который записывал `WriteAllText`, и **измените третий аргумент с `Encoding.Unicode` на `Encoding.BigEndianUnicode`**. Тем самым вы указываете, что данные записываются в обратном порядке, — снова запустив программу, вы увидите, что байты следуют в порядке 05 E9. Также изменится и метка порядка байтов: FE FF. По ней Блокнот или `TextEdit` определяет, как следует интерпретировать байты в файле.

В: Почему после методов `File.ReadAllText` и `File.WriteAllText` мы не используем блок `using` и не вызываем метод `Close`?

О: Класс `File` содержит несколько очень полезных статических методов, которые автоматически открывают файл, читают или пишут данные, а затем **автоматически закрывают его**. Кроме методов `ReadAllText` и `WriteAllText`, также существуют методы `ReadAllBytes` и `WriteAllBytes`, читающие массивы байтов, и методы `ReadAllLines` и `WriteAllLines` для чтения и записи массивов строк; каждая строка в массиве становится отдельной строкой в файле. Все эти методы автоматически открывают и закрывают потоки, так что все операции в файле можно выполнить одной командой.

В: Если в классе `FileStream` имеются методы чтения и записи, то зачем нужны классы `StreamReader` и `StreamWriter`?

О: Класс `FileStream` используется для чтения и записи байтов в двоичный файл. Его методы работают с байтами и массивами байтов. Но есть и программы, работающие только с текстовыми данными, и в таких случаях классы `StreamReader` и `StreamWriter` становятся очень удобными. Методы этих классов созданы специально для чтения или записи строк текста. Без них при необходимости прочитать строку текста из файла вам пришлось бы сначала прочитать массив байтов, а затем писать цикл, ищущий в этом массиве знаки разрыва строк, так что иногда эти классы сильно облегчают жизнь.

Если вы записываете строку, состоящую из символов Юникода с малыми значениями (например, букв латинского алфавита), для каждого символа записывается 1 байт. Для записи символов Юникода с большими значениями (например, эмодзи) выделяются 2 и более байта.

Дамп позволяет просматривать байты в файлах

Шестнадцатеричным дампом данных (hex dump) называется представление содержимого файла в *шестнадцатеричном* виде. Это представление часто используется для анализа внутренней структуры файла.

Как выясняется, шестнадцатеричная система хорошо подходит для вывода байтов из файла. Байт в шестнадцатеричной системе записывается двумя шестнадцатеричными цифрами: значения байта лежат в диапазоне от 0 до 255, или от 00 до ff в шестнадцатеричной системе. Такое представление позволяет уместить большой объем данных в относительно малом пространстве, в формате, который упрощает выявление закономерностей. Двоичные данные полезно разбивать на строки длиной 8, 16 или 32 байта, потому что размер многих двоичных данных обычно кратен 4, 8, 16 или 32... как у всех типов C#. (Например, тип int занимает 4 байта.) Шестнадцатеричный байт позволяет точно понять, из каких частей строятся эти значения.

Создание шестнадцатеричного дампа простого текста

Начнем со знакомого текста, записанного буквами латинского алфавита:

```
When you have eliminated the impossible, whatever remains, however improbable,
must be the truth. - Sherlock Holmes
```

Сначала разбейте текст на сегменты из 16 символов, начиная с первых 16: When you have e1

Затем преобразуйте каждый символ в тексте в кодовый пункт UTF-8. Так как все латинские символы используют **1-байтовые** кодовые пункты UTF-8, каждый из них будет представлен шестнадцатеричным числом из 2 цифр от 00 до 7F. Каждая строка дампа будет выглядеть так:

Смещение сегмента (или позиция в файле), записанная в виде шестнадцатеричного числа. Первые 8 байт 16-байтового сегмента Разделитель упрощает чтение строки Последние 8 байт 16-байтового сегмента Символы текста, включенные в дамп

```
0000: 57 68 65 6e 20 79 6f 75 -- 20 68 61 76 65 20 65 6c When you have e1
```

Повторяйте до тех пор, пока не будут выведены все 16-символьные сегменты из файла:

```
0000: 57 68 65 6e 20 79 6f 75 -- 20 68 61 76 65 20 65 6c When you have e1
0010: 69 6d 69 6e 61 74 65 64 -- 20 74 68 65 20 69 6d 70 iminated the imp
0020: 6f 73 73 69 62 6c 65 2c -- 20 77 68 61 74 65 76 65 ossible, whateve
0030: 72 20 72 65 6d 61 69 6e -- 73 2c 20 68 6f 77 65 76 r remains, howev
0040: 65 72 20 69 6d 70 72 6f -- 62 61 62 6c 65 2c 20 6d er improbable, m
0050: 75 73 74 20 62 65 20 74 -- 68 65 20 74 72 75 74 68 ust be the truth
0060: 2e 20 2d 20 53 68 65 72 -- 6c 6f 63 6b 20 48 6f 6c . - Sherlock Hol
0070: 6d 65 73 0a -- mes.
```

Это и есть наш дамп. Существует много разных программ вывода дампов для разных операционных систем, и все они выводят слегка отличающиеся результаты. Каждая строка этой конкретной программы представляет 16 символов входных данных, использованных для генерирования дампа; в начале строки выводится смещение, а в конце — текст всех символов сегмента. Другие приложения вывода шестнадцатеричных дампов могут выводить информацию в другом формате (например, с визуализацией служебных последовательностей или выводом информации в десятичной системе).

Шестнадцатеричный дамп представляет собой шестнадцатеричное представление данных в файле или в памяти. Он может стать очень полезным инструментом для отладки двоичных данных.

Использование StreamReader для вывода шестнадцатеричного дампа

Построим приложение для выдачи шестнадцатеричного дампа. Приложение читает данные из файла при помощи StreamReader и выводит дампы на консоль. Мы воспользуемся методом ReadBlock класса StreamReader, который читает блок символов в символьный массив; при вызове задается количество читаемых символов. Метод читает либо заданное количество символов, либо весь остаток файла (если не осталось требуемого количества символов). Так как приложение должно выводить данные по 16 символам, размер читаемого блока будет составлять 16 символов.

Создайте новое консольное приложение с именем HexDump. Прежде чем добавлять код, **запустите приложение**, чтобы создать папку с двоичным файлом. При помощи Блокнота или TextEdit создайте **текстовый файл с именем textdata.txt**, добавьте в него текст и поместите в папку с двоичным файлом.

Ниже приведен код метода Main — он читает файл *textdata.txt* и записывает на консоль шестнадцатеричный дампы. Не забудьте добавить в начало файла директиву System.IO;.

```
static void Main(string[] args) {
    var position = 0;
    using (var reader = new StreamReader("textdata.txt")) {
        while (!reader.EndOfStream)
        {
            // Прочитать следующие 16 байт из файла в массив байтов
            var buffer = new char[16];
            var bytesRead = reader.ReadBlock(buffer, 0, 16);

            // Записать шестнадцатеричную позицию (или смещение), затем двоеточие и пробел
            Console.WriteLine("{0:x4}: ", position);
            position += bytesRead;

            // Записать шестнадцатеричное значение каждого символа в массив байтов
            for (var i = 0; i < 16; i++)
            {
                if (i < bytesRead)
                    Console.WriteLine("{0:x2} ", (byte)buffer[i]);
                else
                    Console.WriteLine(" ");
                if (i == 7) Console.WriteLine("-- ");
            }

            // Записать символы в массив байтов
            var bufferContents = new string(buffer);
            Console.WriteLine(" {0}", bufferContents.Substring(0, bytesRead));
        }
    }
}
```

Цикл перебирает символы и выводит каждый символ в выходную строку.

Свойство EndOfStream объекта StreamReader возвращает false, пока в файле остаются символы, которые могут быть прочитаны методом.

Форматная конструкция <<{0:x4}>> преобразует числовое значение в четырехзначное шестнадцатеричное число; таким образом, 1984 преобразуется в строку "07c0".

Метод String.Substring возвращает часть строки. Первый параметр задает начальную позицию (в данном случае начало строки), а во втором передается количество символов, включаемых в подстроку. Класс String содержит перегруженный конструктор, который получает массив char в параметре и преобразует его в строку.

Запустите приложение. На консоль выводится шестнадцатеричный дампы:

```
0000: 45 6c 65 6d 65 6e 74 61 -- 72 79 2c 20 6d 79 20 64      Elementary, my d
0010: 65 61 72 20 57 61 74 73 -- 6f 6e 21                        ear Watson!
```

Использование `Stream.Read` для чтения байтов из потока

Программа прекрасно работает с текстовыми файлами. Скопируйте файл `binarydata.dat`, записанный ранее с использованием `BinaryWriter`, в папку приложения, а затем измените приложение для чтения файла:

```
using (var reader = new StreamReader("binarydata.dat"))
```

Запустите приложение. На этот раз выводится другая информация — но не совсем правильная:

```
0000: fd 29 fd 02 06 48 65 6c -- 6c 6f 21 2f fd 00 74 fd    )?)Hello!/? t?
0010: fd fd 43 45                --                )?)CE
```

В решении упражнения «Возьмите в руку карандаш» эти байты были равны 81 и f6, но `StreamReader` заменил их на fd.

Символы текста («Hello!») выглядят нормально. Но сравните вывод с решением упражнения «Возьмите в руку карандаш» — с байтами что-то не так. Похоже, некоторые байты (86, e8, 81, f6, d8 и f5) заменены байтом fd. Дело в том, что класс `StreamReader` предназначен для чтения текстовых файлов, поэтому он читает только 7-разрядные значения, т. е. байты со значениями до 127 (7F в шестнадцатеричной записи или 1111111 в двоичной).

Исправим проблему — читаем байтов напрямую из потока. Измените блок `using`, чтобы в нем использовался вызов `File.OpenRead`, который открывает файл и возвращает `FileStream`. Свойство `Length` класса `Stream` используется для того, чтобы прочитать все байты в файле, а метод `Read` читает следующие 16 байт в буфер массива байтов:

```
using (Stream input = File.OpenRead("binarydata.dat"))
{
    var buffer = new byte[16];
    while (position < input.Length)
    {
        // Прочитать до следующих 16 байт из файла в массив байтов
        var bytesRead = input.Read(buffer, 0, buffer.Length);
```

Мы использовали явный тип вместо `var`, чтобы наглядно показать, что в приложении используется поток, а именно `FileStream` (расширяющий `Stream`).

Остаток кода остается неизменным, кроме строки, в которой задается значение `bufferContents`:

```
// Записать символы в массив байтов
var bufferContents = Encoding.UTF8.GetString(buffer);
```

Метод `Stream.Read` получает три аргумента: массив байтов для чтения данных (`buffer`), начальный индекс в массиве (0) и количество читаемых байтов (`buffer.Length`).

Класс `Encoding` использовался ранее в этой главе для преобразования массива байтов в строку. Массив байтов содержит 1 байт на символ — т. е. он является действительной строкой в кодировке UTF-8. Это означает, что его содержимое можно преобразовать методом `Encoding.UTF8.GetString`. Так как класс `Encoding` принадлежит пространству имен `System.Text`, в начало файла необходимо добавить директиву `using System.Text`;

Снова запустите приложение. На этот раз оно выводит правильные байты вместо того, чтобы заменять их байтом fd:

```
0000: 86 29 e8 02 06 48 65 6c -- 6c 6f 21 2f 81 00 74 f6    )?)Hello!/? t?
0010: d8 f5 43 45                --                )?)CE
```

Теперь ваше приложение читает из файла все байты, не только символы текста.

Осталось еще одно, что можно сделать для чистки вывода. Многие программы шестнадцатеричного дампа заменяют нетекстовые символы в выводе точками. Добавьте следующую строку в конец цикла `for`:

```
if (buffer[i] < 0x20 || buffer[i] > 0x7F) buffer[i] = (byte)'. ';
```

Снова запустите приложение — на этот раз вопросительные знаки заменяются точками:

```
0000: 86 29 e8 02 06 48 65 6c -- 6c 6f 21 2f 81 00 74 f6    .)...Hello!/.t.
0010: d8 f5 43 45                --                ..CE
```

Аргументы командной строки

Большинство программ вывода шестнадцатеричного дампа – утилиты, предназначенные для запуска из командной строки. Чтобы вывести дамп файла, пользователь передает его имя программе в **аргументе командной строки** – например, `C:\> HexDump myfile.txt`.

Изменим приложение так, чтобы в нем использовались аргументы командной строки. Когда вы создаете консольное приложение, C# предоставляет доступ к аргументам командной строки в **строковом массиве** `args`, передаваемом методу `Main`:

```
static void Main(string[] args)
```

Изменим метод `Main`, чтобы он открывал файл и читал содержимое из потока. **Метод `File.OpenRead`** получает имя файла в параметре, открывает файл для чтения и возвращает поток с содержимым файла.

Измените следующие строки в методе `Main`:

```
static void Main(string[] args)
{
    var position = 0;
    using (Stream input = File.OpenRead(args[0]))
    {
        var buffer = new byte[16];
        int bytesRead;
```

```
        // Прочитать до следующих 16 байт из файла в массив байтов
        while ((bytesRead = input.Read(buffer, 0, buffer.Length)) > 0) {
```

Приложение получает аргументы командной строки в параметре `args` и передает их методу `GetInputStream`.

Также необходимо удалить строку с объявлением `bytesRead` и первую строку блока `while`, в которой вызывается `inputRead` для потока.

Чтобы воспользоваться аргументами командной строки, следует **изменить в IDE свойства отладки** и включить передачу командной строки программе. **Щелкните правой кнопкой мыши на проекте** в решении, а затем:

- ★ **В Windows** – выберите команду `Properties`, щелкните на категории `Debug` и введите имя файла для выдачи дампа в поле `Application arguments` (либо полное имя, либо имя файла в папке с двоичным файлом).
- ★ **В macOS** – выберите команду `Options`, раскройте раздел `Run>>Configurations`, щелкните на `Default` и введите имя файла в поле `Arguments`.

Будьте внимательны: щелкнуть нужно именно на проекте, а не на решении.

Теперь при отладке приложения массив `args` будет содержать аргументы, заданные в настройках проекта. **Убедитесь в том, что при настройке аргументов командной строки было задано правильное имя.**

Запуск приложения из командной строки

Также можно запустить приложение из командной строки, заменив `[имя_файла]` именем файла (полным именем или именем файла в текущем каталоге):

- ★ **В системе Windows** Visual Studio строит исполняемый файл в папке `bin\Debug` (в той, в которой размещается файл для чтения), так что исполняемый файл можно запустить прямо из этой папки. Откройте окно командной строки, перейдите в каталог `bin\Debug` командой `cd` и выполните команду `HexDump [имя_файла]`.
- ★ **На Mac** нужно будет **построить автономное приложение**. Откройте окно терминала, перейдите в папку проекта и выполните следующую команду: `dotnet publish -r osx-x64`.

В выводе будет присутствовать строка следующего вида: `HexDump -> /path-to-binary/osx-x64/publish/`

Откройте окно терминала, перейдите по выведенному полному пути командой `cd`, выполните команду `./HexDump [filename]`.



Упражнение: Hide and Seek

В следующем упражнении мы построим приложение, в котором вы исследуете дом и играете в прятки с игроком, управляемым компьютером. Процедура размещения комнат проверит ваши навыки работы с коллекциями и интерфейсами. Затем будет реализована сериализация состояния игры в файле, чтобы вы могли сохранить и загрузить его.

Сначала вы исследуете виртуальный дом, перемещаясь из комнаты в комнату и исследуя предметы в каждой комнате.



Затем добавляется компьютерный игрок, который ищет ваше убежище. Посмотрите, сколько ходов ему для этого понадобится!

Перейдите к репозиторию GitHub данной книги и загрузите PDF-файл проекта:
<https://github.com/head-first-csharp/fourth-edition>

КЛЮЧЕВЫЕ МОМЕНТЫ

- Юникод — отраслевой стандарт кодирования символов, т. е. преобразования их в байты. Каждому из миллиона с лишним символов Юникода назначается **кодový пункт**, т. е. уникальный числовой идентификатор.
- Большинство файлов и веб-страниц кодируется в **UTF-8** — кодировке переменной длины, в которой символы кодируются одним, двумя, тремя или четырьмя байтами.
- C# и .NET используют UTF-16 для хранения символов и текста в памяти, при этом строка интерпретируется как **коллекция символов, доступная только для чтения**.
- Метод **Encoding.UTF8.GetString** преобразует массив байтов UTF-8 в строку. **Encoding.Unicode** преобразует массив байтов, закодированный в UTF-16, в строку, а **Encoding.UTF32** преобразует массив байтов UTF-32.
- **Служебные последовательности \u** используются для включения Юникода в строки C#. Последовательность **\u** содержит кодировку UTF-16, а **\U** содержит UTF-32 — 4-байтовую кодировку переменной длины.
- **StreamWriter** и **StreamReader** хорошо работают с текстом, но обрабатывают лишь немногие символы за пределами латинского набора символов. Для чтения двоичных данных следует использовать классы **BinaryWriter** и **BinaryReader**.
- Метод **StreamReader.ReadBlock** читает символы в буферный массив байтов. Он выполняется в **блокирующем режиме**, т. е. продолжает работать и не возвращает управление до тех пор, пока не будут прочитаны все запрошенные символы или не будут исчерпаны все данные для чтения.
- **File.OpenRead** возвращает **FileStream**, а метод **FileStream.Read** читает байты из потока.
- Метод **String.Substring** возвращает часть строки. Класс **String** содержит **перегруженный конструктор**, который получает массив **char** в параметре и преобразует его в строку.
- C# предоставляет доступ к аргументам командной строки консольных приложений в **строковом массиве args**, передаваемом методу **Main**.

Лабораторный курс Unity № 5

Отслеживание лучей

Создавая сцену в Unity, вы создаете виртуальный 3D-мир, в котором перемещаются персонажи вашей игры. Но в большинстве игр объекты окружающей обстановки не контролируются игроком напрямую. Как же эти объекты определяют свое место в сцене?

В лабораторных работах № 5 и 6 вы познакомитесь с **системой поиска путей и навигации** — изощренной системой на базе искусственного интеллекта, которая позволяет создавать персонажей, способных находить путь в построенных вами мирах. В этой лабораторной работе мы построим сцену из объектов `GameObject` и используем навигацию для перемещения персонажей по сцене.

Отслеживание лучей (raycasting) будет использоваться для написания кода, реагирующего на геометрию сцены, **захвата ввода** и использования его для перемещения объекта `GameObject` в точку, в которой щелкнул игрок. Что не менее важно, вы **потренируетесь в написании кода C#** с классами, полями, ссылками и другими аспектами, которые были представлены в предыдущих главах.

Создание нового проекта Unity и начало создания сцены

Прежде чем браться за работу, закройте все проекты Unity, открытые в данный момент. Также закройте Visual Studio – Unity откроет среду за вас. Создайте новый проект Unity с использованием 3D-шаблона, выберите макет Wide, чтобы он соответствовал приведенным снимкам экранов, и присвойте ему имя **Unity Labs 5 and 6**, чтобы иметь возможность вернуться к нему в будущем.

Начните с создания игровой области, в которой будет ориентироваться игрок. Щелкните правой кнопкой мыши в окне Hierarchy и создайте плоскость (GameObject>>3D Object>>Plane). Присвойте объекту Plane имя Floor.

Щелкните правой кнопкой мыши на папке Assets в окне Project и **создайте в ней папку с именем Materials**. Затем щелкните на только что созданной папке Materials и выберите команду **Create>>Material**. Присвойте новому материалу имя *FloorMaterial*. Пока оставим этот материал простым – только назначим ему цвет. Выберите объект Floor в окне Project, затем щелкните на белом поле справа от свойства Albedo в окне Inspector.

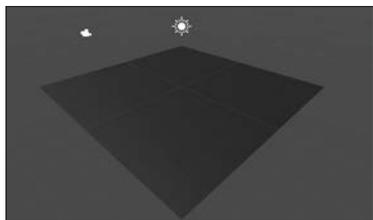


Вы можете воспользоваться пипеткой для того, чтобы выбрать образец цвета в любой точке экрана.



В окне Color выберите цвет пола на внешнем кольце. На снимке мы выбрали цвет с кодом 4E51CB – вы можете ввести это число в поле Hexadecimal.

Перетащите материал **из окна Project на объект GameObject Plane в окне Hierarchy**. Пол должен окраситься в выбранный вами цвет.



Плоскость не имеет измерения Y. Что произойдет, если увеличить масштаб по оси Y? А если масштаб по оси Y будет отрицательным? Нулевым?

Подумайте над вопросом и попробуйте предположить результат. Затем при помощи окна Inspector опробуйте разные значения масштаба Y и посмотрите, работает ли плоскость так, как ожидалось. (Потом не забудьте вернуть исходное значение!)

Плоскость Plane представляет собой плоский квадратный объект с размерами 10×10 единиц (по плоскости X-Z) и высотой 0 единиц (по оси Y). Unity создает плоскость так, что ее центр находится в точке (0, 0, 0). Центральная точка плоскости определяет ее положение в сцене. Как и другие объекты, плоскость можно перемещать по сцене при помощи окна Inspector или воспользоваться инструментами, изменяющими ее позицию и угол поворота. Также можно изменить масштаб, но поскольку плоскость не имеет высоты, изменять можно только масштаб по осям X и Z – любое положительное число, введенное для масштаба по оси Y, игнорируется. Объекты, созданные из меню 3D Object (плоскости, сферы, кубы, цилиндры и несколько других простых фигур), называются примитивными объектами. Чтобы больше узнать о них, откройте руководство Unity из меню Help и найдите страницу «Primitive and placeholder objects». Не торопитесь и откройте эту страницу прямо сейчас. Прочитайте, что в ней говорится о плоскостях, сферах, кубах и цилиндрах.

Настройка камеры

В двух последних лабораторных работах Unity вы узнали, что объект `GameObject` фактически является «контейнером» для компонентов, а главная камера (Main Camera) содержит три компонента: `Transform`, `Camera` и `Audio Listener`. Это вполне логично, потому что все, что требуется от камеры, — находиться в определенном месте и записывать то, что она видит и слышит. Взгляните на компонент `Transform` в окне `Inspector`.

Обратите внимание: поле `Position` содержит значение (0, 1, -10). Щелкните на надписи `Z` в строке `Position` и перетащите ее вверх-вниз. Вы увидите, как камера двигается вперед-назад в окне сцены. Обратите внимание на четыре линии перед камерой. Они представляют **область просмотра** (viewport), т. е. видимую область на экране игрока.



Переместите камеру по сцене, поверните ее при помощи инструментов `Move (W)` и `Rotate (E)`, как это делалось с другими объектами `GameObject` в сцене. Окно `Camera Preview` обновляется в реальном времени, показывая, что видит камера. Следите за окном `Camera Preview` при перемещении камеры. При входе и выходе поля из области просмотра камеры создается впечатление, что он движется.

Воспользуйтесь контекстным меню в окне `Inspector` для сброса компонента `Transform` главной камеры. Позиция камеры и ее угол поворота возвращаются к значениям (0, 0, 0). Камера пересекает плоскость в окне `Scene`.

Направим камеру вниз. Щелкните на надписи `X` рядом с `Rotation`, попробуйте перетащить ее вверх-вниз. Вы увидите, что область просмотра в окне `Camera Preview` двигается. **Задайте камере угол поворота по оси X, равный 90**, в окне `Inspector`, чтобы направить камеру вертикально вниз.



Вы заметите, что окно `Camera Preview` стало пустым, и это понятно, потому что камера направлена вниз под бесконечно тонкую плоскость. Щелкните на надписи `Y` компонента `Transform` и перетащите вверх, пока не увидите всю плоскость в окне `Camera Preview`.

Выберите объект `Floor` в окне `Hierarchy`. Обратите внимание: окно `Camera Preview` исчезает — оно отображается только при выделенной камере. Вы также можете переключаться между окнами `Scene` и `Game`, чтобы увидеть то, что видит камера.

При помощи компонента `Transform` объекта `Plane` в окне `Inspector` **назначьте объекту `GameObject Floor` масштаб (4, 1, 2)**, чтобы его длина была вдвое больше ширины. Так как ширина и высота объекта `Plane` равны 10×10 единиц, с этим масштабом его длина будет составлять 40 единиц, а ширина — 20 единиц. Плоскость снова полностью перекрывает область просмотра, поэтому сдвиньте камеру вверх по оси `Y`, пока вся плоскость не окажется в поле зрения.



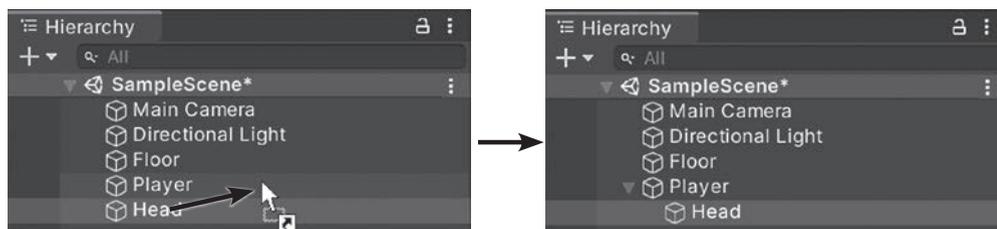
Вы можете переключаться между окнами `Scene` и `Game`, чтобы увидеть, что в данный момент видит камера.

Создание объекта GameObject для игрока

Нам понадобится игровой персонаж, которым будет управлять игрок (далее будем называть его просто «игроком»). Мы создадим простую человекоподобную фигуру с цилиндром вместо тела и сферой вместо головы. Чтобы убедиться в том, что ни один объект в настоящий момент не выделен, щелкните на сцене (или в пустом месте) в окне Hierarchy.

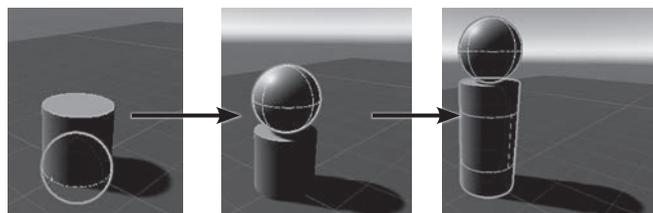
Создайте объект GameObject Cylinder (3D Object>> Cylinder) – в середине сцены появится цилиндр. Присвойте ему имя Player, затем **выберите команду Reset в контекстном меню** компонента Transform, чтобы объекту были присвоены значения по умолчанию. Затем **создайте объект GameObject Sphere** (3D Object>> Sphere). Присвойте ему имя Head и выполните сброс его компонента Transform, как было сделано с цилиндром. Каждый объект будет представлен отдельной строкой в окне Hierarchy.

Но нам нужны не два отдельных объекта GameObject, а один объект GameObject, которым управляет один сценарий C#. Для таких ситуаций в Unity существует концепция **родительских связей**. Щелкните на объекте Head в окне Hierarchy и **перетащите его на Player**. Объект Player становится родителем Head. Теперь объект GameObject Head **вложен** в Player.



Выделите объект Head в окне Hierarchy. Он был создан в точке (0, 0, 0), как и все созданные вами сферы. Вы видите контур сферы, но сама сфера не видна, потому что она скрыта плоскостью и цилиндром. Используйте компонент Transform в окне Inspector, чтобы **изменить позицию Y сферы на 1.5**. Теперь сфера располагается над цилиндром, как раз в подходящем месте для головы игрока.

Выделите объект Player в окне Hierarchy. Так как его позиция Y равна 0, половина цилиндра скрыта плоскостью. Задайте его позицию Y равной 1. Цилиндр поднимается над плоскостью. Обратите внимание на то, что сфера Head поднялась вместе с ним. При перемещении Player объект Head перемещается вместе с ним, потому что перемещение родительского объекта GameObject приводит к перемещению его потомков – более того, *любое* изменение, внесенное в компонент Transform, автоматически применяется к потомкам. Например, при масштабировании родительского объекта его потомки тоже масштабируются. Переключитесь в окно Game, – фигурка игрока находится в середине игровой области.



Когда вы изменяете компонент Transform объекта GameObject, у которого есть вложенные потомки, потомки будут перемещаться, вращаться и масштабироваться вместе с ним.

Знакомство с системой навигации Unity

Одна из основных операций, выполняемых в видеоиграх, – перемещение различных объектов. Игроки, персонажи, враги, предметы, препятствия... все эти объекты могут двигаться. По этой причине в Unity существует сложная система поиска путей и навигации на базе искусственного интеллекта, которая помогает объектам GameObject перемещаться по сцене. Мы воспользуемся системой навигации для того, чтобы игрок двигался к цели.

Система поиска путей и навигации позволяет персонажам осмысленно искать путь в игровом мире. Чтобы воспользоваться ею, необходимо сообщить Unity, куда игрок может идти:

- ★ Во-первых, необходимо точно сообщить Unity, где разрешено перемещаться игроку. Для этого создается объект навигационной сетки NavMesh с полной информацией обо всех областях сцены, в которых разрешено перемещение: склоны, лестницы, препятствия и даже точки, называемые внесеточными звеньями, в которых выполняются особые действия игрока (например, открытие двери).
- ★ Во-вторых, добавьте компонент NavMesh Agent к каждому объекту GameObject, который должен выполнять навигацию. Этот компонент автоматически перемещает объект GameObject по сцене, используя ИИ для поиска самого эффективного пути к цели с обходом препятствий (и возможно, других компонентов NavMesh Agent).
- ★ Иногда навигация по сложным объектам NavMesh требует серьезных вычислений. Вот почему в Unity предусмотрена функция Bake, которая позволяет настроить NavMesh заранее и провести предварительную обработку геометрической информации, чтобы повысить эффективность работы агентов.

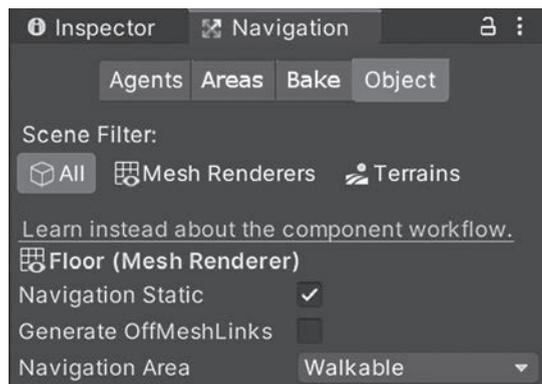
Unity предоставляет сложную систему поиска путей и навигации, которая может перемещать объекты GameObject по сцене в реальном времени с поиском эффективных путей и обходом препятствий.



Создание сетки NavMesh

Создадим сетку навигации NavMesh, которая состоит только из плоскости Floor. Для этого мы воспользуемся окном Navigation. **Выберите команду AI>>Navigation из меню Window**, чтобы добавить окно Navigation в рабочее пространство Unity. Оно должно появиться в виде вкладки на одной панели с окном Inspector. Затем в окне Navigation включите для объекта GameObject Floor параметры *navigation static* и *walkable*.

- * Нажмите **кнопку Object** в верхней части окна Navigation.
- * **Выберите плоскость Floor** в окне Hierarchy.
- * Установите флажок **Navigation Static**. Тем самым вы приказываете Unity включать Floor при предварительном построении NavMesh.
- * **Выберите Walkable** из раскрывающегося списка Navigation Area. Это сообщает Unity, что плоскость Floor является поверхностью, по которой может перемещаться любой объект GameObject с компонентом NavMesh Agent.



Нажмите кнопку **Object**, чтобы пометить объекты GameObject в сцене как **навигационно-статические**; это означает, что они являются частью NavMesh и остаются неподвижными.

Плоскость Floor объявляется как **проходимая**; NavMesh Agent будет знать, как продолжить маршрут.

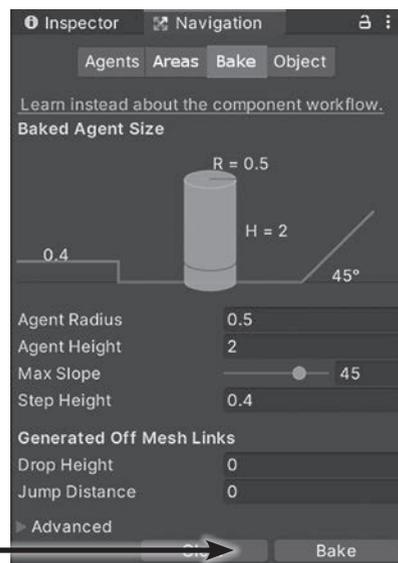
Так как единственной проходимой областью в игре будет пол, настройка раздела Object на этом завершена. Для более сложных сцен с большим количеством проходимых поверхностей или непроходимых препятствий каждый отдельный объект GameObject должен быть снабжен соответствующей пометкой.

Щелкните на кнопке Bake в **верхней части** окна Navigation, чтобы просмотреть возможные параметры построения навигационной сетки.

Теперь щелкните на **другой кнопке Bake** в **нижней части** окна Navigation. Надпись на ней ненадолго заменится текстом Cancel, а потом снова вернется к Bake. А вы заметили, что в окне Scene что-то изменилось? Попробуйте переключаться между окнами Inspector и Navigation. Когда окно Navigation активно, в окне Scene выводится навигационная сетка: NavMesh выделяется как синяя «накладка» на объектах GameObject, которые входят в предварительно построенную сетку NavMesh. В данном случае выделяется плоскость, которая была помечена как навигационно-статическая и проходимая.

Подготовка сетки NavMesh завершена.

Щелкните на **кнопке Bake**, чтобы сгенерировать NavMesh.



Автоматическая навигация в игровой области

Добавим компонент NavMesh Agent к объекту GameObject Player. **Выделите Player** в окне Hierarchy, вернитесь к окну Inspector, щелкните на кнопке **Add Component** и выберите команду **Navigation>>NavMesh Agent**. Высота тела (цилиндр) составляет 2 единицы, а высота сферы (голова) составляет 1 единицу, поэтому высота агента должна быть равна 3 единицам — задайте Height значение 3. Теперь компонент NavMesh Agent готов к перемещению объекта GameObject Player по навигационной сетке.

Создайте папку Scripts и добавьте сценарий с именем MoveToClick.cs. Сценарий позволяет игроку щелкнуть в любой точке игровой области и переместить объект GameObject в эту точку. Вы узнали о приватных полях в главе 5. Сценарий будет использовать такое поле для хранения ссылки на NavMeshAgent. Коду GameObject понадобится ссылка на агента, чтобы он мог сообщить агенту, куда следует перемещаться; мы вызываем метод GetComponent для получения ссылки и сохраняем ее в **приватном поле NavMeshAgent** с именем agent:

```
agent = GetComponent<NavMeshAgent>();
```

Система навигации использует классы из пространства имен UnityEngine.AI, поэтому в начало файла *MoveToClick.cs* необходимо добавить строку:

```
using UnityEngine.AI;
```

Сценарий MoveToClick выглядит так:

```
public class MoveToClick : MonoBehaviour
```

```
{
    private NavMeshAgent agent;

    void Awake()
    {
        agent = GetComponent<NavMeshAgent>();
    }

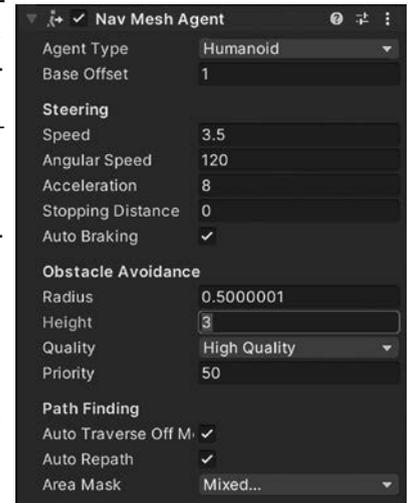
    void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            Camera cameraComponent = GameObject.Find("Main Camera").GetComponent<Camera>();
            Ray ray = cameraComponent.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;
            if (Physics.Raycast(ray, out hit, 100))
            {
                agent.SetDestination(hit.point);
            }
        }
    }
}
```

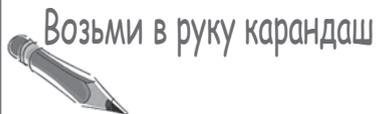
В последней лабораторной работе Unity метод **Start** использовался для задания позиции объекта GameObject при его первом появлении. Однако существует еще один метод, который вызывается до метода **Start** вашего сценария. Метод **Awake** вызывается при создании объекта, тогда как **Start** вызывается при активизации сценария. Сценарий **MoveToClick** использует для инициализации поля метод **Awake**, а не **Start**.

Здесь сценарий обрабатывает **щелчки мышью**. Метод **Input.GetMouseButtonDown** проверяет, нажал ли пользователь кнопку мыши, а аргумент 0 указывает на то, что проверяется левая кнопка. Так как метод **Update** вызывается для каждого кадра, проверка нажатия кнопки мыши выполняется постоянно.

Поэкспериментируйте с полями *Speed*, *Angular Speed*, *Acceleration* и *Stopping Distance* в агенте *NavMesh*. Их можно изменять во время выполнения игры (но помните, что значения, измененные во время игры, не сохраняются). Что произойдет, если присвоить им очень большие значения?

Перетащите сценарий на объект Player и запустите игру. Во время выполнения игры щелкните в **любой точке плоскости**. Компонент NavMesh Agent перемещает вашего игрока к точке щелчка.





В нескольких последних главах много говорилось о ссылках на объекты и ссылочных переменных. Немного поработаем с карандашом, чтобы некоторые идеи и концепции, лежащие в основе ссылок на объекты, закрепились в вашем мозгу.

Добавьте следующее открытое поле в класс MoveToClick:

```
public Camera cameraComponent;
```

Перейдите в окно Hierarchy, щелкните на объекте Player и найдите новое поле Main Camera в компоненте Move To Click (Script). **Перетащите объект Main Camera** из окна Hierarchy **на поле Camera Component** компонента Move To Click (Script) объекта GameObject Player:



Закомментируйте следующую строку:

```
Camera cameraComponent = GameObject.Find("Main Camera").GetComponent<Camera>();
```

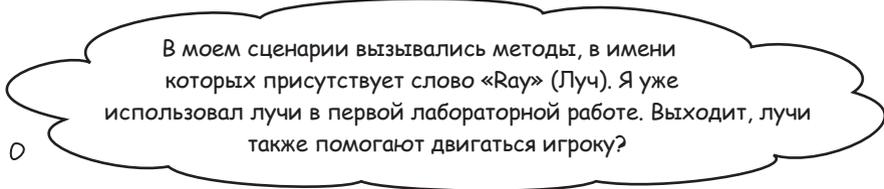
Снова запустите игру. Она все равно работает! Почему? Подумайте и попробуйте найти ответ. Запишите его внизу.

.....

.....

.....

.....



В моем сценарии вызывались методы, в имени которых присутствует слово «Ray» (Луч). Я уже использовал лучи в первой лабораторной работе. Выходит, лучи также помогают двигаться игроку?



Да! Здесь используется чрезвычайно полезный инструмент, называемый отслеживанием лучей.

Во второй лабораторной работе метод Debug.DrawRay помог нам разобраться в том, как работают 3D-векторы; для этого в сцене рисовался луч, начинающийся в точке (0, 0, 0). Метод Update вашего сценария MoveToClick обычно делает нечто похожее. Он использует **метод Physics.Raycast** для «прокладки» луча (вроде того, который использовался для изучения векторов) от камеры до точки щелчка и проверяет, **соприкасается ли луч с полом**. Если результат положительный, метод Physics.Raycast предоставляет точку пола, в которой обнаружен контакт. Тогда сценарий задает поле **точки назначения компонента NavMesh Agent**, в результате чего NavMesh Agent автоматически перемещает игрока к указанной точке.

Ослеживание лучей под увеличительным стеклом



Сценарий MoveToClick вызывает метод **Physics.Raycast** — чрезвычайно полезный инструмент, который Unity предоставляет для реакции на изменения в сцене. Метод проводит виртуальный луч в сцене и сообщает, столкнулся ли он с какими-либо объектами. Параметры метода **Physics.Raycast** сообщают, куда должен быть направлен луч и каково максимальное расстояние для его проведения:

Physics.Raycast(направление_луча, out hit, макс_расстояние)

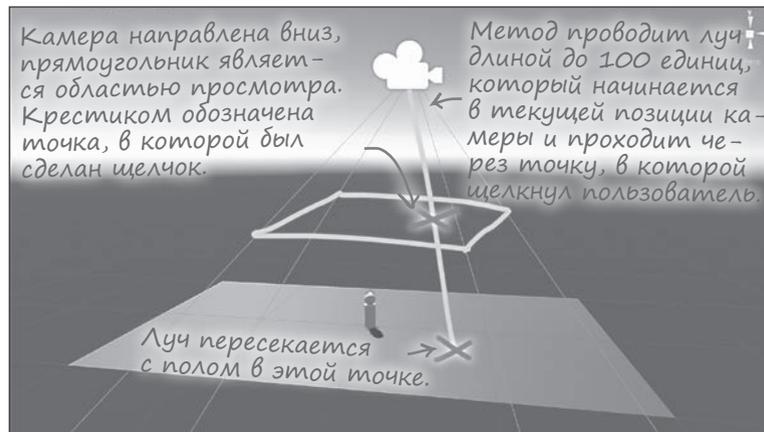
Метод возвращает **true**, если столкновение будет обнаружено, или **false** в противном случае. Ключевое слово **out** используется для сохранения результата в переменной (точно так же, как это делалось с **int.TryParse** в нескольких последних главах. Разберемся поподробнее в том, как работает метод.

Методу **Physics.Raycast** необходимо сообщить, куда должен быть направлен луч. Таким образом, прежде всего необходимо найти камеру, а конкретнее, компонент **Camera** объекта **GameObject Main Camera**. Это делается примерно так же, как мы получали **GameController** в последней лабораторной работе Unity:

```
GameObject.Find("Main Camera").GetComponent<Camera>();
```

Класс **Camera** содержит метод с именем **ScreenPointToRay**, который создает луч от текущей позиции камеры через точку (X,Y) на экране. Метод **Input.mousePosition** предоставляет позицию (X,Y) экрана, в которой был сделан щелчок. В результате переменная **ray** содержит координаты, которые должны передаваться **Physics.Raycast**:

```
Ray ray = cameraComponent.ScreenPointToRay(Input.mousePosition);
```



Итак, мы знаем параметры луча и можем вызвать метод **Physics.Raycast**, чтобы узнать точку столкновения:

```
RaycastHit hit;
if (Physics.Raycast(ray, out hit, 100))
{
    agent.SetDestination(hit.point);
}
```

Метод возвращает логическое значение и использует ключевое слово **out** — собственно, он работает точно так же, как **int.TryParse**. Если метод возвращает **true**, то переменная **hit** содержит точку пола, в которой происходит столкновение с лучом. Присваивание **agent.destination** приказывает **NavMesh Agent** начать перемещение игрока к точке столкновения луча с полом.

Возьми в руку карандаш



В этом упражнении мы изменили класс MoveToClick и добавили поле для главной камеры вместо того, чтобы использовать методы Find и GetComponent. Вы перетаскили Main Camera, после чего мы задали вопрос. Ваш ответ похож на наш?

Снова запустите игру. Она все равно работает! Почему? Подумайте и попробуйте найти ответ. Запишите его внизу.

Когда мой код вызвал метод `mainCamera.GetComponent<Camera>`, он вернул ссылку на объект `GameObject`.

Я заменил ее полем и перетаскил объект `Main Camera` из окна `Hierarchy` в окно `Inspector`, в результате чего поле было инициализировано ссылкой на тот же объект `GameObject`. Это два разных способа присваивания переменной `cameraComponent` ссылки на один и тот же объект, поэтому поведение программы не изменилось.

Мы будем повторно использовать сценарий MoveToClick в последующих лабораторных работах, поэтому после того, как ответы будут записаны, верните сценарий к прежнему состоянию: удалите поле `MainCamera` и восстановите строку, задающую значение переменной `cameraComponent`.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Плоскость `Plane` представляет собой плоский квадратный объект размером 10×10 единиц (по плоскости X-Z) и высотой 0 единиц (по оси Y).
- Вы можете **перемещать главную камеру** для изменения отображаемой части сцены; для этого измените компонент `Transform` (так же, как для любого другого объекта `GameObject`).
- При изменении компонента `Transform` объекта `GameObject`, у которого есть вложенные потомки, потомки будут перемещаться, вращаться и масштабироваться вместе с ним.
- Unity предоставляет **систему поиска путей и навигации на базе искусственного интеллекта**, которая может перемещать объекты `GameObject` по сцене в реальном времени с поиском эффективных путей и обходом препятствий.
- Объект **NavMesh** содержит всю информацию о проходимых областях сцены. Вы можете настроить `NavMesh` заранее и заранее вычислить геометрические подробности, необходимые для повышения эффективности работы агентов.
- Компонент **NavMesh Agent** автоматически перемещает объект `GameObject` по сцене, используя искусственный интеллект для нахождения самого эффективного пути к цели.
- Метод **NavMeshAgent.SetDestination** заставляет агента вычислить путь к новой позиции и начать двигаться к новой точке.
- Unity вызывает метод **Awake** вашего сценария при первой загрузке объекта `GameObject`, задолго до вызова метода `Start` сценария, но после создания экземпляров других объектов `GameObject`. Это место отлично подходит для инициализации ссылок на другие объекты `GameObject`.
- Метод **Input.GetMouseButtonDown** возвращает `true`, если кнопка мыши нажата в данный момент.
- Метод **Physics.Raycast** выполняет отслеживание лучей, для чего он проводит виртуальный луч по сцене и возвращает `true`, если луч столкнулся с каким-либо объектом. Для возвращения информации об объектах, с которыми столкнулся луч, используется ключевое слово `out`.
- Метод **ScreenPointToRay** камеры создает луч, проходящий через точку экрана. Объедините его с `Physics.Raycast`, чтобы определить, куда следует переместить игрока.

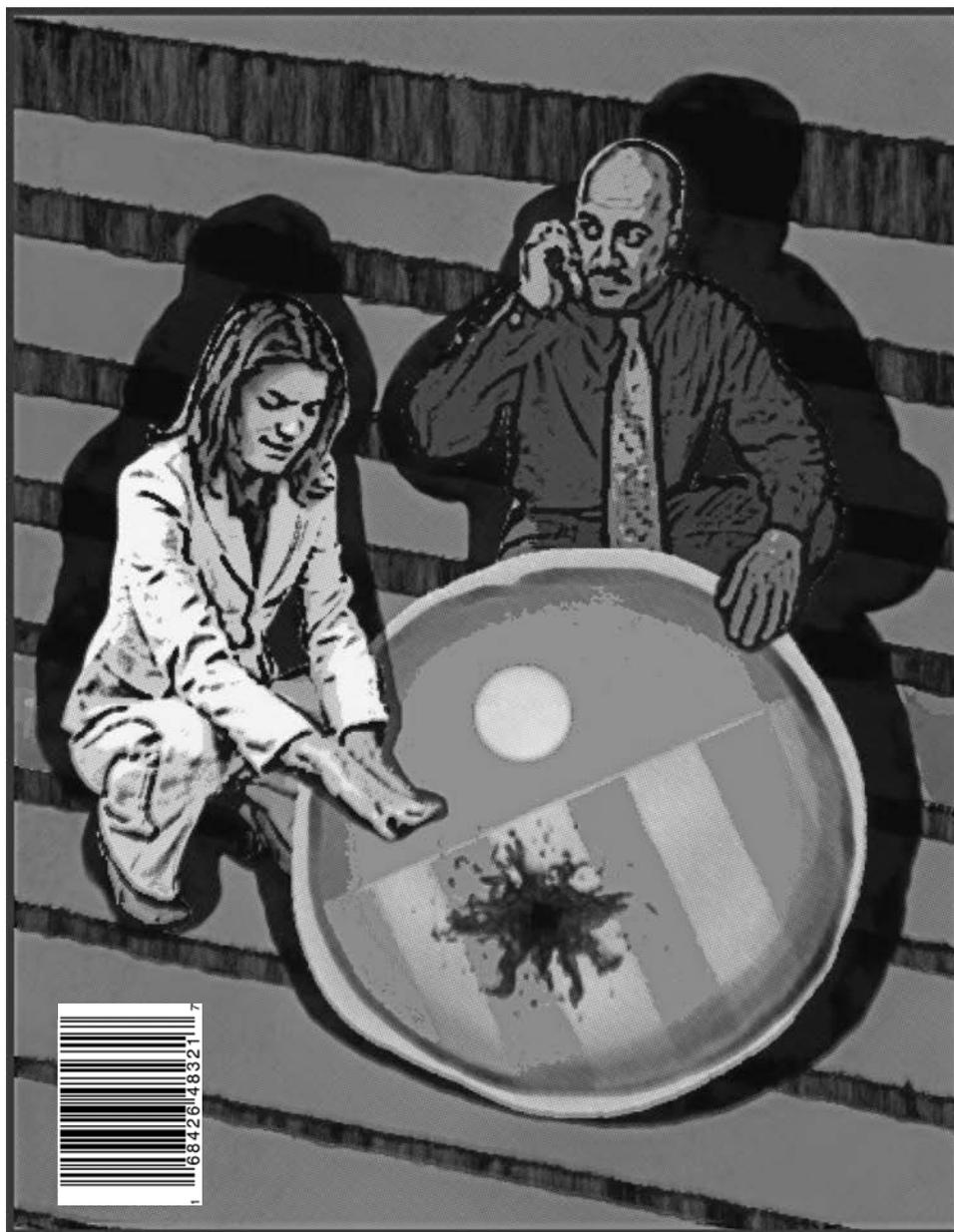
КАПИТАН ВЕЛИКОЛЕПНЫЙ

СМЕРТЬ ОБЪЕКТА

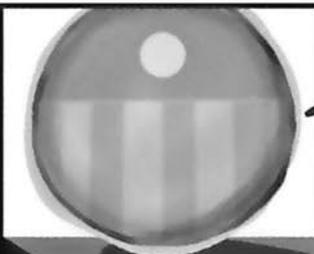
Head First C#

Четыре
доллара

Глава
11



КАПИТАН ВЕЛИКОЛЕТНЫЙ, САМЫЙ ВЫДАЮЩИЙСЯ ОБЪЕКТ ОБЪЕКТИВИА, ПРЕСЛЕДУЕТ СВОЕГО ВРАГА...

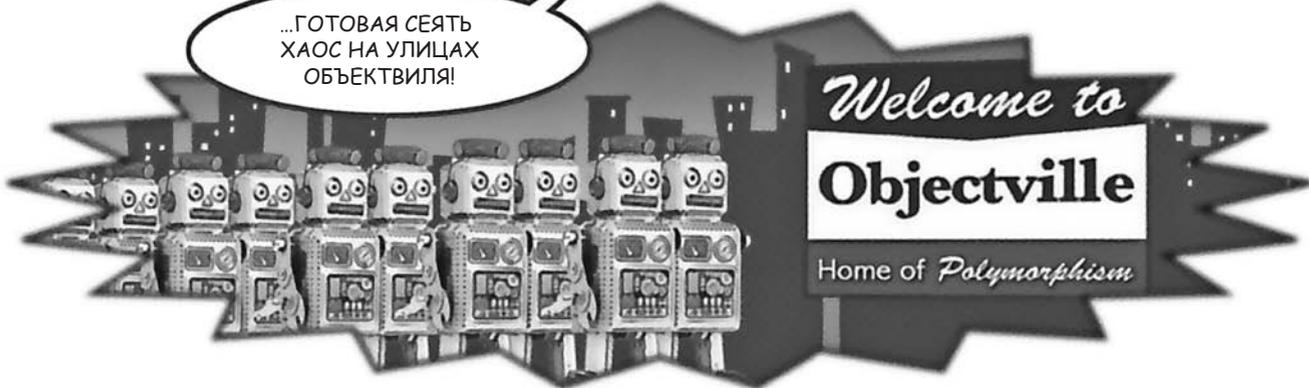


ТЕБЕ НЕ УЙТИ,
ПРОЙДОХА



ТЫ ОПОЗДАЛ! ПТОКА МЫ ГОВОРИМ,
НА ФАБРИКЕ ПОД НАМИ СОБИРАЕТСЯ МОЯ АРМИЯ КЛОНОВ...

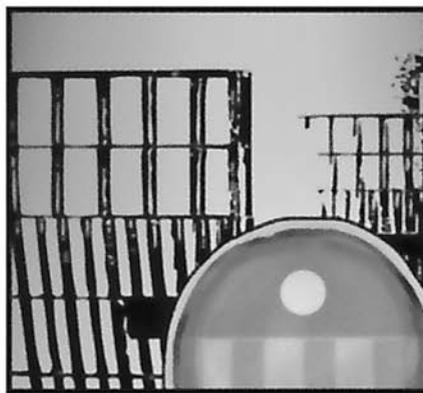
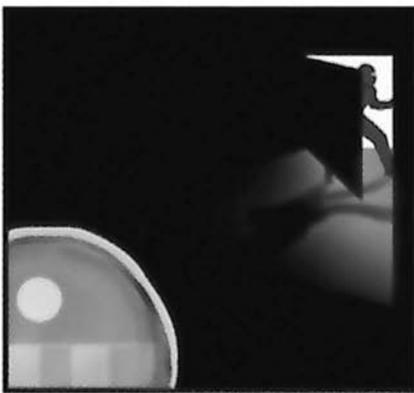
...ГОТОВАЯ СЕЯТЬ
ХАОС НА УЛИЦАХ
ОБЪЕКТИВИА!



БАХ!!

Я УНИЧОЖУ ССЫЛКИ
КЛОНОВ ОДНУ ЗА
ОДНОЙ!





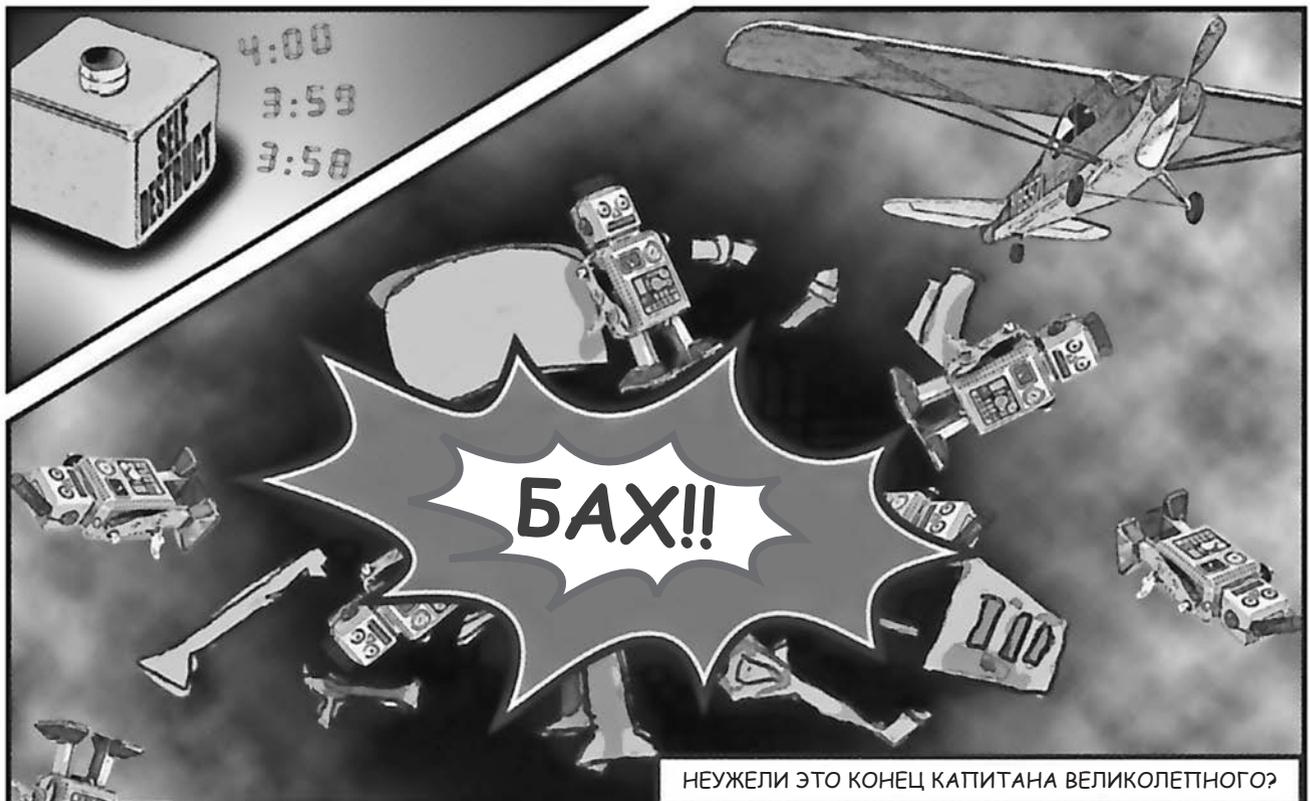
КАПИТАН ВЕЛИКОЛЕТНЫЙ
ЗАГНАЛ ТРОЙДОХУ В УГОЛ...



...НО САМ ПОПАДАЕТ
В ЛОВУШКУ.



ЧЕРЕЗ
НЕСКОЛЬКО МИНУТ И ТЫ,
И МОЯ АРМИЯ СТАНЕТЕ
МУСОРОМ... БОЙТЕСЬ СБОР-
ЩИКА МУСОРА!



НЕУЖЕЛИ ЭТО КОНЕЦ КАПИТАНА ВЕЛИКОЛЕТНОГО?

Жизнь и смерть объекта

Ниже приведен краткий обзор того, что мы знаем о жизни и смерти объектов:

- * При создании объекта среда CLR, которая запускает ваши приложения .NET и управляет памятью, выделяет достаточный объем памяти в куче (специальной области памяти компьютера, зарезервированной для объектов и данных).
- * Объект продолжает «жить» благодаря ссылкам на него, которые могут храниться в переменных, коллекциях, свойствах или полях других объектов.
- * На один объект может существовать много ссылок — как было показано в главе 4, в которой ссылочные переменные lloyd и lucinda указывают на один экземпляр Elephant.
- * Когда последняя ссылка на объект Elephant перестает существовать, CLR помечает объект для уничтожения в ходе сборки мусора.
- * Со временем CLR уничтожает объект Elephant и освобождает занимаемую им память, чтобы она могла использоваться для новых объектов, которые будут создаваться позднее.

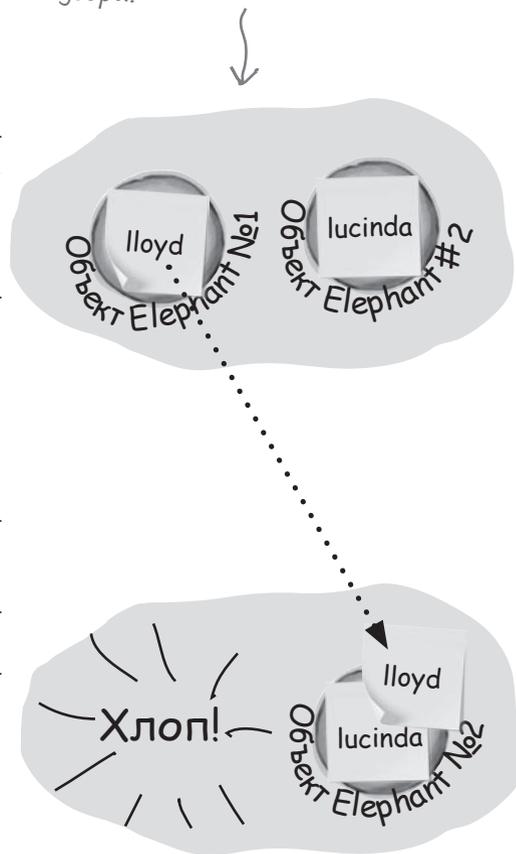
Сейчас все эти факты будут исследованы более подробно. Мы напишем маленькие программы, демонстрирующие, как работает сборка мусора.

Но прежде чем начать эксперименты со сборкой мусора, необходимо сделать шаг назад. Ранее вы узнали, что объекты «помечаются» для сборки мусора, но фактическое удаление объекта может произойти в любой момент (а может не произойти никогда!). Необходимо каким-то образом узнать, когда объект уничтожается сборщиком мусора, а также принудительно запустить процедуру сборки мусора. С этого и начнем.



— У МЕНЯ... ОСТАЛОСЬ...
— ОХ! —
ОДНО... ПОСЛЕДНЕЕ... ДЕЛО...

Диаграмма из главы 4. Мы создали в куче два объекта Elephant, а затем удалили ссылку на один из объектов, чтобы пометить его для сборки мусора. Но что это означает в действительности? Что выполняет сборку мусора?



Для принудительной сборки мусора используйте класс GC (осторожно!)

.NET предоставляет в ваше распоряжение класс GC, управляющий сборкой мусора. Мы будем использовать его статические методы, например метод `GetTotalMemory`, возвращающий значение `long` с *приблизительным* количеством байтов, выделенных в куче (*предположительно*):

```
Console.WriteLine(GC.GetTotalMemory(false));
```

```
class System.GC
Controls the system garbage collector, a service that automatically reclaims unused memory.
```

Возможно, вы подумали: «Почему *приблизительным*? И что значит *предположительно*? Как сборщик мусора может не знать, сколько именно памяти выделено?» В этой формулировке отражается одно из базовых правил сборки мусора: вы можете безусловно, полностью положиться на сборку мусора, но **существует много неизвестных и приближений**.

В этой главе используются некоторые функции GC:

- ★ `GC.GetTotalMemory` возвращает приблизительное количество байтов, предположительно выделенных в куче.
- ★ `GC.GetTotalAllocatedBytes` возвращает приблизительное количество байтов, выделенных с момента запуска программы.
- ★ `GC.Collect` заставляет сборщик мусора немедленно освободить все объекты, на которые нет ни одной ссылки.

По поводу этих методов необходимо сделать важное замечание: мы используем их для обучения и исследований, но использовать их следует только в том случае, если вы *действительно* хорошо знаете, что делаете. Если не знаете — **не вызывайте `GC.Collect` в реальном проекте**. Сборщик мусора .NET является хорошо оптимизированным, тщательно откалиброванным продуктом инженерной мысли. Как правило, когда заходит речь об определении того, когда должны удаляться объекты, сборщик мусора умнее нас, и нам стоит просто верить, что он выполнит свою работу.

Частво Задаваемые Вопросы

В: У меня... есть вопросы. что... как бы это выразиться... что такое Капитан Великолепный?

О: Капитан Великолепный — самый замечательный объект в мире, супергерой и защитник мирных граждан Объеквила, друг мелких зверушек во всем мире.

А если говорить более конкретно, Капитан Великолепный является антропоморфным объектом, вдохновленным одним из самых важных комиксов XXI века, посвященных смерти супергероя, а именно комиксом, вышедшим в 2007 году, когда мы работали над черновиком `Head First C#` и искали хороший подход к обсуждению жизни и смерти объектов. Мы заметили поразительное сходство между формой объектов на диаграммах структуры кучи и щитом одного знаменитого Капитана из комиксов... Так родился Капитан Великолепный. (Если вы не любитель комиксов, не огорчайтесь: вам не обязательно что-то знать о комиксах, чтобы понять материал этой главы.)

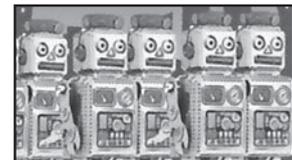
Обычный скромный объект.



Капитан Великолепный, самый знаменитый объект в мире.

В: Почему «клоны» похожи на роботов? Разве они не должны выглядеть как люди?

О: Да, в своем комиксе мы придали клонам следующий вид:



Потому что мы не хотели включать графические изображения уничтожаемых людей. В общем, *расслабьтесь*. Комиксы в этой главе помогают закрепить в вашем мозгу важнейшие концепции `C#` и `.NET`. История — всего лишь инструмент, который помогает провести ряд полезных аналогий.

Финализатор объекта — последний шанс что-то сделать

Иногда бывает нужно гарантировать, что некая операция (например, **освобождение неуправляемых ресурсов**) будет выполнена до того, как объект будет уничтожен сборщиком мусора.

Специальный метод, называемый **финализатором** (finalizer), позволяет написать код, который всегда будет выполняться при уничтожении объекта. Код финализатора выполняется последним, что бы ни произошло.

Немного поэкспериментируем с финализаторами. **Создайте новое консольное приложение** и добавьте класс с финализатором:

```
class EvilClone
{
    public static int CloneCount = 0;
    public int CloneID { get; } = ++CloneCount;

    public EvilClone() => Console.WriteLine("Clone #{0} is wreaking havoc", CloneID);

    ~EvilClone()
    {
        Console.WriteLine("Clone #{0} destroyed", CloneID);
    }
}
```

В общем случае никогда не следует писать финализатор для объекта, который владеет только управляемыми ресурсами. Всеми ресурсами, которые встречались вам в книге до настоящего момента, управляла CLR. Однако в отдельных случаях программистам приходится обращаться к ресурсам Windows, не входящим в пространство имен .NET. Например, если в интернете вам встретится код, в котором перед объявлением стоит директива [DllImport], возможно, вы используете неуправляемый ресурс. Некоторые из этих ресурсов, не находящихся под управлением .NET, при отсутствии необходимой «зачистки» могут оставить систему в нестабильном состоянии. Финализаторы нужны именно для таких ситуаций.

Если оператор ++ находится перед именем переменной, то переменная инкрементируется до выполнения команды. Как вы думаете, почему мы так поступили?

Это финализатор (иногда называемый деструктором). Он объявляется как метод, имя которого начинается с тильды (~); финализатор не может иметь ни возвращаемого значения, ни параметров. Финализатор объекта выполняется непосредственно перед уничтожением объекта сборщиком мусора.

Метод Main создает экземпляры EvilClone, *разыменовывает* их (удаляет ссылки на них) и уничтожает объекты при помощи сборщика мусора:

```
class Program
{
    static void Main(string[] args)
    {
        var stopwatch = System.Diagnostics.Stopwatch.StartNew();
        var clones = new List<EvilClone>();
        while (true)
        {
            switch (Console.ReadKey(true).KeyChar)
            {
                case 'a':
                    clones.Add(new EvilClone());
                    break;
                case 'c':
                    Console.WriteLine("Clearing list at time {0}", stopwatch.ElapsedMilliseconds);
                    clones.Clear();
                    break;
                case 'g':
                    Console.WriteLine("Collecting at time {0}", stopwatch.ElapsedMilliseconds);
                    GC.Collect();
                    break;
            }
        }
    }
}
```

Класс Stopwatch поможет получить представление о том, насколько быстро выполняется сборка мусора. Класс Stopwatch точно измеряет затраченное время; для этого он запускает отсчет времени и возвращает количество миллисекунд, прошедших с момента запуска.

При нажатии клавиши 'a' приложение создает новый экземпляр EvilClone и добавляет его в список clones.

Нажатие 'c' приказывает приложению очистить список. При этом удаляются все ссылки на клоны, которые были созданы и добавлены в список.

Нажатие 'g' приказывает CLR уничтожить все объекты, помеченные для сборки мусора.

Запустите приложение и несколько раз нажмите **a**, чтобы создать несколько объектов EvilClone и добавить их в List. Затем нажмите **c**, чтобы очистить List и удалить все ссылки на объекты EvilClone. Нажмите **c** несколько раз — существует *незначительная* вероятность того, что CLR уничтожит некоторые из разыменованных объектов, но *скорее всего*, они будут уничтожены только тогда, когда вы нажмете **g** для вызова GC.Collect.

Когда ИМЕННО выполняется финализатор?

Финализатор объекта выполняется **после** того, как перестанут существовать все ссылки на объект, но **до** уничтожения объекта сборщиком мусора. Сборка мусора происходит только после освобождения **всех** ссылок на объект, но она не всегда происходит *сразу же* после исчезновения последней ссылки. Предположим, имеется объект, и на этот объект существует ссылка. CLR отправляет сборщика мусора на работу, тот проверяет ваш объект. Так как объект используется (на него существует ссылка), сборщик мусора игнорирует его и двигается дальше. Объект продолжает находиться в памяти. Затем что-то происходит. Последний объект, содержащий ссылку на *ваш* объект, удаляет эту ссылку. Теперь ваш объект остается в памяти без единой ссылки. К нему невозможно обратиться. По сути, это **мертвый объект**.

Но тут возникает нюанс: *сборкой мусора управляет CLR*, а не ваши объекты. Таким образом, если сборщик мусора не активизируется снова, скажем, через несколько секунд или даже минут, ваш объект продолжит оставаться в памяти. Он не может использоваться, но еще не был уничтожен сборщиком мусора. **И финализатор объекта (еще) не может быть выполнен.**

Наконец, CLR снова отправляет сборщика мусора на работу. Он проверяет ваш объект, обнаруживает, что ссылок на него не осталось, и выполняет финализатор... возможно, через несколько минут после удаления или изменения последней ссылки на объект. После выполнения финализатора объект официально считается мертвым, и сборщик мусора уничтожает его.

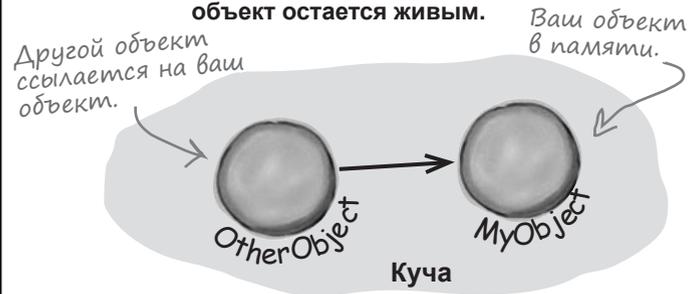
Вы можете ПОРЕКОМЕНДОВАТЬ .NET выполнить сборку мусора

.NET позволяет *порекомендовать* провести сборку мусора. **В большинстве случаев этот метод лучше не использовать, потому что сборка мусора настроена так, чтобы она реагировала на многие условия в CLR, и пытаться запускать ее напрямую нежелательно.** Но просто чтобы увидеть, как работает финализатор, можно активизировать сборку мусора самостоятельно при помощи метода GC.Collect.

Будьте внимательны. Метод не *заставляет* CLR немедленно провести сборку мусора. Он просто говорит: «Выполни сборку мусора при первой возможности».

Жизнь и смерть объекта на диаграмме

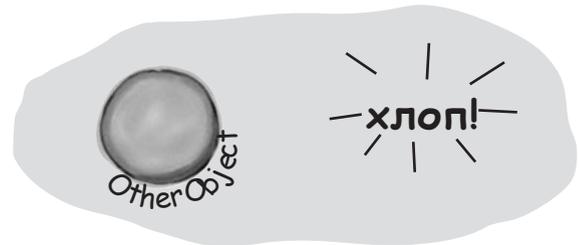
1. Ваш объект благополучно существует в куче. Другой объект хранит ссылку на него, поэтому ваш объект остается живым.



2. Другой объект изменяет свою ссылку. Теперь не остается других объектов, ссылающихся на ваш объект.



3. CLR помечает объект для сборки мусора.



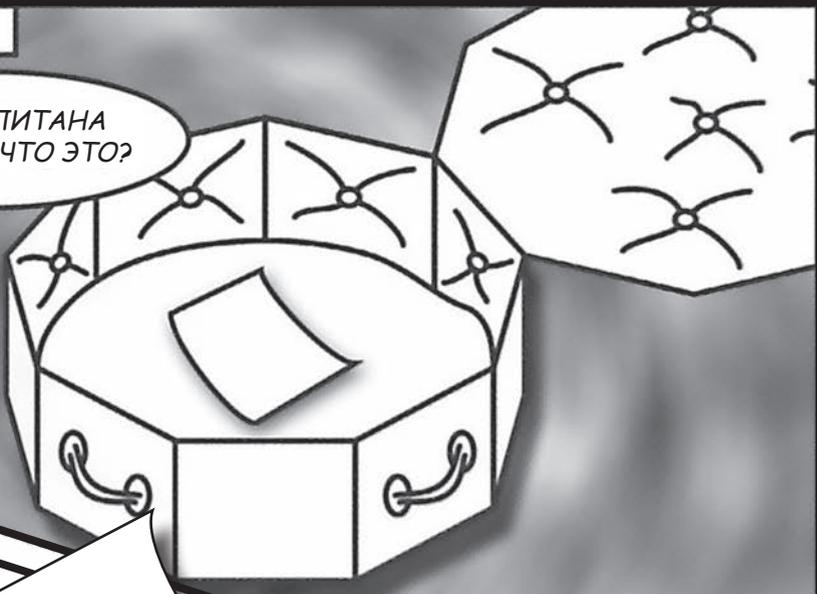
4. Через какое-то время сборщик мусора выполняет финализатор объекта и удаляет объект из кучи.

Мы используем метод GC.Collect как учебный инструмент, который помогает вам понять, как работает сборка мусора. Использовать его за пределами учебных программ определенно не стоит (если только вы не понимаете, как работает сборка мусора в .NET, на более глубоком уровне, чем мы рассматриваем в этой книге.)

ПОЗДНЕЕ В ПОХОРОННОМ БЮРО



ГРОБ КАПИТАНА
ПУСТ... НО ЧТО ЭТО?



6e 61 6d 65 73 70 61 63 65
20 51 7b 0d 0a 5b 53 65 72
69 61 6c 69 7a 61 62 6c 65
5d 70 75 62 6c 69 63 20 63
6c 61 73 73 20 4d 73 67 7b
0d 0a 70 75 62 6c 69 63 20
73 74 72 69 6e 67 20 61 3b
70 75 62 6c 69 63 20 73 74
6c 69 6e 67 20 62 3b 70 75 62
20 63 3b 70 75 62 6c 69 6e
69 6e 74 20 69 3b 0d 0a 70
62 6c 69 63 20 76 6f 69 64 73
53 68 6f 77 28 65 42 6f 78 2e 53 68
73 61 67 65 42 6f 78 2e 53 68
6f 77 28 63 2e 53 75 62 73 74
72 69 6e 67 28 31 2c 32 29

КАПИТАН НАШЕЛ
ВОЗМОЖНОСТЬ СЕРИАЛИ-
ЗОВАТЬ СВОЮ СУЩНОСТЬ
ПЕРЕД СМЕРТЬЮ?

Финализаторы не могут зависеть от других объектов

При написании финализатора нельзя полагаться на то, что он будет выполнен в какой-то конкретный момент. Даже при вызове GC.Collect вы только *рекомендуете* выполнить сборщика мусора. Нет никаких гарантий, что это произойдет немедленно. А когда это произойдет, вы не знаете, в каком порядке будут уничтожаться объекты.

Что это означает в практическом смысле? Представьте, что два объекта содержат ссылки друг на друга. Если объект 1 будет уничтожен первым, то ссылка объекта 2 указывает на несуществующий объект. Но если объект 2 уничтожается первым, то ссылка объекта 1 становится недействительной. Это означает, что *вы не можете зависеть от ссылок в финализаторе вашего объекта*. А следовательно, в финализаторе не стоит делать ничего, что зависело бы от действительности ссылок.

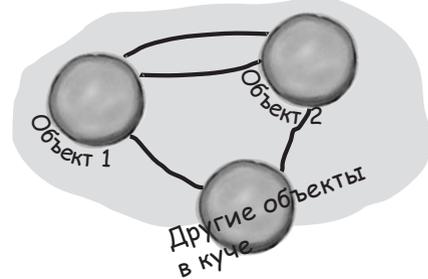
Не используйте финализаторы для сериализации

Сериализация — очень хороший пример того, что **не стоит делать в финализаторе**. Если ваш объект содержит ссылки на другие объекты, сериализация зависит от нахождения в памяти *всех* этих объектов... а также всех объектов, на которые они ссылаются, и объектов, на которые ссылаются эти объекты, и т. д. Таким образом, при попытке выполнения сериализации в процессе сборки мусора некоторые **важные части** программы могут оказаться недоступными, потому что какие-то объекты были уничтожены *до* выполнения финализатора.

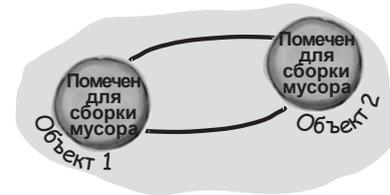
К счастью, C# предоставляет хорошее решение для подобных проблем: IDisposable. Все операции, которые могут изменить основные данные или зависят от нахождения в памяти других объектов, должны выполняться в методе Dispose, а не в финализаторе.

Некоторые разработчики рассматривают финализатор как своего рода «страховку» для метода Dispose. И это выглядит логично — вы видели на примере объекта Clone, что факт реализации IDisposable еще не гарантирует вызова метода Dispose. Однако вы должны действовать внимательно — если метод Dispose зависит от нахождения в куче других объектов, вызов Dispose из финализатора может создать проблемы. Лучшее решение всех проблем такого рода — **всегда использовать команду using** каждый раз, когда вы создаете объект IDisposable.

Вначале два объекта содержат ссылки друг на друга.



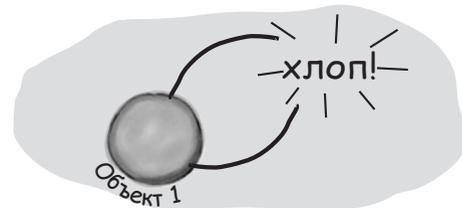
Если все остальные объекты в куче удаляют свои ссылки на объекты 1 и 2, оба объекта будут помечены для сборки мусора.



Если объект 1 уничтожается первым, то его данные будут недоступны в момент выполнения финализатора объекта 2 средой CLR.

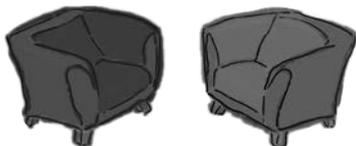


С другой стороны, объект 2 может исчезнуть раньше объекта 1. Порядок уничтожения объектов заранее неизвестен.



И именно по этой причине финализатор одного объекта не может зависеть от нахождения в куче любых других объектов.

Беседа у камина



Кто важнее: метод `Dispose` или финализатор?

Dispose:

Честно говоря, я немного удивлен, что меня сюда пригласили. Мне казалось, что программисты уже пришли к согласию. Имею в виду, что я просто намного ценнее для разработчиков `C#`, чем ты. Будем откровенны: ты просто слабоват. Ты даже не можешь рассчитывать на то, что другие объекты еще существуют на момент твоего вызова. Выходит, ты недостаточно надежен.

Интерфейс существует именно **потому**, что я важен. В нем вообще нет других методов!

Здесь ты прав, программист должен знать, когда он захочет воспользоваться моими услугами, и либо вызвать меня напрямую, либо воспользоваться командой для `using`. Но они всегда знают, когда я буду выполняться, и используют меня, чтобы сделать все необходимое для наведения порядка после освобождения объектов. Я мощен, надежен и прост в использовании. Тройная польза. А ты? Никто не знает, когда ты будешь выполняться или в каком состоянии окажется приложение после того, как ты наконец-то соизволишь показаться.

Ты думаешь, что ты большая шишка, потому что всегда выполняешься со сборщиком мусора, но по крайней мере я могу зависеть от других объектов.

Финализатор:

Прошу прощения? Это сильно сказано. Я «слабоват»? Что ж, я не хотел ввязываться в дискуссию, но если уж мы опустились до такого... прежде всего мне не нужен интерфейс. Давай признаем, что без интерфейса `IDisposable`... ты просто обычный бесполезный метод.

Конечно, конечно... продолжай убеждать себя в этом. И что произойдет, если кто-нибудь забудет использовать команду `using` при создании экземпляра? Раз — и тебя вообще нет.

Дескрипторы — то, что используют ваши программы при прямом взаимодействии с системой Windows. Так как среда .NET о них ничего не знает, она не сможет освободить их за вас.

Но если нужно сделать что-то в самый последний момент, непосредственно перед уничтожением объекта сборщиком мусора, без меня не обойтись. Я могу освобождать сетевые ресурсы и дескрипторы Windows... и вообще все, что необходимо освободить, чтобы не возникли проблемы. Я могу гарантировать, что объекты будут уничтожаться более корректно, и это не игрушки.

Да, это верно — но я выполняюсь всегда. Тебе необходим кто-то другой, кто тебя вызовет. А мне не нужен никто и ничего!



Часть Задаваемые Вопросы

В: Может ли финализатор использовать все поля и методы объекта?

О: Да. Хотя метод-финализатор не может получать параметры, вы можете использовать любые его поля в объекте, напрямую или при помощи `this`, но будьте внимательны, потому что если эти поля ссылаются на другие объекты, то другие объекты могут быть уже уничтожены сборщиком мусора. Таким образом, ваш финализатор может вызывать другие методы или свойства объекта... при условии, что эти методы и свойства *не зависят от других объектов*.

В: Что происходит при выдаче исключений в финализаторе?

О: Хороший вопрос. Исключения в финализаторах работают точно так же, как в любой другой точке вашего кода. Попробуйте заменить финализатор `EvilClone` другим, в котором выдается исключение:

```
~EvilClone() => throw new Exception();
```

Затем снова запустите приложение, создайте несколько экземпляров `EvilClone`, очистите список и запустите сборщик мусора. Ваше приложение прервется в финализаторе, как и при возникновении любых других исключений. (Спойлер: в следующей главе вы научитесь *перехватывать* исключения, т. е. обнаруживать факт их возникновения и выполнять код обработки.)

В: С какой частотой сборщик мусора выполняется автоматически?

О: Простой ответ: неизвестно. Сборщик мусора не выполняется с какой-то прогнозируемой периодичностью и не находится под контролем разработчика. Вы можете быть уверены только в том, что он будет выполнен при нормальном выходе из программы. Даже при вызове `GC.Collect` (чего обычно делать не рекомендуется) вы только *рекомендуете CLR* выполнить сборку мусора.

В: Через какое время после вызова `GC.Collect` начнется сборка мусора?

О: При вызове `GC.Collect` вы сообщаете `.NET`, что сборка мусора должна быть выполнена как можно скорее. **Обычно** это происходит тогда, когда `.NET` сделает то, чем занимается в настоящий момент. Скорее всего, это произойдет достаточно скоро, но точно управлять началом сборки мусора вы не можете.

В: Если какой-то код должен быть выполнен абсолютно точно, я помещаю его в финализатор?

О: Нет. Может оказаться, что финализатор выполнен не будет. Выполнение финализаторов при сборке мусора может быть заблокировано. Или может завершиться весь процесс. Если только вы не освобождаете управляемые ресурсы в своем коде, почти всегда лучше использовать `IDisposable` и команды `using`.

ТЕМ ВРЕМЕНЕМ НА УЛИЦАХ ОБЪЕКТИВЛЯ...

КАПИТАН
ВЕЛИКОЛЕТНЬИЙ...
ОН ВЕРНУЛСЯ!

С НИМ ЧТО-ТО
НЕ ТАК. ОН НЕ ПОХОЖ
НА СЕБЯ ПРЕЖНЕГО... И СО
СВЕРХСИЛАМИ ТВОРИТСЯ
ЧТО-ТО СТРАННОЕ.

КАПИТАНУ ВЕЛИКОЛЕТНЬИМУ
ПОНАДОБИЛОСЬ СТОЛЬКО ВРЕМЕНИ,
ЧТОБЫ ДОБРАТЬСЯ СЮДА, ЧТО
ПУШИСТИК САМ СЛЕЗ
С ДЕРЕВА.

МЯУ!

ПОЗЖЕ...

ЕЩЕ ПОЗЖЕ...

ТАЙНОЕ УБЕЖИЩЕ
КАПИТАНА
ВЕЛИКОЛЕТНОГО
СОВЕРШЕННО
СЕКРЕТНО

УФ... ПЫХ... ФУ...
МНЕ НАДО
ПЕРЕВЕСТИ ДУХ.

ЧТО ПРОИЗОШЛО? ПО-
ЧЕМУ СИЛЫ КАПИТАНА
ВЕЛИКОЛЕТНОГО РАБО-
ТАЮТ НЕ ТАК, КАК ПРЕ-
ЖДЕ? ЭТО КАТАСТРОФА?

Структура похожа на объект...

Мы говорили о куче, в которой существуют ваши объекты. Тем не менее это не единственная часть памяти, в которой могут размещаться объекты. Один из типов .NET, не упоминавшийся до настоящего момента, — *структуры* — поможет нам разобраться с другими аспектами жизни и смерти в C#. Структуры (ключевое слово `struct`) очень похожи на объекты. У них, как и у объектов, могут быть поля и свойства. Их даже можно передавать методам, которые получают параметр объектного типа:

```
public struct AlmostSuperhero : IDisposable {
    private bool superStrength;
    public int SuperSpeed { get; private set; }

    public void RemoveVillain(Villain villain)
    {
        Console.WriteLine("OK, {0}, surrender now!",
            villain.Name);
        if (villain.Surrendered)
            villain.GoToJail();
        else
            villain.StartEpicBattle();
    }

    public void Dispose() => Console.WriteLine("Nooooooo!");
}
```

Структуры могут реализовать интерфейсы, но не могут расширять другие классы. Кроме того, структуры не могут расширяться.

Структуры могут содержать поля и свойства...

...и определять методы.

Структуры даже могут реализовать интерфейсы (например, `IDisposable`).

...но не является объектом

Однако структуры *не являются* объектами. Они могут содержать методы и поля, но *не могут* иметь финализаторы. Они также не могут наследовать от классов и других структур, и другие классы или структуры не могут наследовать от них: оператор `:` может использоваться в объявлении структуры, но за ним могут следовать только один или несколько интерфейсов.

Все структуры расширяют класс `System.ValueType`, который, в свою очередь, расширяет `System.Object`. Вот почему каждая структура поддерживает метод `ToString` — он получает его от `Object`. Но это единственное наследование, разрешенное для структур.

Структуры не могут наследовать от других объектов.



Автономный объект можно смоделировать в виде структуры, но структуры плохо подходят для сложных иерархий с наследованием.

Мощь объектов связана с их способностью моделировать поведение сущностей реального мира посредством наследования и полиморфизма. Структуры лучше всего подходят для хранения данных, но отсутствие наследования и ссылок может стать серьезным ограничением.

Значения копируются, ссылки присваиваются

Вы уже видели, какую важную роль ссылки играют в области сборки мусора, — стоит вам присвоить новое значение вместо последней ссылки на объект, и объект помечается для сборки мусора. Но мы также знаем, что эти правила не имеют смысла для значений. Если вы хотите лучше понять, как объекты и значения живут и умирают в памяти CLR, необходимо повнимательнее присмотреться к значениям и ссылкам: чем они похожи и, что важнее, — чем они отличаются.

Вы уже примерно представляете, чем одни типы отличаются от других. С одной стороны, существуют **типы значений**, такие как `int`, `bool` и `decimal`. С другой стороны, существуют **объекты**, такие как `List`, `Stream` и `Exception`. И работают они совсем не одинаково, верно?

Когда вы используете оператор `=` для того, чтобы присвоить одну переменную типа значения другой, **создается копия значения**, и в дальнейшем две переменные никак не связаны друг с другом. С другой стороны, когда вы используете оператор `=` со ссылками, **обе ссылки будут указывать на один объект**.

- 1 Объявления переменных и присваивания работают для типов значений и объектных типов одинаково:

```
int howMany = 25;
bool Scary = true;
List<double> temps = new List<double>();
throw new NotImplementedException();
```

← *int и bool являются типами значений, List и Exception — объектные типы.*

- 2 Но когда вы начинаете присваивать значения, проявляются различия между ними. Для всех типов значений используется копирование. В следующем примере вам все должно быть знакомо:

Изменение переменной fifteenMore не влияет на howMany, и наоборот.

```
int fifteenMore = howMany;
fifteenMore += 15;
Console.WriteLine("howMany has {0}, fifteenMore has {1}",
    howMany, fifteenMore);
```

← *Эта строка копирует значение, хранящееся в переменной fifteenMore, в переменную howMany и увеличивает его на 15.*

Результат ясно показывает, что переменные `fifteenMore` и `howMany` *не* связаны:

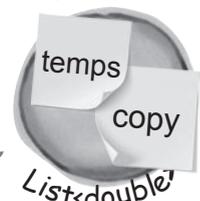
→ `howMany has 25, fifteenMore has 40`

- 3 Но как вы уже знаете, при работе с объектами присваиваются ссылки, а не значения:

После выполнения этой строки переменная copy ссылается на тот же объект, что и переменная temps.

```
temps.Add(56.5D);
temps.Add(27.4D);
List<double> copy = temps;
copy.Add(62.9D);
```

→ *Обе ссылки указывают на один объект.*



Таким образом, изменение `List` означает, что обновление будет видимым по обоим ссылкам, так как они обе указывают на один объект `List`. Чтобы убедиться в этом, выведите следующую строку:

```
Console.WriteLine("temps has {0}, copy has {1}", temps.Count(), copy.Count());
```

Следующий результат демонстрирует, что `copy` и `temps` действительно указывают на **один и тот же** объект:

`temps has 3, copy has 3`

← *При вызове copy.Add метод добавил новую температуру к объекту, на который указывают как copy, так и temps.*

Структуры относятся к типам значений; объекты относятся к ссылочным типам

Присмотримся повнимательнее к тому, как работают структуры, чтобы вы хотя бы в общих чертах представляли, в каких случаях стоит использовать структуру вместо объекта. При создании структуры вы создаете **тип значения**. Это означает, что, присваивая одну структурную переменную другой, вы создаете новую копию структуры в новой переменной. Итак, хотя структура *похожа* на объект, она не работает как объект.

1 Создайте структуру с именем Dog.

Ниже приведена простая структура для хранения информации о собаке. Она выглядит как объект, но объектом не является. Добавьте ее в **новое консольное приложение**.

```
public struct Dog {

    public string Name { get; set; }
    public string Breed { get; set; }

    public Dog(string name, string breed) {
        this.Name = name;
        this.Breed = breed;
    }

    public void Speak() {
        Console.WriteLine("My name is {0} and I'm a {1}.", Name, Breed);
    }
}
```

Сделайте это!

2 Создайте класс с именем Canine.

Создайте точную копию структуры Dog, но **замените struct на class**, а затем **замените Dog на Canine**. Не забудьте переименовать конструктор Dog. Теперь у вас имеется *класс* Canine, с которым можно экспериментировать; он почти эквивалентен *структуре* Dog.

3 Добавьте метод Main, в котором создаются копии данных Dog и Canine.

Код метода Main выглядит так:

```
Canine spot = new Canine("Spot", "pug");
Canine bob = spot;
bob.Name = "Spike";
bob.Breed = "beagle";
spot.Speak();
Dog jake = new Dog("Jake", "poodle");
Dog betty = jake;
betty.Name = "Betty";
betty.Breed = "pit bull";
jake.Speak();
```

MINI Возьми в руку карандаш

4 Прежде чем запускать программу...

Напишите, что, по-вашему мнению, будет выведено на консоль при выполнении следующего кода:

```
.....
.....
```



Возьми в руку карандаш

Как вы думаете, что будет выведено на консоль?

Решение

My name is Spike and I'm a beagle.

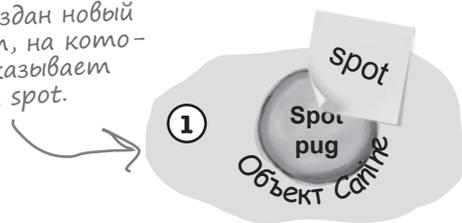
My name is Jake and I'm a poodle.

Вот что произошло...

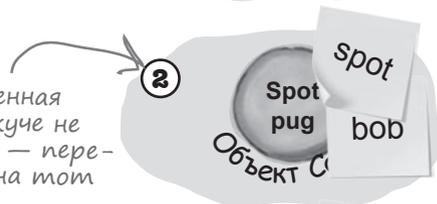
Ссылки bob и spot указывают на один и тот же объект, поэтому по обеим ссылкам изменяются одни и те же поля и вызывается один и тот же метод Speak. Структуры работают не так. При создании betty была создана новая копия данных из jake. Две структуры полностью независимы друг от друга.

```
Canine spot = new Canine("Spot", "pug"); ①
Canine bob = spot; ②
bob.Name = "Spike";
bob.Breed = "beagle"; ③
spot.Speak();
```

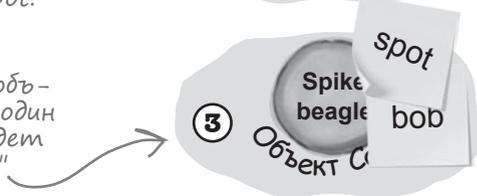
Был создан новый объект, на который указывает ссылка spot.



Новая ссылочная переменная bob была создана, но в куче не появился новый объект — переменная bob указывает на тот же объект, что и spot.



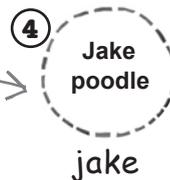
Так как spot и bob указывают на один объект, spot.Speak и bob.Speak вызывают один и тот же метод, и в обоих случаях будет получен один и тот же вывод со "Spike" и "beagle".



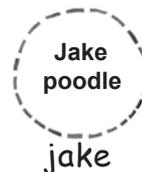
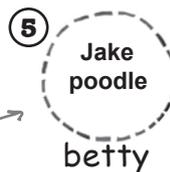
```
Dog jake = new Dog("Jake", "poodle"); ④
Dog betty = jake; ⑤
betty.Name = "Betty"; ⑥
betty.Breed = "pit bull";
jake.Speak();
```

Присваивая одну структуру другой, вы создаете новую **КОПИЮ** данных, содержащихся в структуре. Дело в том, что структура относится к **ТИПУ ЗНАЧЕНИЯ** (а не к объектному/ссылочному типу).

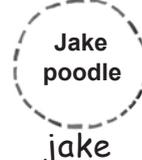
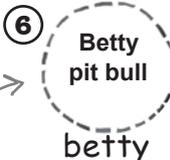
Когда вы создаете новую структуру, это очень похоже на создание объекта — вы получаете переменную, которая может использоваться для обращения к полям и методам.



И это очень серьезное различие. Добавляя переменную betty, вы создаете новое значение.



Так как вы создали новую копию данных, изменение полей betty не отразилось на jake.





За сценой

Помните: во время выполнения вашей программы CLR активно управляет памятью, работает с кучей и собирает мусор.

Стек и куча: подробнее о памяти

Давайте быстро вспомним, чем структура отличается от объекта. Вы уже видели, что новую копию структуры можно создать простым присваиванием, тогда как с объектами это невозможно. Но что при этом происходит за кулисами?

CLR распределяет данные между двумя областями памяти: кучей и стеком. Вы уже знаете, что объекты существуют в **куче**. CLR также резервирует другую часть памяти, называемую **стеком**; в ней сохраняются локальные переменные, объявленные в методе, и параметры, передаваемые этим методам. Стек можно рассматривать как набор ячеек, в которых могут сохраняться значения. При вызове метода CLR добавляет новые ячейки на вершину стека. При возврате управления ячейки вызванного метода удаляются.

Ког

Пример кода, который вам может встретиться в программе.

```
Canine spot = new Canine("Spot", "pug");
Dog jake = new Dog("Jake", "poodle");
```

Так будет выглядеть стек после выполнения этих двух строк.

Стек

Здесь хранятся структуры и локальные переменные.



```
Canine spot = new Canine("Spot", "pug");
Dog jake = new Dog("Jake", "poodle");
Dog betty = jake;
```

При создании новой структуры — или любой другой переменной, относящейся к типу значения, — в стек добавляется новая «ячейка». Она содержит копию значения соответствующего типа.

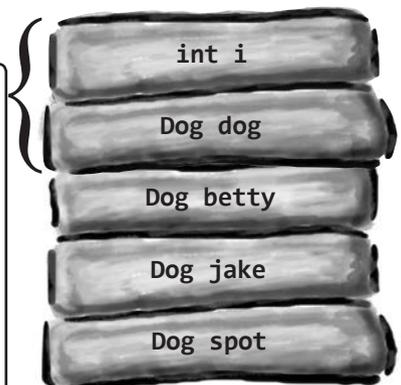


```
Canine spot = new Canine("Spot", "pug");
Dog jake = new Dog("Jake", "poodle");
Dog betty = jake;
SpeakThreeTimes(jake);
```

```
public SpeakThreeTimes(Dog dog) {
    int i;
    for (i = 0; i < 5; i++)
        dog.Speak();
}
```

При вызове метода CLR помещает его локальные переменные на вершину стека. В данном случае код вызывает метод `SpeakThreeTimes`. Метод получает один параметр (`dog`) и создает одну переменную (`i`); CLR сохраняет их в стек.

При возврате управления из метода CLR извлекает `i` и `dog` из стека — так проходит жизнь и смерть значений в CLR.





Теория — это, конечно, здорово, но я уверен, что у всех этих разговоров о стеке, куче, значениях и ссылках есть и практические причины.

Важно понимать, чем структура, копируемая по значению, отличается от объекта, копируемого по ссылке.

В отдельных случаях требуется написать метод, который может получать тип значения *или* ссылочный тип, — например, метод, способный работать со структурой Dog или с объектом Canine. Если вы окажетесь в такой ситуации, используйте ключевое слово `object`:

```
public void WalkDogOrCanine(object getsWalked) { ... }
```

Если передать такому методу структуру, эта структура будет **упакована** в специальный объект-«обертку», которая позволяет ей существовать в куче. Пока обертка находится в куче, вы мало что сможете сделать со структурой. Необходимо «распаковать» обертку, чтобы работать со структурой. К счастью, все это происходит *автоматически* при присваивании объекту типа значения или передаче типа значения методу, который ожидает получить объект.

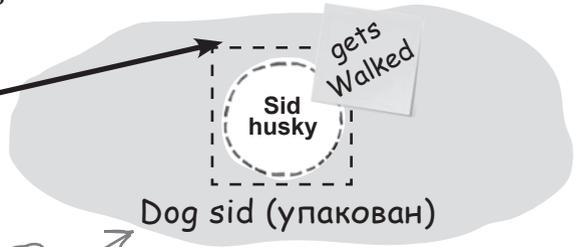
Также можно воспользоваться ключевым словом `<is>`, чтобы узнать, является ли объект структурой или любым другим типом значения, упакованным и помещенным в кучу.

1 Так выглядят стек и куча после создания объектной переменной и присваивания ей структуры Dog.

```
Dog sid = new Dog("Sid", "husky");
WalkDogOrCanine(sid);
```



После того как структура будет упакована, существуют две копии данных: копия в стеке и копия, упакованная в куче.



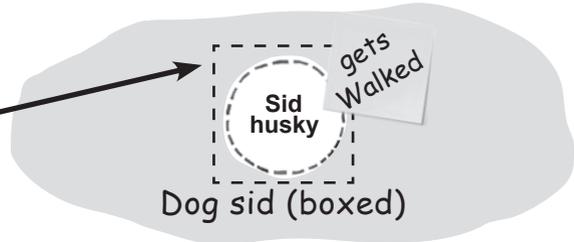
The `WalkDogOrCanine` получает ссылку на объект, поэтому структура Dog была упакована до ее передачи. После приведения ее обратно к типу Dog происходит ее распаковка.

2 Если вы хотите **распаковать объект**, для этого достаточно преобразовать его к правильному типу; объект будет распакован автоматически. Ключевое слово `is` нормально работает со структурами, но будьте осторожны, потому что ключевое слово `as` не работает с типами значений.

```
if (getsWalked is Dog doggo) doggo.Speak();
```



После выполнения этой строки появляется третья копия данных в новой структуре с именем `doggo`, которой назначается отдельная ячейка в стеке.





При вызове метод ищет свои аргументы в стеке.

Стек играет важную роль в управлении данными приложения средой CLR. При этом мы приняли как нечто само собой разумеющееся, что мы можем написать метод, вызывающий другой метод, который, в свою очередь, вызывает другой метод. Более того, метод даже может вызывать самого себя (это называется *рекурсией*). Именно стек предоставляет нашим программам такую возможность.

Ниже приведены три метода из программы, моделирующей поведение собаки. Метод `FeedDog` вызывает метод `Eat`, `gn` вызывает метод `CheckBowl`.

Запомните терминологию: параметр задает значение, необходимое методу; аргумент — это фактическое значение или ссылка, передаваемые методу при вызове.

Так будет выглядеть стек при вызове `Eat` из `FeedDog`; метод вызывает `CheckBowl`, который, в свою очередь, вызывает `Console.WriteLine`.

```
public double FeedDog(Canine dogToFeed, Bowl dogBowl) {
    double eaten = Eat(dogToFeed.MealSize, dogBowl);
    return eaten + .05D; // Небольшая часть всегда теряется.
}

public void Eat(double mealSize, Bowl dogBowl) {
    dogBowl.Capacity -= mealSize;
    CheckBowl(dogBowl.Capacity);
}

public void CheckBowl(double capacity) {
    if (capacity < 12.5D) {
        string message = "My bowl's almost empty!";
        Console.WriteLine(message);
    }
}
```



Метод `FeedDog` получает два параметра, ссылку на `Canine` и ссылку на `Bowl`. Таким образом, при вызове в стеке оказываются два переданных аргумента.

`FeedDog` передает два аргумента методу `Eat`, поэтому эти аргументы также должны быть занесены в стек.

Когда вызовов становится все больше, а одни методы начинают вызывать другие методы, которые, в свою очередь, вызывают еще какие-то методы, размер стека увеличивается.

Когда метод `WriteLine` завершается, его аргументы извлекаются из стека. После этого метод `Eat` продолжает выполнение так, словно управление никуда не передавалось.

```
var v = Vector3.zero;
```

```
struct UnityEngine.Vector3
Representation of 3D vectors and points.
```

Так как `Vector3` в `Unity` является структурой, создание нескольких векторов не вызовет дополнительной сборки мусора.

Перейдите к проекту `Unity` и наведите указатель мыши на `Vector3` — это структура. Сборка мусора может серьезно понизить быстродействие приложения, а большое количество объектов в игре порождает лишние сборки мусора и снижает частоту кадров. В играх часто используется МНОЖЕСТВО векторов. Преобразование их в структуры означает, что их данные будут храниться в стеке, так что создание даже миллионов векторов не вызовет дополнительной сборки мусора, замедляющие игру.

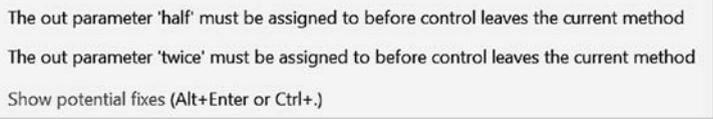
Параметры out и Возвращение нескольких значений методом

Сделайте это!

Раз уж речь зашла о параметрах и аргументах, существуют и другие способы передачи значений в программу и из нее. Все они требуют включения **модификаторов** в объявления методов. В одном из самых распространенных способов **модификатор out** используется для задания выходного параметра. Модификатор **out** вам уже знаком — он используется каждый раз, когда вы вызываете метод `int.TryParse`. Кроме того, вы можете использовать модификатор **out** в своих собственных методах. Создайте консольное приложение и добавьте в форму объявления пустого метода. Обратите внимание на модификаторы **out** у обоих параметров:

Метод может вернуть несколько значений при помощи параметров **out**.

```
public static int ReturnThreeValues(int value, out double half, out int twice)
{
    return value + 1;
}
```



Взгляните внимательно на описание двух ошибок:

- * Выходному параметру 'half' должно быть присвоено значение перед возвратом управления из текущего метода.
- * Выходному параметру 'twice' должно быть присвоено значение перед возвратом управления из текущего метода.

Каждый раз, когда вы используете параметр **out**, **всегда** необходимо присвоить ему значение до возврата из метода — по аналогии с тем, как метод, возвращающий значение, всегда должен содержать команду `return`.

Полный код приложения:

```
public static int ReturnThreeValues(int value, out double half, out int twice)
{
    half = value / 2f;
    twice = value * 2;
    return value + 1;
}
```

Значения всех параметров с модификатором **out** должны быть присвоены до выхода из метода.

Знакомый код, который уже использовался в книге; модификатор **out** используется с `int.TryParse` для преобразования строки в `int`.

```
static void Main(string[] args)
{
    Console.WriteLine("Enter a number: ");
    if (int.TryParse(Console.ReadLine(), out int input))
    {
        var output1 = ReturnThreeValues(input, out double output2, out int output3);

        Console.WriteLine("Outputs: plus one = {0}, half = {1:F}, twice = {2}",
            output1, output2, output3);
    }
}
```

Также модификатор **out** используется при вызове нового метода.

При запуске приложения будет получен следующий результат:

```
Enter a number: 17
Outputs: plus one = 18, half = 8.50, twice = 34
```

Передача по ссылке с модификатором ref

Как вы уже неоднократно видели, каждый раз, когда вы передаете методу `int`, `double`, `struct` или любой другой тип значения, методу передается копия этого значения. У такого механизма передачи существует специальное название: **передача по значению**. Это означает, что значение аргумента полностью копируется на сторону вызова.

Однако существует еще один способ передачи аргументов методам — так называемая **передача по ссылке**. Ключевое слово `ref` позволяет методу работать напрямую с переданным ему аргументом. Как и в случае с модификатором `out`, вы должны использовать `ref` при объявлении метода, а также при его вызове. При этом неважно, относится аргумент к типу значения или к ссылочному типу, — любая переменная, передаваемая в параметре `ref` метода, будет напрямую изменяться этим методом.

Чтобы понять, как работает этот механизм, создайте новое консольное приложение с классом `Guy` и следующими методами:

```
class Guy
{
    public string Name { get; set; }
    public int Age { get; set; }
    public override string ToString() => $"a {Age}-year-old named {Name}";
}
```

Во внутренней реализации аргумент `out` почти не отличается от аргумента `ref`, кроме того что ему не обязательно присваивать значение перед вызовом метода и оно должно присваиваться перед возвратом из метода.

```
class Program
{
    static void ModifyAnIntAndGuy(ref int valueRef, ref Guy guyRef)
    {
        valueRef += 10;
        guyRef.Name = "Bob";
        guyRef.Age = 37;
    }

    static void Main(string[] args)
    {
        var i = 1;
        var guy = new Guy() { Name = "Joe", Age = 26 };
        Console.WriteLine("i is {0} and guy is {1}", i, guy);
        ModifyAnIntAndGuy(ref i, ref guy);
        Console.WriteLine("Now i is {0} and guy is {1}", i, guy);
    }
}
```

Когда этот метод задает значения `valueRef` и `guyRef`, на самом деле он изменяет значения переменных в том методе, из которого он был вызван.

Когда метод `Main` вызывает `ModifyAnIntAndGuy`, переменные `i` и `guy` передаются по ссылке. Метод работает с ними как с любыми другими переменными. Но поскольку они передавались по ссылке, метод обновляет исходные значения, а не их копии в стеке. При завершении метода переменные `i` и `guy` в методе `Main` обновляются соответствующим образом.

Запустите приложение — оно выводит на консоль следующий результат:

```
i is 1 and guy is My name is Joe
Now i is 11 and guy is My name is Bob
```

Вторая строка отличается от первой, потому что `ModifyAnIntAndGuy` изменяет ссылки на переменные в методе `Main`.

Типы значений содержат метод `TryParse` с параметрами `out`

Вы пользовались методом `int.TryParse` для преобразования строк в значения `int`. Сходные функции поддерживаются другими типами значений: `double.TryParse` пытается преобразовать строки в значения `double`, `bool.TryParse` делает то же самое с логическими значениями и т. д. для `decimal.TryParse`, `float.TryParse`, `long.TryParse`, `byte.TryParse` и других. Также вспомните, что в главе 10 мы использовали команду `switch` для преобразования строки "Spades" в значение из перечисления `Suits.Spades`. Статический метод `Enum.TryParse` делает то же самое, если не считать перечисления.

Необязательные параметры и значения по умолчанию

Достаточно часто ваши методы снова и снова вызываются с одними и теми же аргументами, но методу все равно нужен параметр, потому что в отдельных случаях значение все же изменяется. Было бы полезно иметь возможность задать значение по умолчанию, чтобы аргумент можно было задавать только при вызове метода с новым значением.

Необязательные параметры нужны именно для таких случаев. Чтобы задать необязательный параметр в объявлении метода, поставьте после параметра знак = и укажите значение по умолчанию. Необязательных параметров может быть сколько угодно, но все они должны следовать после обязательных.

Пример метода с использованием необязательных параметров:

```
static void CheckTemperature(double temp, double tooHigh = 99.5, double tooLow = 96.5)
{
    if (temp < tooHigh && temp > tooLow)
        Console.WriteLine("{0} degrees F - feeling good!", temp);
    else
        Console.WriteLine("Uh-oh {0} degrees F -- better see a doctor!", temp);
}
```

Необязательные параметры имеют значения по умолчанию, заданные в объявлении.

Метод имеет два необязательных параметра: `tooHigh` имеет значение 99.5, а `tooLow` — значение по умолчанию 96.5. При вызове `CheckTemperature` с одним аргументом значения по умолчанию используются как для `tooHigh`, так и для `tooLow`. При вызове с двумя аргументами второй аргумент становится значением `tooHigh`, тогда как для `tooLow` сохраняется значение по умолчанию. Если при вызове передаются все три аргумента, то для всех трех параметров значения задаются явно.

Если вы хотите использовать некоторые (но не все) значения по умолчанию, используйте **именованные аргументы** для передачи значений только тех параметров, которые вам нужны. Все, что для этого потребуется, — присвоить каждому параметру имя, за которым следует двоеточие. Если вы используете более одного именованного аргумента, разделите их запятыми, как и любые другие аргументы.

Добавьте метод `CheckTemperature` в консольное приложение, затем добавьте следующий метод `Main`:

```
static void Main(string[] args)
{
    // Значения подходят для среднего человека
    CheckTemperature(101.3);

    // Температура собаки от 100.5 до 102.5 по Фаренгейту
    CheckTemperature(101.3, 102.5, 100.5);

    // Температура Боба всегда ниже обычной, задаем tooLow значение 95.5
    CheckTemperature(96.2, tooLow: 95.5);
}
```

Программа выводит следующий результат с разными значениями необязательных параметров:

```
Uh-oh 101.3 degrees F -- better see a doctor!
101.3 degrees F - feeling good!
96.2 degrees F - feeling good!
```

Если вы хотите, чтобы у метода были значения по умолчанию, используйте необязательные параметры и именованные аргументы.

Ссылка null не указывает ни на какой объект



Когда вы создаете новую ссылку, но не присваиваете ей значение, так что она ни на что не указывает, у этой ссылки есть значение. Изначально ссылка содержит `null` — собственно, это и означает, что она ни на что не указывает. Поэкспериментируем с `null`-ссылками.

1 Создайте новое консольное приложение и добавьте класс `Guy`, который использовался для экспериментов с ключевым словом `ref`.

2 Добавьте следующий код, который создает новый объект `Guy`, но не инициализирует его свойство `Name`.

```
static void Main(string[] args)
{
    Guy guy;
    guy = new Guy() { Age = 25 };
    Console.WriteLine("guy.Name is {0} letters long", guy.Name.Length);
}
```

3 Установите точку прерывания в последней строке метода `Main` и запустите отладку приложения. При достижении точки прерывания наведите указатель мыши на объект `guy`, чтобы просмотреть значения его свойств:

```
7 static void Main(string[] args)
8 {
9     Guy guy;
10    guy = new Guy() { Age = 25 };
11    Console.WriteLine("guy.Name is {0} letters long", guy.Name.Length);
12 }
13
14
```

Строка является ссылочным типом. Так как ей не было присвоено значение при инициализации объекта `Guy`, она сохраняет свое значение по умолчанию: `null`.

guy	(a 25-year-old named)
Age	25
Name	null

4 Продолжите выполнение кода. Метод `Console.WriteLine` пытается обратиться к свойству `Length` объекта `String`, на который ссылается свойство `guy.Name`, и программа выдает исключение:

```
Console.WriteLine("guy.Name is {0} letters long", guy.Name.Length);
```

Exception Thrown

System.NullReferenceException: 'Object reference not set to an instance of an object.'

NullTest.Program.Guy.Name.get returned null.

View Details | Copy Details | Start Live Share session...

Когда среда CLR выдает исключение **NullReferenceException** (которое разработчики часто обозначают сокращением **NRE**), она сообщает вам о том, что программа попыталась обратиться к компоненту объекта, но ссылка, которая для этого использовалась, содержала `null`. Разработчики стараются предотвратить исключения `null`-ссылок.



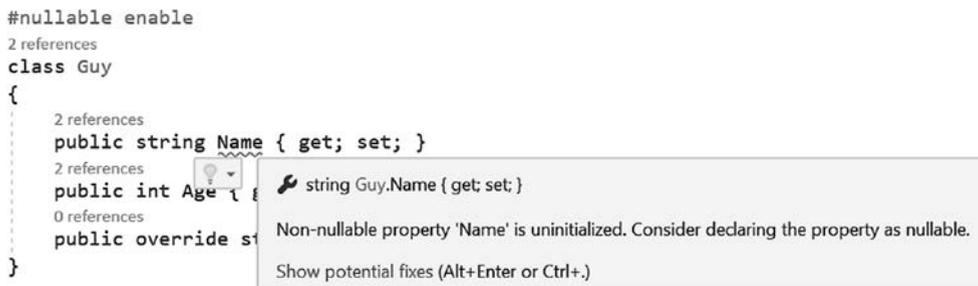
А как бы вы подошли к предотвращению исключений `null`-ссылок в программах?

Ссылочные типы, не допускающие null, помогут избежать NRE

Самый простой способ избежать исключений null-ссылок (или NRE) — **спроектировать код так, чтобы ссылки не могли содержать null**. К счастью, компилятор C# предоставляет очень полезный инструмент для решения проблем с null. Добавьте следующий фрагмент в начало класса Guy (он может располагаться как до объявления пространства имен, так и после него):

```
#nullable enable
```

Строка, начинающаяся с #, является **директивой**, т. е. средством передачи конкретных параметров компилятору. В данном случае она приказывает компилятору рассматривать все ссылки как **ссылочные типы, не допускающие null**. Сразу же после добавления директивы Visual Studio подчеркивает свойство Name предупреждающей красной линией. Наведите указатель мыши на свойство, чтобы просмотреть предупреждение:



Компилятор C# проделал нечто очень интересное: он воспользовался *анализом путей передачи управления*, чтобы определить, возможен ли путь, на котором *свойство Name будет содержать null*. Это будет означать, что код теоретически может выдать NRE.

Чтобы избавиться от предупреждения, можно принудительно объявить свойство Name со **ссылочным типом, допускающим null**. Для этого следует добавить символ ? после типа:

```
public string? Name { get; set; } ← Объявление свойства Name с типом, допускающим null, подавляет предупреждение, но не решает проблему.
```

Впрочем, хотя сообщение об ошибке пропадет, возможность выдачи исключений остается.

Используйте инкапсуляцию для того, чтобы свойство было заведомо отлично от null

В главе 5 вы узнали о том, как при помощи инкапсуляции предотвратить присваивание некорректных значений компонентам класса. Объявите свойство Name приватным, а затем добавьте конструктор, в котором ему будет присваиваться значение:

```
class Guy
{
    public string Name { get; private set; }
    public int Age { get; private set; }
    public override string ToString() => $"a {Age}-year-old named {Name}";

    public Guy(int age, string name)
    {
        Age = age;
        Name = name;
    }
}
```

← Объявление set-метода свойства Name приватным и добавление конструктора гарантирует, что свойству всегда будет присвоено значение. Таким образом, оно никогда не будет равно null, поэтому предупреждение компилятора исчезает.

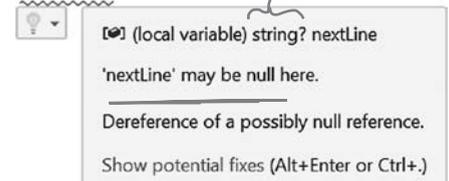
Инкапсуляция свойства Name предотвращает возможность присваивания ему null, и предупреждение исчезает.

Оператор объединения с null ??

Иногда без обработки null просто не обойтись. Например, в главе 10 вы узнали о чтении данных из строк с использованием класса `StringReader`. Создайте новое консольное приложение и добавьте следующий код:

```
#nullable enable
class Program
{
    static void Main(string[] args)
    {
        using (var stringReader = new StringReader(""))
        {
            var nextLine = stringReader.ReadLine();
            Console.WriteLine("Line length is: {0}", nextLine.Length);
        }
    }
}
```

Мы включили поддержку типов, не допускающих null. Теперь компилятор C# говорит, что строка `nextLine` допускает null и может содержать null при обращении к свойству `Length`.



Запустите код — возникает исключение NRE. Что с этим можно сделать?

?? проверяет null и возвращает альтернативу

Один из способов предотвратить обращение по ссылке null (**разыменование** ссылки) — использование **оператора объединения с null ??** для проверки выражения, которое теоретически может быть равно null (в данном случае `stringReader.ReadLine()`), и возвращения альтернативного значения, если оно равно null. Измените первую строку блока `using` и добавьте `?? String.Empty` в конец строки:

```
var nextLine = stringReader.ReadLine() ?? String.Empty;
```

`String.Empty` — статическое поле класса `String`, возвращающее пустую строку "".

Как только вы добавите оператор, предупреждение исчезает. Дело в том, что оператор объединения с null приказывает компилятору C# выполнить строку `stringReader.ReadLine()`; и использовать возвращенное значение, если оно отлично от null, — или заменить его значением, предоставленным вами (пустой строкой в данном случае) в противном случае.

Оператор ??= присваивает значение переменной, только если она равна null

При работе со значениями null достаточно часто встречается код, который проверяет значение на равенство null и присваивает значение, отличное от null, для предотвращения NRE. Например, если вы хотите изменить вашу программу, чтобы она выводила первую строку кода, это можно сделать так:

```
if (nextLine == null)
    nextLine = "(the first line is null)";

// Этот код работает с nextLine при условии, что переменная отлична от null.
```

Эту условную команду можно переписать с использованием **оператора ??=**:

```
nextLine ??= "(the first line was empty)";
```

Оператор `??=` проверяет переменную, свойство или поле в левой части выражения (в данном случае `nextLine`) на равенство null. Если левая часть равна null, то оператор присваивает значение в правой части выражения, а если нет — исходное значение остается неизменным.

Безопасная работа с типами значений, допускающими null

Когда вы объявляете `int`, `bool` или другой тип значения и не задаете исходное значение, среда CLR присваивает значение по умолчанию — например, 0 или `true`. Но представьте, что вы пишете код для обработки сетевых опросов пользователей, в котором имеется необязательный вопрос «да/нет». Получается, что вам нужно представить логическое значение, которое может быть равно `true` или `false`, но может вообще не иметь никакого значения?

И здесь вам могут пригодиться типы значений, допускающие `null`. Тип значения, допускающий `null`, содержит либо определенное значение, либо `null`. В нем используется обобщенная структура `Nullable<T>`, которая может использоваться в качестве *обертки для значения* (или может содержать значение и предоставлять компоненты для обращения и работы с ним). Если присвоить `null` типу значения, допускающему `null`, он не будет иметь определенного значения, но `Nullable<T>` предоставляет удобные компоненты для безопасной работы с ним *даже в этом случае*.

Логическое значение, допускающее `null`, объявляется так:

```
Nullable<bool> optionalYesNoAnswer = null;
```

В C# также предусмотрена сокращенная запись: для типа значения `T` структуру `Nullable<T>` можно объявить в виде `T?`:

```
bool? anotherYesNoAnswer = false;
```

Структура `Nullable<T>` содержит свойство `Value`, которое задает или получает значение. Тип `bool?` содержит значение типа `bool`, `int?` содержит значение типа `int`, и т. д. Также он содержит свойство с именем `HasValue`, которое возвращает `true`, если значение отлично от `null`.

Также тип значения всегда можно преобразовать к типу, допускающему `null`:

```
int? myNullableInt = 9321;
```

Чтобы получить значение обратно, воспользуйтесь удобным свойством `Value`:

```
int = myNullableInt.Value;
```

Но вызов `Value` в конечном итоге просто преобразует тип `int` к `(int)myNullableInt`, и если значение равно `null`, будет выдано исключение `InvalidOperationException`. Именно поэтому `Nullable<T>` также содержит свойство `HasValue`, которое возвращает `true`, если значение отлично от `null`, или `false` в противном случае. Также можно воспользоваться удобным методом `GetValueOrDefault`, который безопасно возвращает значение по умолчанию, если `Nullable` не содержит значения. Также возможно передать значение по умолчанию или воспользоваться нормальным значением по умолчанию для типа.

Nullable<bool>	
Value:	DateTime
HasValue:	bool
...	
GetValueOrDefault():	DateTime
...	

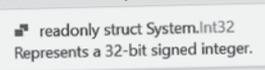
Nullable<T> — структура, позволяющая сохранить тип значения ИЛИ значение null. На диаграмме представлены некоторые методы и свойства Nullable<bool>.

T? является синонимом для Nullable<T>

Когда к любому типу значения присоединяется вопросительный знак (например, `int?` или `decimal?`), компилятор преобразует его в структуру `Nullable<T>` (`Nullable<int>` или `Nullable<decimal>`). В этом нетрудно убедиться: добавьте в программу переменную `Nullable<bool>`, установите точку прерывания и включите отслеживание этой переменной в отладчике. Вы увидите, что в окне `Watch` в IDE выводится `bool?`. Это пример синонима — и не первый, с которым вы встречаетесь в книге. Наведите указатель мыши на любое вхождение `int`. Вы увидите, что он преобразуется в структуру с именем `System.Int32`:

```
int value = 3;
```

`int.Parse()` и `TryParse()` являются компонентами этой структуры.



Проделайте это для каждого из типов, упомянутых в начале главы 4. Обратите внимание на то, что все они являются синонимами для структур, кроме строк, которым соответствует класс `System.String`, а не тип значения.

Капитан... уже не такой Великолепный

К этому моменту вы уже должны достаточно понимать, что происходит с ослабевшим и не столь могущественным Капитаном Великолепным. Собственно, это вообще не Капитан Великолепный, а упакованная структура:



и



- 1 Структуры не могут наследовать от классов. Неудивительно, что суперсилы Капитана не впечатляют! Он не получил никакого унаследованного поведения.
- 2 Структуры копируются по значению. Это одна из самых полезных особенностей структур. Она особенно полезна для инкапсуляции.
- 3 Ключевое слово `as` может использоваться с объектами. Поддержка полиморфизма объектами основана на том, что объект может функционировать как любой из объектов, от которых он наследует.
- 4 Объекты не копируются при присваивании. Когда вы присваиваете одной объектной переменной другую, вы копируете ссылку на *ту же* переменную.

Важный момент: при помощи ключевого слова «`is`» можно проверить, реализует ли структура заданный интерфейс, — это один из аспектов полиморфизма, поддерживаемых структурами.

Простота копирования — одно из преимуществ структур (и других типов значений).



Часть Задаваемые Вопросы

В: Вернемся чуть назад. Зачем мне вообще знать о стеке?

О: Затем, что понимание различий между стеком и кучей позволяет правильно понять, как работают ссылочные типы и типы значений. Легко забыть, что структуры и объекты работают по-разному: когда вы используете с ними знак =, они выглядят очень похоже. Понимание того, как работают некоторые механизмы в .NET и CLR во внутренней реализации, поможет разобраться в том, **почему** ссылочные типы отличаются от типов значений.

В: А упаковка? Почему она важна для меня?

О: Потому, что вы должны знать, когда данные попадают в стек и когда происходит их копирование. Упаковка требует дополнительных затрат памяти и времени. Если она выполняется в вашей программе всего несколько раз (или несколько сотен раз), вы не заметите разницы. Но представьте, что ваша программа в цикле выполняет одну и ту же операцию миллионы раз в секунду. Пример не такой уж надуманный: в играх Unity может происходить именно это. Если вы обнаружите, что ваши программы занимают все больше памяти или начинают работать все медленнее, возможно, их эффективность можно повысить за счет предотвращения упаковки в многократно повторяющихся частях программы.

В: Я понимаю, как мы получаем новую копию структуры, когда одна структурная переменная присваивается другой. Но зачем мне это может понадобиться?

О: Одна из областей, в которых это действительно полезно, — **инкапсуляция**. Взгляните на следующий код:

```
private Point location;
public Point Location {
    get { return location; }
}
```

Если бы поле Point было классом, такая инкапсуляция была бы просто ужасной. Объявление location с private ни на что не влияет, потому что вы создаете открытое свойство, доступное только для чтения, которое возвращает ссылку на location, — и выходит, что любой другой объект сможет обратиться к приватным данным.

К счастью, Point является структурой. А это означает, что открытое свойство Location возвращает новую копию данных. Объект, который использует эту копию, может делать с ней все что угодно — эти изменения никак не отразятся на приватном поле location.

В: Как узнать, когда лучше использовать структуру, а когда класс?

О: В большинстве случаев программисты используют классы. У структур много ограничений, которые сильно усложняют работу с ними в крупномасштабных проектах. Структуры не поддерживают наследование или абстракцию, поддержка полиморфизма ограничена, а вы уже знаете, насколько это важно для написания кода.

Структуры по-настоящему полезны для небольших, ограниченных типов данных, с которыми выполняются повторяющиеся операции. Хорошим примером служат векторы Unity; в некоторых играх они используются снова и снова, иногда миллионы раз. Повторное использование вектора с присваиванием его той же переменной приводит к повторному использованию той же памяти в стеке. Если бы Vector3 был классом, то CLR пришлось бы выделять память в куче для каждого нового экземпляра Vector3, а в системе постоянно происходила бы сборка мусора. Таким образом, реализовав Vector3 в виде структуры, а не класса, группа разработчиков Unity обеспечила более высокую частоту кадров в ваших играх, а вам для этого не пришлось ничего делать.

Структуры приносят пользу при инкапсуляции, потому что свойство, доступное только для чтения, которое возвращает структуру, всегда создает его новую копию.

MINI Возьми в руку карандаш

Предполагается, что этот метод будет уничтожать объект EvilClone, помечая его для сборщика мусора, но он не работает. Почему?

```
void SetCloneToNull(EvilClone clone) => clone = null;
```

У бассейна



Ваша **задача** — взять фрагменты кода из бассейна и разместить их в пустых строках. Любой фрагмент можно использовать несколько раз, использовать все фрагменты не обязательно. Ваша **цель** — добиться того, чтобы программа вывела на консоль приведенный ниже результат.

```
class Program {
    static void Main(string[] args) =>
        new Faucet();
}

public class Faucet {
    public Faucet() {
        Table wine = new Table();
        Hinge book = new Hinge();
        wine.Set(book);
        book.Set(wine);
        wine.Lamp(10);
        book.garden.Lamp("back in");
        book.bulb *= 2;
        wine.Lamp("minutes");
        wine.Lamp(book);
    }
}
```

```
public _____ Table {
    public string stairs;
    public Hinge floor;

    public void Set(Hinge b) => floor = b;

    public void Lamp(object oil) {
        if (oil _____ int oilInt)
            _____ .bulb = oilInt;
        else if (oil _____ string oilString)
            stairs = oilString;
        else if (oil _____ Hinge _____)
            Console.WriteLine(
                $"{vine.Table()} {_____ .bulb} {stairs}");
    }
}

public _____ Hinge {
    public int bulb;
    public Table garden;

    public void Set(Table a) => garden = a;

    public string Table() {
        return _____ .stairs;
    }
}
```

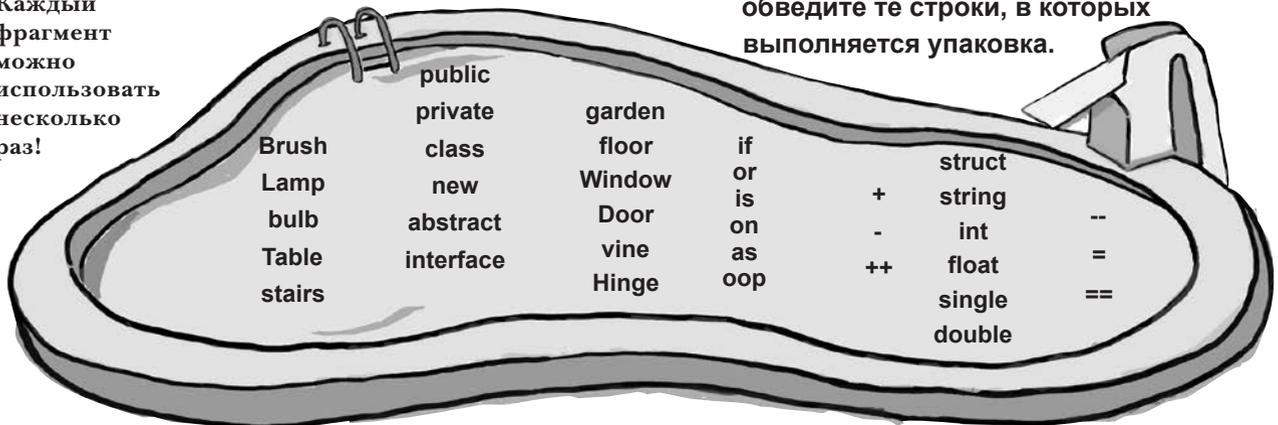
Результат

back in 20 minutes

Результат, который должно выводить приложение.

Каждый фрагмент можно использовать несколько раз!

Дополнительный вопрос: обведите те строки, в которых выполняется упаковка.



У бассейна. Решение



Метод Lamp присваивает различные строки и int. Если вызвать его с аргументом int, он присвоит значение полю Bulb в объекте, на который указывает Hinge.

```
class Program {
    static void Main(string[] args) =>
        new Faucet();
}

public class Faucet {
    public Faucet() {
        Table wine = new Table();
        Hinge book = new Hinge();
        wine.Set(book);
        book.Set(wine);
        wine.Lamp(10);
        book.garden.Lamp("back in");
        book.bulb *= 2;
        wine.Lamp("minutes");
        wine.Lamp(book);
    }
}

public struct Table {
```

Вот почему тип Table объявлен как структура. Если бы он был классом, то поле wine указывало бы на тот же объект, что и book.Garden, в результате чего эта команда перезаписала бы строку "back in".

Дополнительное задание:
 обведите строки, в которых происходит упаковка.

```
public string stairs;
public Hinge floor;

public void Set(Hinge b) => floor = b;

public void Lamp(object oil) {
    if (oil is int oilInt)
        floor.bulb = oilInt;
    else if (oil is string oilString)
        stairs = oilString;
    else if (oil is Hinge vine)
        Console.WriteLine(
            $"{vine.Table()} {floor.bulb} {stairs}");
}
}
```

Если передать Lamp строку, то полю Stairs присваивается текущее содержимое этой строки.

```
public class Hinge {
    public int bulb;
    public Table garden;

    public void Set(Table a) => garden = a;

    public string Table() {
        return garden.stairs;
    }
}
```

Результат

back in 20 minutes

Hinge и Table содержат методы Set, тело которых определяется выражением. Hinge.Set задает поле Table с именем Garden, тогда как Table.Set задает поле Hinge с именем Floor.

Так как метод Lamp получает параметр object, при передаче int или строки автоматически выполняется упаковка. С другой стороны, book не упаковывается, потому что уже является объектом — экземпляром класса Hinge.

Возьми в руку карандаш



Решение

Параметр clone находится в стеке, так что присвоение ему никак не отражается на куче.

Предполагается, что этот метод будет уничтожать объект EvilClone, помечая его для сборщика мусора, но он не работает. Почему?

```
void SetCloneToNull(EvilClone clone) => clone = null;
```

Этот метод только присваивает null своему параметру, но этот параметр всего лишь является ссылкой на EvilClone. В сущности, мы прикрепили к объекту наклейку, а потом сняли ее.

Методы расширения добавляют новое поведение в СУЩЕСТВУЮЩИЕ классы

Помните модификатор `sealed` из главы 7? Он указывает, что класс не может расширяться.

Иногда требуется расширить класс, наследование от которого невозможно, например запечатанного (`sealed`) класса (многие классы .NET являются запечатанными и не могут использоваться для наследования). C# предоставляет для подобных случаев гибкий инструмент: **методы расширения**. Когда вы добавляете в свой проект класс с методами расширения, он **добавляет новые методы** в уже существующие классы. От вас потребуется лишь создать статический класс и добавить статический метод, который получает экземпляр класса в первом параметре с использованием ключевого слова `this`.

Допустим, у вас имеется запечатанный класс `OrdinaryHuman`:

```
sealed class OrdinaryHuman {
    private int age;
    int weight;

    public OrdinaryHuman(int weight){
        this.weight = weight;
    }

    public void GoToWork() { /* Код для выхода на работу */ }
    public void PayBills() { /* Код для оплаты счетов */ }
}
```

Класс `OrdinaryHuman` объявлен запечатанным и субклассироваться не может. Но что, если нам потребуется добавить в него метод?

Чтобы использовать метод расширения, определите первый параметр с ключевым словом «`this`».

Так как расширяется класс `OrdinaryHuman`, первый параметр объявляется в виде «`this OrdinaryHuman`».

Метод `AmazeballsSerum` добавляет метод расширения в `OrdinaryHuman`:

```
static class AmazeballsSerum {
    public static string BreakWalls(this OrdinaryHuman h, double wallDensity) {
        return ("I broke through a wall of {wallDensity} density.");
    }
}
```

Методы расширения всегда являются статическими, и они должны размещаться в статических классах.

Как только класс `AmazeballsSerum` будет добавлен в проект, у `OrdinaryHuman` появляется метод `BreakWalls`. Теперь вы можете использовать его в своем методе `Main`:

```
static void Main(string[] args){
    OrdinaryHuman steve = new OrdinaryHuman(185);
    Console.WriteLine(steve.BreakWalls(89.2));
}
```

Вот и все! От вас требуется лишь добавить класс `AmazeballsSerum` в ваш проект, и внезапно в классе `OrdinaryHuman` появляется новенький метод `BreakWalls`.

Когда программа создает экземпляр класса `OrdinaryHuman`, она может обратиться к методу `BreakWalls` напрямую — при условии, что класс `AmazeballsSerum` входит в проект. Убедитесь сами! Создайте новое консольное приложение, добавьте в него два класса и метод `Main`. Зайдите в отладчике в метод `BreakWalls` и посмотрите, что в нем происходит.

Ранее в этой книге мы «по волшебству» добавляли методы в классы простым включением директивы `using` в начало кода. А вы помните, где это было?

Часть Задаваемые Вопросы

В: Объясните, почему бы не добавить нужные мне методы прямо в код моего класса, вместо того чтобы использовать методы расширения?

О: Вы можете так поступить, и, наверное, это стоит сделать, если речь идет только о добавлении метода в один класс. Методами расширения следует пользоваться осмотрительно, и только в тех случаях, когда изменение класса, с которым вы работаете, по какой-то причине абсолютно невозможно (например, если он является частью .NET Framework или другого стороннего кода.) Методы расширения по-настоящему проявляют свою мощь тогда, когда вам требуется расширить поведение **того, что вам обычно недоступно**, например типа или объекта, входящего в .NET Framework или другую библиотеку.

В: Зачем вообще использовать методы расширения? Почему бы не расширить класс наследованием?

О: Если вы можете расширить класс, то обычно стоит поступить именно так — методы расширения не предназначены для замены наследования. Но обычно они оказываются очень удобными, когда у вас имеются классы, которые невозможно расширить. С методами расширения вы можете изменить поведение целых групп объектов и даже добавлять новую функциональность в некоторые фундаментальные классы .NET Framework.

Расширение класса предоставляет новое поведение, но вам придется использовать новый subclass, если вы хотите использовать это поведение.

В: Влияет ли мой метод расширения на все экземпляры класса или только на какой-то отдельный экземпляр?

О: На все экземпляры расширяемого класса. Более того, после того, как вы создадите метод расширения, он появится в IDE наряду с обычными методами расширяемого класса.

Когда я добавил директиву «using system.linq» в начало кода, во всех моих коллекциях и последовательностях неожиданно появились методы LINQ. Выходит, я использовал методы расширения?

И еще один факт, который стоит помнить: создание метода расширения не предоставит вам доступа к внутреннему устройству класса.

Да! В основе LINQ лежат методы расширения.

Кроме расширения классов, также можно расширять **интерфейсы**. Все, что для этого нужно, — использовать имя интерфейса вместо класса после ключевого слова **this** в первом параметре метода расширения. Метод расширения будет добавляться в **каждый класс, реализующий этот интерфейс**. Именно так поступила команда .NET при создании LINQ: все методы LINQ являются статическими методами расширения для интерфейса `IEnumerable<T>`.

Когда вы включаете директиву `using System.Linq;` в начало своего кода, ваш код «видит» статический класс с именем `System.Linq.Enumerable`. Мы использовали некоторые из его методов (например, `Enumerable.Range`), но он также содержит методы расширения. Откройте IDE, введите имя `Enumerable.First` и просмотрите объявление. Оно начинается с описания (`extension`), которое сообщает, что это метод расширения, а его первый параметр использует ключевое слово `this`, как и в написанном вами методе расширения. Эта схема применяется во всех методах LINQ.

Enumerable.First

Enumerable.First — метод расширения, в объявлении которого используется ключевое слово «this».

```

First<>
FirstOrDefault<>
(extension) TSource Enumerable.First<TSource>(this IEnumerable<TSource> source) (+ 1 generic overload)
Returns the first element of a sequence.
    
```

Эта кнопка в окне IntelliSense выводит только методы расширения.



Расширение фундаментального типа: string

Чтобы изучить работу методов расширения, расширим класс String. **Создайте новый проект консольного приложения** и добавьте файл с именем HumanExtensions.cs.

Сделайте это!

1 Разместите все методы расширения в отдельном пространстве имен.

Проследите за тем, чтобы класс был объявлен открытым (а следовательно, видимым при добавлении объявления using).

Все методы расширения желательно разместить в специальном пространстве имен отдельно от основного кода. При таком подходе вы сможете легко найти их для использования в других программах. Создайте статический класс, в котором будут находиться ваши методы:

```
namespace AmazingExtensions {
    public static class ExtendAHuman {
```

Использование отдельного пространства имен — хороший организационный прием. Кроме того, класс, в котором определяется ваш метод расширения, должен быть статическим.

2 Создайте статический метод расширения, определите его первый параметр с ключевым словом this и расширяемым типом.

Два главных факта, о которых необходимо помнить при объявлении метода расширения: метод должен быть статическим и он должен получать расширяемый класс в первом параметре:

Метод расширения тоже должен быть статическим.

```
public static bool IsDistressCall(this string s) {
```

«this string s» означает, что мы расширяем класс String, а параметр s используется для обращения к используемой строке для вызова метода.

3 Завершите метод расширения.

Метод проверяет, содержит ли строка слово «Help!» (просьба о помощи, на которую должен откликнуться каждый супергерой):

```
    if (s.Contains("Help!"))
        return true;
    else
        return false;
}
}
```

Использует метод String.Contains для проверки того, содержит ли строка слово «Help!» — определено это не та операция, которая обычно выполняется со строками.

4 Используйте новый метод расширения IsDistressCall.

Добавьте директиву using AmazingExtensions; в начало файла с классом Program. Затем добавьте в класс код, который создает строку и вызывает метод IsDistressCall. Вы увидите метод расширения в окне IntelliSense:

0 references

```
static void Main(string[] args)
```

```
{
    string message = "Evil clones are wreaking havoc. Help!";
    message.IsDistressCall
}
```

IsDistressCall (extension) bool string.IsDistressCall()

Как только вы добавите директиву using для включения пространства имен со статическим классом, в окне IntelliSense появится новый метод расширения. Именно по этому принципу работает LINQ.



Развлечения с Магнитами

Расставьте магниты так, чтобы программа выводила следующий результат:

a buck begets more bucks

```

namespace Upside {
    public static class Margin {
        public static void SendIt
    }
    public static string ToPrice
}

namespace Sideways
{
    using Upside;
    class Program {
        static void Main(string[] args) {
            int i = 1;
            (this int n) {
                s.SendIt();
                string s = i.ToPrice();
                b.Green().SendIt();
            }
            else
                return " more bucks";
            i = 3;
            Console.Write(s);
            if (n == 1)
                return "a buck ";
            bool b = true;
            (this bool b) {
                b = false;
                if (b == true)
                    return "be";
                b.Green().SendIt();
            }
            public static string Green
            {
                i.ToPrice()
                .SendIt();
            }
            (this string s) {
                else
                    return "gets";
            }
        }
    }
}
    
```

МЫ ВОССТАНОВИЛИ
КЛАСС СУПЕРГЕРОЯ, НО КАК
ВЕРНУТЬ КАПИТАНА?



ЭВРИКА!
Я ПРОАНАЛИЗИРОВАЛА
КОД — КАПИТАН ВЕЛИКО-
ЛЕТНЫЙ ИСПОЛЬЗОВАЛ
СВОЮ СМЕРТЬ ДЛЯ ТОГО,
ЧТОБЫ СЕРИАЛИЗОВАТЬ
СЕБЯ!



The UNIVERSE

НОВАЯ ЖИЗНЬ КАПИТАНА ВЕЛИКОЛЕПНОГО

СМЕРТЬ ЕЩЕ НЕ КОНЕЦ

Лаки Бернс
ШТАТНЫЙ КОРРЕСПОНДЕНТ

ОБЪЕКТВИЛЬ

Капитан Великолепный десериализует себя — потрясающее возвращение к жизни!

После умопомрачительного поворота событий Капитан Великолепный вернулся в Объектвиль. В прошлом месяце оказалось, что гроб Капитана пуст, а там, где должно было лежать его тело, лежала только странная записка. Анализ записки показал, что ДНК Капитана Великолепного — все ее поля и значения — была точно отражена в двоичном формате.

Сегодня эти данные неожиданно воплотились в жизнь. Капитан вернулся — ему удалось десериализовать себя по своей записке. Когда мы его спросили, как ему пришел в голову такой план, Капитан только пожал плечами и пробормотал: «Глава 10». Источники, близкие к Капитану, отказались комментировать смысл этой загадочной реплики, но признали, что до своего неудачного нападения на Пройдоху Капитан провел много времени за чтением книг, изучая методы Dispose и тему долгосрочного хранения данных. Мы ожидаем, что Капитан Великолепный...



Капитан
Великолепный
вернулся!



Развлечения с магнитами. Решение

Расставьте магниты так, чтобы программа выводила следующий результат:

a buck begets more bucks

Пространство имен *Upside* имеет расширение. Пространство имен *Sideways* содержит точку входа.

Класс *Margin* расширяет строку и добавляет метод *SendIt*, который просто выводит строку на консоль. Также он расширяет *int* и добавляет метод *ToPrice*, который возвращает строку "a buck", если значение *int* равно 1, или "more bucks" в противном случае.

```
namespace Upside {
    public static class Margin {
        public static void SendIt (this string s) {
            Console.Write(s);
        }
        public static string ToPrice (this int n) {
            if (n == 1)
                return "a buck ";
            else
                return " more bucks";
        }
        public static string Green (this bool b) {
            if (b == true)
                return "be";
            else
                return "gets";
        }
    }
}
```

Метод *Main* расширяет *bool* — он возвращает строку "be", если *bool* содержит *true*, или "gets" для значения *false*.

```
namespace Sideways
{
    using Upside;
    class Program {
        static void Main(string[] args) {
            int i = 1;
            string s = i.ToPrice();
            s.SendIt();
            bool b = true;
            b.Green().SendIt();
            b = false;
            b.Green().SendIt();
            i = 3;
            i.ToPrice().SendIt();
        }
    }
}
```

Метод *Main* использует расширения, добавленные в класс *Margin*.

Борьба с огнем надоедает

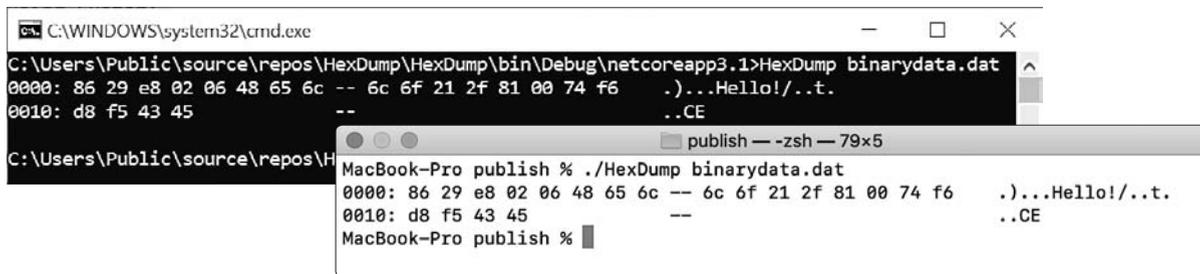
Я знаю, что этот страшный клоун где-то прячется.
Хорошо, что я написала код для обработки
СтрахИсключение.



Программисты не должны уподобляться пожарным. Вы усердно работали, штудировали справочники и руководства и, наконец, достигли вершины. Но вам до сих пор продолжают звонить с работы по ночам, потому что **программа упала** или **работает не так, как должна работать**. Ничто так не выбивает из колеи, как необходимость устранять странные ошибки... но благодаря **обработке исключений** вы сможете написать код, который сам будет **разбираться с возможными проблемами**. А еще лучше, что вы можете планировать такие проблемы и **восстанавливать работоспособность программы** при их возникновении.

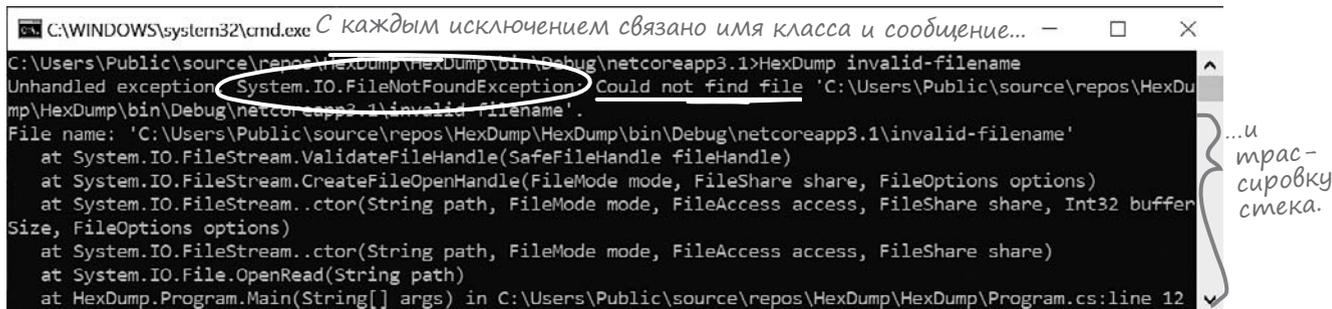
Программа вывода шестнадцатеричного дампа читает имя файла из командной строки

В конце главы 10 мы построили программу вывода шестнадцатеричного дампа, которая использовала аргументы командной строки для вывода произвольного файла. Мы воспользовались свойствами проекта в IDE для задания аргументов в отладчике. Также вы увидели, как запустить программу из командной строки Windows или окна терминала macOS.



Но что произойдет, если передать HexDump недействительное имя файла?

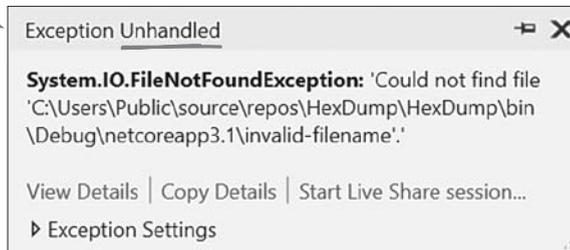
Когда вы изменяли приложение HexDump для поддержки аргументов командной строки, мы рекомендовали особенно внимательно следить за тем, чтобы программе передавалось правильное имя файла. Что произойдет, если файл с указанным именем не существует? Попробуйте запустить приложение, но теперь передать ему аргумент `invalid-filename`. На этот раз программа *выдает исключение*.

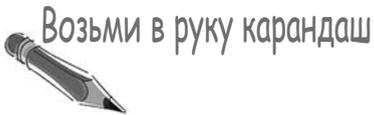


Задайте в настройках проекта недействительное имя файла и запустите приложение в отладчике IDE. В этом случае будет выдано исключение с тем же именем класса (`System.IO.FileNotFoundException`) и похожее сообщение «Файл не найден».

Необработанное исключение означает, что приложение столкнулось с непредвиденной проблемой.

IDE останавливает отладчик в строке, в которой произошло исключение, и выводит информацию в окне **Exception Unhandled**. Вы даже сможете просмотреть трассировку стека в окне **Call Stack**.





```

class HoneyBee
{
    public double Capacity { get; set; }
    public string Name { get; set; }

    public HoneyBee(double capacity, string name)
    {
        Capacity = capacity;
        Name = name;
    }

    public static void Main(string[] args)
    {
        object myBee = new HoneyBee(36.5, "Zippo");
        float howMuchHoney = (float)myBee;

        HoneyBee anotherBee = new HoneyBee(12.5, "Buzzy");
        double beeName = double.Parse(anotherBee.Name);

        double totalHoney = 36.5 + 12.5;
        string beesWeCanFeed = "";
        for (int i = 1; i < (int)totalHoney; i++)
        {
            beesWeCanFeed += i.ToString();
        }
        int numberOfBees = int.Parse(beesWeCanFeed);

        int drones = 4;
        int queens = 0;
        int dronesPerQueen = drones / queens;

        anotherBee = null;
        if (dronesPerQueen < 10)
        {
            anotherBee.Capacity = 12.6;
        }
    }
}

```

Этот код не работает. Программа выдает пять разных исключений; справа приведены сообщения об ошибках, выводимые в IDE или на консоль. Ваша задача — **соединить строки, содержащие ошибки, с исключением, которое генерируется этими строками.** Помните,

что **код не работает**. Если добавить этот класс в приложение и запустить его метод Main, выполнение прервется после первого выданного исключения. Просто прочитайте код и сопоставьте каждое исключение со строкой, в которой оно *было бы* выдано при запуске.

Метод `double.Parse` преобразует строку в `double`, поэтому при передаче строки (например, "32.7") будет возвращено эквивалентное значение `double` (32.7). Как вы думаете, что произойдет при передаче строки, которая не может быть преобразована в `double`?

System.OverflowException: 'Value was either too large or too small for an Int32.' ①

System.NullReferenceException: 'Object reference not set to an instance of an object.' ②

System.InvalidCastException: 'Unable to cast object of type 'ExceptionTests.HoneyBee' to type 'System.Single'.' ③

System.DivideByZeroException: 'Attempted to divide by zero.' ④

System.FormatException: 'Input string was not in a correct format.' ⑤



Возьми в руку карандаш Решение

Вам было предложено соединить каждую проблемную строку кода с исключением, которое генерируется этой строкой.

Код преобразования myBee к типу float компилируется, но способа преобразования объекта HoneyBee в float не существует. При выполнении кода CLR понятия не имеет, как выполнить это преобразование, поэтому выдается исключение `InvalidCastException`.

```
object myBee = new HoneyBee(36.5, "Zippo");
float howMuchHoney = (float)myBee;
```

System.InvalidCastException: 'Unable to cast object of type 'ExceptionTests.HoneyBee' to type 'System.Single'.'

3

Подсказки для IDE: Команда Set Next Statement

Чтобы воспроизвести эти исключения в IDE, скопируйте этот код и запустите его. Установите точку прерывания в первой строке кода, затем щелкните правой кнопкой мыши в строке кода, которую вы хотите выполнить, и выберите команду **Set Next Statement** — при продолжении выполнения приложение перейдет сразу к указанной команде.

```
HoneyBee anotherBee = new HoneyBee(12.5, "Buzzy");
double beeName = double.Parse(anotherBee.Name);
```

System.FormatException: 'Input string was not in a correct format.'

5

Метод Parse ожидает, что вы предоставите ему строку в определенном формате. Когда вы передаете ему строку "Buzzy", он не знает, как преобразовать ее в число, и поэтому выдает исключение `FormatException`.

```
double totalHoney = 36.5 + 12.5;
string beesWeCanFeed = "";
for (int i = 1; i < (int)totalHoney; i++)
{
    beesWeCanFeed += i.ToString();
}
```

```
int numberOfBees = int.Parse(beesWeCanFeed);
```

System.OverflowException: 'Value was either too large or too small for an Int32.'

1

Цикл for создает строку с именем `beesWeCanFeed` с числом, состоящим более чем из 60 цифр. Такое большое число никак не может поместиться в `int`, и попытка преобразования его к `int` приведет к выдаче `OverflowException`.

Все эти исключения никогда не будут выданы подряд — программа выдает первое исключение и останавливается. Второе исключение встретится вам только в том случае, если вы исправите причину первого.

Возьми в руку карандаш Решение

```
int drones = 4;
int queens = 0;
int dronesPerQueen = drones / queens;
```

Сгенерировать исключение `DivideByZeroException` несложно — для этого достаточно разделить любое число на ноль.

System.DivideByZeroException: 'Attempted to divide by zero.'

4

При делении любого целого числа на ноль всегда выдается такое исключение. Даже если значение `queens` неизвестно заранее, исключение можно предотвратить — проверьте, не равно ли значение нулю, прежде чем делить на него.



Даже при простом взгляде на код я вижу, что он пытается делить на ноль. Уверен, что этого исключения можно избежать.

Ошибки `DivideByZero` можно было бы избежать.

Даже если просто взглянуть на код, становится понятно, что здесь что-то не так. Если задуматься, то же самое относится и к другим исключениям — вся суть упражнения «Возьми в руку карандаш» заключалась в поиске этих исключений без запуска кода. Каждое из этих исключений можно было **предотвратить**. Чем больше вы знаете об исключениях, тем лучше вы будете справляться с предотвращением сбоев в приложениях.

```
anotherBee = null;
if (dronesPerQueen < 10)
{
    anotherBee.Capacity = 12.6;
}
```

Присваивание `null` переменной `anotherBee` сообщает C#, что она ни на что не указывает. Выдавая исключение `NullReferenceException`, C# сообщает вам, что не существует объекта, для которого можно было бы вызвать метод `DoMyJob`.

System.NullReferenceException: 'Object reference not set to an instance of an object.'

2

Когда ваша программа выдает исключение, CLR генерирует объект Exception

Мы рассматривали механизм, который используется CLR для оповещения о том, что в программе что-то пошло не так: **исключения**. Когда в программе происходит исключение, создается объект, представляющий проблему. Как нетрудно предположить, это объект класса `Exception`.

Допустим, у вас имеется массив с четырьмя элементами и вы пытаетесь обратиться к 16-му элементу (с индексом 15, так как индексы начинаются с нуля):

```
int[] anArray = {3, 4, 1, 11};
int aValue = anArray[15];
```

Очевидно, в этом коде возникнет проблема.

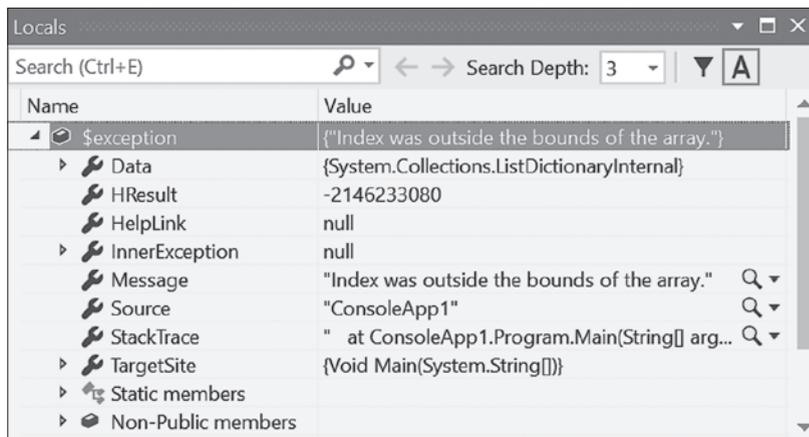
Ис-кю-че-ни-е, суц. Человек или объект, включающиеся из общего утверждения или не подчиняющиеся общему правилу.

Как только в вашей программе произойдет исключение, CLR создает объект с полной информацией о проблеме.



Объект `Exception` содержит сообщение с описанием проблемы и трассировкой стека, т. е. списком всех вызовов, приведших к команде, в которой произошло исключение.

Когда IDE останавливает выполнение из-за того, что в коде произошло исключение, вы можете просмотреть подробную информацию об исключении в разделе **\$exception** в окне **Locals**. В этом окне выводятся все переменные, находящиеся в области видимости (т. е. доступные для текущей команды).



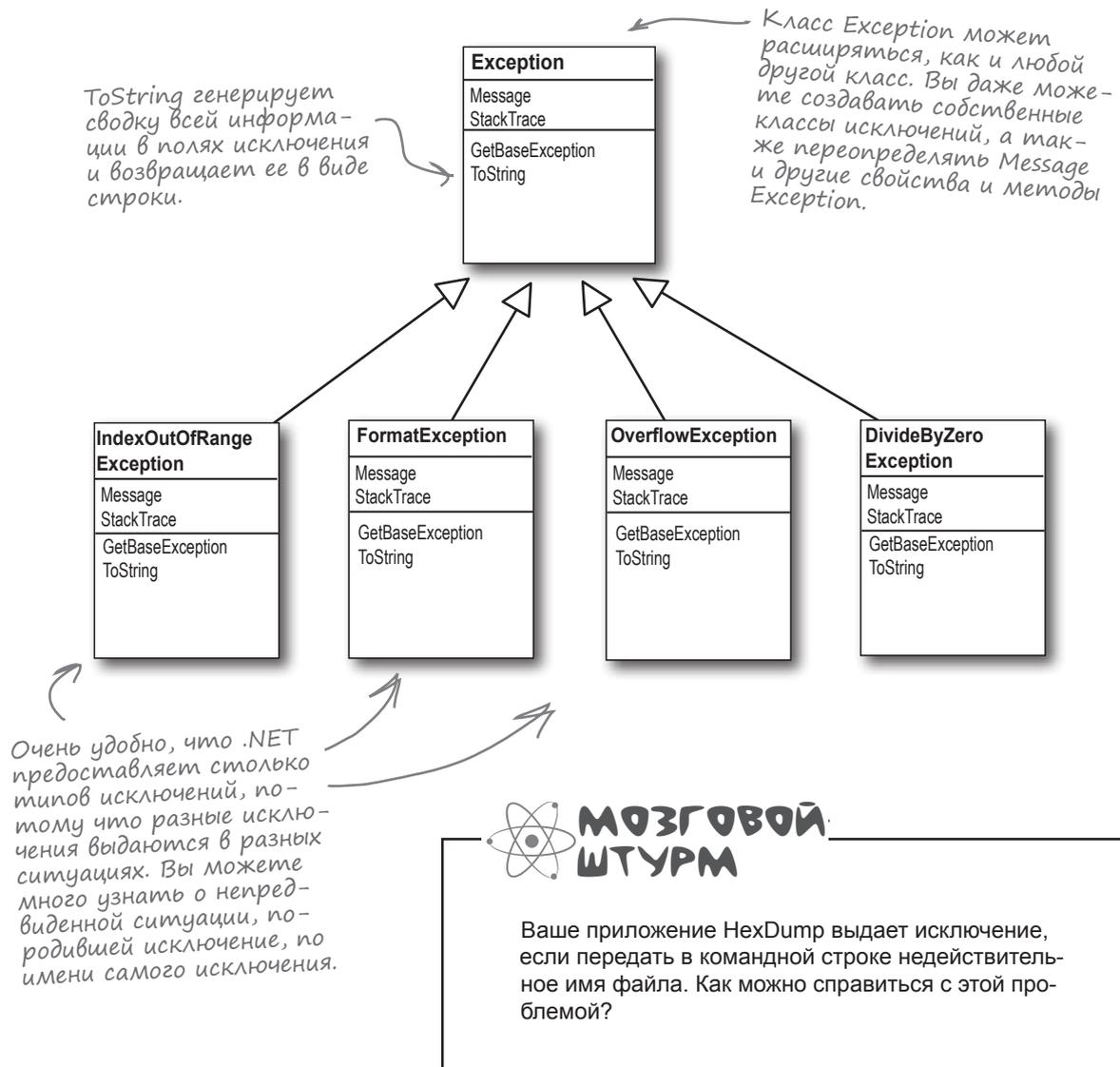
CLR идет на все хлопоты с созданием объекта, потому что среда должна предоставить полную информацию о причине возникновения исключения. Возможно, вам придется внести изменения в коде или какие-то исправления в обработку конкретной ситуации в вашей программе.

В данном случае выдается исключение **`IndexOutOfRangeException`**, которое раскрывает суть проблемы: вы пытаетесь обратиться к элементу с индексом, выходящим за границу массива. У вас же имеется информация о том, где именно в коде возникла проблема; это упростит ее поиск и исправление (даже если ваша программа содержит тысячи строк кода).

Все объекты Exception наследуют от System.Exception

.NET поддерживает множество разных исключений, о которых приходится сообщать пользователю. Так как многие из них обладают рядом сходных признаков, в игру вступает механизм наследования. В .NET определяется базовый класс Exception, от которого наследуют все конкретные типы исключений.

В классе Exception есть пара полезных компонентов. Свойство Message содержит удобочитаемое сообщение о проблеме. Свойство StackTrace сообщает, какой код выполнялся в момент возникновения исключения и какая последовательность вызовов привела к исключению. (Существуют и другие свойства, но пока мы ограничимся этими.)





Отладка любого исключения должна начинаться с изучения всей полученной информации. При выполнении консольного приложения эта информация выводится на консоль. Повнимательнее присмотримся к диагностической информации, выведенной приложением (мы переместили приложение в папку C:\HexDump, чтобы сократить пути в трассировке стека):

```
C:\HexDump\bin\Debug\netcoreapp3.1> HexDump invalid-filename
Unhandled exception. System.IO.FileNotFoundException: Could not find file 'C:\HexDump\bin\Debug\netcoreapp3.1\invalid-filename'.
File name: 'C:\HexDump\bin\Debug\netcoreapp3.1\invalid-filename'
   at System.IO.FileStream.ValidateFileHandle(SafeFileHandle fileHandle)
   at System.IO.FileStream.CreateFileOpenHandle(FileMode mode, FileShare share, FileOptions options)
   at System.IO.FileStream..ctor(String path, FileMode mode, ... , FileOptions options)
   at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess access, FileShare share)
   at System.IO.File.OpenRead(String path)
   at HexDump.Program.Main(String[] args) in C:\HexDump\Program.cs:line 12
```

Мы заметили в этой трассировке следующее:

- Класс исключения: System.IO.FileNotFoundException
- Сообщение об исключении: Could not find file 'C:\HexDump\bin\Debug\netcoreapp3.1\invalid-filename'.
- Дополнительная диагностическая информация: File name: 'C:\HexDump\bin\Debug\netcoreapp3.1\invalid-filename'
- Первые пять строк в трассировке стека относятся к классам из пространства имен System.IO
- Последняя строка трассировки стека относится к нашему пространству имен HexDump, и она включает номер строки в программе. Эта строка выглядит так: using (Stream input = File.Opened(args[0]))

Воспроизведите ошибку в отладчике

В конце главы 10 мы показали, как задать аргументы приложения при запуске приложения в отладчике. Задайте значение `invalid-filename`, затем установите точку прерывания в строке приложения, в которой произошло исключение. Запустите приложение, и когда управление будет передано в точку прерывания, выполните команду в пошаговом режиме. В IDE выводится информация об исключении.

Запуская программу из командной строки Mac, не забудьте заново опубликовать ее командой «dotnet publish -r osx -x64» из каталога решения.

Добавьте код для предотвращения исключения

Приложение выдает исключение, потому что оно пытается прочитать данные из несуществующего файла. Чтобы **предотвратить** исключение, следует сначала проверить, существует ли файл. Если он не существует, то вместо открытия потока с содержимым файла вызовом `File.OpenRead` будет использован вызов `Console.OpenStandardInput`, возвращающий поток **стандартного ввода** вашего приложения. Начните с **добавления метода `GetInputStream`** в приложение:

```
static Stream GetInputStream(string[] args) {
    if ((args.Length != 1) || !File.Exists(args[0]))
        return Console.OpenStandardInput();
    else
        return File.OpenRead(args[0]);
}
```

Console.OpenStandardInput возвращает объект **Stream**, связанный со **стандартным вводом приложения**. Если вы передаете входные данные приложения по каналу или запустили его в IDE и ввели **console** или **terminal**, все введенные или переданные по каналу данные попадут в поток.

Затем измените строку, в которой выдавалось исключение, и включите в нее вызов нового метода:

```
using (Stream input = GetInputStream(args))
```

Теперь запустите приложение в IDE. На этот раз оно не выдает исключение, а читает данные из стандартного ввода. Протестируйте его:

- Введите данные и нажмите Enter. Приложение выводит шестнадцатеричный дамп введенных данных, завершающийся разрывом строки (0d 0a в Windows, 0a на Mac). Поток `stdin` добавляет данные только после разрыва строки, так что приложение будет выводить новый дамп для каждой строки.
- Запустите приложение из командной строки: `HexDump << input.txt` (или `./HexDump << input.txt` на Mac). Приложение направляет данные из файла `input.txt` в поток `stdin` и выводит дамп всего содержимого файла.

Часто Задаваемые Вопросы

В: Так что же такое исключение?

О: Это объект, который создается CLR при возникновении проблемы. Также вы можете генерировать исключения в своем коде — собственно, вы уже делали это ранее с ключевым словом `throw`.

В: Исключение является *намеренно*?

О: Да, исключение является объектом. CLR создает его для передачи всей информации о том, что происходило при выполнении команды, породившей исключение. Его свойства, которые вы видели при просмотре расширенного описания `Exception` в окне Locals, содержат информацию об исключении. Например, свойство `Message` содержит полезную строку вида «Попытка деления на ноль» или «Слишком большое или слишком малое значение для `Int.32`».

В: Почему существует столько разных объектов `Exception`?

О: Потому, что вариантов непредвиденного поведения вашего кода очень много. Во многих ситуациях ваша программа просто аварийно завершится. Если бы вы не знали, почему произошел сбой, это бы очень сильно затруднило диагностику. Выдавая разные виды исключений в разных обстоятельствах, CLR предоставляет много ценной информации, упрощающей поиск и исправление проблем.

В: Означает ли это, что когда мой код выдает исключение, это не всегда происходит по моей вине?

О: Точно. Иногда входные данные не соответствуют вашим ожиданиям — как в примере работы с массивом, который оказался короче, чем предполагалось. Кроме того, не забывайте, что с вашими программами работают живые люди, поведение которых часто непредсказуемо. При помощи исключений `C#` упрощает обработку этих непредвиденных ситуаций, чтобы выполнение вашего кода шло гладко и он не выдавал загадочные и бесполезные сообщения об ошибке.

В: Выходит, исключения должны мне помочь, а не создавать проблемы, из-за которых мне приходится по ночам отлаживать свои приложения?

О: Да! Исключения помогают вам ожидать неожиданное. Многие разработчики огорчаются, когда в их коде происходит исключение. Вместо этого им стоило бы рассматривать исключение как помощь в выявлении проблем и отладке программ со стороны `C#`.

Похоже, исключения не всегда плохи. Иногда они помогают выявлять ошибки, но во многих случаях они лишь сообщают мне, что в программе произошло что-то неожиданное для меня.



Все верно. Исключения — полезный инструмент для выявления тех мест, в которых поведение вашего кода не соответствует ожидаемому.

Многие неопытные программисты пугаются при виде исключений. Однако исключения очень удобны, и вы можете обратить их себе на пользу. Исключение предоставляет вам многочисленные подсказки, по которым вы сможете определить, почему ваш код реагирует непредвиденным образом. Так вы узнаете о новых ситуациях, которые должны обрабатываться в программах, и получаете возможность **что-то с ними сделать**.

**Вся суть
исключе-
ний — поиск
и исправле-
ние ситуаций,
в которых
поведение ва-
шего кода не
соответствует
ожиданиям.**

Для некоторых файлов вывод дампа невозможен

В главе 9 мы рассказывали, как повесить **надежность** вашего кода, чтобы он справлялся с поврежденными данными, некорректно сформированным вводом, ошибками пользователя и другими непредвиденными ситуациями. Попытка вывода дампа стандартного ввода при отсутствии файла в командной строке или с указанием несуществующего файла — отличный пример повышения надежности программы.

Остались ли еще какие-то ситуации, которые необходимо обработать? Например, если файл существует, но недоступен для чтения? Посмотрим, что произойдет, если запретить чтение файла, а потом попытаться прочитать его:

- ★ **В Windows:** щелкните правой кнопкой мыши в Проводнике Windows, выберите команду Свойства, перейдите на вкладку Безопасность и щелкните на кнопке Изменить, чтобы изменить разрешения доступа к файлу. Установите все флажки Запретить.
- ★ **На Mac:** в окне терминала перейдите в папку с файлом, дамп которого вы хотите вывести, и выполните следующую команду, заменив binarydata.dat именем файла: `chmod 000 binarydata.dat`.

Теперь, когда с файла сняты разрешения чтения, попробуйте снова запустить приложение в IDE или в командной строке.

Программа выдает исключение: трассировка стека показывает, что **команда using вызвала метод GetInputStream**, в результате чего FileStream выдает исключение System.UnauthorizedAccessException:

```
C:\HexDump\bin\Debug\netcoreapp3.1>hexdump binarydata.dat
Unhandled exception. System.UnauthorizedAccessException: Access to the path 'C:\HexDump\bin\Debug\netcoreapp3.1\binarydata.dat' is denied.
   at System.IO.FileStream.ValidateFileHandle(SafeFileHandle fileHandle)
   at System.IO.FileStream.CreateFileOpenHandle(FileMode mode, ..., FileOptions options)
   at System.IO.FileStream..ctor(String path, ..., Int32 bufferSize, FileOptions options)
   at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess access, FileShare share)
   at System.IO.File.OpenRead(String path)
   at HexDump.Program.GetInputStream(String[] args) in C:\HexDump\Program.cs:line 14
   at HexDump.Program.Main(String[] args) in C:\HexDump\Program.cs:line 20
```



Одну минуту. Конечно, программа аварийно завершится. Я передаю ей нечитаемый файл. Пользователи постоянно ошибаются. И с этим ничего не поделаешь... **Не так ли?**

Вообще говоря, кое-что сделать **МОЖНО**

Да, пользователи постоянно ошибаются. Они вводят неправильную информацию, некорректные входные данные и из любопытства щелкают на том, о существовании чего вы даже не подозревали. Такова суровая реальность жизни, но это не означает, что с ней ничего не поделать. C# предоставляет удобные **средства обработки исключений**, которые повышают надежность вашей программы. Хотя вы *не можете* управлять тем, что пользователи делают с вашим приложением, вы *можете* принять меры к тому, чтобы приложение не завершилось аварийно при их возникновении.

Обработка исключений с try и catch

Чтобы добавить обработку исключений в своем коде, воспользуйтесь ключевыми словами try и catch и создайте блок кода, который будет выполняться при выдаче исключения.

Код try/catch фактически сообщает компилятору C#: «**Проверьте** (try) этот код и при появлении исключения **перехватите** (catch) его *этим* кодом». Проверяемая часть кода называется **блоком try**, а часть, обрабатывающая исключения, — **блоком catch**. В блоке catch можно выполнять различные действия, например вывести сообщение об ошибке, вместо того чтобы дать программе аварийно завершиться.

Еще раз взгляните на последние три строки трассировки стека в сценарии HexDump, чтобы понять, где следует разместить код обработки исключений:

```
at System.IO.File.OpenRead(String path)
at HexDump.Program.GetInputStream(String[] args) in Program.cs:line 14
at HexDump.Program.Main(String[] args) in Program.cs:line 20
```

Исключение UnauthorizedAccessException вызвано строкой GetInputStream, из которой вызывается File.OpenRead. Так как это исключение невозможно предотвратить, изменим GetInputStream для использования блока try/catch:

```
static Stream GetInputStream(string[] args)
{
    if ((args.Length != 1) || !File.Exists(args[0]))
        return Console.OpenStandardInput();
    else
    {
        try
        {
            return File.OpenRead(args[0]);
        }
        catch (UnauthorizedAccessException ex)
        {
            Console.Error.WriteLine("Unable to read {0}, dumping from stdin: {1}",
                args[0], ex.Message);
            return Console.OpenStandardInput();
        }
    }
}
```

Это блок try. Обработка исключений начинается с ключевого слова «try». В данном случае за ним размещается существующий код.

Разместите код, который может выдать исключение, в блоке try. Если исключения не произойдет, все работает как обычно и команды в блоке catch игнорируются. Если команда в блоке try выдает исключение, то остальная часть блока try не выполняется.

Ключевое слово catch сигнализирует о том, что блок, следующий непосредственно за ним, содержит обработчик исключения.

Если в блоке try выдается исключение, программа немедленно переходит к команде catch и начинает выполнять блок catch.

Мы сознательно не стали усложнять свой обработчик исключений. Сначала вызов Console.Error выводит строку в поток ошибок (stderr), сообщая пользователю о произошедшей ошибке, после чего программа возвращается к чтению данных из стандартного ввода. Обратите внимание: **в блоке catch присутствует команда return**. Метод возвращает Stream, поэтому и при обработке исключения он все равно должен вернуть Stream; в противном случае компилятор выдаст ошибку «Не все ветви кода возвращают значение».



Если при выдаче исключения происходит автоматический переход к блоку catch, что произойдет с объектами и данными, с которыми вы работали до выдачи исключения?

Отслеживание передачи управления в try/catch

Важный аспект обработки исключений заключается в том, что когда команда в вашем блоке `try` выдает исключение, весь остальной код в блоке **обходится**. Управление немедленно передается первой строке блока `catch`. Отладчик IDE поможет разобраться в том, как работает эта схема.

Принципы отладки

- 1 Замените метод `GetInputStream` в приложении `HexDump` методом, который был приведен выше.
- 2 Измените конфигурацию проекта и задайте аргумент так, чтобы он содержал путь к файлу, чтение которого запрещено.
- 3 Установите точку прерывания в первой команде `GetInputStream`. Запустите отладку проекта.
- 4 При достижении точки прерывания выполните несколько следующих команд в пошаговом режиме, пока не достигнете вызова `File.OpenRead`. Выполните следующую команду без захода в метод — приложение переходит к первой строке блока `catch`.

Точка прерывания, установленная в первой строке `GetInputStream`.

Пошаговое выполнение метода. При достижении `File.OpenRead` будет выдано исключение, в результате чего управление передается блоку `catch`.

```

HexDump - Program.cs
Program.cs
HexDump
HexDump.Program
GetInputStream(string[] args)
1 reference
9 static Stream GetInputStream(string[] args)
10 {
11     if ((args.Length != 1) || !File.Exists(args[0]))
12         return Console.OpenStandardInput();
13     else
14     {
15         try
16         {
17             return File.OpenRead(args[0]);
18         }
19         catch (UnauthorizedAccessException ex) < 13ms elapsed
20         {
21             Console.Error.WriteLine("Unable to read {0}, dumping from stdin: {1}",
22                                     args[0], ex.Message);
23             return Console.OpenStandardInput();
24         }
25     }
26 }
110 % No issues found Ln: 19 Ch: 17 SPC CRLF
    
```

- 5 Продолжайте выполнять блок `catch`. Он выводит строку на консоль, после чего возвращает `Console.OpenStandardInput` и возобновляет выполнение метода `Main`.

Когда блок `finally` выполняется ВСЕГДА

При возникновении исключения возможны два варианта развития событий. **Необработанное** исключение ведет к аварийному завершению программы. В противном случае управление переходит к блоку `catch`. Но что происходит с остальным кодом блока `try`? Представьте, что в этой части закрывался поток или освобождались важные ресурсы. Тогда код должен быть выполнен, несмотря на исключение. На помощь в этой ситуации приходит блок `finally`. Он **выполняется всегда**, независимо от выдачи или отсутствия исключения. Воспользуемся отладчиком и разберемся в том, как работает блок `finally`.

1 Создайте новый проект консольного приложения.

Добавьте директиву `using System.IO`; в начало файла, затем добавьте метод `Main`:

```
static void Main(string[] args)
{
    var firstLine = "No first line was read";
    try
    {
        var lines = File.ReadAllLines(args[0]);
        firstLine = (lines.Length > 0) ? lines[0] : "The file was empty";
    }
    catch (Exception ex)
    {
        Console.Error.WriteLine("Could not read lines from the file: {0}", ex);
    }
    finally
    {
        Console.WriteLine(firstLine);
    }
}
```

Принципы отладки

WriteLine вызывает метод ToString объекта Exception, который возвращает имя исключения, сообщение и трассировку стека.

Этот блок finally будет выполнен независимо от того, обнаружит блок try исключение или нет.

Исключение появляется в окне Locals, как и прежде.

2 Добавьте точку прерывания в первую строку метода `Main`.

Запустите отладку приложения и выполняйте его в пошаговом режиме. Первая строка блока `try` пытается обратиться к `args[0]`, но так как аргументы командной строки не заданы, массив `args` пуст и программа выдает исключение, а именно `System.IndexOutOfRangeException` с сообщением «Индекс выходит за границу массива». После вывода сообщения **выполняется блок `finally`**, и программа завершается.

3 Задайте аргумент командной строки, содержащий путь к существующему файлу.

Используйте свойства проекта для передачи аргумента командной строки приложению. Присвойте ему полный путь к существующему файлу. Убедитесь в том, что имя файла не содержит пробелов; в противном случае приложение интерпретирует его как два аргумента. Снова включите отладку своего приложения — после завершения блока `try` будет **выполнен блок `finally`**.

4 Задайте аргумент командной строки, содержащий путь к несуществующему файлу.

Вернитесь к свойствам проекта и измените аргумент командной строки, чтобы приложению передавалось имя несуществующего файла. Снова запустите свое приложение. На этот раз будет выдано другое исключение: `System.IO.FileNotFoundException`. Затем **выполняется блок `finally`**.

Перехват всех исключений

Вы только что заставили свое приложение выдавать исключения двух типов (`IndexOutOfRangeException` и `FileNotFoundException`), и оба типа были обработаны. Взгляните на блок `catch`:

```
catch (Exception ex)
```

Это условие **перехватывает все исключения**: тип после блока `catch` указывает, какой тип исключений должен обрабатываться, а поскольку все исключения расширяют класс `System.Exception`, определение типа `Exception` в условии приказывает блоку `try/catch` перехватывать любые исключения.

Предотвращение перехвата всех исключений при помощи множественных блоков `catch`

Всегда лучше постараться спрогнозировать конкретные исключения, которые будут выдаваться вашей программой, и обработать их. Например, мы знаем, что если имя файла не задано, этот код будет выдавать исключение `IndexOutOfRangeException`, а если задано недопустимое имя файла, код выдает исключение `FileNotFoundException`. Ранее в этой главе также было показано, что при попытке чтения нечитаемого файла CLR выдает `UnauthorizedAccessException`. Чтобы в приложении по-разному обрабатывались разные виды исключений, добавьте несколько **блоков `catch`** в свой код:

```
static void Main(string[] args)
{
    var firstLine = "No first line was read";
    try
    {
        var lines = File.ReadAllLines(args[0]);
        firstLine = (lines.Length > 0) ? lines[0] : "The file was empty";
    }
    catch (IndexOutOfRangeException)
    {
        Console.Error.WriteLine("Please specify a filename.");
    }
    catch (FileNotFoundException)
    {
        Console.Error.WriteLine("Unable to find file: {0}", args[0]);
    }
    catch (UnauthorizedAccessException ex) ← Только в этом блоке catch задается
    {                                     имя переменной для объекта Exception.
        Console.Error.WriteLine("File {0} could not be accessed: {1}",
                                args[0], ex.Message);
    }
    finally
    {
        Console.WriteLine(firstLine);
    }
}
```

Обработчик исключения со-держит три блока `catch` для обработки трех разных типов исключений.

Теперь ваше приложение выводит разные сообщения об ошибках в зависимости от того, какое исключение было обработано программой. Обратите внимание на то, что в первых двух блоках `catch` **не указано имя переменной** (например, `ex`, как в третьем блоке). Имя переменной необходимо задавать только в том случае, если вы собираетесь работать с объектом `Exception`.



Можно ли предотвратить какие-либо из этих исключений (вместо того, чтобы обрабатывать их)?

Часть Задаваемые Вопросы

В: Выходит, каждый раз, когда в моей программе возникает исключение, она прервет работу, если только я не напишу код для перехвата этого исключения. Что же в этом хорошего?

О: Одно из главных преимуществ исключений — то, что они не дают пройти мимо проблемы. В больших и сложных приложениях трудно следить за всеми объектами, с которыми работает программа. Исключения привлекают внимание к проблемам и помогают понять причины их возникновения, чтобы вы всегда знали, соответствует ли поведение программы вашим ожиданиям.

Появление исключения в программе предупреждает: что-то идет не так, как планировалось. Может быть, ссылка указывает не на тот объект, на который нужно, или пользователь ввел совсем не то значение, которое требовалось, или даже файл, с которым ведется работа, вдруг стал недоступен. Если в программе происходит нечто подобное, а вы об этом даже не знаете, скорее всего, результат работы программы изменится, а ее поведение с этого момента будет сильно отличаться от того, чего вы ожидали при написании программы.

А теперь представьте, что вы даже не знаете обо всех этих ошибках. Пользователь вводит некорректные данные и начинает жаловаться, что приложение нестабильно. Вот почему так хорошо, что исключения прерывают работу вашей программы. Они заставляют вас разобрататься с проблемой на том этапе, когда ее еще можно решить легко и безболезненно.

В: Напомните, для чего мне использовать объект `Exception`?

О: Объект `Exception` содержит информацию о том, что пошло не так. Вы можете воспользоваться типом объекта исключения для определения того, какая проблема возникла в программе, и написать обработчик исключения, который сохранит работоспособность программы.

В: Чем *обработанное* исключение отличается от *необработанного*?

О: При появлении исключений исполнительная среда начинает искать в коде блок `catch`, обрабатывающий это исключение. Если такой блок присутствует, блок `catch` выполнит все действия, предусмотренные вами для этого конкретного исключения. Так как вы заранее написали блок `catch` для такого исключения, оно считается обработанным.

Если же исполнительная среда не находит блок `catch`, программа просто прекращает работу и выдает сообщение об ошибке. В этом случае речь идет о *необработанном* исключении.

В: Что происходит, если в условии `catch` не указан конкретный тип исключения?

О: Это называется **универсальным исключением**. Такой блок `catch` перехватывает любые исключения, выдаваемые блоком `try`. Таким образом, если вам не нужно объявлять переменную для использования объекта `Exception`, простейший универсальный обработчик выглядит так:

```
catch
{
    // Обработка исключения
}
```

В: Зачем указывать в блоке `catch` тип обрабатываемого исключения? Разве не безопаснее использовать универсальный код?

О: Старайтесь как можно реже перехватывать `Exception`. Знаете старую поговорку, что щепотка профилактики лучше горсти лекарств? Это правило относится и к исключениям. Попытка обработать все исключения сразу обычно является признаком плохого стиля программирования. Например, лучше воспользоваться методом `File.Exists` для проверки наличия файла, а не обрабатывать исключение `FileNotFoundException`. Хотя бывают исключения, которые попросту неизбежны, большинство из них можно предотвратить.

В: Но если блок `catch` без заданного исключения может обрабатывать любые исключения, зачем указывать конкретный тип?

О: Не существует универсального способа обработки всех исключений. Например, для устранения проблемы с делением на ноль в блоке `catch` нужно изменить значения некоторых переменных и сохранить некоторые данные. А чтобы обработать исключение `null-ссылки`, возможно, вам придется создать новый экземпляр.

Необработанное исключение может привести к непредсказуемому поведению программы. Вот почему программа прерывает работу при возникновении исключения.

У бассейна



Ваша **задача** — взять фрагменты кода из бассейна и разместить их в пустых строках программы. Любой фрагмент можно использовать несколько раз, использовать все фрагменты не обязательно. Ваша **цель** — добиться того, чтобы программа вывела приведенный ниже результат.

Результат: **—————> G'day Mate!**

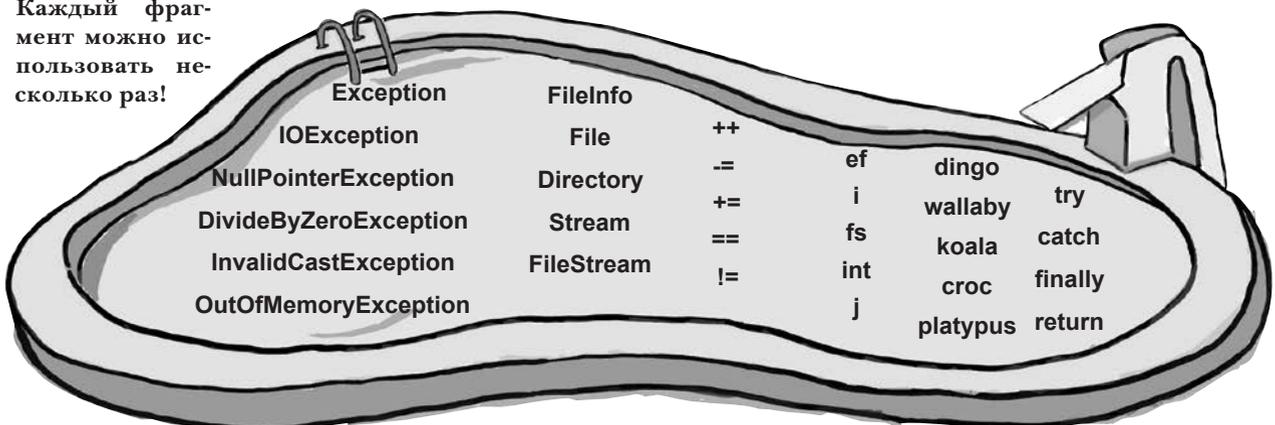
```
using System.IO;

class Program {
    public static void Main() {
        Kangaroo joe = new Kangaroo();
        int koala = joe.Wombat(
            joe.Wombat(joe.Wombat(1)));
        try {
            Console.WriteLine((15 / koala)
                + " eggs per pound");
        }
        catch (_____ ) {
            Console.WriteLine("G'Day Mate!");
        }
    }
}
```

```
class Kangaroo {
    _____ fs;
    int croc;
    int dingo = 0;

    public int Wombat(int wallaby) {
        _____ ;
        try {
            if (_____ > 0) {
                fs = File.OpenWrite("wobbiegong");
                croc = 0;
            } else if (_____ < 0) {
                croc = 3;
            } else {
                _____ = _____.OpenRead("wobbiegong");
                croc = 1;
            }
        }
        catch (IOException) {
            croc = -3;
        }
        catch {
            croc = 4;
        }
        finally {
            if (_____ > 2) {
                croc _____ dingo;
            }
        }
        _____ ;
    }
}
```

Каждый фрагмент можно использовать несколько раз!



У бассейна. Решение



На то, что это `FileStream`, указывает наличие метода `OpenRead` и выдача исключения `IOException`.

Код открывает файл с именем «wobbiegong» и оставляет его открытым при первом вызове. Позднее он открывает файл снова. Но файл нигде не закрывается, в результате чего выдается исключение `IOException`.

Не забывайте: универсального перехвата исключений следует избегать. Также следует избегать того, что нам пришлось сделать, чтобы головоломка осталась интересной, — например, бессодержательных имен переменных.

```
using System.IO;

class Program {
    public static void Main() {
        Kangaroo joe = new Kangaroo();
        int koala = joe.Wombat(
            joe.Wombat(joe.Wombat(1)));
        try {
            Console.WriteLine((15 / koala)
                + " eggs per pound");
        }
        catch ( DivideByZeroException ) {
            Console.WriteLine("G'Day Mate!");
        }
    }
}

class Kangaroo {
    FileStream fs;
    int croc;
    int dingo = 0;

    public int Wombat(int wallaby) {
        dingo ++;
        try {
            if ( wallaby > 0 ) {
                fs = File.OpenWrite("wobbiegong");
                croc = 0;
            } else if ( wallaby < 0 ) {
                croc = 3;
            } else {
                fs = File.OpenRead("wobbiegong");
                croc = 1;
            }
        }
        catch (IOException) {
            croc = -3;
        }
        catch {
            croc = 4;
        }
        finally {
            if ( dingo > 2 ) {
                croc -= dingo;
            }
        }
        return croc ;
    }
}

```

Метод `joe.Wombat` вызывается три раза, на третий раз он возвращает 0. Это приводит к тому, что `WriteLine` выдает исключение `DivideByZeroException`.

Блок `catch` перехватывает только исключения с делением на ноль.

Вы уже знаете, что файлы всегда необходимо закрывать при завершении работы с ними. Если этого не сделать, файл останется заблокированным и при попытке его повторного открытия произойдет исключение `IOException`.



Вы продолжаете говорить о **небезопасном** коде, но разве безопасно оставлять исключение необработанным? Зачем писать обработку исключений, которая не обрабатывает все возможные типы исключений?

Необработанные исключения всплывают.

Хотите верить, хотите нет, но иногда оставлять исключения необработанными даже полезно. Реальные программы содержат сложную логику, и часто бывает трудно правильно восстановить работоспособность программы, когда что-то пошло не так, особенно если проблема происходит на нижних уровнях программы. Обрабатывая только конкретные исключения и избегая универсальных обработчиков, вы позволяете непредвиденным исключениям **всплывать**: вместо того чтобы обрабатываться в текущем методе, они перехватываются следующей командой в стеке вызовов. Если вы будете обрабатывать ожидаемые исключения и позволите необработанным исключениям всплывать, это будет способствовать построению более надежных приложений.

Иногда бывает полезно **заново выдать** исключение; это означает, что исключение обрабатывается в методе, но будет всплывать до команды, из которой было вызвано. Все, что необходимо для повторной выдачи исключений, — вызвать `throw`; внутри блока `catch`. Перехваченное исключение немедленно всплывает:

```
try {
    // код, который может выдать исключение
} catch (DivideByZeroException d) {
    Console.Error.WriteLine($"Got an error: {d.Message}");
    throw;
}
```

Команда `throw` заставляет исключение `DivideByZeroException` всплыть до кода, вызвавшего этот блок `try/catch`.

Подсказка для карьерного роста: на многих собеседованиях на вакансии по программированию на C# встречается вопрос о том, как обрабатывать исключения в конструкторе.



Будьте осторожны!

Уберите небезопасный код из конструктора!

К настоящему моменту вы уже знаете, что конструктор не имеет возвращаемого значения, даже `void`. Дело в том, что конструктор ничего не возвращает. Его единственной целью является инициализация объекта, и это создает проблемы для обработки исключений внутри конструктора. Если исключение выдается в конструкторе, команда, которая пытается создать экземпляр класса, **этот экземпляр не получит**.

Использование исключения, подходящего для конкретной ситуации

При использовании IDE для генерирования метода добавляется код следующего вида:

```
private void MyGeneratedMethod()
{
    throw new NotImplementedException();
}
```

Исключение **NotImplementedException** используется каждый раз, когда в программе вызывается не-реализованная операция или метод. Оно прекрасно подходит для временных реализаций — при виде этого исключения вы знаете, что есть код, который необходимо написать. Это лишь одно из многих исключений, предоставляемых .NET.

Выбор подходящего исключения упростит чтение вашего кода, а обработка исключений станет более ясной и надежной. Например, код в методе, проверяющем его параметры, может выдать исключение **ArgumentException**, у которого имеется перегруженный конструктор с параметром, сообщаящим, какой аргумент создал проблему. Вспомните класс **Guy** из главы 3: его метод **ReceiveCash** проверял параметр **amount**, чтобы убедиться в том, что он получает положительную сумму. В такой ситуации уместно выдать исключение **ArgumentException**:

```
public void ReceiveCash(int amount)
{
    if (amount <= 0)
        throw new ArgumentException("Must receive a positive value", "amount");
    Cash += amount;
}
```

*Имя недействительного аргумента передается конструктору **ArgumentException**.*

Посмотрите список исключений, являющихся частью .NET API, — любые из этих исключений вы сможете выдать в своем коде: <https://docs.microsoft.com/en-us/dotnet/api/system.systemexception>.

Перехват специализированных исключений, расширяющих **System.Exception**

Иногда программа должна выдавать исключение из-за специального условия, которое может возникнуть при выполнении. Вернемся к классу **Guy** из главы 3. Допустим, вы используете его в приложении, которое полностью зависит от того, что **Guy** всегда имеет положительную сумму денег. Вы можете добавить специализированное исключение, **расширяющее **System.Exception****:

```
class OutOfCashException : System.Exception {
    public OutOfCashException(string message) : base(message) { }
}
```

*Специализированное исключение **OutOfCashException** расширяет базовый конструктор **System.Exception** параметром **message**.*

Теперь можно выдать это новое исключение и перехватить его точно так же, как и любое другое исключение:

```
class Guy
{
    public string Name;
    public int Cash;

    public int GiveCash(int amount)
    {
        if (Cash <= 0) throw new OutOfCashException($"{Name} ran out of cash");
        ...
    }
}
```

*Теперь **Guy** выдает специализированное исключение, и любой метод, который вызывает **GiveCash**, может обработать это исключение в собственном блоке **try/catch**.*

```
class Program {
    public static void Main(string[] args) {
        Console.WriteLine("when it ");
        ExTestDrive.Zero("yes");
        Console.WriteLine(" it ");
        ExTestDrive.Zero("no");
        Console.WriteLine(".");
    }
}
```



Развлечения с Магнитами

Разместите магниты так, чтобы программа выводила на консоль следующий результат:

when it thaws it throws.

```
class MyException : Exception { }
```

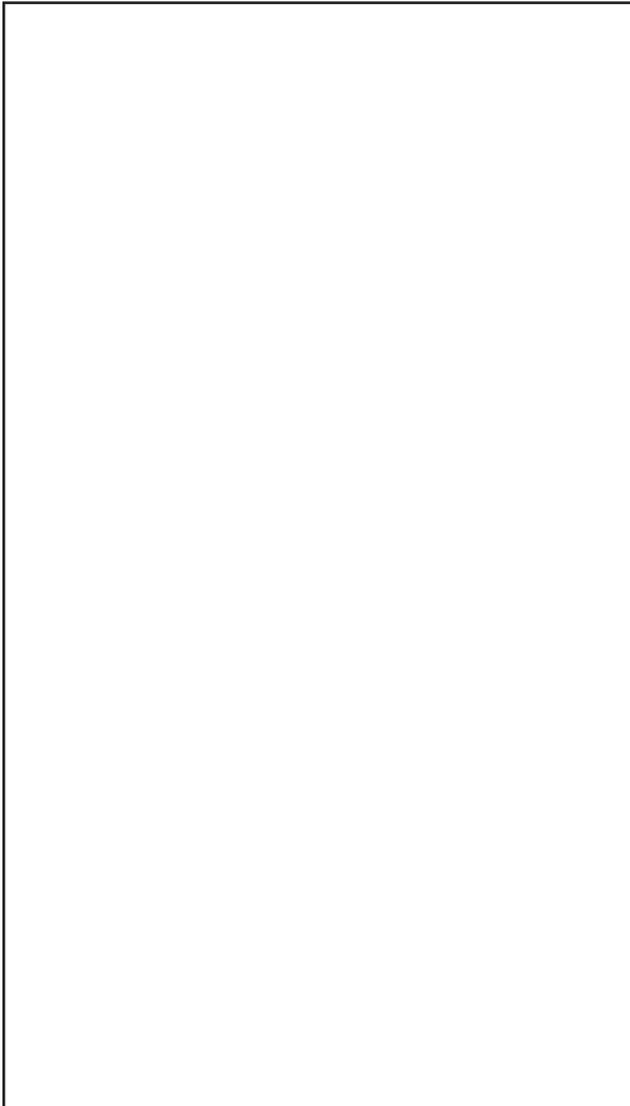


Diagram showing code snippets and their assembly into a final program structure:

- try {
- Console.WriteLine("r");
- DoRisky(test);
- finally {
- Console.WriteLine("o");
- Console.WriteLine("w");
- Console.WriteLine("s");
- throw new MyException();

```
class ExTestDrive {
    public static void Zero(string test) {
        static void DoRisky(String t) {
            Console.WriteLine("h");
            try {
                if (t == "yes") {
                    Console.WriteLine("a");
                }
            } catch (MyException) {
                Console.WriteLine("t");
            }
        }
    }
}
```

```
class Program {  
    public static void Main(string[] args) {  
        Console.WriteLine("when it ");  
        ExTestDrive.Zero("yes");  
        Console.WriteLine(" it ");  
        ExTestDrive.Zero("no");  
        Console.WriteLine(".");  
    }  
}
```



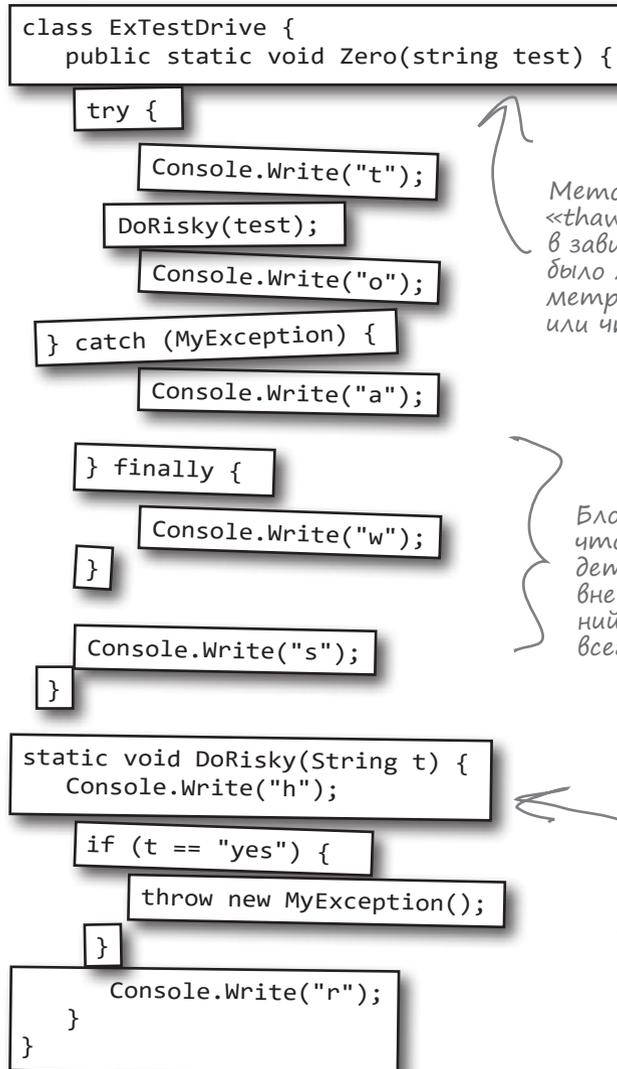
Развлечения с Магнитами. Решение

Разместите магниты так, чтобы программа выводила на консоль следующий результат:

when it thaws it throws.

```
class MyException : Exception { }
```

↑
Эта строка определяет специализированное исключение с именем MyException, которое перехватывает в блоке catch в коде.



Метод Zero выводит либо «thaws», либо «throws» в зависимости от того, было ли передано в параметре test значение "yes" или что-то другое.

Блок finally гарантирует, что метод всегда выведет "w", а "s" выводится вне обработчика исключений (и поэтому выводится всегда).

Эта строка выполняется только в том случае, если DoRisky не выдаст исключение.

Метод DoRisky выдает исключение только при передаче строки "yes".



IDisposable использует try/finally для обеспечения вызова метода Dispose

За сценой

Помните, как мы анализировали этот код из главы 10?

```
using System.IO;
using System.Text;
```

```
class Program
{
    static void Main(string[] args)
    {
        using (var ms = new MemoryStream())
        {
            using (var sw = new StreamWriter(ms))
            {
                sw.WriteLine("The value is {0:0.00}", 123.45678);
            }
            Console.WriteLine(Encoding.UTF8.GetString(ms.ToArray()));
        }
    }
}
```

Эта команда using переместилась вовнутрь, чтобы метод StreamWriter.Close вызывался перед методом MemoryStream.Close.

Мы разбирались в том, как работают команды using, и в данном случае нам пришлось вложить одну команду using в другую, чтобы гарантировать, что экземпляр StreamWriter будет освобожден ранее MemoryStream. Мы поступили так, потому что оба класса, StreamWriter и MemoryStream, реализуют интерфейс IDisposable, и включили вызов метода Close в их методы Dispose. Каждая команда using гарантирует, что метод Dispose будет вызван в конце ее блока; это гарантирует, что потоки всегда будут закрываться.

Теперь вы знаете, как работает обработка исключений, и сможете понять, как работает команда using. Команда using является примером **синтаксического сахара**: C# предоставляет удобную сокращенную запись, которая упрощает чтение кода:

```
using (var sw = new StreamWriter(ms))
{
    sw.WriteLine("The value is {0:0.00}", 123.45678);
}
```

Компилятор C# генерирует откомпилированный код, который выглядит (приблизительно) так:

```
try {
    var sw = new StreamWriter(ms)
    sw.WriteLine("The value is {0:0.00}", 123.45678);
} finally {
    sw.Dispose();
}
```

Размещение команды Dispose в блоке finally гарантирует, что он будет выполнен в любом случае, даже при возникновении исключения.

IDisposable — эффективный инструмент для предотвращения исключения.

ИЗБЕГАЙТЕ ЛИШНИХ ИСКЛЮЧЕНИЙ... ВСЕГДА ИСПОЛЬЗУЙТЕ БЛОК USING ПРИ РАБОТЕ С ПОТОКОМ!

Или все, что угодно, что реализует IDisposable.

Фильтры исключений повышают точность обработки исключений

Допустим, вы строите игру, действие которой происходит в классической мафиозной атмосфере 1930-х годов. Класс LoanShark (Ростовщик) должен собирать деньги от экземпляров Guy при помощи метода Guy.GiveCash и обрабатывать исключения OutOfCashException (Денег нет) классическим уроком по правилам мафии.

Каждый ростовщик знает золотое правило: не пытайтесь выбивать деньги из босса мафии. В таких ситуациях вам могут пригодиться **фильтры исключений**. Фильтр исключения использует ключевое слово `when`, чтобы приказать обработчику исключения перехватывать исключение только в конкретных условиях.

Пример использования фильтра исключений:

```
try
{
    loanShark += guy.GiveCash(amount);
    emergencyReserves -= amount;
} catch (OutOfCashException) when (guy.Name == "Al Capone")
{
    Console.WriteLine("Don't mess with the mafia boss");
    loanShark += amount;
} catch (OutOfCashException ex)
{
    Console.Error.WriteLine($"Time to teach {guy.Name} a lesson: {ex.Message}");
}
```

Этот фильтр исключений перехватывает OutOfCashException только в том случае, если guy.Name содержит "Al Capone", в противном случае исключение пройдет насквозь к следующему блоку catch.

Лучше не серди нас, парень.



Если try/catch — такая замечательная штука, почему бы IDE не заключать в них все подряд? Тогда нам не пришлось бы писать все эти блоки try/catch самостоятельно. Не так ли?

Всегда лучше определить самый точный обработчик исключений из всех возможных.

Обработка исключений не сводится к простой выдаче сообщения об ошибке. Иногда разные исключения должны обрабатываться по-разному — например, программа HexDump обрабатывала исключение FileNotFoundException не так, как исключение UnauthorizedAccessException. В планировании для обработки исключений всегда учитываются **непредвиденные ситуации**. Иногда такие ситуации можно предотвратить, иногда они обрабатываются, а иногда исключения должны всплывать. Из всего сказанного следует вынести важный урок: не существует единого подхода к обработке непредвиденных условий «на все случаи жизни»; именно по этой причине IDE не пытается упаковывать все подряд в блоки try/catch.

Вот почему существует так много разных классов, производных от Exception. И возможно, вам тоже захочется написать собственные классы, наследующие от Exception.



Часть Задаваемые Вопросы

В: Мне все еще непонятно, когда перехватывать исключения, когда предотвращать и когда разрешать им всплывать.

О: Не существует единственно правильного ответа или правила, которое можно соблюдать в каждой ситуации, — это всегда зависит от того, что должен делать ваш код.

Впрочем, и это утверждение работает не на сто процентов. Существует правило: лучше предотвращать все исключения, если это возможно. Не всегда можно предвидеть все неожиданности, особенно когда вы имеете дело с вводом от пользователей или решениями, которые они принимают.

Решения относительно того, разрешать ли исключениям всплывать или обрабатывать их в классе, часто сводятся к разделению обязанностей. Насколько разумно классу знать о существовании определенного исключения? Это зависит от того, что делает класс. К счастью, средства рефакторинга IDE всегда помогут вам изменить этот код, если вы решите, что некоторые исключения лучше передавать на верхние уровни, чем перехватывать.

В: Объясните, что такое «синтаксический сахар»?

О: Когда разработчики упоминают «синтаксический сахар», они обычно имеют в виду, что язык программирования предоставляет удобную и понятную сокращенную запись для кода, который обычно бывает громоздким и неудобным. Термином «синтаксис» обозначаются ключевые слова C# и правила, которые ими управляют. Команда using официально является частью синтаксиса C# и подчиняется следующему правилу: за ним должно следовать объявление переменной, которое создает экземпляр типа, реализующего IDisposable, а затем следует блок кода. Под «сахаром» имеется в виду тот факт, что компилятор C# преобразует сокращенную запись в намного более громоздкий вариант, если бы ее пришлось записывать вручную.

В: Возможно ли использовать объект с командой using, если он не реализует IDisposable?

О: Нет. Командой using можно создавать только объекты, реализующие IDisposable, потому что они были разработаны в расчете друг на друга. Добавление команды using аналогично созданию нового экземпляра класса, не считая того, что в конце блока всегда вызывается метод Dispose. Вот почему класс должен реализовать интерфейс Disposable.

В: В блоке using можно размещать любые команды?

О: Разумеется. Команда using всего лишь гарантирует уничтожение любого созданного вами объекта, но что вы делаете с этими объектами, зависит исключительно от вас. Можно даже создать объект при помощи команды using и не использовать его внутри блока (конечно, это было бы довольно бессмысленно, так что мы так поступать не рекомендуем).

В: Можно ли вызвать Dispose вне команды using?

О: Да. На самом деле использовать команду using вообще не обязательно. Вы можете самостоятельно вызвать Dispose, завершив работу с объектом. Метод освободит указанные ресурсы, как и при ручном вызове метода Close для потока (так делалось в главе 10). Команда using всего лишь облегчает чтение и понимание кода, а также предотвращает проблемы, которые могут возникнуть, если объект не будет своевременно удален.

В: Так как команда using, по сути, генерирует блок try/catch, который вызывает метод Dispose, можно ли выполнять обработку исключений внутри блока using?

О: Да. Она работает точно так же, как вложенные блоки try/catch, которые использовались ранее в главе с методом GetInputStream.

В: Вы упомянули блок try/finally. Означает ли это, что команды try и finally могут фигурировать без catch?

О: Да! Блок try можно скомбинировать непосредственно с блоком finally, как показано ниже:

```
try {
    DoSomethingRisky();
    SomethingElseRisky();
}
finally {
    AlwaysExecuteThis();
}
```

Если метод DoSomethingRisky() выдаст исключение, немедленно будет выполнен блок finally.

В: Метод Dispose работает только с файлами и потоками?

О: Нет. Многие классы реализуют интерфейс IDisposable, и при работе с ними всегда нужно использовать команду using. Если вы напишете класс, который нужно утилизировать определенным способом, также реализуйте интерфейс IDisposable.

Не существует единственно правильного подхода к планированию непредвиденного.

Наихудший блок catch: универсальный перехват с комментариями

Блок `catch` позволяет программе продолжить работу, если разработчик предусмотрел такую возможность. Появившееся исключение обрабатывается, и вместо аварийной остановки и сообщения об ошибке программа продолжает работать. Как ни странно, это не всегда хорошо.

Рассмотрим класс `Calculator`, с которым явно что-то не так. Что же происходит?

```
class Calculator
{
    public void Divide(int dividend, int divisor)
    {
        try
        {
            this.quotient = dividend / divisor;
        } catch {

            // TODO: придумать, как предотвратить ввод
            // пользователем нулевого делителя.

        }
    }
}
```

Проблема: если делитель равен нулю, выдается исключение `DivideByZeroException`.

Программист подумал, что сможет скрыть исключения при помощи пустого блока `catch`, но он всего лишь создал проблему тому, кто позднее будет разбираться с ошибкой.

Исключения нужно обрабатывать, а не скрывать

Тот факт, что программа продолжает работу, еще не означает, что исключение было *обработано*. Приведенный выше код не будет аварийно остановлен... по крайней мере, не в методе `Divide`. Но что, если этот метод вызывается другим методом, который пытается вывести результат? Если делитель равен нулю, метод, скорее всего, вернет неправильное (и неожиданное) значение.

Нужно не просто добавить комментарий и скрыть исключение, а **обработать исключение**. Даже если вы не можете справиться с проблемой сейчас, *не оставляйте блок catch пустым или закомментированным!* Это лишь усложнит исправление ошибки в будущем. Лучше пусть программа выдаст сообщение об исключении, — так вам будет проще разобраться, что именно работает не так.

Помните, что если исключение не обрабатывается, оно поднимается вверх в стеке вызовов. В некоторых ситуациях это становится совершенно нормальным вариантом обработки исключения. Предоставить исключению возможность всплывать наверх почти всегда лучше, чем пустой блок `catch`.

Временные решения допустимы (но только временно)

Иногда, столкнувшись с проблемой, вы понимаете, что нужно что-то делать, но не знаете, что именно. В этом случае имеет смысл зарегистрировать ошибку в журнале, снабдив ее примечанием. Это не так хорошо, как обработка исключения, но лучше, чем ничего.

Временное решение для калькулятора может выглядеть так:

```
class Calculator
{
    public void Divide(int dividend, int divisor)
    {
        try
        {
            this.quotient = dividend / divisor;
        } catch (Exception ex) {
            using (StreamWriter sw = new StreamWriter(@"C:\Logs\errors.txt");
            {
                sw.WriteLine(ex.getMessage());
            }
        }
    }
}
```

Остановитесь и проанализируйте блок catch. Что произойдет, если StreamWriter не сможет сделать запись в папку C:\Logs\? Для снижения риска можно вложить внутрь еще один блок try/catch. Можете предложить лучший способ?

```
using (StreamWriter sw = new StreamWriter(@"C:\Logs\errors.txt");
{
    sw.WriteLine(ex.getMessage());
}
```

...к сожалению, в реальной жизни «временные» решения зачастую становятся постоянными.

Проблема никуда не исчезла, но по крайней мере стало ясно, где она возникла. Однако не лучше ли было разобратсья, почему ваш метод Divide вызывается с нулевым знаменателем, и устранить эту возможность?

Я понял! Мы используем обработку исключений, чтобы пометить проблемную область.



Обработка исключения далеко не всегда означает УСТРАНЕНИЕ исключения.

Плохо, когда в вашей программе происходит фатальный сбой. Но еще хуже, если вы не понимаете, почему программа аварийно завершается или что она делает с данными пользователя. Поэтому всегда нужно обрабатывать ошибки, которые вы можете предсказать, и записывать в журнал информацию об ошибках, с которыми вы пока не можете бороться. Хотя журнальная информация помогает в отслеживании проблем, лучше с самого начала предотвратить эти проблемы — это более надежное и дальновидное решение.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Любая команда может выдать **исключение**, если что-то пойдет не так **во время выполнения**.
- Для обработки исключений используется блок `try/catch`. Необработанные исключения приводят к аварийному завершению программы и появлению окна с информацией об ошибке.
- При возникновении исключения в блоке кода, следующем за ключевым словом `try`, управление немедленно передается первой команде **обработчика исключения**, т. е. блока кода после `catch`.
- **Объект Exception** содержит информацию о перехваченном исключении. Объявив переменную `Exception` в команде `catch`, вы получаете доступ к информации о любом исключении, выданном в блоке `try`:


```
try {
    // statements that might
    // throw exceptions
} catch (IOException ex) {
    // if an exception is thrown,
    // ex has information about it
}
```
- Существуют **различные виды исключений**, которые могут перехватываться в программе. Каждому виду соответствует объект, расширяющий `System.Exception`.
- Старайтесь избегать **универсальных обработчиков исключений**, перехватывающих `Exception`. Обработывайте исключения конкретных типов.
- **Блок finally** после обработчиков исключений выполняется всегда, независимо от того, выдается исключение или нет.
- Каждой команде `try` может соответствовать несколько блоков `catch`:


```
try { ... }
catch (NullReferenceException ex) {
    // Эти команды срабатывают при выдаче
    // исключения NullReferenceException
}
catch (OverflowException ex) { ... }
catch (FileNotFoundException) { ... }
catch (ArgumentException) { ... }
```
- Ваш код может выдавать исключения командой `throw`:


```
throw new Exception("Exception message");
```
- Ваш код также может повторно выдать перехваченное исключение командой `throw`, но эта возможность работает только внутри блока `catch`. Повторная выдача исключения сохраняет состояние стека вызовов.
- Чтобы определить **пользовательское исключение**, добавьте класс, расширяющий базовый класс `System.Exception`:


```
class CustomException : Exception { }
```
- В большинстве случаев хватает исключений, встроенных в .NET, таких как `ArgumentException`. Используя разные типы исключений, вы **предоставляете пользователю больше информации** для диагностики.
- **Фильтр исключений** использует ключевое слово `when`, чтобы приказать обработчику исключений перехватывать исключение только при определенных условиях.
- Команда `using` представляет собой «**синтаксический сахар**», который заставляет компилятор C# сгенерировать эквивалент блока `finally` с вызовом метода `Dispose`.

Лабораторный курс Unity № 6

Перемещение по сцене

В последней лабораторной работе Unity была создана сцена с полом (плоскость) и игроком (сфера с цилиндром). При этом использовался объект NavMesh, NavMesh Agent и отслеживание лучей, чтобы игрок следовал за щелчками в сцене.

Продолжим работу с той точки, в которой прервалась последняя лабораторная работа Unity. Цель этих лабораторных работ — помочь вам в изучении **системы поиска путей и навигации**: сложной системы на базе искусственного интеллекта, которая позволяет создавать персонажей, способных находить путь в построенных вами игровых мирах. В этой работе мы воспользуемся навигационной системой Unity, чтобы объекты GameObject сами перемещались по сцене.

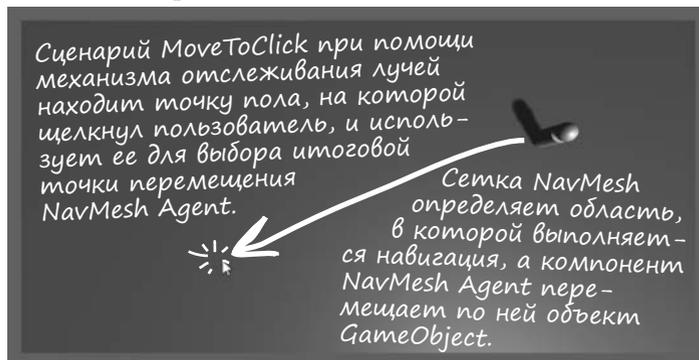
Попутно вы освоите ряд полезных инструментов: мы создадим более сложную сцену и заранее сгенерируем сетку NavMesh, которая позволяет объекту перемещаться по сцене, построим статические и перемещающиеся препятствия и, что самое важное, — **получим опыт написания кода** C#.

Продолжим с того места, на котором прервалась последняя лабораторная работа Unity

В последней лабораторной работе Unity мы создали игрока со сферической головой и цилиндрическим телом. Затем был добавлен компонент NavMesh Agent для перемещения игрока по сцене; для определения точки пола, на которой был сделан щелчок, используется отслеживание лучей. Мы добавим в сцену объекты GameObject, включая лестницы и препятствия, чтобы вы видели, как искусственный интеллект навигации Unity справляется с ними. Затем мы добавим движущееся препятствие, чтобы компоненту NavMesh Agent пришлось действительно постараться.

Откройте проект Unity, сохраненный в конце последней лабораторной работы Unity. Если вы сохраняли лабораторные работы Unity, выполняя их одну за другой, скорее всего, вы сможете с ходу взяться за работу! А если нет, потратьте несколько минут и снова просмотрите последнюю лабораторную работу Unity — а также код, написанный для нее.

Если вы читаете нашу книгу, потому что собираетесь стать профессиональным разработчиком, умение читать и проводить рефакторинг кода старых проектов является очень важным навыком — и не только для разработки игр!



Часть Задаваемые Вопросы

В: В последней лабораторной работе Unity было много новых концепций. Нельзя ли снова пробежаться по ним, чтобы я был уверен, что ничего не забыл?

О: Конечно. Сцена Unity, созданная в последней лабораторной работе, состоит из четырех частей. Разобраться в том, как они сочетаются друг с другом, не так просто, поэтому перечислим их поочередно:

1. Сетка **NavMesh**, определяющая все «проходимые» места, по которым персонаж может перемещаться в сцене. Чтобы создать ее, мы назначили пол проходимой поверхностью, а затем обработали сетку.
2. **NavMesh Agent** — компонент, который может «взять под контроль» объект GameObject и перемещать его по сетке NavMesh простым вызовом метода `SetDestination`. Мы добавили его к объекту GameObject Player.
3. Метод **ScreenPointToRay** камеры создает луч, который проходит через точку на экране. Мы добавили в метод `Update` код, который проверяет, нажата ли кнопка мыши в настоящий момент. Если кнопка нажата, то текущая позиция мыши используется для вычисления луча.
4. **Отслеживание лучей** — механизм, прокладывающий луч в сцене. Unity предоставляет метод `Physics.Raycast`, который получает луч, прокладывает его на определенное расстояние, и если луч столкнулся с каким-то объектом, сообщает вам об этом.

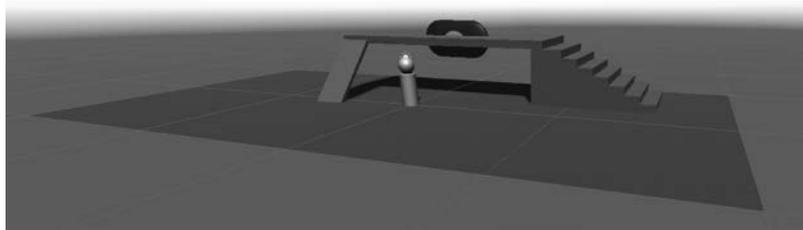
В: Как же эти части работают друг с другом?

О: Каждый раз, когда вы пытаетесь определить, как разные части системы взаимодействуют друг с другом, начинать следует с **понимания общей цели**. В данном случае цель заключается в том, чтобы игрок мог щелкнуть в любой точке пола, а объект GameObject автоматически переместился в выбранную точку. Разобьем ее на отдельные шаги. Код должен:

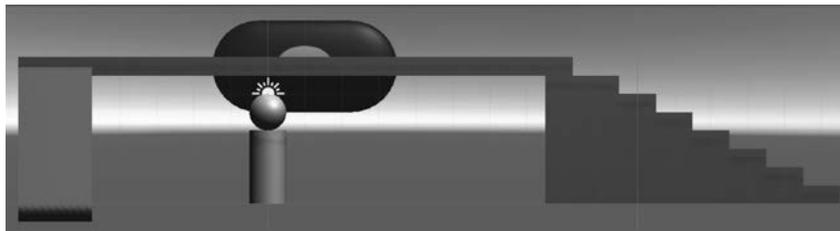
- **Обнаружить, что пользователь щелкнул кнопкой мыши.** Ваш код использует `Input.GetMouseButtonDown` для обнаружения щелчка кнопки мыши.
- **Определить, какой точке сцены соответствует точка щелчка.** Для прокладки луча и определения точки сцены, в которой был сделан щелчок, используются методы `Camera.ScreenPointToRay` и `Physics.Raycast`.
- **Приказать NavMesh Agent установить эту точку в качестве конечной точки для перемещения.** Метод `NavMeshAgent.SetDestination` заставляет агента вычислить новый путь и начать движение к новой точке. Метод `MoveToClick` представляет собой **адаптированную версию кода из страницы руководства Unity** для метода `NavMeshAgent.SetDestination`. Выделите немного времени и прочитайте ее — **выберите команду Help>>Scripting Reference** из главного меню, а затем проведите поиск по имени `NavMeshAgent.SetDestination`.

Добавление платформы в сцену

Немного поэкспериментируем с системой навигации Unity. Для этого мы добавим новые объекты GameObject и построим из них платформу с лестницей, наклонной плоскостью и препятствием. Вот как это выглядит:



Чтобы вам было немного проще разобраться в происходящем, мы переключимся в **изометрический** режим, в котором перспектива отсутствует. В **режиме перспективы** объекты, находящиеся дальше от зрителя, кажутся маленькими, тогда как близкие объекты — большими. В изометрическом режиме объекты всегда имеют одинаковый размер независимо от их расстояния до камеры.



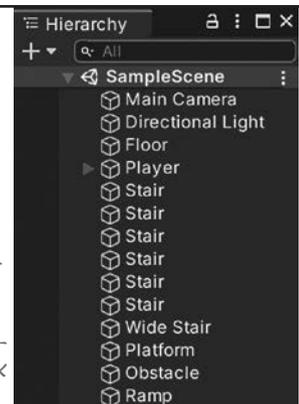
Добавьте в сцену 10 объектов GameObject. Создайте новый материал с именем **Platform** в папке Materials с цветом альбеда CС472F и добавьте его ко **всем** объектам GameObject, за исключением объекта Obstacle, который будет использовать **новый материал с именем 8 Ball** с картой 8 Ball Texture из первой лабораторной работы Unity. В следующей таблице перечислены имена, типы и позиции объектов:

Name	Type	Position	Rotation	Scale
Stair	Cube	(15, 0.25, 5)	(0, 0, 0)	(1, 0.5, 5)
Stair	Cube	(14, 0.5, 5)	(0, 0, 0)	(1, 1, 5)
Stair	Cube	(13, 0.75, 5)	(0, 0, 0)	(1, 1.5, 5)
Stair	Cube	(12, 1, 5)	(0, 0, 0)	(1, 2, 5)
Stair	Cube	(11, 1.25, 5)	(0, 0, 0)	(1, 2.5, 5)
Stair	Cube	(10, 1.5, 5)	(0, 0, 0)	(1, 3, 5)
Wide stair	Cube	(8.5, 1.75, 5)	(0, 0, 0)	(2, 3.5, 5)
Platform	Cube	(0.75, 3.75, 5)	(0, 0, 0)	(15, 0.5, 5)
Obstacle	Capsule	(1, 3.75, 5)	(0, 0, 90)	(2.5, 2.5, 0.75)
Ramp	Cube	(-5.75, 1.75, 0.75)	(-46, 0, 0)	(2, 0.25, 6)

Попробуйте создать первую ступеньку, а затем продублировать ее 5 раз и изменить ее настройки. Объект Capsule напоминает цилиндр с полусферами на обоих торцах.

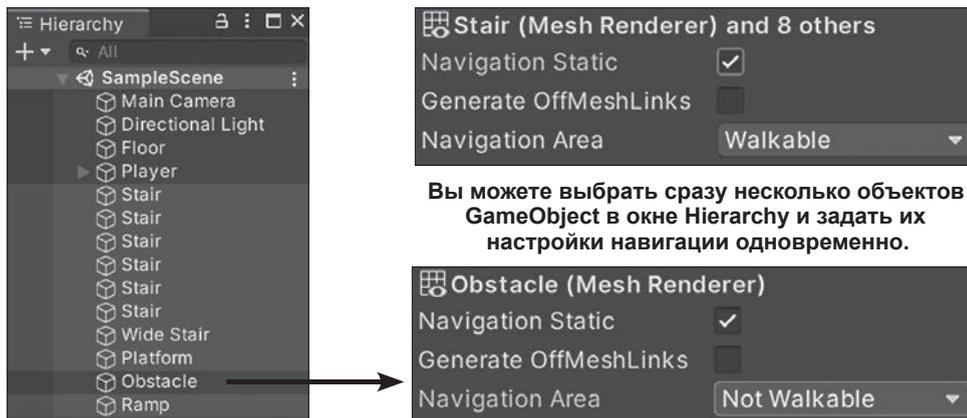
Иногда проще увидеть, что происходит в сцене, при переключении в изометрический режим. Если вы перестанете ориентироваться в представлении, макет всегда можно сбросить к исходному состоянию.

Когда вы запустите Unity, в метке (**< Persp**) под манипулятором Scene выводится имя представления. Три линии (**<**) показывают, что манипулятор находится в режиме **перспективы**. Щелкните на конусах, чтобы переключиться в режим **задней проекции** (**< Back**). Если щелкнуть на линиях, они заменяются тремя параллельными линиями (**≡**), это означает, что сцена переключилась в **изометрический режим**.

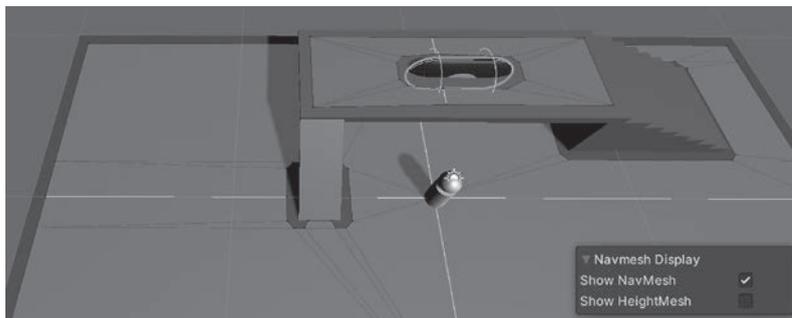


Изменение настроек предварительного построения

Используйте комбинацию Shift+щелчок на всех новых объектах GameObject, добавленных в сцену, затем снимите выделение с объекта Obstacle, щелкнув на нем с нажатой клавишей Control (или клавишей Command на Mac). Перейдите в окно Navigation и щелкните на кнопке Object, а затем **назначьте их проходимыми** — установите флажок Navigation Static и выберите в списке Navigation Area пункт Walkable.



Теперь выполните те же действия, которые использовались до **предварительного построения сетки NavMesh**: щелкните на кнопке Bake в верхней части окна Navigation, чтобы переключиться в режим предварительного построения, а затем щелкните на кнопке Bake в нижней части.

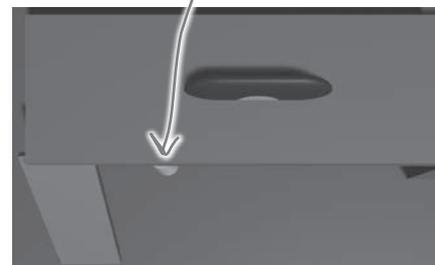


Сработало! Теперь сетка NavMesh отображается поверх платформы, а вокруг препятствия имеется свободное пространство. Попробуйте запустить игру. Щелкните на верхней части платформы и посмотрите, что произойдет.

Гмм, что-то не то. Все работает не так, как ожидалось. Когда вы щелкаете на верхней поверхности платформы, игрок перемещается *под* нее. Если внимательно присмотреться к сетке NavMesh, отображаемой при просмотре окна Navigation, мы видим, что лестница и наклонная плоскость не включены в NavMesh. Игрок не может добраться до точки, на которой был сделан щелчок, поэтому искусственный интеллект подводит его как можно ближе.



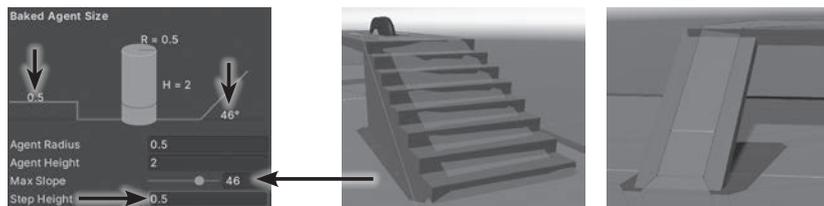
При щелчке на верхней поверхности платформы игрок заходит под нее.



Включение лестницы и наклонной плоскости в NavMesh

Искусственный интеллект не может заставить вашего персонажа подняться или спуститься по наклонной плоскости или лестнице. К счастью, система поиска путей Unity позволяет справиться с обоими случаями. Для этого достаточно внести несколько небольших изменений в настройки предварительного построения NavMesh. Начнем с лестницы. Вернитесь к окну Bake и обратите внимание на то, что значение по умолчанию для Step Height равно 0.4. Внимательно присмотритесь к параметрам ступеней вашей лестницы — их высота составляет 0.5 единицы. Чтобы приказать системе навигации включить ступени высотой 0.5 единицы, **измените значение в поле Step Height на 0.5**. Вы увидите, что изображение ступеней на диаграмме становится выше, а число над ним увеличивается с используемого по умолчанию 0.4 до 0.5.

Также необходимо включить в NavMesh наклонную плоскость. При создании объектов GameObject для платформы мы присвоили наклонной плоскости угол поворота X, равный -46 ; это означает наклон под углом 46 градусов. Параметр Max Slope по умолчанию равен 45, это означает, что в построение будут включаться склоны, наклонные плоскости или другие возвышения с углом, не превышающим 45 градусов. **Измените Max Slope на 46 и снова проведите построение NavMesh**. На этот раз в сетку будут включены наклонная плоскость и ступени.



Запустите игру и протестируйте изменения в NavMesh.



Упражнение

Еще одно упражнение по программированию для Unity! Ранее мы направили камеру сверху вниз, назначив ей угол поворота по оси X, равный 90. Чтобы игрок мог лучше рассмотреть своего персонажа, было бы неплохо реализовать управление камерой при помощи клавиш со стрелками и колеса мыши. Вы уже знаете почти все необходимое — нужно лишь добавить немного кода. *Задача кажется сложной, но она вам по силам!*

- **Создайте новый сценарий с именем MoveCamera и перетащите его на камеру.** Он должен содержать поле Transform с именем Player. Перетащите объект GameObject Player из окна Hierarchy на поле Player в окне Inspector. Так как поле имеет тип Transform, операция копирует ссылку на компонент Transform объекта GameObject Player.
- **Реализуйте вращение камеры вокруг игрока с использованием клавиш со стрелками.** Input.GetKey(KeyCode.LeftArrow) вернет true, если игрок в настоящее время нажимает клавишу \leftarrow ; для проверки других направлений используются значения RightArrow, UpArrow и DownArrow. Используйте этот метод так же, как вы использовали Input.GetMouseButtonDown в сценарии MoveToClick для проверки щелчков мышью. Когда игрок нажимает клавишу, вызовите transform.RotateAround для выполнения поворота вокруг позиции игрока. Позиция игрока передается в первом аргументе; используйте Vector3.left, Vector3.right, Vector3.up или Vector3.down как значение второго аргумента, а поле Angle (со значением 3F) — как значение третьего аргумента.
- **Реализуйте изменение масштаба камеры с использованием колеса мыши.** Input.GetAxis("Mouse ScrollWheel") возвращает число (обычно в диапазоне от -0.4 до 0.4), представляющее смещение колеса (или 0, если оно не двигалось). Добавьте поле float с именем ZoomSpeed, равное 0.25F. Проверьте, сместилось ли колесо мыши. Если оно сместилось, выполните векторные вычисления для масштабирования камеры, умножив transform.position на $(1F + scrollWheelValue * ZoomSpeed)$.
- **Направьте камеру на игрока.** Метод transform.LookAt направляет объект GameObject в заданную позицию. Затем сбросьте компонент Transform объекта Main Camera в позицию $(0, 1, -10)$ с поворотом $(0, 0, 0)$.

Не забывайте: посмотреть в решение — не значит жульничать!

Ничего страшного, если ваш код отличается от нашего. У любой задачи по программированию существует множество решений! Главное — не жалейте времени и разберитесь в том, как работает этот код.



Упражнение Решение

Еще одно упражнение по программированию для Unity! Ранее мы направили камеру сверху вниз, назначив ей угол поворота по оси X, равный 90. Чтобы игрок мог лучше рассмотреть своего персонажа, было бы неплохо реализовать управление камерой при помощи клавиш со стрелками и колеса мыши. Вы уже знаете почти все необходимое — нужно лишь добавить немного кода. *Задача кажется сложной, но она вам по силам!*

```
public class MoveCamera : MonoBehaviour
```

```
{  
    public Transform Player;  
    public float Angle = 3F;  
    public float ZoomSpeed = 0.25F;
```

```
    void Update()
```

```
{  
    var scrollWheelValue = Input.GetAxis("Mouse ScrollWheel");  
    if (scrollWheelValue != 0)  
    {  
        transform.position *= (1F + scrollWheelValue * ZoomSpeed);
```

```
    }  
    if (Input.GetKey(KeyCode.RightArrow))
```

```
{  
        transform.RotateAround(Player.position, Vector3.up, Angle);
```

```
    }  
    if (Input.GetKey(KeyCode.LeftArrow))
```

```
{  
        transform.RotateAround(Player.position, Vector3.down, Angle);
```

```
    }  
    if (Input.GetKey(KeyCode.UpArrow))
```

```
{  
        transform.RotateAround(Player.position, Vector3.right, Angle);
```

```
    }  
    if (Input.GetKey(KeyCode.DownArrow))
```

```
{  
        transform.RotateAround(Player.position, Vector3.left, Angle);
```

```
    }  
    transform.LookAt(Player.position);
```

```
}
```

Не забудьте перетащить Player на поле компонента сценария объекта Main Camera.

Попробуйте поэкспериментировать с разными углами и скоростями масштабирования. Найдите вариант, который покажется вам наиболее подходящим.

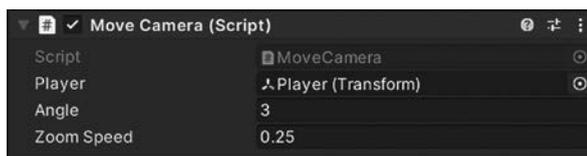
Вы не забыли сбросить позицию и углы поворота главной камеры? Если забыли, начало движения игрока может быть немного «дерганым» (из-за способа вычисления угла в методе Camera.LookAt).

Перед вами пример того, как простые арифметические операции с векторами упрощают задачу. Позиция объекта GameObject определяется вектором, поэтому умножение его на 1.02 несколько отдаляет его от нулевой точки, а умножение на 0.98 — приближает.

Происходит то же, что при использовании transform.RotateAround в двух последних лабораторных работах, только вместо Vector3.zero (0, 0, 0) поворот выполняется вокруг игрока.

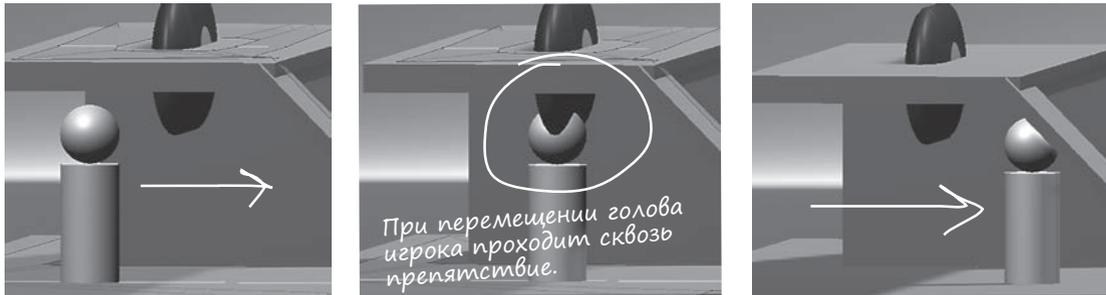
Мы предложили вам создать поле Player с типом Transform. Здесь вы получаете ссылку на компонент Transform объекта GameObject Player, так что позиция Player соответствует позиции игрока.

Используйте клавиши со стрелками для перемещения камеры, чтобы она была направлена на игрока. Вы можете смотреть прямо сквозь плоскость пола!

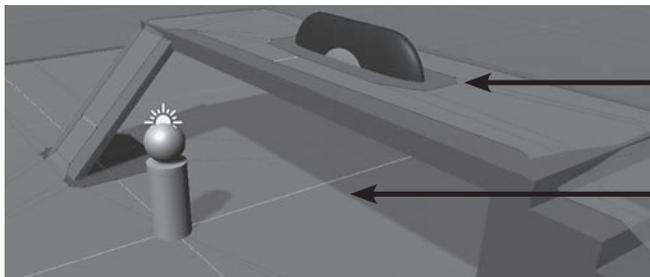


Решение проблем с высотой в NavMesh

Итак, теперь вы можете управлять камерой, и мы можем лучше рассмотреть, что происходит под платформой, — и здесь определенно что-то не так. Запустите игру, поверните камеру и увеличьте изображение так, чтобы хорошо видеть выступ препятствия под платформой. Шелкните на полу с одной стороны препятствия, затем с другой. Все выглядит так, словно игрок проходит прямо сквозь препятствие! И еще он проходит через конец наклонной плоскости.

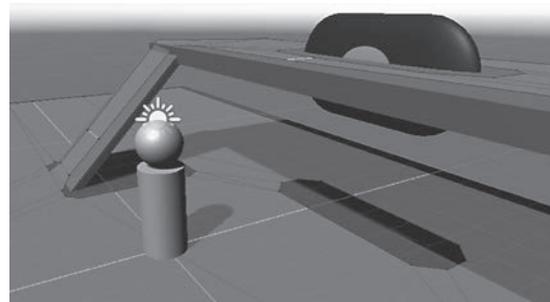
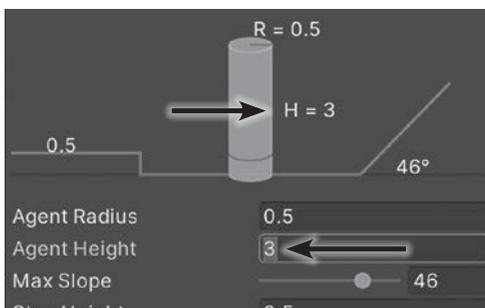


Но если завести игрока на верх платформы, он прекрасно обходит препятствие. Что происходит? Внимательно присмотритесь ко всем частям NavMesh выше и ниже препятствия. Заметили какие-нибудь отличия?



Над платформой вокруг препятствия в NavMesh существует промежуток, а внизу никаких промежутков нет, поэтому игрок просто проходит сквозь препятствие.

Вернитесь к той части последней лабораторной работы, в которой вы настраивали компонент NavMesh Agent, а конкретно к той, в которой Height присваивалось значение 3. Теперь то же самое необходимо проделать для NavMesh. Вернитесь к настройкам Bake в окне Navigation, **задайте в поле Agent Height значение 3, а затем снова проведите предварительное построение сетки.**



Тем самым вы создаете промежуток в сетке NavMesh под препятствием и расширяете этот промежуток. Теперь игрок не столкнется ни с препятствием, ни с наклонной плоскостью при перемещении под платформой.

Добавление препятствия в сетку NavMesh

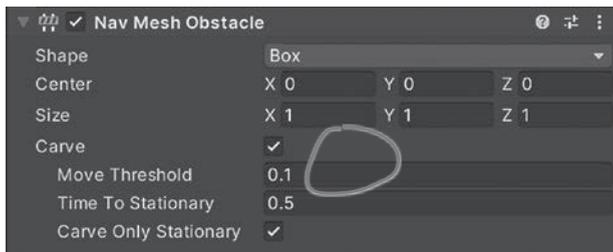
Мы уже добавили статическое препятствие в середине платформы: для этого была создана растянутая капсула (Capsule), помеченная как непроходимая, после чего мы предварительно подготовили объект NavMesh с отверстием, окружающим препятствие, чтобы игроку приходилось обходить его. Что, если вам понадобится подвижное препятствие? Попробуйте переместить его — сетка NavMesh не изменится! Отверстие в ней все еще находится там, *где препятствие находилось*, а не там, где оно находится в настоящий момент. Если вы снова предварительно построите NavMesh, то отверстие просто появится в новой позиции препятствия. Чтобы добавить подвижное препятствие, необходимо добавить компонент NavMesh Obstacle к объекту GameObject.

Давайте сделаем это. **Добавьте в сцену куб** (Cube) с позицией (-5.75, 1, -1) и масштабом (2, 2, 0.25). Создайте для него новый материал с темно-серым цветом (333333), присвойте новому объекту GameObject имя *Moving Obstacle*. Он будет действовать как своего рода «ворота» в нижней части наклонной плоскости: в верхнем положении препятствие освобождает путь, а в нижнем — блокирует его.



Это препятствие NavMesh создает движущийся проем в NavMesh, который мешает игроку перемещаться вверх по наклонной плоскости. Мы добавим сценарий, который позволяет игроку перетаскивать его мышью, чтобы блокировать и освобождать наклонную плоскость.

Осталось сделать еще одно. Щелкните на кнопке Add Component в нижней части окна Inspector и выберите команду Navigation >> Nav Mesh Obstacle, чтобы **добавить компонент NavMesh Obstacle** к объекту GameObject Cube.



Свойства Shape, Center и Size позволяют создать препятствие, которое только частично блокирует NavMesh Agent. Если у вас имеется объект GameObject нестандартной формы, вы можете добавить несколько компонентов NavMesh Obstacle, чтобы создать несколько разных отверстий в сетке NavMesh.

Если оставить всем настройкам значения по умолчанию, вы получите препятствие, через которое объект NavMesh Agent пройти не сможет — он столкнется с препятствием и остановится. **Проверьте поле Carve** — оно заставляет препятствие *создать в сетке NavMesh подвижное отверстие*, которое следует за объектом GameObject. Теперь объект GameObject подвижного препятствия может помешать игроку подняться или спуститься по наклонной плоскости. Так как высота NavMesh выбрана равной 3, если препятствие имеет высоту менее 3 единиц, оно создаст отверстие в находящейся под ним сетке NavMesh. Если высота препятствия превышает порог, отверстие исчезает.

В руководстве Unity приведено подробное — и доступное! — объяснение различных компонентов. Щелкните на кнопке Open Reference (?) в верхней части панели NavMesh Obstacle в окне Inspector, чтобы открыть руководство. Не пожалейте времени и ознакомьтесь с ним — в нем отлично описаны различные варианты.

Добавление сценария для перемещения препятствия вверх и вниз

В сценарии используется метод **OnMouseDown**. Он работает так же, как и метод **OnMouseDown** из последней лабораторной работы, если не считать того, что он вызывается при перетаскивании объекта **GameObject**.

```
public class MoveObstacle : MonoBehaviour
{
    void OnMouseDown()
    {
        transform.position += new Vector3(0, Input.GetAxis("Mouse Y"), 0);
        if (transform.position.y < 1) {
            transform.position = new Vector3(transform.position.x, 1, transform.position.z);
        }
        if (transform.position.y > 5) {
            transform.position = new Vector3(transform.position.x, 5, transform.position.z);
        }
    }
}
```

Ранее для работы с колесом мыши использовался метод **Input.GetAxis**. Теперь перемещение мыши вверх-вниз (по оси Y) используется для перемещения препятствия посредством изменения его позиции Y.

Первая команда **if** не позволяет прямоугольнику уходить ниже уровня пола, а вторая не позволяет ему подниматься слишком высоко. Сможете разобраться в том, как они работают?

Перетащите сценарий на объект **GameObject Moving Obstacle** и запустите игру. Ой-ой, что-то не так. Вы можете щелкать и перетаскивать препятствие вверх-вниз, но при этом также перемещается игрок. Чтобы исправить ошибку, добавим **тег** к **GameObject**.



Теперь изменим сценарий **MoveToClick**, чтобы проверить тег:

```
hit.collider if (Physics.Raycast(ray, out hit, 100))
содержит {
ссылку на → if (hit.collider.gameObject.tag != "Obstacle")
объект, {
с которым agent.SetDestination(hit.point);
столкнулся }
луч. }
```

Снова запустите игру. Если щелкнуть на препятствии, вы сможете перетащить его вверх и вниз, и оно остановится, когда достигнет пола или поднимется слишком высоко. Щелкните в любом другом месте, и игрок пойдет туда, как и прежде. Теперь вы можете поэкспериментировать с настройками **NavMesh Obstacle** (это будет проще сделать, если вы уменьшите значение **Speed** в настройках **NavMesh Agent** объекта **Player**):

- ★ Запустите игру. Щелкните в строке **Moving Obstacle** в окне **Hierarchy** и снимите флажок **Carve**. Переместите игрока на верх наклонной плоскости, щелкните на нижней части наклонной плоскости — игрок столкнется с препятствием и остановится. Перетащите препятствие вверх, и игрок снова начнет двигаться.
- ★ Теперь установите флажок **Carve** и попробуйте сделать то же самое. При перемещении препятствия вверх и вниз игрок пересчитывает свой маршрут, выбирая долгий путь для обхода препятствия, если оно опущено, и меняя курс в реальном времени при перемещении препятствия.

Назначьте тег объекту препятствия так же, как это было сделано в последней лабораторной работе, но на этот раз выберите команду «Add tag...» в раскрывающемся списке, а затем воспользуйтесь кнопкой **+** для добавления нового тега **Obstacle**. Теперь вы можете воспользоваться раскрывающимся списком для назначения тега объекту **GameObject**.

Часто задаваемые вопросы

В: Как работает сценарий **MoveObstacle**? Он использует **+=** для обновления **transform.position** — означает ли это, что мы используем векторные вычисления?

О: Да, и это отличная возможность лучше понять векторную арифметику. **Input.GetAxis** возвращает положительное число, если мышь движется вверх, или отрицательное, если мышь движется вниз (попробуйте добавить команду **Debug.Log**, чтобы увидеть возвращаемое значение). Препятствие начинается в позиции $(-5.75, 1, -1)$. Если игрок перемещает мышь вверх и метод **GetAxis** возвращает 0.372 , операция **+=** прибавляет к позиции вектор $(0, 0.372, 0)$. Это означает, что оба значения **X** суммируются для получения нового значения **X**, после чего то же самое делается со значениями **Y** и **Z**. Таким образом, новая позиция **Y** равна $1 + 0.372 = 1.372$, а поскольку к значениям **X** и **Z** прибавляются нули, изменяется только значение **Y**, что приводит к перемещению вверх.

Проявите фантазию!

Сможете ли вы найти возможности для того, чтобы улучшить игру (а заодно попрактиковаться в программировании)? Несколько идей, которые помогут вам проявить фантазию:

- ★ Дополните сцену — добавьте в нее больше наклонных плоскостей, лестниц, платформ и препятствий. Найдите нестандартные возможности для использования материалов. Поищите в интернете новые текстурные карты. Постарайтесь, чтобы сцена выглядела интересно!
- ★ Ускорьте перемещение NavMesh Agent, когда игрок удерживает нажатой клавишу Shift. Проведите в документации Scripting Reference поиск по тексту «KeyCode», чтобы найти коды левой/правой клавиш Shift.
- ★ В предыдущей лабораторной работе использовались методы OnMouseDown, Rotate, RotateAround и Destroy. Попробуйте использовать их для создания препятствий, которые вращаются или исчезают по щелчку.
- ★ Реальной игры пока нет — игрок просто перемещается по сцене. Сможете ли вы предложить способ **преобразовать вашу программу в полосу препятствий с таймером?**

Вы уже знаете о Unity вполне достаточно, чтобы начать строить интересные игры, — это отличный способ накопить практический опыт, чтобы стать более эффективным разработчиком.

Вам предоставляется возможность поэкспериментировать. Проявление творческой фантазии — очень эффективный способ повышения вашей квалификации программиста.

КЛЮЧЕВЫЕ МОМЕНТЫ

- При предварительном построении NavMesh указывается максимальный угол наклона и высота ступени, чтобы агенты NavMesh Agent могли перемещаться по наклонным плоскостям и лестницам.
- Также вы можете задать высоту агента, чтобы в сетке создавались отверстия вокруг препятствий, слишком низких для того, чтобы агент обходил их.
- При перемещении объекта GameObject по сцене NavMesh Agent избегает препятствий (и возможно, других агентов NavMesh Agent).
- На метке под манипулятором Scene изображен значок, который показывает, что сцена находится в режиме перспективы (дальние объекты выглядят меньше, чем ближние) или в изометрическом режиме (все объекты сохраняют свои размеры независимо от дальности). Значок может использоваться для переключения между двумя режимами.
- Метод `transform.LookAt` направляет объект GameObject в заданную позицию. С его помощью можно направить камеру на объект GameObject в сцене.
- Вызов `Input.GetAxis("Mouse ScrollWheel")` возвращает число (обычно в диапазоне от -0.4 до 0.4), представляющее величину смещения колеса мыши (или 0, если колесо не двигалось).
- Вызов `Input.GetAxis("Mouse Y")` позволяет отслеживать перемещения мыши вверх/вниз. Его можно объединить с методом `OnMouseDown`, чтобы организовать перемещение объектов GameObject мышью.
- Добавьте компонент NavMesh Obstacle, чтобы сделать препятствия, которые могут создавать подвижные отверстия в NavMesh.
- Класс `Input` содержит методы для получения ввода во время выполнения метода `Update` (например, `Input.GetAxis` для перемещения мыши и `Input.GetKey` для ввода с клавиатуры).



Упражнение: поиск пар — сражение с боссом

Если вы играли в видеоигры (а мы в этом твердо уверены!), наверняка вам пришлось часто сражаться с боссами — такие сражения обычно происходят в конце уровня или его части, а противник оказывается сильнее и опаснее тех, которые встречались вам ранее. Перед вами последнее упражнение, завершающее эту книгу, — считайте, что это своего рода «сражение с боссом».

В главе 1 мы построили игру с поиском пар животных. Приложение стало неплохим началом, но в нем... чего-то не хватает. А вы сможете придумать, как преобразовать это приложение в реальную игру? Перейдите к репозиторию GitHub данной книги и загрузите PDF-файл проекта — или, если вы предпочитаете сражаться с боссами в режиме Hard, попробуйте справиться с задачей самостоятельно.

Find all of the matching animals

6.7s

Matches found: 3
Time: 8.8s

Книга подходит к концу, но обучение продолжается. Мы подготовили еще немало загружаемых материалов по важным темам C#. В частности, в продолжение вашего пути изучения Unity в них представлены дополнительные лабораторные работы Unity и даже новое «сражение с боссом» из области Unity.

Надеемся, вы многое узнали из этой книги, но что еще важнее, мы верим, что ваш путь изучения C# только начинается. Хороший разработчик никогда не перестает учиться.

За дополнительной информацией обратитесь к репозиторию GitHub данной книги:
<https://github.com/head-first-csharp/fourth-edition>.

Спасибо за то, что вы прочитали нашу книгу!

Похлопайте себя по плечу — это настоящее достижение! Мы надеемся, что это путешествие было для вас таким же приятным, как и для нас, и что вам понравились все проекты и код, который был при этом написан.

Постойте, это еще не все! Путешествие только началось...

В некоторых главах приводились ссылки на дополнительные проекты, которые можно загрузить с нашей страницы GitHub: <https://github.com/head-first-csharp/fourth-edition>

На GitHub также доступен **большой объем дополнительных материалов**. Из них вы сможете узнать много нового и найдете ряд новых проектов!

Не останавливайтесь на пути изучения C#! В загружаемых документах PDF продолжится ваша история изучения C# и рассматриваются некоторые **важные темы C#**, среди которых:

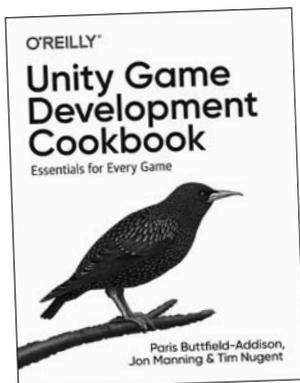
- ★ обработчики событий;
- ★ делегаты;
- ★ паттерн MVM (включая проект аркадной игры в стиле «ретро»);

...и многое другое!

И раз уж вы зашли на GitHub, обратите внимание на **дополнительные материалы о Unity**. Вы сможете загрузить:

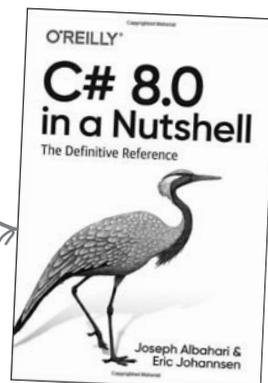
- ★ PDF-версии всех лабораторных работ Unity из книги.
- ★ **Новые лабораторные работы Unity**, в которых используется физика, коллизии и многое другое!
- ★ Завершающее испытание, которое проверит все ваши навыки разработки Unity.
- ★ **Полный проект лабораторной работы Unity**, в котором с нуля создается компьютерная игра.

А также обратите внимание на пару важных (и отлично написанных!) книг от издательства O'REILLY®, созданных нашими друзьями и коллегами.



«Сборник рецептов для разработчика Unity» поможет вам поднять ваши навыки работы с Unity на следующий уровень. Множество исключительно полезных приемов и инструментов представлено в ней в форме «рецептов», которые вы можете с ходу применить в своих проектах.

«C# 8.0 in a Nutshell» должен прочитать каждый разработчик C#. Если вы хотите действительно глубоко изучить тот или иной аспект C#, вряд ли что-нибудь поможет вам лучше, чем эта книга.



Обратите внимание на следующие замечательные ресурсы C# и .NET!

Вступайте в сообщество разработчиков .NET: <https://dotnet.microsoft.com/platform/community>.

Смотрите живые трансляции и общайтесь в чатах с группой разработчиков, занимающихся созданием .NET и C#: <https://dotnet.microsoft.com/platform/community/standup>.

Обращайтесь к документации за новыми знаниями: <https://docs.microsoft.com/en-us/dotnet>

Приложение 1. Проекты ASP.NET Core Blazor

Visual Studio для пользователей Mac

Мы очень серьезно относимся к яблокам.



Ваш Mac — полноправный гражданин мира C# и .NET. Мы писали эту книгу, ни на минуту не забывая о пользователях Mac; именно поэтому в книгу было включено это приложение. Большинство проектов в книге относится к категории консольных приложений, работающих **как в Windows, так и на Mac.** В некоторых главах приводятся проекты, построенные на базе технологий настольных приложений Windows. В этом приложении приводятся **альтернативные версии** этих проектов, **включая полную замену главы 1**, в которых C# используется для создания приложений Blazor Webassembly, выполняемых в браузере и эквивалентных Windows-приложениям. Все это делается в **Visual Studio for Mac** — превосходном инструменте для написания кода и **ценного учебного инструмента** для изучения C#. А теперь займемся программированием!

Зачем вам изучать C#

C# — простой современный язык, на котором можно решать совершенно невероятные задачи. Изучая C#, вы не ограничиваетесь только новым языком. C# открывает перед вами целый мир .NET — невероятно мощной платформы с открытым кодом для построения самых разнообразных приложений.

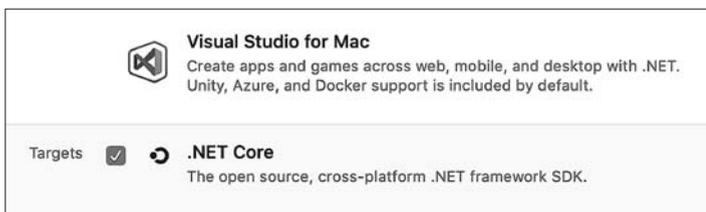
Visual Studio станет вашим окном в C#

Если вы еще не установили Visual Studio 2019, самое время это сделать.

Откройте страницу <https://visualstudio.microsoft.com> и **загрузите Visual Studio for Mac**. (Если она уже установлена на вашем компьютере, запустите программу установки Visual Studio для Mac, чтобы обновить набор установленных компонентов.)

Установка .NET Core

После того как программа установки Visual Studio for Mac будет загружена, запустите ее, чтобы установить Visual Studio. Проследите за тем, чтобы была выбрана целевая платформа .NET Core.



Убедитесь в том, что вы устанавливаете Visual Studio for Mac, а не Visual Studio Code.

Visual Studio Code — прекрасный кросс-платформенный редактор с открытым кодом, но он не так хорошо подходит для разработки .NET, как Visual Studio. Вот почему мы будем использовать Visual Studio в книге как учебный и аналитический инструмент.

Веб-приложения Blazor также можно строить в Visual Studio for Windows

В большинстве проектов этой книги создаются консольные приложения .NET Core, которые работают как в системе Windows, так и в macOS. В некоторых главах рассматриваются проекты, построенные на базе технологии WPF (Windows Presentation Foundation). Так как технология WPF работает только в Windows, мы написали это приложение, чтобы вы могли создавать эквивалентные проекты на Mac на базе веб-технологий, а именно проекты ASP.NET Core Blazor WebAssembly.

А если вы работаете в Windows, но хотите научиться строить веб-приложения с расширенной функциональностью на базе Blazor? Тогда вам повезло! **Проекты из этого приложения можно строить и в Visual Studio for Windows**. Перейдите в программу установки Visual Studio и убедитесь в том, что **флажок ASP.NET and web development установлен**. Ваши снимки экрана IDE не будут полностью совпадать с приведенными в руководстве, но весь код останется неизменным.



В проектах веб-приложений вы будете использовать HTML и CSS, но знания HTML или CSS от вас не потребуется.

Эта книга написана для тех, кто хочет изучить C#. В проектах из этого руководства вы будете создавать веб-приложения Blazor, включающие страницы, оформленные с применением HTML и CSS. Не беспокойтесь, если вы еще не работали с HTML или CSS, — работа с книгой не требует никаких предварительных знаний в области веб-дизайна. Мы предоставим все необходимое для создания всех страниц проектов веб-приложений. Впрочем, попутно вы можете кое-что узнать о HTML.

Visual Studio — инструмент для написания кода и изучения C#

Код C# можно писать и в Блокноте или в другом текстовом редакторе, но существует более удобный вариант. **IDE** (сокращение от Integrated Development Environment, т. е. «интегрированная среда разработки») включает в себя текстовый редактор, визуальный конструктор, менеджер файлов, отладчик... словно многофункциональный инструмент для всех операций, которые могут понадобиться при написании кода.

Перечислим лишь некоторые из задач, в решении которых вам поможет Visual Studio:

- 1 **БЫСТРО создавать приложения.** Язык C# гибок и прост в изучении, а Visual Studio позволяет автоматизировать многие операции, которые обычно приходится выполнять вручную. Примеры операций, которые Visual Studio делает за вас:
 - * Управление файлами проекта.
 - * Удобное редактирование кода проекта.
 - * Управление графикой, аудио, значками и другими ресурсами проекта.
 - * Отладка кода с возможностью выполнения в пошаговом режиме.



- 2 **Написание и выполнение кода C#.** Visual Studio IDE — один из самых простых и удобных инструментов для написания кода. Группа разработки из Microsoft приложила колоссальные усилия к тому, чтобы написание кода было по возможности простым и удобным.
- 3 **Построение восхитительных в визуальном отношении программ.** В этом приложении вы будете строить веб-приложения, работающие в браузере. Мы будем использовать **Blazor** — технологию, которая позволяет строить интерактивные веб-приложения на C#. **При объединении C# с HTML и CSS** в вашем распоряжении появляется впечатляющий инструментарий веб-разработки.
- 4 **Изучение и исследование C# и .NET.** Visual Studio представляет собой инструмент разработки мирового уровня, но к счастью для нас, это также великолепный учебный инструмент. Мы будем использовать IDE для исследования C#, что позволит вам *быстро* закрепить важные концепции программирования в вашем мозге.

В книге Visual Studio будет называться просто «IDE».

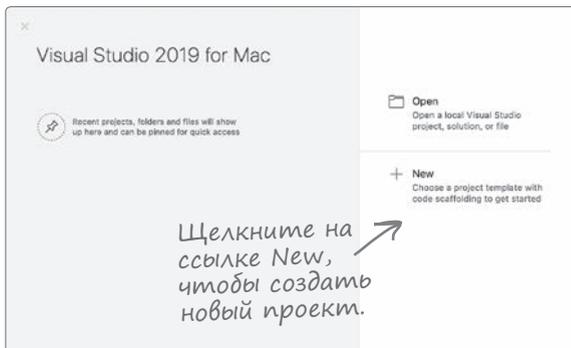
Visual Studio — замечательная среда разработки, но мы также будем использовать ее как учебный инструмент для изучения C#.

Создание вашего первого проекта в Visual Studio for Mac

Лучший способ изучения C# — непосредственное написание кода, поэтому мы воспользуемся Visual Studio для создания нового проекта... и немедленно начнем писать код!

1 Создайте новый проект Console Project.

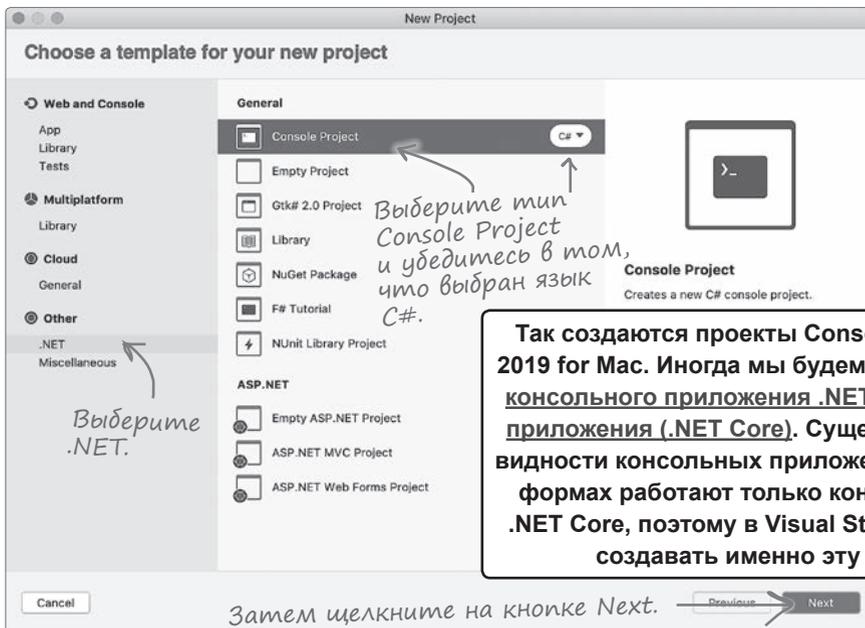
Запустите Visual Studio 2019 for Mac. При первом запуске на экране появится окно для создания нового проекта или открытия уже существующего. Щелкните на ссылке **New**, чтобы создать новый проект. Если вы закроете окно, не беспокойтесь: чтобы вызвать его обратно, достаточно выбрать в меню команду `File>>New Solution...` ($\hat{=} \&N$).



Сделайте это!

Когда вы видите в тексте врезку (Сделайте это! или Принципы отладки), откройте Visual Studio и выполните приведенные инструкции. Мы точно расскажем, что следует делать и на что нужно обратить внимание, чтобы извлечь максимум пользы из приведенного примера.

Выберите на панели слева платформу **.NET**, а затем тип проекта **Console Project**.

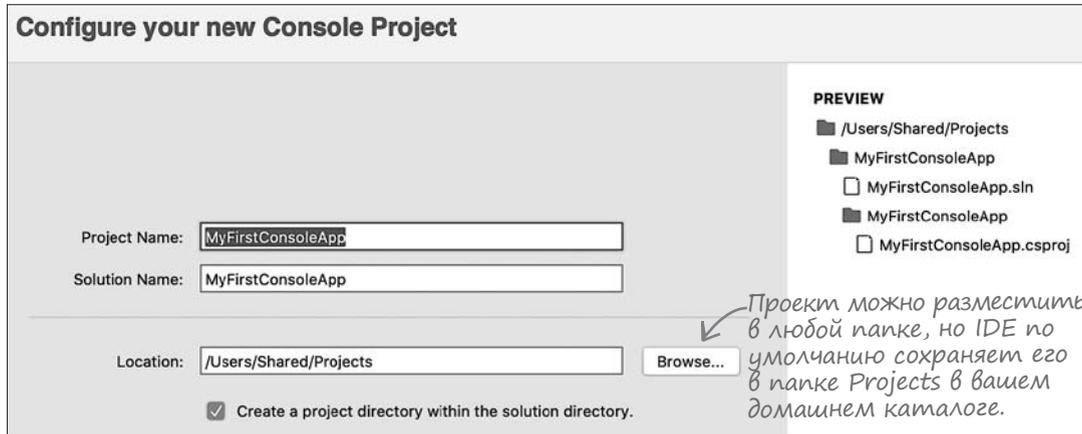


Так создаются проекты Console App в Visual Studio 2019 for Mac. Иногда мы будем называть его проектом консольного приложения .NET Core или консольного приложения (.NET Core). Существуют и другие разновидности консольных приложений, но на разных платформах работают только консольные приложения **.NET Core**, поэтому в Visual Studio for Mac мы будем создавать именно эту разновидность.

2

Присвойте новому проекту имя MyFirstConsoleApp.

Введите **MyFirstConsoleApp** в поле Project Name и нажмите кнопку Create, чтобы создать проект.



3

Просмотрите код нового приложения.

При создании нового проекта Visual Studio дает вам отправную точку для дальнейшей работы. Как только создание новых файлов для приложения будет завершено, на экране должен открыться файл с именем *Program.cs* со следующим кодом:

```

1  using System;
2
3  namespace MyFirstConsoleApp
4  {
5      class MainClass
6      {
7          public static void Main(string[] args)
8          {
9              Console.WriteLine("Hello World!");
10         }
11     }
12 }
13
    
```

При создании нового проекта консольного приложения Visual Studio автоматически добавляет класс с именем MainClass.

Класс изначально содержит метод с именем Main с одной командой для вывода строки текста на консоль. Классы и методы намного подробнее рассматриваются в главе 2.



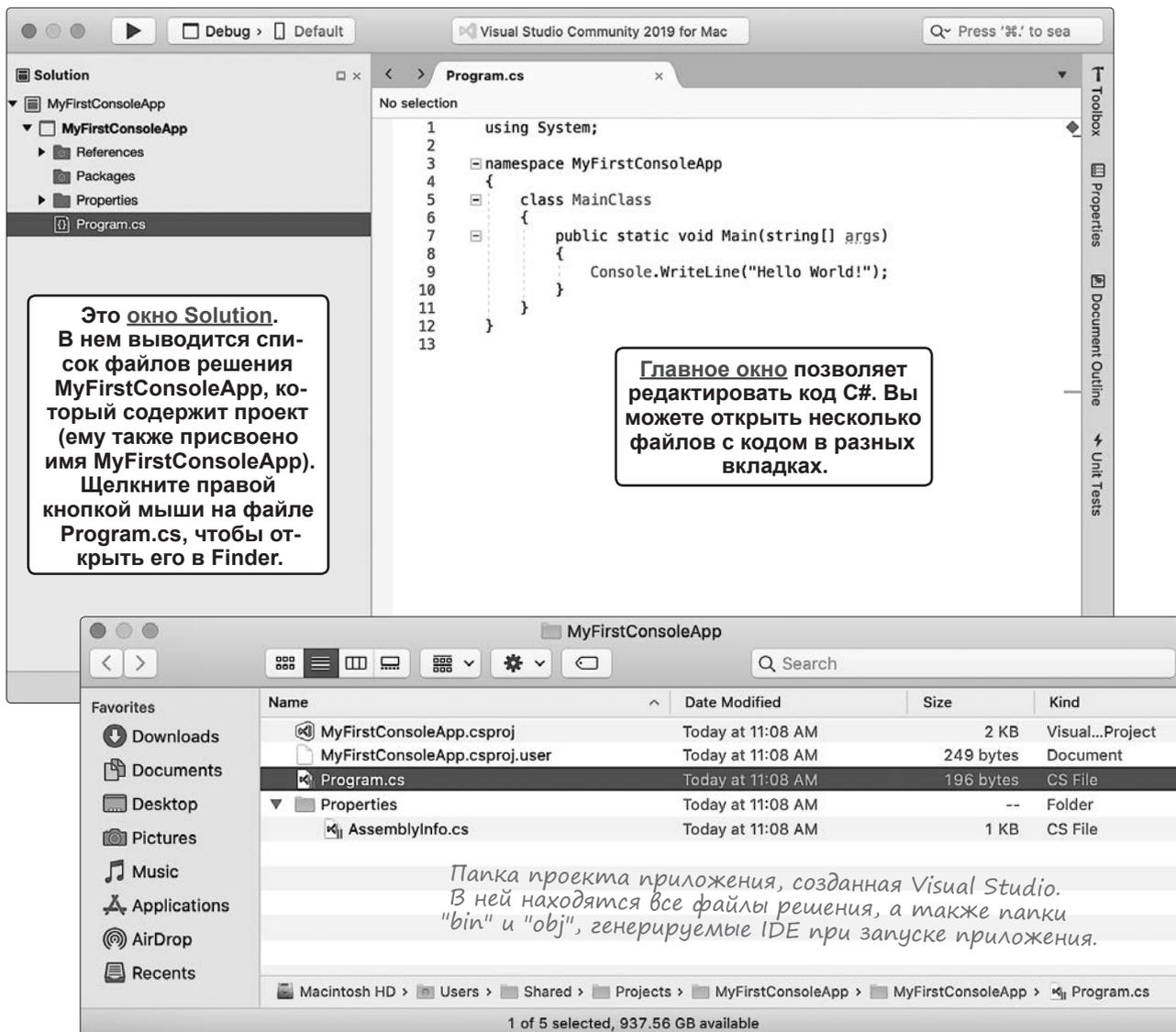
Будьте осторожны!

В Windows используется другое имя главного класса.

Когда вы создаете консольное приложение в Visual Studio for Windows, сгенерированный код почти не отличается от кода, генерируемого в Visual Studio for Mac. Одно из отличий заключается в том, что на Mac главный класс называется *MainClass*, а в Windows ему присваивается имя *Program*. В большинстве проектов книги это ни на что не влияет (исключения будут особо отмечаться в тексте).

Запуск приложений в Visual Studio IDE

- 1 **Рассмотрите среду Visual Studio IDE — и файлы, которые она создала за вас.** При создании нового проекта Visual Studio автоматически создает несколько файлов за вас и объединяет их в **решение** (solution). В окне Solution в левой части IDE показаны эти файлы, а решение (MyFirstConsoleApp) находится на верхнем уровне иерархии. Решение содержит **проект**, имя которого совпадает с именем решения.



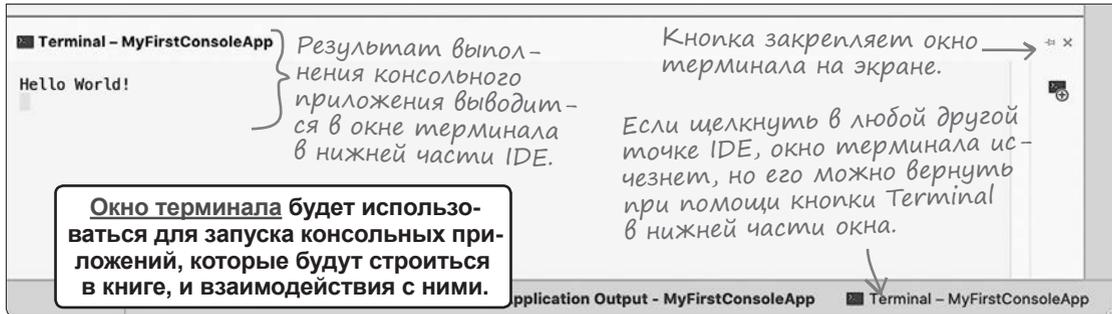
2 **Запустите новое приложение.**

Приложение, которое среда Visual Studio for Mac создала за вас, готово к запуску. Найдите в верхней части Visual Studio IDE кнопку Run (кнопка с треугольником) и **нажмите** ее:



3 **Просмотрите результаты выполнения программы.**

Когда вы запускаете вашу программу, в нижней части IDE появляется окно терминала с результатом выполнения вашей программы:



Лучший способ изучить язык — писать на нем побольше кода, поэтому в этой книге мы будем строить множество программ. Многие из них будут проектами консольных приложений, поэтому давайте повнимательнее посмотрим, что же было сделано.

В верхней части окна терминала выводятся **результаты выполнения программы**:

Hello World!

Щелкните в любой точке кода, чтобы скрыть окно терминала. Затем нажмите кнопку  в нижней части IDE, чтобы открыть его снова, — вы увидите тот же вывод вашей программы. IDE автоматически скрывает окно терминала после выхода из приложения.

Нажмите кнопку Run, чтобы снова запустить вашу программу. Выберите команду Start Debugging из меню Run или воспользуйтесь комбинацией клавиш (**⌘↵**). Так будут запускаться все проекты консольных приложений в книге.

Подсказки для IDE: Открытие терминала из IDE

В окне терминала выводятся результаты консольных приложений, но этим его функциональность не ограничивается. Щелкните на кнопке  в правой части окна терминала или выберите команду View>>Terminal из меню, когда приложение не выполняется. Вы увидите оболочку терминала macOS внутри IDE, которая может использоваться для выполнения командной оболочки macOS:



Щелкните на кнопке  еще несколько раз — IDE откроет несколько окон терминала одновременно. Вы можете переключаться между ними командой меню View>>Other Windows или кнопкой на панели в нижней части IDE:

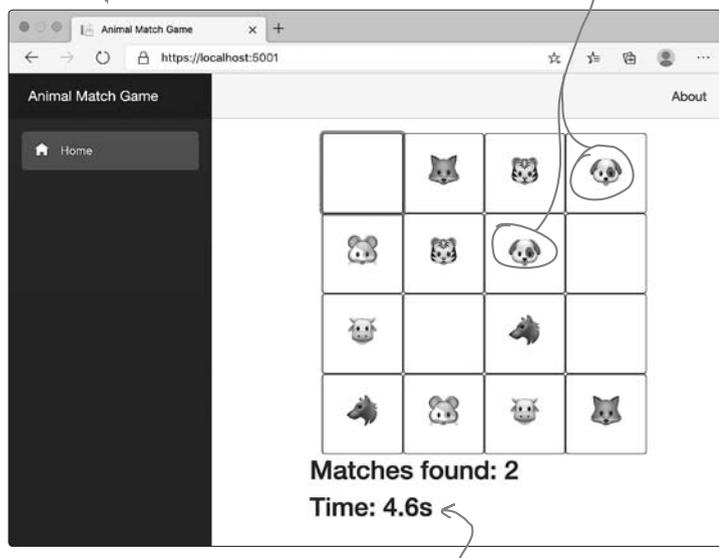


Давайте построим игру!

Вы только что построили свое первое приложение C#, и это замечательно! А теперь попробуем создать что-то посложнее. Мы построим игру, в которой игрок должен подбирать **пары животных**. На экране выводится квадратная сетка с 16 животными, а игрок щелкает на парах, чтобы они исчезали с экрана.

Игра с подбором пар животных, которую мы построим.

На экране выводятся восемь пар разных животных, случайным образом распределенных в сетке. Игрок щелкает на двух животных: если эти животные одинаковы, то пара исчезает со страницы.



Таймер следит за тем, сколько времени потребовалось игроку для завершения игры. Цель игрока — найти все пары за минимальное количество времени.

Игра является приложением Blazor WebAssembly

Консольные приложения прекрасно подходят для ввода и вывода текста. Если вы хотите построить визуальное приложение, которое отображается как страница в браузере, придется использовать другую технологию. Вот почему игра с подбором пар животных будет приложением **Blazor WebAssembly**. Blazor позволяет создавать полнофункциональные веб-приложения, которые могут работать в любом современном браузере. Во многих главах книги будет представлено одно приложение Blazor. В этом разделе мы кратко представим Blazor и опишем средства для построения как полнофункциональных веб-приложений, так и консольных приложений.

Построение проектов разных типов является важной частью изучения C#. Мы выбрали Blazor для проектов Mac в этой книге, потому что эта платформа предоставляет средства для построения полнофункциональных веб-приложений, работающих в любом современном браузере.

Но язык C# не ограничивается веб-программированием и консольными приложениями! У каждого проекта в этом руководстве существует эквивалентный проект Windows.

Вы работаете в Windows, но хотите изучить Blazor и строить веб-приложения на C#? Что же, вам повезло! Все проекты, представленные в этом руководстве, также могут быть построены в Visual Studio for Windows.

К тому времени, когда вы завершите этот проект, вы гораздо лучше освоите те инструменты, которые будут использованы в книге для изучения и исследования возможностей C#.

Как построить игру

В оставшейся части этого раздела будет рассмотрен процесс построения игры с поиском пар животных, который состоит из нескольких шагов:

1. Сначала мы создадим проект Blazor WebAssembly App в Visual Studio.
2. Затем мы создадим макет страницы и напишем код C# для случайной расстановки животных в окне.
3. Игрок щелкает на парных эмодзи, чтобы удалить их из окна.
4. Мы напишем код C#, проверяющий условие победы.
5. Наконец, чтобы игра стала более интересной, мы добавим в нее таймер.

Проект может занять от 15 минут до часа в зависимости от того, с какой скоростью вы вводите текст. Нам лучше думается, когда мы не торопимся, так что выделите побольше времени.



Обращайте внимание на врезки «Разработка игр... и не только», встречающиеся в тексте книги. Мы будем использовать принципы проектирования для изучения и исследования важных концепций и идей программирования, применимых в любых проектах, не только в видеоиграх.



Что такое игра?

Разработка игр... и не только

Вроде бы ответ на этот вопрос совершенно очевиден. Но задумайтесь на минутку — все не так просто, как кажется на первый взгляд.

- У всех ли игр есть победитель? У каждой ли игры есть конец? Не обязательно. Как насчет имитатора полета? Игры, в которой вы строите парк развлечений? Или таких игр, как The Sims?
- Игры всегда интересны? Не для всех. Некоторым игрокам нравится процесс «гринда», когда им приходится делать одно и то же раз за разом; другим это кажется ужасным.
- Всегда ли игры сопряжены с принятием решений, конфликтами или решением задач? Нет, не всегда. «Симуляторы ходьбы» — класс игр, в которых игрок просто исследует игровую среду, и часто в них вообще нет никаких головоломок или конфликтов.
- Обычно довольно трудно четко определить, что же именно следует считать игрой. В учебниках по разработке игр можно найти множество разных определений. Для наших целей определим смысл «игры» (по крайней мере для хороших игр) следующим образом:

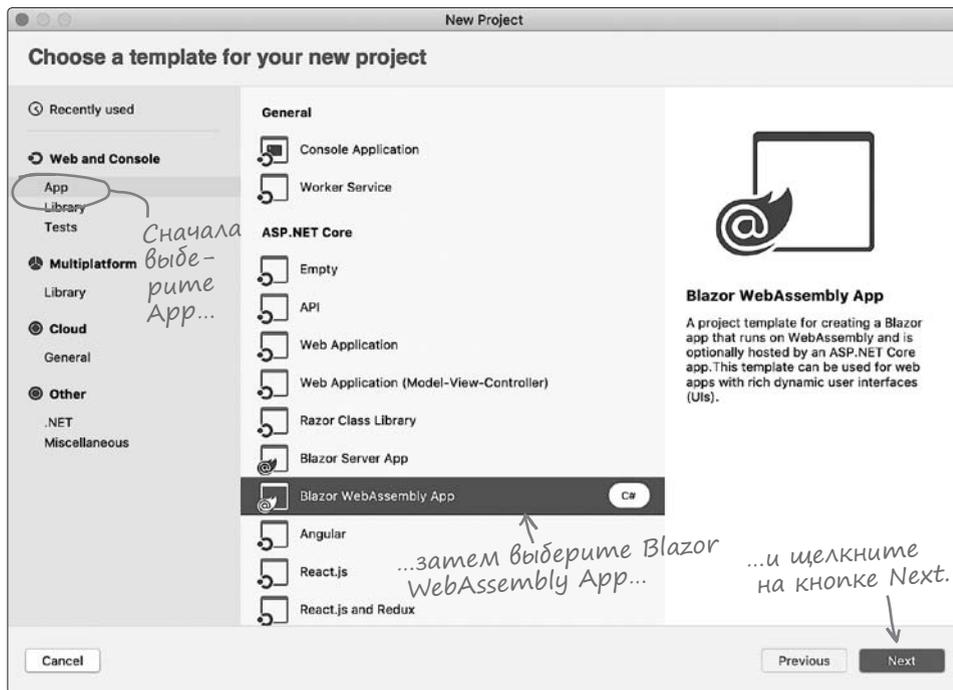
Игра — программа, взаимодействовать с которой не менее увлекательно, чем строить ее.



Создание проекта Blazor WebAssembly в Visual Studio

Построение новой игры начинается с создания нового проекта в Visual Studio.

- 1 Выберите **File >> New Solution...** ($\uparrow \#N$), чтобы вызвать на экран окно New Project. Это окно уже встречалось вам при создании проекта консольного приложения.



Выберите категорию **App** в разделе «Web and Console» слева, затем выберите вариант **Blazor WebAssembly App** и щелкните на кнопке **Next**.

- 2 IDE открывает страницу с параметрами конфигурации.



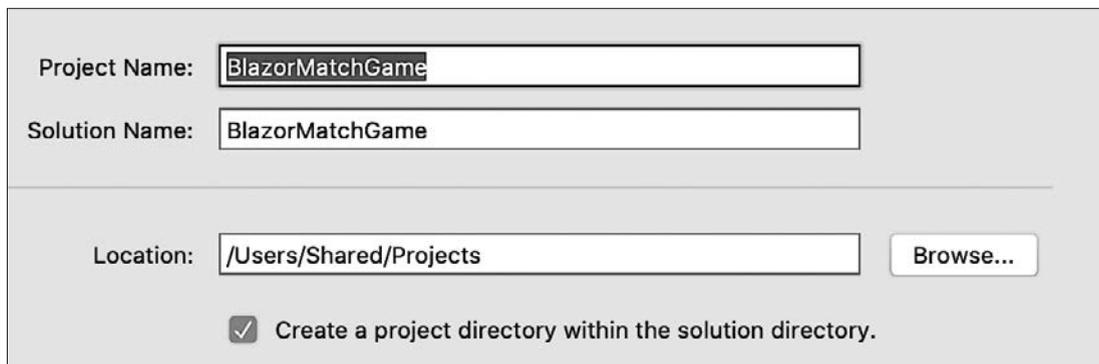
Оставьте всем параметрам значения по умолчанию и щелкните на кнопке **Next**.



Если у вас возникнут какие-либо проблемы с этим проектом, зайдите на нашу страницу [GitHub](https://github.com/head-first-csharp/fourth-edition) и найдите ссылку на видеоролик: <https://github.com/head-first-csharp/fourth-edition>

начинаем программировать на C#

3 Введите имя проекта **BlazorMatchGame**, как это делалось в проекте консольного приложения.



Project Name:

Solution Name:

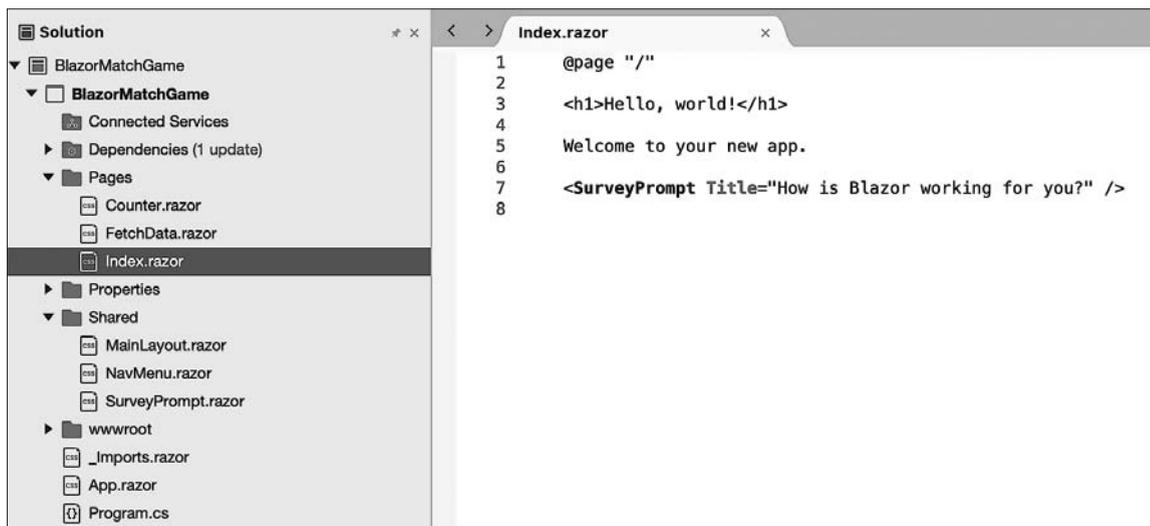
Location:

Create a project directory within the solution directory.

Затем щелкните на кнопке **Create**, чтобы создать решение.



4 IDE создает новый проект **BlazorMatchGame** и выводит его содержимое, как и в консольном приложении. **Раскройте папку Pages** в окне Solution, чтобы просмотреть ее содержимое, а затем сделайте **двойной щелчок** на файле **Index.razor**, чтобы открыть его в редакторе.



Запуск веб-приложения Blazor в браузере

Запущенное веб-приложение состоит из двух частей: **сервера** и **веб-приложения**. Visual Studio запускает их по нажатию одной кнопки.

1 Выберите браузер для запуска веб-приложения.

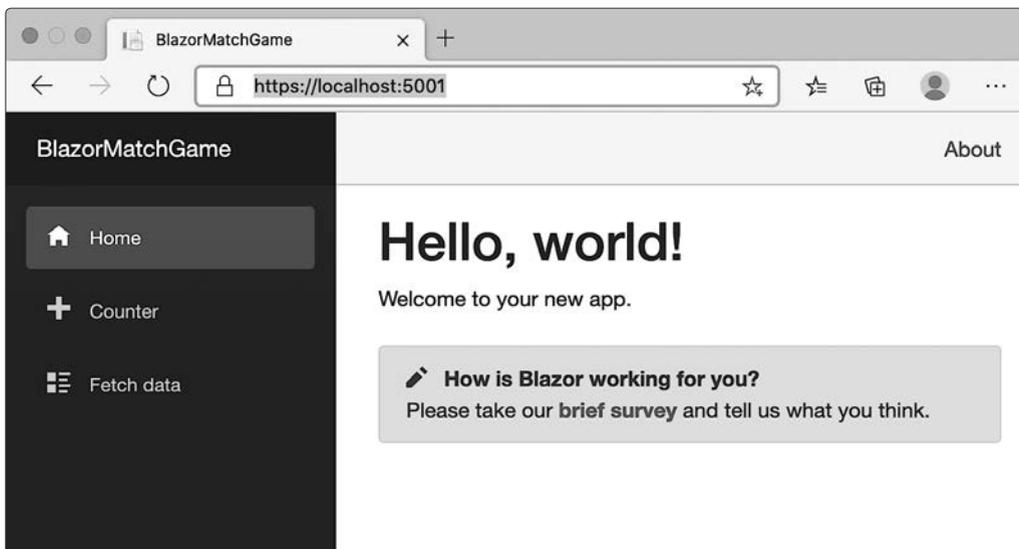
Найдите кнопку Run (кнопка с треугольником) в верхней части Visual Studio IDE.



Браузер по умолчанию отображается рядом с подсказкой Debug >. Щелчок на имени браузера открывает раскрывающийся список установленных браузеров – выберите **Microsoft Edge** или **Google Chrome**.

2 Запустите веб-приложение.

Щелкните на **кнопке Run**, чтобы запустить приложение. Также можно выбрать команду Start Debugging (⌘⇧↵) из меню Run. IDE сначала открывает окно Build Output (в нижней части, как с окном терминала), а затем окно Application Output. После этого открывается браузер, в котором выполняется ваше приложение.



Будьте осторожны!

Запускайте веб-приложения в Microsoft Edge и Google Chrome.

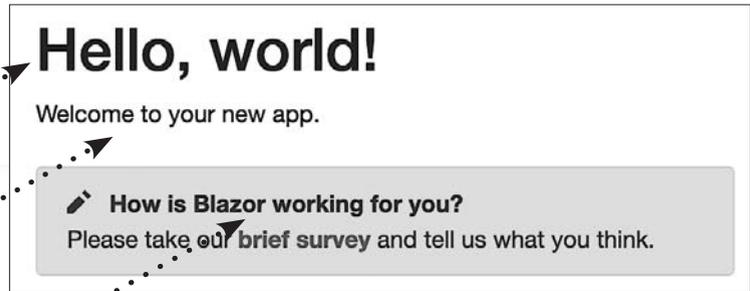
В Safari веб-приложения прекрасно работают, но вы не сможете их отлаживать. Отладка веб-приложений поддерживается только в Microsoft Edge и Google Chrome. Edge можно загрузить по адресу <https://microsoft.com/edge>, а Chrome — по адресу <https://google.com/chrome>; оба браузера бесплатные.

3 Сравните код в Index.razor с тем, что вы видите в браузере.

Веб-приложение в браузере состоит из двух частей: **навигационное меню** в левой части содержит ссылки на различные страницы (Home, Counter и Fetch Data), а справа отображается страница. Сравните разметку HTML в файле Index.razor с приложением, отображаемым в браузере.

```

1 @page "/"
2
3 <h1>Hello, world!</h1>
4
5 Welcome to your new app.
6
7 <SurveyPrompt Title="How is Blazor working for you?" />
    
```



4 Замените «Hello, world!» другим текстом.

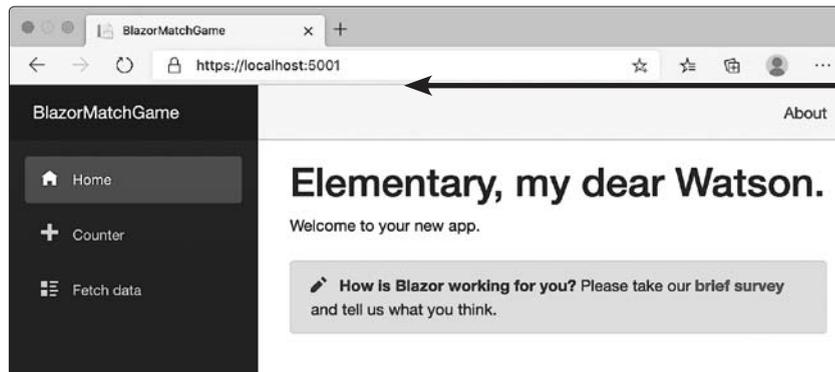
Измените третью строку файла Index.razor, чтобы она содержала другой текст, например такой:

```
<h1>Elementary, my dear Watson.</h1>
```

Теперь вернитесь в браузер и перезагрузите страницу. Подождите немного... нет, ничего не изменилось — все равно выводится текст «Hello, world!». Дело в том, что вы изменили код, *но не обновили сервер*.

Щелкните на кнопке Stop  или выберите команду Stop (⇧⌘↵) из меню Run. Теперь вернитесь и перезагрузите страницу в браузере; так как приложение было остановлено, выводится страница с сообщением о недоступности сайта.

Снова запустите приложение и перезагрузите страницу в браузере. На этот раз вы увидите обновленный текст.

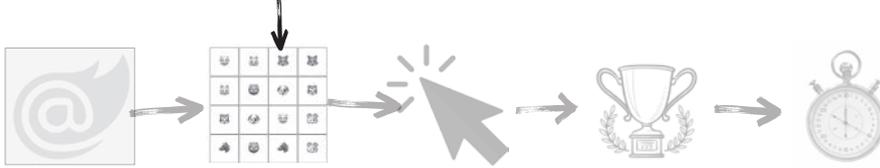


Попробуйте скопировать URL-адрес из браузера, откройте новое окно Safari и вставьте его. Ваше приложение будет работать и здесь. Два разных браузера подключаются к одному серверу.

Нет ли у вас лишних экземпляров открытых браузеров? Visual Studio открывает новый браузер каждый раз, когда вы запускаете веб-приложение Blazor. Возьмите в привычку закрывать браузер (⌘Q), прежде чем останавливать приложение (⇧⌘↵).

начинаем построение игры

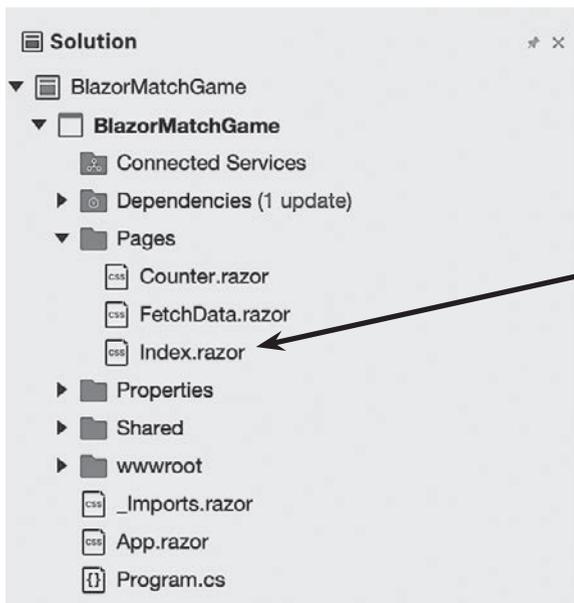
ВЫ НАХОДИТЕСЬ ЗДЕСЬ



СОЗДАНИЕ ПРОЕКТА СЛУЧАЙНАЯ РАССТАНОВКА ЖИВОТНЫХ ОБРАБОТКА ЩЕЛЧКОВ ПРОВЕРКА ПОБЕДЫ ИГРОКА ДОБАВЛЕНИЕ ТАЙМЕРА

Теперь можно переходить к написанию кода игры

Новое приложение создано, а среда Visual Studio сгенерировала необходимые файлы. Теперь добавим код C#, который обеспечит работу игры (а также разметку HTML, определяющую ее оформление).



Мы переходим к работе с кодом C#, который находится в файле Index.razor. Файл с расширением .razor содержит страницу разметки Razor. Razor объединяет разметку страницы HTML с кодом C#, все это в одном файле. Мы добавим в этот файл код C#, который определяет поведение игры, включая код добавления эмодзи на страницу, обработки щелчков мышью и управления таймером.

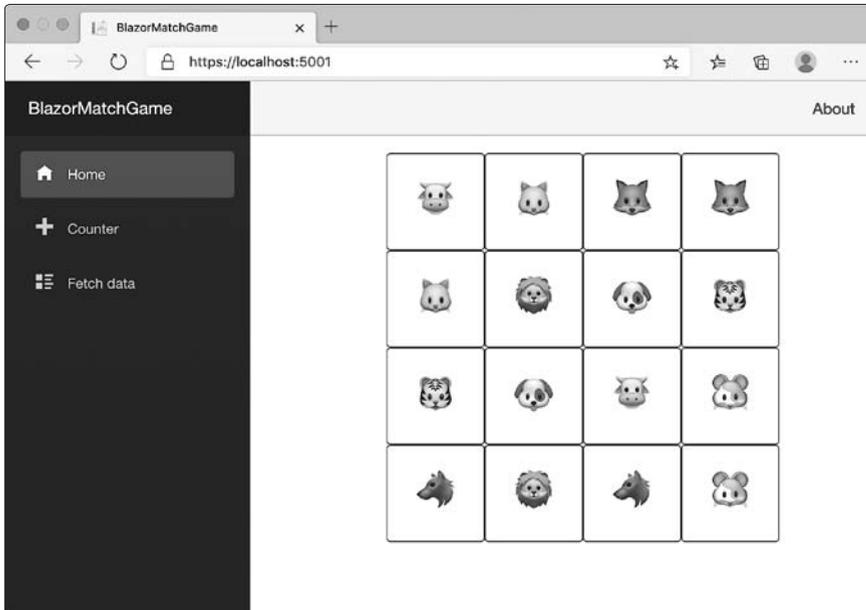
↑
Когда вы создавали свое консольное приложение ранее в этой главе, код C# содержался в файле с именем Program.cs — когда вы видите расширение .cs, оно указывает на то, что файл содержит код C#.



Будьте осторожны!

Когда вы вводите код C#, он должен быть абсолютно правильным.

Некоторые считают, что вы становитесь настоящим разработчиком только после того, как впервые проведете несколько часов в поисках неправильно поставленной точки. Регистр символов важен: SetUpGame и setUpGame — разные имена. Лишние запятые, круглые скобки, точки с запятой и т. д. могут нарушить работоспособность вашего кода или, что еще хуже, изменить ваш код, чтобы он успешно строился, но работал совершенно не так, как предполагалось. Функция IDE IntelliSense помогает избежать подобных проблем... но и она не сможет сделать все за вас.



Как работает разметка страницы в игре

В игре с поиском пар используется сетка – по крайней мере на первый взгляд. На самом деле макет состоит из 16 квадратных кнопок. Если вы измените размеры окна браузера и сделаете его очень узким, кнопки выстроятся в один длинный столбец.

Чтобы определить макет страницы, мы сначала создаем контейнер шириной 400 пикселей («пиксел» CSS составляет 1/96 дюйма, когда браузер использует масштаб по умолчанию), который содержит кнопки шириной 100 пикселей. Мы приведем весь код C# и HTML, который необходимо ввести в IDE. **Внимательно следите за кодом**, который *вскоре* будет добавлен в проект, здесь происходит все «волшебство» с объединением кода C# с HTML:

```
<div class="container">
  <div class="row">
    @foreach (var animal in animalEmoji)
    {
      <div class="col-3">
        <button type="button" class="btn btn-outline-dark">
          <h1>@animal</h1>
        </button>
      </div>
    }
  </div>
</div>
```

Символ @ включает код C# в страницу Razor. Это цикл foreach, который многократно выполняет один и тот же код, чтобы сгенерировать кнопку для каждого эмодзи в списке.

Цикл foreach повторяет весь код, заключенный в фигурные скобки {}, по одному разу для каждого эмодзи в списке, заменяя @animal каждым эмодзи в списке. Так как список состоит из 16 эмодзи, в результате будет сгенерирована серия из 16 кнопок.

Visual Studio помогает в написании кода C#

Blazor позволяет создавать интерактивные, полнофункциональные приложения, сочетающие разметку HTML с кодом C#. К счастью, Visual Studio IDE содержит полезную функциональность, которая помогает писать код C#.

1 Добавьте код C# в файл Index.razor.

Начните с **добавления блока @code** в конец файла *Index.razor*. (Текущее содержимое файла пока не трогайте — оно будет удалено позднее.) Перейдите к последней строке файла и введите @code {. IDE вставляет закрывающую фигурную скобку } за вас. Нажмите клавишу Enter, чтобы добавить строку между двумя скобками:

```
9 @code {
10 |
11 }
```

2 Пользуйтесь окном IntelliSense в IDE, упрощающим написание кода C#.

Установите курсор в строке между { фигурными скобками } и введите букву **L**. IDE открывает **окно IntelliSense** с вариантами автозавершения. Выберите в окне вариант List<>:

```
@code {
  L
}
◆ LinkedListNode<>
◆ List<>
◆ LoaderOptimization
◆ LoaderOptimizationAttribute
```

Окно IntelliSense в IDE помогает писать код C#, предлагая полезные варианты автозавершения. Варианты перебираются клавишами со стрелками, а выбор осуществляется клавишей Enter (также можно воспользоваться мышью).

IDE подставляет List в код. Добавьте **открывающую угловую скобку** (знак «меньше») <; IDE автоматически добавляет закрывающую скобку > и устанавливает курсор между скобками.

3 Начните создавать список List для хранения эмодзи.

Введите s, чтобы открыть следующее окно IntelliSense:

```
@code {
  List<s>
}
◆ string
◆ struct
◆ svm
```

Выберите вариант string — IDE добавляет его между угловыми скобками. Нажмите клавишу со стрелкой →, затем пробел и **введите** animalEmoji = new. Снова нажмите клавишу пробел, чтобы открыть следующее окно IntelliSense. **Нажмите Enter**, чтобы выбрать значение по умолчанию, List<string>, из списка вариантов.

```
@code {
  List<string> animalEmoji = new
}
◆ List<>
◆ List<string>
◆ LoaderOptimization
◆ LoaderOptimizationAttribute
```

После этого код должен выглядеть так: List<string> animalEmoji = new List<string>

4 Завершите создание списка эмодзи.

Начните с **добавления блока @code** в конце файла *Index.razor*. Перейдите к последней строке и **введите** @code {. IDE вставит закрывающую фигурную скобку } за вас. Нажмите Enter, чтобы вставить строку между скобками, затем:

- * Введите **открывающую круглую скобку** (– IDE автоматически вставляет парную закрывающую скобку.
- * Нажмите клавишу →, чтобы выйти из круглых скобок.
- * Введите **открывающую фигурную скобку** { – IDE также вставляет закрывающую скобку.
- * Нажмите Enter, чтобы добавить строку между фигурными скобками, после чего **вставьте точку с запятой ;** после закрывающей фигурной скобки.

В итоге последние шесть строк в конце файла *Index.razor* должны выглядеть так:

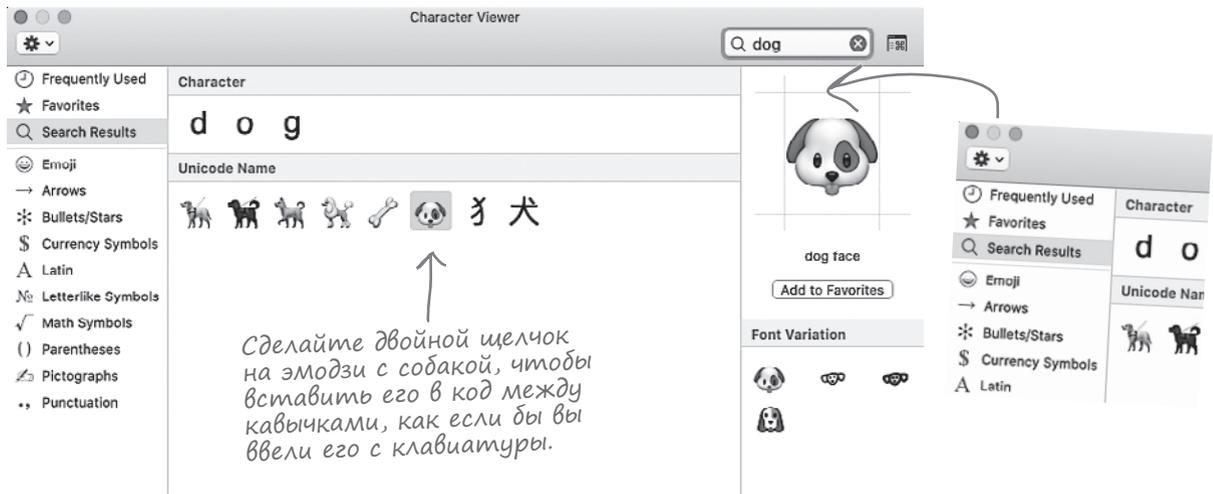
```
@code {
    List<string> animalEmoji = new List<string>()
    {
        ...
    };
}
```

За дополнительной информацией о командах обращайтесь к главе 2.

Поздравляем – вы только что создали свою первую **команду C#**. Но все только начинается! Вы создали список для хранения эмодзи. **Введите двойную кавычку "** в пустой строке – IDE добавит закрывающую кавычку.

5 Используйте Character Viewer для ввода эмодзи.

Затем **выберите команду Edit >> Emoji & Symbols (^⌘Space)**, чтобы запустить программу macOS Character Viewer. Установите курсор между кавычками, а затем перейдите в Character Viewer и проведите поиск по строке «dog»:



Последние шесть строк в конце файла *Index.razor* принимают следующий вид:

```
@code {
    List<string> animalEmoji = new List<string>()
    {
        ...
        "🐶"
    };
}
```

← О том, как работают списки List, рассказано в главе 8.

Завершение создания списка эмодзи и вывод их в приложении

Вы только что добавили эмодзи с изображением собаки в список `animalEmoji`. Теперь добавим второй эмодзи с собакой — поставьте запятую после второй кавычки, затем пробел, еще одну кавычку, эмодзи, кавычку и запятую:

```
@code {
    List<string> animalEmoji = new List<string>()
    {
        "🐶", "🐶",
    };
}
```

Добавьте после этой строки другую, которая выглядит точно так же, но вместо эмодзи с собакой в ней используются эмодзи с волком. Добавьте еще шесть аналогичных строк с парами коров, лис, кошек, львов, тигров и хомяков. В результате список `animalEmoji` должен содержать восемь пар эмодзи:

```
@code {
    List<string> animalEmoji = new List<string>()
    {
        "🐶", "🐶",
        "🐺", "🐺",
        "🐮", "🐮",
        "🐱", "🐱",
        "🐼", "🐼",
        "🐯", "🐯",
        "🐹", "🐹",
    };
}
```

Подсказки для IDE: отступы

IDE автоматически расставляет отступы во всем вводимом коде C#. Но когда вы вводите эмодзи или теги HTML, может оказаться, что автоматические отступы не всегда выглядят так, как вы хотите. Проблема легко решается: выделите текст и нажмите `→` (Tab), чтобы создать отступ, или `⇧→` (Shift+Tab), чтобы удалить отступ.

Замена содержимого страницы

Удалите следующие строки в начале страницы:

```
<h1>Elementary, my dear Watson.</h1>
Welcome to your new app.
<SurveyPrompt Title="How is Blazor working for you?" />
```

Затем установите курсор в третьей строке и введите `<st` — IDE открывает окно IntelliSense:

```
1 @page "/"
2
3 <st
4 > datalist
5 > strong
6 style
7
```

IDE помогает создавать разметку HTML страницы — в данном случае создается **тег HTML**. Если вы не знаете HTML, это нормально; в книге будет приведен весь код, необходимый для работы приложений.

Выберите в списке пункт **style**, затем введите `>`. IDE добавляет *закрывающий тег HTML*: `<style></style>`

Установите курсор между `<style>` и `</style>`, нажмите Enter; **аккуратно введите следующий код**. Убедитесь в том, что код вашего приложения точно соответствует приведенному.

```
<style>
  .container {
    width: 400px;
  }

  button {
    width: 100px;
    height: 100px;
    font-size: 50px;
  }
</style>
```

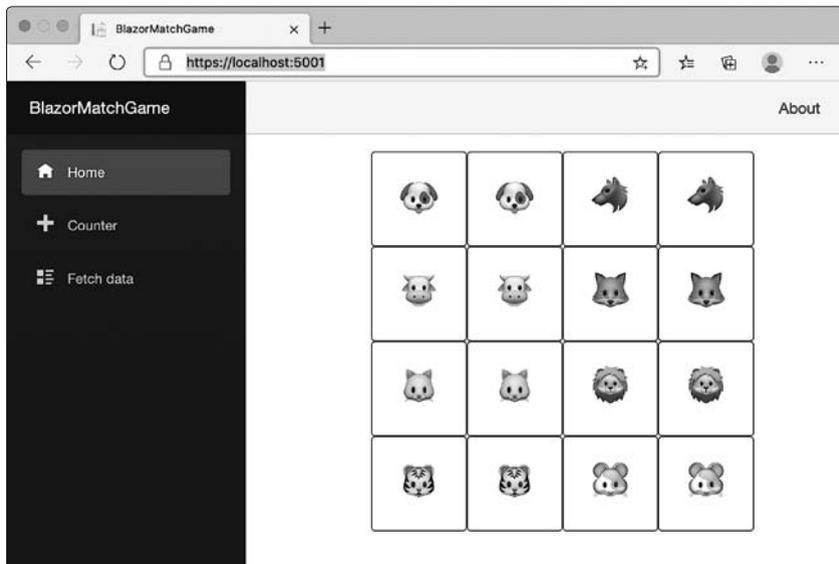
Экран игры строится из кнопок. Она реализуется очень простой таблицей стилей CSS, которые задают общую ширину контейнера, а также высоту и ширину каждой кнопки. Так как ширина контейнера составляет 400 пикселей, а ширина каждой кнопки — 100 пикселей, страница допускает размещение только четырех столбцов в строке, после чего добавляется разрыв строки; таким образом образуется некое подобие сетки.

Перейдите к следующей строке и воспользуйтесь окном IntelliSense для **ввода открывающего и закрывающего тега `<div>`**, как было сделано ранее для `<style>`. Затем **тщательно введите следующий код** (проследите за тем, чтобы он точно совпадал с приведенным в тексте):

```
<div class="container">
  <div class="row">
    @foreach (var animal in animalEmoji)
    {
      <div class="col-3">
        <button type="button" class="btn btn-outline-dark">
          <h1>@animal</h1>
        </button>
      </div>
    }
  </div>
</div>
```

Если ранее вы уже работали с HTML, вы заметите конструкции `@foreach` и `@animal`, непохожие на стандартные элементы HTML. Это код **Blazor — код C#, встраиваемый непосредственно в HTML.**

Каждая кнопка на странице содержит изображение животного. Игрок нажимает кнопки, чтобы выбрать совпадающие пары.



Убедитесь в том, что экран приложения похож на этот снимок экрана. Тогда вы будете точно знать, что код введен без опечаток.

Перестановка животных в случайном порядке

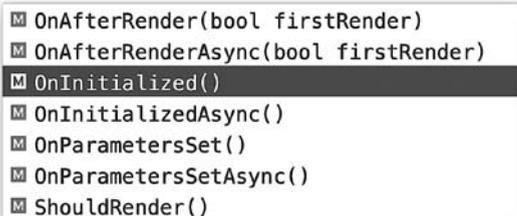
Если бы все пары животных располагались по соседству, игра была бы слишком простой. Добавим код C# для случайной перестановки животных, чтобы они располагались в случайном порядке каждый раз, когда игрок перезагружает страницу.

- 1 Установите курсор после точки с запятой ; над закрывающей фигурной скобкой } в конце файла *Index.razor* и **дважды нажмите Enter**. Затем воспользуйтесь окном IntelliSense, как это делалось ранее, для ввода следующей строки кода:

```
List<string> shuffledAnimals = new List<string>();
```

- 2 Затем **введите** `protected override` (IntelliSense может автоматически завершить эти ключевые слова). Как только вы введете эти ключевые слова и пробел, откроется окно IntelliSense — **выберите** `OnInitialized()` из списка:

`protected override`



IDE автоматически заполняет код **метода** с именем `OnInitialized` (методы более подробно рассматриваются в главе 2):

```
protected override void OnInitialized()
{
    base.OnInitialized();
}
```

- 3 **Замените** вызов `base.OnInitialized()` **вызовом** `SetUpGame()`, чтобы ваш метод выглядел так:

```
protected override void OnInitialized()
{
    SetUpGame();
}
```

Затем **добавьте следующий метод** `SetUpGame` под методом `OnInitialized` — и снова окно IntelliSense поможет вам в этом:

```
private void SetUpGame()
{
    Random random = new Random();
    shuffledAnimals = animalEmoji
        .OrderBy(item => random.Next())
        .ToList();
}
```

← За дополнительной информацией о методах обращайтесь к главе 2.

В процессе ввода метода `SetUpGame` можно заметить, что IDE часто открывает окна IntelliSense, чтобы упростить ввод кода. Чем больше вы пользуетесь Visual Studio для написания кода C#, тем полезнее будут эти окна: со временем вы убедитесь, что они значительно ускоряют работу. А пока используйте их для предотвращения опечаток при вводе — ваш код должен **полностью совпадать** с приведенным в книге, иначе приложение не будет работать.



Вскоре вы узнаете о методах C# намного больше.

Вы только что добавили метод в свое приложение. Впрочем, если вы еще не на сто процентов понимаете, что такое метод, это вполне нормально. В следующей главе методы и структура кода C# рассматриваются намного подробнее.

4

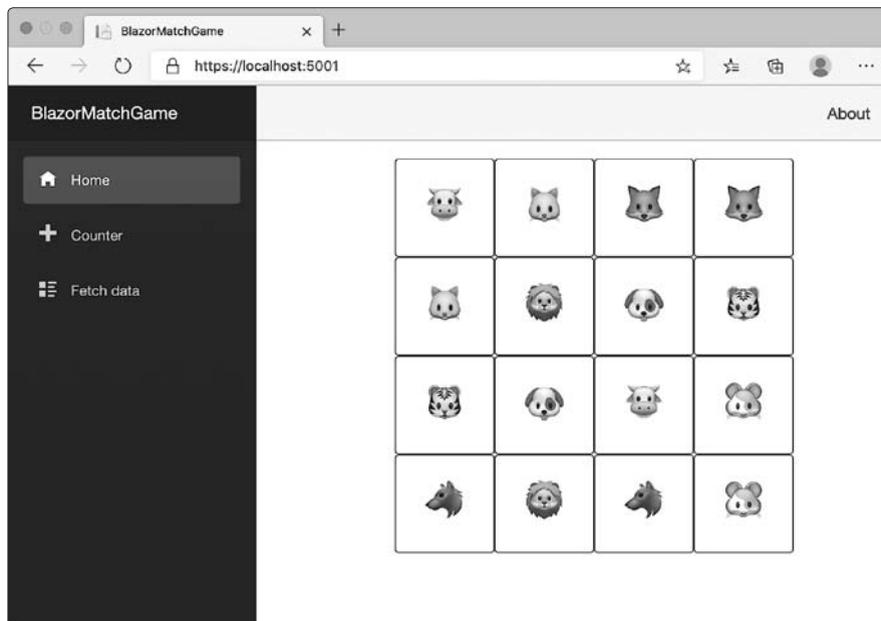
Прокрутите окно к разметке HTML и найдите следующий код: `@foreach (var animal in animalEmoji)`

Сделайте двойной щелчок на имени `animalEmoji`, чтобы выделить его, и введите символ `s`. IDE открывает окно IntelliSense. Выберите в списке пункт `shuffledAnimals`:

```
@foreach (var animal in s)
{
    <div class="col-md-3" style="border: 1px solid #ccc; padding: 5px; text-align: center; background-color: #f9f9f9;">
        <button type="button" class="btn btn-primary" style="width: 100%; text-align: center;">
            <h1>@animal
        </button>
    </div>
}
```

IntelliSense suggestions: `% sbyte`, `% short`, **`shuffledAnimals`**, `sim`, `% sizeof`, `% stackalloc`.
 Additional info: `(field) List<string> Index.shuffledAnimals`

Снова запустите приложение. Животные в списке переставляются в случайном порядке. **Перезагрузите страницу** в браузере — порядок размещения эмодзи изменится. Каждый раз, когда вы перезагружаете страницу, эмодзи будут выводиться в новом порядке.



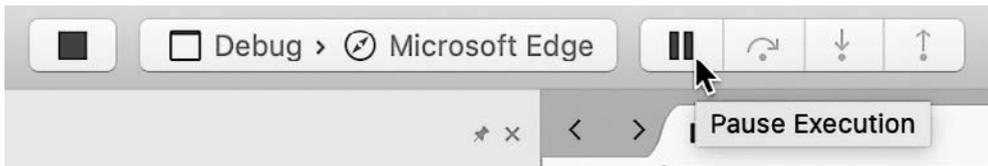
Еще раз: убедитесь в том, что экран запущенного приложения похож на этот снимок экрана. Тогда вы будете точно знать, что код введен без опечаток. Двигайтесь дальше только после того, как ваша игра будет переставлять животных в случайном порядке при каждой перезагрузке страницы браузера.

Игра работает в отладчике

Когда вы щелкаете на кнопке Run (▶) или выбираете команду Start Debugging (⌘↵) из меню Run, чтобы запустить программу, Visual Studio переходит в режим отладки.

Чтобы понять, что приложение работает в режиме отладки, достаточно взглянуть на элементы отладки на панели инструментов. Кнопка Start заменяется кнопкой Stop <>, раскрывающийся список для выбора браузера блокируется, и появляются новые элементы управления.

Наведите указатель мыши на кнопку Pause Execution, чтобы увидеть подсказку:



Чтобы остановить приложение, щелкните на кнопке Stop или выберите команду Stop (⇧⌘↵) из меню Run.



Ого, игра уже неплохо выглядит!

Все готово для следующего этапа работы.

Построение новой игры не сводится к написанию кода — вы также работаете над проектом. Эффективный способ реализации проектов основан на построении их с небольшими приращениями и тщательным анализом промежуточных результатов, гарантирующим, что работа идет в правильном направлении. При таком подходе у вас будет достаточно возможностей, чтобы сменить курс при необходимости.

Еще одно упражнение, в котором вам придется поработать карандашом. Не жалейте времени на выполнение всех этих упражнений, потому что они помогут быстрее закрепить важные концепции C# в вашем мозге.

* КТО И ЧТО ДЕЛАЕТ? *

Поздравляем — вы создали работающую программу! Разумеется, программирование несколько сложнее простого копирования кода из книги. Но даже если вы никогда не писали код прежде, вас удивит, сколько всего вы уже понимаете. Соедините линией каждую команду C# в левом столбце с описанием того, что делает эта команда, в правом столбце. Мы привели решение для первой команды, чтобы вам было проще.

Команда C#

Что делает

```
List<string> animalEmoji = new List<string>()
{
    "🐼", "🐼",
    "🐼", "🐼",
    "🐼", "🐼",
    "🐼", "🐼",
    "🐼", "🐼",
    "🐼", "🐼",
    "🐼", "🐼",
    "🐼", "🐼",
};
```

Создает второй список для хранения переставленных эмодзи.

Создает копии эмодзи, переставляет их в случайном порядке и сохраняет в списке shuffledAnimals.

```
List<string> shuffledAnimals = new List<string>();
```

Начало метода, выполняющего настройку игры.

```
protected override void OnInitialized()
{
    SetupGame();
}
```

Создает список из восьми пар эмодзи.

```
private void SetupGame()
{
```

Настраивает игру при каждой перезагрузке страницы.

```
    Random random = new Random();
```

Создает новый генератор случайных чисел.

```
    shuffledAnimals = animalEmoji
        .OrderBy(item => random.Next())
        .ToList();
```

Конец метода, выполняющего настройку игры.

```
}
```

КТО И ЧТО ДЕЛАЕТ? решение

Команда C#	Что делает
<pre>List<string> animalEmoji = new List<string>() { "🐼", "🐼", "🐼", "🐼", "🐼", "🐼", "🐼", "🐼", "🐼", "🐼", "🐼", "🐼", "🐼", "🐼", };</pre>	Создает второй список для хранения переставленных эмодзи.
<pre>List<string> shuffledAnimals = new List<string>();</pre>	Создает копии эмодзи, переставляет их в случайном порядке и сохраняет в списке shuffledAnimals.
<pre>protected override void OnInitialized() { SetUpGame(); }</pre>	Начало метода, выполняющего настройку игры.
<pre>private void SetUpGame() { Random random = new Random(); shuffledAnimals = animalEmoji .OrderBy(item => random.Next()) .ToList(); }</pre>	Создает список из восьми пар эмодзи. Настраивает игру при каждой перезагрузке страницы. Создает новый генератор случайных чисел. Конец метода, выполняющего настройку игры.

MINI

Возьми в руку карандаш



Следующее упражнение поможет вам лучше понять код C#.

1. Возьмите лист бумаги и поверните его набок, чтобы он лежал в альбомной ориентации. Нарисуйте вертикальную линию в середине.
2. Запишите весь код C# в левой части, оставляя свободное место между командами. (Особая точность с эмодзи не нужна.)
3. В правой части листа запишите каждый из ответов «Что делает» рядом с командой, с которой он соединен. Прочитайте оба столбца — код должен постепенно приобретать смысл.



Я не уверена насчет этих упражнений «Возьми в руку карандаш» и с соединением ответов. Разве не лучше просто дать мне код, который можно ввести в IDE?

Отработка навыков понимания кода повысит вашу квалификацию разработчика.

Письменные упражнения являются **обязательными**. Они предоставляют вашему мозгу новый способ усвоения информации. Однако они делают нечто еще более важное: они предоставляют вам возможность *ошибаться*. Ошибки — важная часть обучения, и мы тоже допустили множество ошибок (возможно, вы даже найдете одну-две опечатки в этой книге!). Никто не пишет идеальный код с первого раза — действительно хорошие программисты всегда предполагают, что написанный сегодня код, возможно, потребует изменить завтра. Позднее в книге вы узнаете о рефакторинге — приеме программирования, сутью которого становится улучшение вашего кода после того, как он будет написан.

В таких списках «ключевых моментов» приводятся краткие сводки многих идей и средств, которые были представлены в этой главе.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Visual Studio — **интегрированная среда разработки (IDE) компании Microsoft**, которая упрощает редактирование файлов с кодом C# и выполнения различных операций с ними.
- **Консольные приложения .NET Core** — кросс-платформенные приложения с текстовым вводом и выводом.
- Приложения **Blazor WebAssembly** позволяют строить полнофункциональные интерактивные веб-приложения из кода C# и разметки HTML.
- Функция IDE **IntelliSense** помогает быстрее и точнее вводить код.
- Visual Studio может **выполнять приложения Blazor** в режиме отладки, открывая браузер для отображения приложения.
- Пользовательские интерфейсы приложений Blazor строятся в формате **HTML** — языке разметки для построения веб-страниц.
- **Razor** позволяет включать код C# прямо в разметку HTML. Файлы страниц Razor имеют расширение *.razor*.
- Используйте синтаксис **@** для встраивания кода C# в страницу Razor.
- **Цикл foreach** в страницах Razor повторяет блок кода HTML для каждого элемента в списке.



Включать проект в систему управления версиями не обязательно.

Добавление нового проекта в систему управления версиями

В этой книге мы будем строить много разных проектов. Только представьте, как удобно было бы иметь место, в котором их можно было бы сохранить или загружать позднее с любого устройства? А если вы допустите ошибку, разве не будет удобно вернуться к предыдущей версии вашего кода? Что ж, вам повезло! Именно эту задачу решает **система управления версиями**: она предоставляет простые средства для создания резервной копии всего кода и отслеживания всех вносимых изменений. Visual Studio позволяет легко добавлять проекты в систему управления версиями.

Git — популярная система управления версиями, и Visual Studio может публиковать ваш исходный код в любом **репозитории** Git. Мы считаем **GitHub** одним из самых удобных провайдеров Git. Для сохранения кода вам понадобится учетная запись GitHub. Если у вас еще нет учетной записи, зайдите на сайт <https://github.com> и создайте ее.

Когда ваша учетная запись GitHub будет настроена, вы можете воспользоваться встроенными средствами управления версиями вашей IDE. **Выберите команду *Version Control* >> *Publish in Version Control***, чтобы открыть окно Clone Repository:

Возможно, вы работаете на компьютере офисной сети, которая не имеет доступа к GitHub (рекомендованному нами провайдеру Git). А может быть, вам просто не хочется этого делать. Как бы то ни было, этот шаг можно пропустить — или же опубликовать код в приватном репозитории, если вы хотите хранить резервную копию, но не хотите, чтобы она была доступна для других.

После того как вы создадите удаленный репозиторий на GitHub, вы сможете вставить сюда URL-адрес.

Введите свое имя пользователя.

В документации Visual Studio for Mac приведено полное руководство по созданию проектов на GitHub и публикации их из Visual Studio. Оно включает пошаговые инструкции для создания удаленного репозитория на GitHub и публикации проектов в репозитории Git непосредственно из Visual Studio. Мы считаем, что все ваши проекты стоит публиковать на GitHub — это позволит вам легко вернуться к ним в будущем. <https://docs.microsoft.com/en-us/visualstudio/mac/set-up-git-repository>

ВЫ НАХОДИТЕСЬ ЗДЕСЬ

начинаем программировать на C#



СОЗДАНИЕ
ПРОЕКТА



СЛУЧАЙНАЯ
РАССТАНОВКА
ЖИВОТНЫХ



ОБРАБОТКА
ЩЕЛЧКОВ



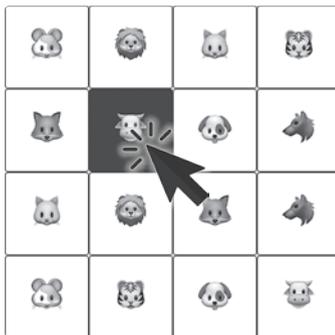
ПРОВЕРКА
ПОБЕДЫ
ИГРОКА



ДОБАВЛЕНИЕ
ТАЙМЕРА

Добавление кода C# для обработки щелчков

Мы создали кнопки со случайными эмодзи. Теперь необходимо сделать так, чтобы эти кнопки реагировали на щелчки пользователя. Вот как это делается:



Игрок щелкает на первой кнопке.

Игрок щелкает на парах кнопок. Когда он щелкает на первой кнопке, игра сохраняет информацию о животном, связанном с этой кнопкой.



Игрок щелкает на второй кнопке.

Игра определяет животное на второй кнопке и сравнивает его с животным, сохраненным при первом щелчке.



Игра проверяет совпадение.

Если животные *совпадают*, игра перебирает все эмодзи в списке. Она находит в списке эмодзи, соответствующие найденной паре, и заменяет их пустыми строками.

Если животные *не совпадают*, игра ничего не делает.

В *любом случае* последнее найденное животное сбрасывается, чтобы повторить процедуру для следующего щелчка.

Назначение обработчиков щелчков кнопкам

Когда вы щелкаете на кнопке, приложение должно что-то сделать. В веб-страницах щелчок является **событием**. У веб-страниц также существуют и другие события, например завершение загрузки страницы или изменение ввода. **Обработчик события** представляет собой код C#, который выполняется каждый раз при возникновении определенного события. Мы добавим обработчик события, реализующий функциональность кнопки.

Не беспокойтесь, если вы не на сто процентов понимаете, что делает этот код C#! Пока сосредоточьтесь на том, чтобы ваш код полностью совпадал с приведенным.

Код обработчика события

Добавьте следующий код в страницу Razor, непосредственно перед закрывающей фигурной скобкой } в нижней части:

```
string lastAnimalFound = string.Empty;

private void ButtonClick(string animal)
{
    if (lastAnimalFound == string.Empty)
    {
        // Выбор первого элемента пары. Запомнить его.
        lastAnimalFound = animal;
    }
    else if (lastAnimalFound == animal)
    {
        // Совпадение обнаружено! Выполнить сброс для следующей пары.
        lastAnimalFound = string.Empty;

        // Найденные животные заменяются пустыми строками, чтобы скрыть эмодзи на экране.
        shuffledAnimals = shuffledAnimals
            .Select(a => a.Replace(animal, string.Empty))
            .ToList();
    }
    else
    {
        // Пользователь выбрал непарных животных.
        // Сбросить сохраненный выбор.
        lastAnimalFound = string.Empty;
    }
}
```

Строки с // содержат **комментарии**. Они ничего не делают — комментарии приводятся только для того, чтобы код стал более понятным. Мы включили их для удобочитаемости кода.

Это запрос LINQ. Технология LINQ более подробно рассматривается в главе 9.

Связывание обработчиков событий с кнопками

Теперь необходимо внести изменения, чтобы при щелчках на кнопках вызывался метод ButtonClick:

```
@foreach (var animal in animalEmoji)
{
    <div class="col-3">
        <button @onclick="@(() => ButtonClick(animal))"
            type="button" class="btn btn-outline-dark">
            <h1>@shuffledAnimals</h1>
        </button>
    </div>
}
```

Включите атрибут @onclick в разметку HTML в foreach. Будьте внимательны с круглыми скобками.

Когда мы предлагаем вам обновить что-то в блоке кода, остальной код выделяется более светлым, а изменяемый код — жирным шрифтом.



Обработчик события под увеличительным стеклом

Разберемся с тем, как работают обработчики событий. Мы связали код из обработчика события с предшествующими объяснениями того, как игра обнаруживает щелчки. Взгляните на следующий код и сравните его с кодом, только что введенным в IDE. Попробуйте понять его логику — ничего страшного, если что-то останется непонятным, достаточно составить общее представление о том, как добавленные фрагменты сочетаются друг с другом. Это упражнение развивает ваши навыки понимания кода C#.

Игрок щелкает на первой кнопке.

Код проверяет, является ли эта кнопка первой, на которой был сделан щелчок. В таком случае информация о животном, связанном с кнопкой, сохраняется в `lastAnimalFound`.

```
if (lastAnimalFound == string.Empty)
{
    lastAnimalFound = animal;
}
```



Игрок щелкает на второй кнопке.

Команды между открывающей { и закрывающей } фигурной скобкой выполняются только в том случае, если игрок щелкнул на кнопке с животным, совпадающим с тем, которое было связано с первой кнопкой.

```
else if (lastAnimalFound == animal)
{
}
```



Игра проверяет совпадение.

Код C# выполняется только в том случае, если второе животное совпадает с первым. Он перебирает список эмодзи и заменяет элементы, совпадающие с текущей парой, пустыми строками.

```
shuffledAnimals = shuffledAnimals
    .Select(a => a.Replace(animal, string.Empty))
    .ToList();
```

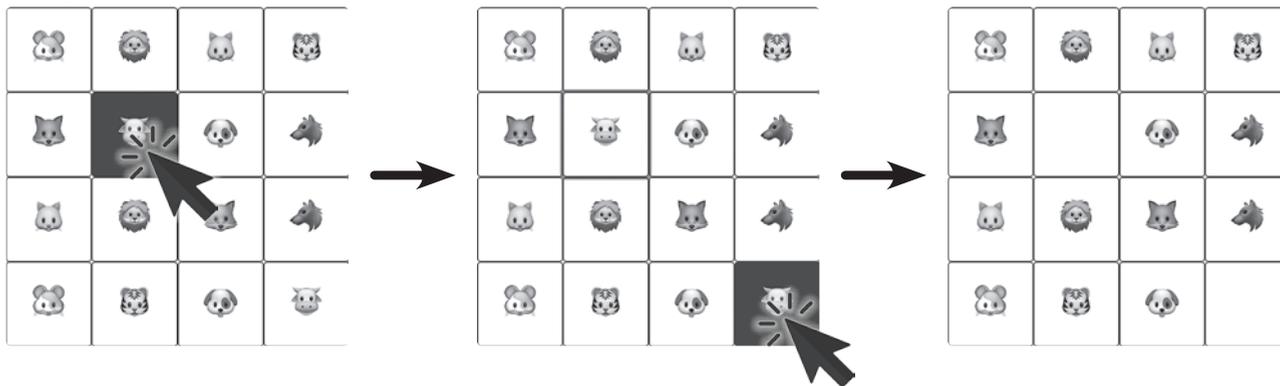
Эта команда встречается в коде дважды: в части, которая выполняется, если второе животное совпадает с первым, и в части, выполняемой при несовпадении. Эмодзи последнего найденного животного заменяется пустой строкой, так что следующий щелчок на кнопке зафиксирует первое животное в паре.

```
→ lastAnimalFound = string.Empty;
```

**В этом коде присутствует ошибка! А вы сможете ее обнаружить?
Мы найдем ошибку и исправим ее в следующем разделе**

Тестирование обработчика события

Снова запустите приложение. Когда страница появится на экране, протестируйте обработчик события: щелкните на кнопке, а затем щелкните на кнопке с парным эмодзи. Оба изображения должны исчезнуть со страницы.



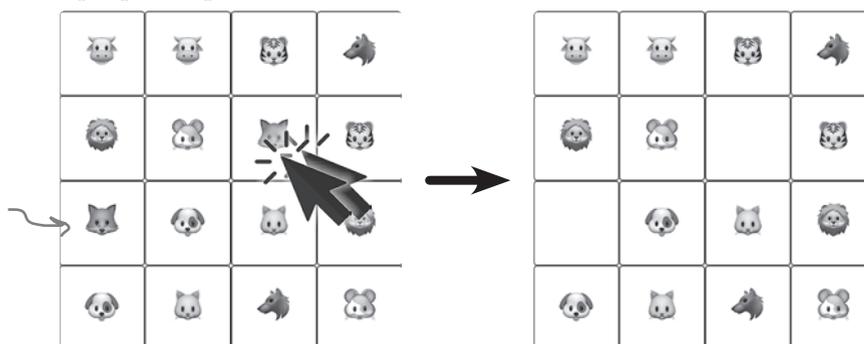
Щелкните на другой паре, затем на следующей и т. д. Приложение позволяет щелкать на парах, пока все кнопки не останутся пустыми. Поздравляем, вы нашли все пары!



Но что произойдет, если щелкнуть на одной кнопке дважды?

Перезагрузите страницу в браузере, чтобы сбросить игру. Но на этот раз вместо того, чтобы выбрать пару, снова щелкните на той же кнопке. В игре обнаруживается ошибка! Этот щелчок должен быть проигнорирован, но вместо этого программа работает так, словно было найдено совпадение.

Если снова щелкнуть на той же кнопке, игра действительно так, словно вы нашли совпадение. Но такого быть не должно!



Диагностика проблемы в отладчике

У каждой ошибки есть объяснение – в программе ничего не происходит без причин, но не каждую ошибку легко обнаружить.

Понимание ошибки – первый шаг к ее исправлению. К счастью, для этого существует превосходный инструмент – отладчик Visual Studio.

1 Подумайте, что могло пойти не так.

Первое, на что следует обратить внимание, – что ошибка **воспроизводится**: каждый раз, когда вы повторно щелкаете на одной кнопке, программа работает так, словно вы щелкнули на паре.

Второй важный момент: вы **достаточно хорошо** представляете, где находится ошибка. Проблема возникла только *после* того, как вы добавили код обработки события Click, так что начинать следует именно с него.

2 Добавьте точку прерывания в только что написанный код обработчика события Click.

Щелкните в первой строке метода ButtonClick и выберите **команду меню Run>>Toggle Breakpoint (⌘\)**. Строка изменяет цвет, а на левых полях появляется точка:

```

62     private void ButtonClick(string animal)
63     {
64         if (lastAnimalFound == string.Empty)
65         {
66             //First selection of the pair. Remember it.
67             lastAnimalFound = animal;
68         }
    
```

Когда в строке устанавливается точка прерывания, IDE меняет цвет фона и отображает точку на полях слева.



Анатомия отладчика

Когда ваше приложение приостанавливается в отладчике (это называется «прерыванием»), на панели инструментов появляются элементы отладки. В книге вы еще не раз потренируетесь в их использовании, поэтому запоминать их назначение сейчас не обязательно. Пока просто прочитайте описания, наведите указатель мыши на каждый элемент, чтобы увидеть подсказку, и найдите соответствующие сочетания клавиш в меню Run (например, **↑⌘O** для Step Over).

Кнопка Continue Execution перезапускает приложение.

Кнопка Pause Execution приостанавливает приложение.

Кнопка Step Over также выполняет следующую команду, но если эта команда является вызовом метода, то будет выполнен весь метод в целом.

Кнопка Step Out завершает выполнение текущего метода и прерывает программу в строке, следующей за той, из которой он был вызван.

Кнопка Step Into выполняет следующую команду. Если эта команда является вызовом метода, то выполняется только первая команда в этом методе.



Отладка обработчика события

После установки точки прерывания мы сможем составить некоторое представление о том, что же произошло не так в вашем коде.

3 Щелкните на кнопке с животным, чтобы активизировать точку прерывания.

Если приложение уже выполняется, остановите его и закройте все окна браузера. Затем снова **запустите приложение** и щелкните на любой кнопке с животным. Среда Visual Studio активизируется, а ее окно выходит на передний план. Строка, в которой установлена точка прерывания, выделяется другим цветом:

```

62     private void ButtonClick(string animal)
63     {
64     |     if (lastAnimalFound == string.Empty)
65     |     {

```

Переместите указатель мыши к первой строке метода (начинающейся со слов `private void`) и наведите указатель мыши на `animal`. На экране появляется маленькое окно с животным, на котором был сделан щелчок:

Наведите указатель мыши на «animal», чтобы увидеть эмодзи, на котором был сделан щелчок.

```

private void ButtonClick(string animal,
{

```



Нажмите кнопку **Step Over** или выберите команду меню `Run>Step Over` (⇧⌘O). Цветовое выделение переходит к строке `{`. Снова выполните ту же команду, чтобы переместить выделение к следующей команде:

```

64     if (lastAnimalFound == string.Empty)
65     {
66         //First selection of the pair. Remember it.
67     |     lastAnimalFound = animal;
68     }

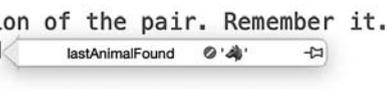
```

Повторите команду пошагового выполнения еще раз, а затем наведите указатель мыши на `lastAnimalFound`:

```

66         //First selection of the pair. Remember it.
67     lastAnimalFound
68     }

```



Выполненная команда задает значение переменной `lastAnimalFound`, чтобы оно соответствовало `animal`.

Так в коде сохраняется первое животное, на котором щелкнул игрок.

4 Продолжите выполнение.

Нажмите кнопку **Continue Execution** или выберите команду меню `Run>>Continue Debugging` (⌘↵). Переключитесь обратно в браузер — игра продолжит выполняться с точки прерывания.

5

Щелкните на парном животном

Найдите кнопку с парным эмодзи и щелкните на ней. IDE активизирует точку прерывания и снова приостанавливает приложение. Нажмите кнопку **Step Over** – первый блок пропускается, и управление передается второму:

```

69 else if (lastAnimalFound == animal)
70 {
71     // Совпадение обнаружено! Выполнить сброс для следующей пары
72     lastAnimalFound = string.Empty;
    
```

Наведите указатель мыши на `lastAnimalFound` и `animal` – в обеих переменных должны храниться одинаковые эмодзи. Так обработчик события узнает о найденном совпадении. **Выполните команду Step Over еще три раза.**

```

74 // Найденные животные заменяются пустыми строками.
75 shuffledAnimals = shuffledAnimals
76     .Select(a => a.Replace(animal, string.Empty))
77     .ToList();
    
```

Теперь наведите указатель мыши на `shuffledAnimals`. В открывшемся окне перечислено несколько вариантов. Щелкните на треугольнике рядом с `shuffledAnimals`, чтобы раскрыть раздел, а затем раскройте `_items`, чтобы просмотреть всех животных:

shuffledAnimals	System.Collections.Generic.List<string>
_items	string[(16)]
0	'🐱'
1	'🐱'
2	'🐱'
3	'🐱'
4	'🐱'
5	'🐱'
6	'🐱'
7	'🐱'
8	'🐱'
9	'🐱'
10	'🐱'
11	'🐱'
12	'🐱'
13	'🐱'
14	'🐱'
15	'🐱'

shuffledAnimals представляет собой список List со всеми животными, находящимися в игре на данный момент. Используйте кнопки с треугольниками, чтобы сначала раскрыть shuffledAnimals, а затем вернуть _items, чтобы увидеть содержащиеся в них элементы.

После того как вы раскроете shuffledAnimals и _items, используйте отладчик для просмотра содержимого List. О том, что такое списки List и как они работают, рассказано в главе 8.

Нажмите **Step Over** еще раз, чтобы выполнить команду удаления совпадений из списка. Затем снова наведите указатель мыши на `shuffledAnimals` и просмотрите элементы. Среди них два пустых (`null`) значения там, где находились совпадающие эмодзи:

6	'🐱'
7	(null)
8	'🐱'

Мы проанализировали большой объем доказательств и нашли важные улики. Как вы думаете, что стало причиной проблемы?

Поиск ошибки, породившей проблему...

Пришло время побывать в роли Шерлока Холмса. Мы собрали немало улик. Вот что нам известно на данный момент:



|| о следу

1. Каждый раз, когда вы щелкаете на кнопке, выполняется обработчик события.
2. Обработчик события использует `animal` для определения того, на каком животном вы щелкнули первым.
3. Обработчик события использует `lastAnimalFound` для определения того, на каком животном был сделан второй щелчок.
4. Если значение `animal` равно `lastAnimalFound`, обработчик делает вывод, что найдено совпадение, и удаляет соответствующих животных из списка.

Что же произойдет, если щелкнуть на одной кнопке с животным дважды? Давайте узнаем! Повторите те же действия, которые выполнялись ранее, но на этот раз **щелкните на одном животном дважды**. Посмотрите, что произойдет на шаге (5).

Наведите указатель мыши на `animal` и `lastAnimalFound`, как это делалось ранее. Они совпадают! *Это происходит из-за того, чтоб обработчик не различает разные кнопки с одинаковыми животными.*

...и ее исправление!

Теперь, когда мы знаем, из-за чего происходит ошибка, становится ясно, как ее исправить: нужно научить обработчик события различать две кнопки с одинаковыми эмодзи.

Для начала **вносите следующие изменения** в обработчик `ButtonClick` (будьте внимательны и не пропустите какое-нибудь изменение):

```
string lastAnimalFound = string.Empty;
string lastDescription = string.Empty;

private void ButtonClick(string animal, string animalDescription)
{
    if (lastAnimalFound == string.Empty)
    {
        // Выбор первого элемента пары. Запомнить его.
        lastAnimalFound = animal;
        lastDescription = animalDescription;
    }
    else if ((lastAnimalFound == animal) && (animalDescription != lastDescription))
```

← Теперь каждая кнопка получает не только эмодзи с животным, но и описание, а обработчик события использует `lastDescription` для отслеживания описания.

← Проверяем, что совпадают как животные, так и описания.

Заменим цикл `foreach` другой разновидностью: циклом `for` — он подсчитывает животных:

```
<div class="row">

    @for (var animalNumber = 0; animalNumber < shuffledAnimals.Count; animalNumber++)
    {
        var animal = shuffledAnimals[animalNumber];
        var uniqueDescription = $"Button #{animalNumber}";

        <div class="col-3">
            <button onclick="@(() => ButtonClick(animal, uniqueDescription))"
                type="button" class="btn btn-outline-dark">@animal</button>
```

← Замените цикл `foreach` циклом `for`. Циклы рассматриваются в главе 2.

Теперь снова включите отладку приложения, как делалось ранее. На этот раз, если щелкнуть на одном животном дважды, управление будет передано в конец обработчика события. **Ошибка исправлена!**

Часто задаваемые вопросы

Обращайте внимание на врезки с вопросами и ответами. Они часто дают ответы на самые насущные вопросы, а также поднимают другие вопросы, которые могли прийти в голову другим читателям. Собственно, многие из них — вполне реальные вопросы от читателей предыдущего издания этой книги!

В: Вы упоминали о запуске сервера и веб-приложения. Что имеется в виду?

О: Когда вы запускаете свое приложение, IDE запускает выбранный вами браузер. В адресной строке браузера выводится URL-адрес вида `https://localhost:5001/` — если скопировать URL-адрес и вставить его в адресную строку другого браузера, в этом браузере также будет запущена ваша игра. Дело в том, что в браузере выполняется веб-приложение или веб-страница, которая выполняется исключительно в браузере. Как и любая веб-страница, она должна размещаться на веб-сервере.

В: К какому веб-серверу подключается мой браузер?

О: Ваш браузер подключается к серверу, работающему *внутри Visual Studio*. Щелкните на кнопке Application Output в нижней части IDE, чтобы открыть окно с результатом выполняемого приложения, — в данном случае это приложение, включающее сервер, который управляет вашим веб-приложением. Прокрутите окно или найдите в нем строку, в которой выводится информация о прослушивании входных подключений браузера:

```
Now listening on: https://localhost:5001
```

В: Когда я нажимаю `⌘→` (Command+Tab) для переключения между приложениями macOS, в системе обнаруживаются открытые экземпляры Edge или Chrome. Что происходит?

О: Каждый раз, когда вы останавливаете и перезапускаете свое приложение в Visual Studio, запускается новый экземпляр браузера, потому что для отладки должно устанавливаться отдельное подключение. Вы можете подключаться к другим экземплярам браузера, но отладка возможна только в браузере, запущенном IDE. Вы можете убедиться в этом сами: запустите, остановите и перезапустите свое приложение в IDE, затем установите точку прерывания. Только один из браузеров действительно будет приостановлен при достижении точки прерывания.

В: Веб-приложения Blazor выглядят намного сложнее консольных приложений. Они действительно работают так же?

О: Да. Если разобраться, весь код C# работает по одному принципу: сначала выполняется одна команда, потом следующая, потом следующая за ней и т. д. Одна из причин, по которым веб-приложения кажутся более сложными, заключается в том, что некоторые методы вызываются только при возникновении определенных событий, например при загрузке страницы или щелчке на кнопке. После того как метод будет вызван, он работает точно так же, как в консольном приложении, и вы можете убедиться в этом, установив в нем точку прерывания.

Если только вы не обладаете сверхъестественной способностью идеально программировать без единой ошибки, вы уже видели окно Errors в нижней части IDE. Оно открывается при попытке запуска проекта, содержащего ошибки. Вот как выглядело это окно, когда мы попытались исправить ошибку, но случайно допустили опечатку: `string lastDescription = string.Empty;`



Наличие ошибок всегда можно проверить, **построив** ваш код: либо запустите его, либо выберите команду Build All (`⌘B`) в меню Build. Если окно Error не появится, значит, код было **построен** успешно; это необходимо IDE для преобразования кода в **двоичную** форму, т. е. исполняемый файл, который может быть запущен системой macOS.

Давайте намеренно включим ошибку в код. Перейдите в первую строку метода `SetUpGame`, а затем введите в отдельной строке символы `Хyz`.

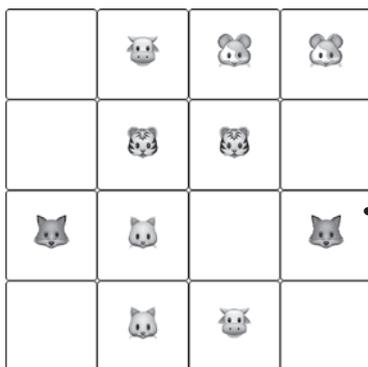
Попробуйте построить ваш код. IDE открывает окно Errors с сообщением `⊗ Errors: 1` в верхней части и с одной ошибкой. Если щелкнуть в другом месте, окно Errors исчезнет, но не беспокойтесь, вы всегда можете открыть его, щелкнув на кнопке `⊗ Errors` в нижней части IDE.



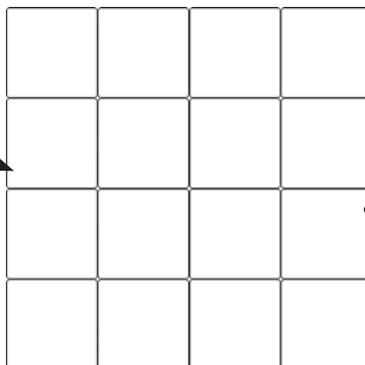
Добавление кода для сброса игры при победе

Игра работает так, как положено, – сначала игроку предоставляется сетка, заполненная животными. Он щелкает на парах животных, исчезающих при успешном составлении пары. Но что произойдет, когда будут найдены все совпадения? Необходимо каким-то образом сбросить состояние игры, чтобы игрок мог сделать следующую попытку.

Игрок щелкает на парах, найденные пары исчезают



Рано или поздно игрок находит все пары



Когда будет найдена последняя пара, игра возвращается в исходное состояние



Когда вы встречаете врезку «Мозговой штурм», как следует поразмыслите над заданным вопросом.



МОЗГОВОЙ ШТУРМ

Проанализируйте код C# и разметку HTML. Как вы думаете, какие части необходимо изменить, чтобы вернуть игру в исходное состояние, когда игрок найдет все совпадающие пары?



Упражнение

Ниже приведены четыре блока кода, которые следует включить в приложение. Когда все блоки окажутся на своих местах, игра будет возвращаться в исходное состояние после того, как игрок найдет все пары.

```
int matchesFound = 0;
```

```
matchesFound = 0;
```

```
matchesFound++;
if (matchesFound == 8)
{
    SetUpGame();
}
```

```
<div class="row">
  <h2>Matches found: @matchesFound</h2>
</div>
```

Ваша задача — определить, где следует разместить все четыре блока. Мы скопировали части кода и добавили четыре прямоугольника, по одному для каждого блока. Как определить, в каком прямоугольнике должен находиться каждый из четырех блоков?

```
<div class="container">
  <div class="row">
    @for (var animalNumber = 0; animalNumber < shuffledAnimals.Count; animalNumber++)
    {
      var animal = shuffledAnimals[animalNumber];
      var uniqueDescription = $"Button #{animalNumber}";

      <div class="col-3">
        <button @onclick="@(() => ButtonClick(animal, uniqueDescription))"
          type="button" class="btn btn-outline-dark">
          <h1>@animal</h1>
        </button>
      </div>
    }
  </div>
  _____
</div>
```

Какой из четырех блоков кода размещается в этом блоке?

```
List<string> shuffledAnimals = new List<string>();
```

```
private void SetUpGame()
{
```

```
    Random random = new Random();
    shuffledAnimals = animalEmoji
        .OrderBy(item => random.Next())
        .ToList();
    _____
}
```

Это упражнение не предназначено для выполнения «на бумаге» — оно выполняется изменением кода в IDE. Когда вы видите надпись Упражнение с кроссовками, это означает, что вы должны вернуться в IDE и приступить к написанию кода C#.

```
else if ((lastAnimalFound == animal) && (animalDescription != lastDescription))
{
    // Совпадение обнаружено! Выполнить сброс для следующей пары.
    lastAnimalFound = string.Empty;

    // Найденные животные заменяются пустыми строками.
    shuffledAnimals = shuffledAnimals
        .Select(a => a.Replace(animal, string.Empty))
        .ToList();
    _____
}
```

Когда вы выполняете упражнение по программированию, помните: подсматривать в решение — это не жульничество! Раздражение не способствует эффективному обучению — все мы легко можем споткнуться на какой-то мелочи, а решение поможет вам справиться с препятствием.



Упражнение
Решение

Ниже показано, как будет выглядеть код, когда все блоки окажутся на своих местах. **Добавьте все четыре блока в игру**, если это еще не было сделано ранее, чтобы игра автоматически возвращалась в исходное состояние при обнаружении игроком всех совпадений.

```
<div class="container">
  <div class="row">
    @for (var animalNumber = 0; animalNumber < shuffledAnimals.Count; animalNumber++)
    {
      var animal = shuffledAnimals[animalNumber];
      var uniqueDescription = $"Button #{animalNumber}";

      <div class="col-3">
        <button @onclick="@(() => ButtonClick(animal, uniqueDescription))"
          type="button" class="btn btn-outline-dark">
          <h1>@animal</h1>
        </button>
      </div>
    }
  </div>
  <div class="row">
    <h2>Matches found: @matchesFound</h2>
  </div>
</div>
```

Эта разметка Razor использует @matchesFound для того, чтобы страница выводила количество найденных совпадений под сеткой с кнопками.

```
List<string> shuffledAnimals = new List<string>();
```

Здесь игра хранит количество совпадений, найденных игроком на данный момент.

```
int matchesFound = 0;
```

```
private void SetUpGame()
{
```

```
    Random random = new Random();
    shuffledAnimals = animalEmoji
        .OrderBy(item => random.Next())
        .ToList();
```

```
    matchesFound = 0;
```

При настройке или сбросе игры количество найденных совпадений обнуляется.

```
}
```

```
else if ((lastAnimalFound == animal) && (animalDescription != lastDescription))
```

```
{
    // Совпадение обнаружено! Выполнить сброс для следующей пары.
    lastAnimalFound = string.Empty;
```

Каждый раз, когда игрок находит пару, этот блок увеличивает matchesFound на 1. Если найдены все 8 совпадений, игра возвращается в исходное состояние.

```
    // Найденные животные заменяются пустыми строками.
    shuffledAnimals = shuffledAnimals
        .Select(a => a.Replace(animal, string.Empty))
        .ToList();
```

```
    matchesFound++;
    if (matchesFound == 8)
    {
        SetUpGame();
    }
```



Вы достигли очередной контрольной точки вашего проекта! Возможно, игра еще не закончена, но она работает. Самое время сделать шаг назад и подумать над тем, как улучшить программу. Какие изменения вы бы предложили для того, чтобы сделать ее более интересной?

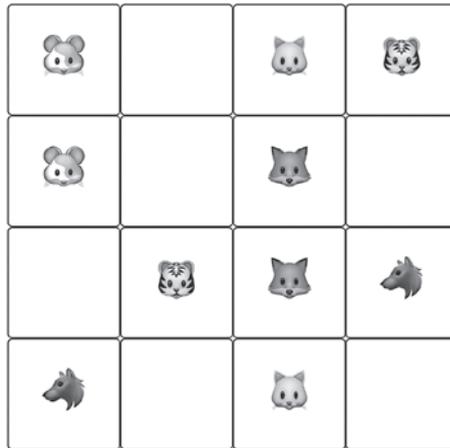
ВЫ НАХОДИТЕСЬ ЗДЕСЬ



СОЗДАНИЕ СЛУЧАЙНАЯ ОБРА- ПРОВЕРКА ДОБАВЛЕНИЕ
 ПРОЕКТА РАССТАНОВ- БОТКА ПОБЕДЫ ТАЙМЕРА
 КА ЖИВОТ-ЩЕЛЧКОВ ИГРОКА
 НЫХ

Добавление таймера

Наша игра станет более интересной, если игрок сможет попытаться побить свой рекорд. Добавим **таймер**, который срабатывает с фиксированным интервалом и многократно вызывает метод.



Matches found: 3

Time: 8.8s

↑
 Сделаем игру чуть более азартной!
 В нижней части окна выводится время, прошедшее с момента запуска игры. Показания таймера постоянно увеличиваются, а останавливается таймер только после нахождения последней пары.



Таймер срабатывает после истечения заданного интервала, а назначенный метод вызывается снова и снова. Мы используем таймер, который запускается вместе с запуском игры и перестает работать после нахождения последнего животного.

Добавление таймера в код игры

Добавьте



- Сначала найдите следующую строку в верхней части файла *Index.razor*: `@page "/"`:
Добавьте следующую строку под ней — это необходимо для использования таймера в коде C#:

```
@using System.Timers
```

- Далее необходимо обновить разметку HTML для вывода времени. Вставьте следующий фрагмент сразу же после первого блока, добавленного в упражнении:

```
</div>
<div class="row">
  <h2>Matches found: @matchesFound</h2>
</div>
<div class="row">
  <h2>Time: @timeDisplay</h2>
</div>
</div>
```

- В страницу нужно добавить таймер. Кроме того, понадобится переменная для хранения затраченного времени:

```
List<string> shuffledAnimals = new List<string>();
int matchesFound = 0;
Timer timer;
int tenthsOfSecondsElapsed = 0;
string timeDisplay;
```

- Таймеру необходимо сообщить, с какой частотой он должен срабатывать и какой метод должен вызываться. Это будет происходить в методе `OnInitialized`, который вызывается однократно после загрузки страницы:

```
protected override void OnInitialized()
{
  timer = new Timer(100);
  timer.Elapsed += Timer_Tick;

  SetUpGame();
}
```

- Обнулите затраченное время при инициализации игры:

```
private void SetUpGame()
{
  Random random = new Random();
  shuffledAnimals = animalEmoji
    .OrderBy(item => random.Next())
    .ToList();

  matchesFound = 0;
  tenthsOfSecondsElapsed = 0;
}
```

- ⑥ Таймер необходимо останавливать и запускать. Добавьте следующую строку в начало метода `ButtonClick`, чтобы таймер запускался при щелчке на первой кнопке:

```
if (lastAnimalFound == string.Empty)
{
    // Выбор первого элемента пары. Запомнить его.
    lastAnimalFound = animal;
    lastDescription = animalDescription;

    timer.Start();
}
```

Добавьте следующие две строки в метод `ButtonClick`, чтобы остановить таймер и вывести сообщение «Play Again?» после того, как игрок найдет последнее совпадение:

```
matchesFound++;
if (matchesFound == 8)
{
    timer.Stop();
    timeDisplay += " - Play Again?";

    SetUpGame();
}
```

- ⑦ Наконец, таймер должен знать, что делать при каждом срабатывании. Подобно тому как у кнопок есть обработчики событий `Click`, у таймера есть обработчик события `Tick`, который выполняется при каждом срабатывании таймера.

Добавьте следующий код в конец страницы, непосредственно перед закрывающей фигурной скобкой `}`:

```
private void Timer_Tick(Object source, ElapsedEventArgs e)
{
    InvokeAsync(() =>
    {
        tenthsOfSecondsElapsed++;
        timeDisplay = (tenthsOfSecondsElapsed / 10F)
            .ToString("0.0s");
        StateHasChanged();
    });
}
```

Таймер запускается, когда игрок щелкает на первом животном, и останавливается при обнаружении последнего совпадения. Таймер не изменяет игровой процесс, но делает его более интересным.

Очистка меню навигации

Ваша игра работает! Но вы заметили, что в приложении также присутствуют другие страницы? Попробуйте щелкнуть на ссылках «Counter» или «Fetch data» в навигационном меню слева. Когда вы создавали проект Blazor WebAssembly App, среда Visual Studio добавила дополнительные страницы. Их можно безопасно удалить.

Для начала раскройте папку **wwwroot** и отредактируйте файл *index.html*. Найдите строку, которая начинается с тега `<title>`, и измените ее, чтобы привести к виду `<title>Animal Matching Game</title>`.

Раскройте папку **Shared** в решении и сделайте двойной щелчок на файле *NavMenu.razor*. Найдите следующую строку:

```
<a class="navbar-brand" href="">BlazorMatchGame</a>
```

и замените ее следующей:

```
<a class="navbar-brand" href="">Animal Matching Game</a>
```

Затем удалите следующие строки:

```
<li class="nav-item px-3">  
  <NavLink class="nav-link" href="counter">  
    <span class="oi oi-plus" aria-hidden="true"></span> Counter  
  </NavLink>  
</li>  
<li class="nav-item px-3">  
  <NavLink class="nav-link" href="fetchdata">  
    <span class="oi oi-list-rich" aria-hidden="true"></span> Fetch data  
  </NavLink>  
</li>
```

Наконец, удерживая нажатой клавишу ⌘ (Command), выделите несколько файлов, щелкая на них в окне Solution: *Counter.razor* и *FetchData.razor* в папке Pages, *SurveyPrompt.razor* в папке Shared, а также всю папку **sample-data** в папке **wwwroot**. Когда файлы будут выделены, щелкните правой кнопкой мыши на любом из них и выберите команду меню **Delete** (⌘ ⌫), чтобы удалить их.

И теперь игра готова!



Да, это очень удобно — разбить игру на меньшие части, с которыми можно справиться поочередно.

Любой крупный проект всегда рекомендуется разбивать на меньшие части.

Один из самых полезных навыков программирования — умение взглянуть на большую и сложную задачу и разбить ее на ряд меньших, легко решаемых задач.

В самом начале большого проекта легко впасть в уныние: «Ого, да это бесконечная работа!» Но если вы сможете выделить меньшую подзадачу и начнете трудиться над ней, это станет отправной точкой для работы. А когда эта часть будет завершена, можно перейти к следующей меньшей части, потом к следующей и т. д. Во время построения каждой части вы будете все больше узнавать о проекте в целом.

Еще лучше, если...

Игра получилась вполне достойной! Но любую игру — да, собственно, практически любую программу — можно усовершенствовать. Несколько предложений, которые, как нам кажется, могли бы улучшить нашу игру:

- ★ Добавьте больше видов животных, чтобы в игре не использовались одни и те же изображения.
- ★ Храните лучшее время, чтобы игрок мог попытаться побить свой рекорд.
- ★ Реализуйте обратный отсчет времени, чтобы время игрока было ограничено.



Возьми в руку карандаш

А сможете ли вы предложить собственные улучшения для игры? Это очень полезное упражнение — подумайте несколько минут и запишите не менее трех улучшений для игры с поиском пар.

Мы абсолютно серьезно: не пожалейте времени и сделайте это. Когда вы делаете шаг назад и размышляете о только что завершенном проекте, это отлично помогает закрепить в мозге только что усвоенный материал.

КЛЮЧЕВЫЕ МОМЕНТЫ

- **Обработчик события** представляет собой метод, который вызывается вашим приложением при возникновении некоторого события (щелчка кнопкой мыши, перезагрузки страницы, срабатывания таймера и т. д.).
- В окне **Errors** в IDE выводятся описания ошибок, которые препятствуют построению вашего приложения.
- **Таймеры** многократно выполняют обработчик события Tick с заданным интервалом.
- **foreach** — разновидность циклов для перебора коллекций элементов.
- **for** — разновидность цикла, хорошо подходящая для отсчета фиксированного количества элементов.
- Когда в вашей программе происходит ошибка, соберите информацию и постарайтесь определить, что является ее причиной.
- **Воспроизводимые** ошибки проще устранять.
- **Отладчик IDE** позволяет приостановить приложение на конкретной команде, чтобы найти потенциальные проблемы.
- При установке **точки прерывания** отладчик приостанавливает выполнение на команде, в которой она была установлена.
- Visual Studio упрощает использование **систем управления версиями** для резервного копирования кода и контроля вносимых вами изменений.
- Вы можете сохранять свой код в **удаленном репозитории Git**. Мы создали репозиторий с исходным кодом всех проектов в книге на GitHub.

На всякий случай напомним: в книге мы часто называем Visual Studio просто «IDE».



Самое время сохранить ваш код в Git! Вы всегда сможете вернуться к своему проекту, если захотите повторно использовать какую-то часть его кода.

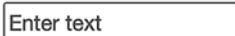
Это Blazor-версия проекта настольного приложения Windows из главы 2.

Из главы 2 Близкое знакомство с C#

В последней части главы 2 был приведен проект для экспериментов с разными типами элементов управления в Windows. Мы воспользуемся Blazor, чтобы построить аналогичный проект для экспериментов с элементами управления.

Элементы управления определяют механику ваших пользовательских интерфейсов

В предыдущей главе для построения игры использовались элементы управления Button. Однако существует много разных способов использования элементов, а решения, принятые вами при выборе элементов, могут очень сильно изменить ваше приложение. Звучит неожиданно? На самом деле все это очень похоже на принятие решений при проектировании игр. Если вы создаете настольную игру, которой нужен генератор случайных чисел, вы можете воспользоваться кубиками или картами. Если вы работаете над игрой-платформером, вы можете решить, что игровой персонаж должен уметь прыгать, делать двойной прыжок, делать прыжок от стены или летать (или выполнять разные действия в разное время). То же относится к приложениям: если вы разрабатываете приложение, в котором пользователь должен ввести число, вы можете выбрать для этой цели разные элементы, и от вашего выбора зависят впечатления пользователя при работе с приложением.



- * В текстовом поле пользователь может ввести любой текст по своему усмотрению. При этом в данном примере необходимо проверить, что пользователь вводит только числа, а не произвольный текст.



- * Ползунки предназначены исключительно для выбора чисел. В принципе телефонные номера тоже являются обычными числами, так что формально ползунок может использоваться для выбора телефона. Как вы думаете, хорошая ли это мысль?

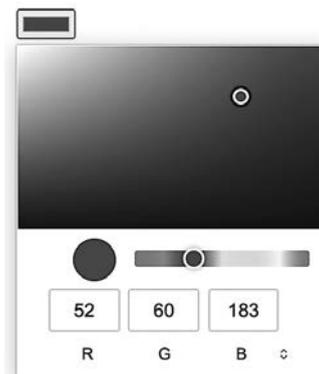


- * Переключатели ограничивают выбор пользователя несколькими фиксированными вариантами. Они часто выглядят как круги с точкой внутри, но при помощи стилизового оформления им можно придать вид обычных кнопок.

Элементы управления — компоненты пользовательского интерфейса (UI), строительные блоки для построения UI. От выбора элементов управления зависит механика вашего приложения.

Позаимствуем идею механики из видеоигр. Она поможет лучше понять возможные варианты и принять правильные решения в любых приложениях (не только в видеоиграх).

- * Селекторы — элементы управления, специально разработанные для выбора определенного типа значений из списка. Например, селекторы даты позволяют задать дату с выбором составляющих года, месяца и даты, а селекторы цветов предоставляют возможность выбрать цвет при помощи ползунка, представляющего спектр, или ввести числовой код.



Создание нового проекта Blazor WebAssembly App

Ранее в этом приложении мы создали проект *Blazor WebAssembly App* для игры с поиском пар. Теперь мы сделаем то же самое для этого проекта.

Ниже подробно описана последовательность действий для создания проекта Blazor WebAssembly App, изменения текста заголовка главной страницы и удаления лишних файлов, созданных Visual Studio. Мы не будем повторять эту инструкцию для всех остальных проектов в этом руководстве – вы сможете самостоятельно повторить ее для всех будущих проектов Blazor WebAssembly App.

1 Создайте новый проект Blazor WebAssembly App.

Запустите Visual Studio 2019 for Mac или выберите команду меню *File>>New Solution...* (⌘⌘N), чтобы **открыть окно New Project**. Щелкните на кнопке **New**, чтобы создать новый проект. Введите имя проекта **ExperimentWithControlsBlazor**.

2 Измените заголовок и меню навигации.

В конце проекта игры с поиском пар мы изменили текст заголовка и панели навигации. Сделаем то же самое в этом проекте. Раскройте папку **wwwroot** и отредактируйте файл *Index.html*. Найдите строку, которая начинается с тега `<title>`, и измените ее, чтобы привести ее к виду `<title>Experiment with Controls</title>`.

Раскройте папку **Shared** в решении и сделайте двойной щелчок на файле *NavMenu.razor*. Найдите следующую строку:

```
<a class="navbar-brand" href="">ExperimentWithControlsBlazor</a>
```

и замените ее следующей:

```
<a class="navbar-brand" href="">Experiment With Controls</a>
```

3 Удалите лишние команды меню навигации и соответствующие файлы.

Это то же самое, что было сделано в конце проекта с поиском пар. Сделайте двойной щелчок на файле *NavMenu.razor* и удалите следующие строки:

```
<li class="nav-item px-3">
  <NavLink class="nav-link" href="counter">
    <span class="oi oi-plus" aria-hidden="true"></span> Counter
  </NavLink>
</li>
<li class="nav-item px-3">
  <NavLink class="nav-link" href="fetchdata">
    <span class="oi oi-list-rich" aria-hidden="true"></span> Fetch data
  </NavLink>
</li>
```

Наконец, удерживая нажатой клавишу ⌘ (Command), выделите несколько файлов, щелкая на них в окне Solution: *Counter.razor* и *FetchData.razor* в папке Pages, *SurveyPrompt.razor* в папке Shared, а также всю папку **sample-data** в папке wwwroot. Когда файлы будут выделены, щелкните правой кнопкой мыши на любом из них и выберите команду меню **Delete** (⌘⌘), чтобы удалить их.

Создание страницы с ползунком

Многие программы зависят от ввода чисел пользователем. Одним из самых популярных элементов управления для ввода чисел является **ползунок** (slider). Создадим новую страницу Razor, использующую ползунок для обновления значения.

Отредактируйте страницу Razor — точно так же, как в игре с поиском пар из главы 1.

1 Замените содержимое страницы Index.razor.

Откройте файл `Index.razor` и **замените** все его содержимое следующей разметкой HTML:

```
@page "/"

<div class="container">
  <div class="row">
    <h1>Experiment with controls</h1>
  </div>
  <div class="row mt-2">
    <div class="col-sm-6">
      Pick a number:
    </div>
    <div class="col-sm-6">
      <input type="range"/>
    </div>
  </div>
  <div class="row mt-5">
    <h2>
      Here's the value:
    </h2>
  </div>
</div>
```

Добавление mt-2 в атрибут class добавляет надстройку над строкой верхний отступ.

Атрибут `class="row"` в этом теге приказывает странице сгенерировать весь контент, заключенный между открывающим тегом `<div class="row">` и закрывающим тегом `</div>`, в одну строку на странице.

Это тег **элемента ввода**. Его **атрибут type** определяет, какая разновидность элемента ввода будет отображаться на странице. Если присвоить **типу** значение `range`, отображается ползунок:

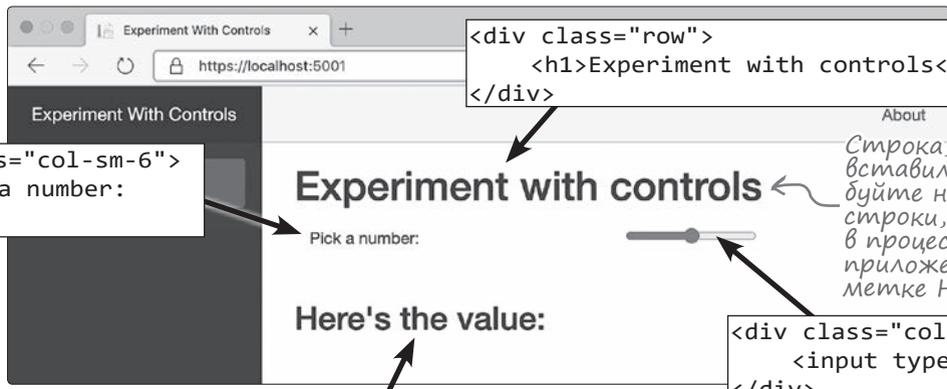
`<input type="range"/>`

Внешний вид элементов HTML иногда зависит от используемого браузера. Ползунок в Edge выглядит так:



2 Запустите приложение.

Запустите приложение, как это было сделано в главе 1. Сравните разметку HTML со страницей, отображаемой в браузере, — сопоставьте отдельные блоки `<div>` с тем, что отображается на странице.



The screenshot shows a browser window with the URL `https://localhost:5001`. The page content is "Experiment With Controls" with a heading "Experiment with controls", a "Pick a number:" label, a range slider, and a heading "Here's the value:". Overlaid HTML code boxes map the code to the UI:

- `<div class="row"> <h1>Experiment with controls</h1> </div>` maps to the main heading.
- `<div class="col-sm-6"> Pick a number: </div>` maps to the label.
- `<div class="col-sm-6"> <input type="range"/> </div>` maps to the slider.
- `<div class="row mt-5"> <h2>Here's the value:</h2> </div>` maps to the value heading.

Строка, которую мы вставили выше. Попробуйте найти другие строки, измененные в процессе работы над разметке HTML.

3 Добавьте код C# в страницу.

Вернитесь к файлу *Index.razor* и добавьте следующий код C# в конец файла:

```
@code
{
    private string DisplayValue = "";

    private void UpdateValue(ChangeEventArgs e)
    {
        DisplayValue = e.Value.ToString();
    }
}
```

Метод `UpdateValue` является обработчиком `ChangeEvent`. Он получает один параметр, который может использоваться вашим методом для выполнения каких-либо действий с измененными данными.

Обработчик события `Change` обновляет `DisplayValue` каждый раз, когда он вызывается со значением.

4 Свяжите ползунок с только что добавленным обработчиком события `Change`.

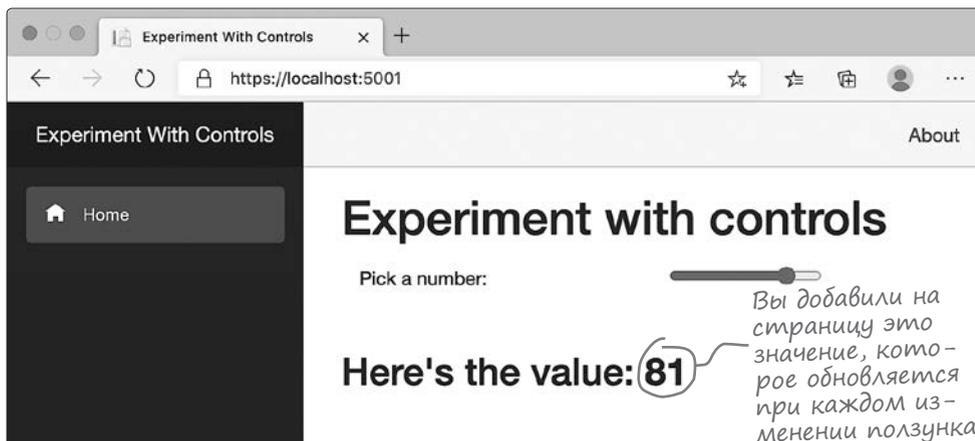
Добавьте атрибут `@onchange` к тегу `<input>`:

```
@page "/"

<div class="container">
    <div class="row">
        <h1>Experiment with controls</h1>
    </div>
    <div class="row mt-2">
        <div class="col-sm-6">
            Pick a number:
        </div>
        <div class="col-sm-6">
            <input type="range" @onchange="UpdateValue" />
        </div>
    </div>
    <div class="row mt-5">
        <h2>
            Here's the value: <strong>@DisplayValue</strong>
        </h2>
    </div>
</div>
```

Когда вы используете `@onchange` для связывания элемента управления с обработчиком события `Change`, ваша страница будет вызывать обработчик события при каждом изменении элемента управления.

При каждом изменении `DisplayValue` значение, отображаемое на странице, тоже будет изменяться.



Добавление текстового поля

Этот проект создавался для того, чтобы вы могли поэкспериментировать с разными видами элементов управления. Добавьте элемент текстового поля, чтобы пользователь мог ввести текст в приложении. Введенный им текст будет воспроизведен в нижней части страницы.

1 Добавьте текстовое поле в разметку HTML-страницы.

Добавьте тег `<input.../>`, практически идентичный тому, который был добавлен для ползунка. Существует только одно отличие: атрибуту `text` присваивается значение `"text"` вместо `"range"`. Разметка HTML выглядит так:

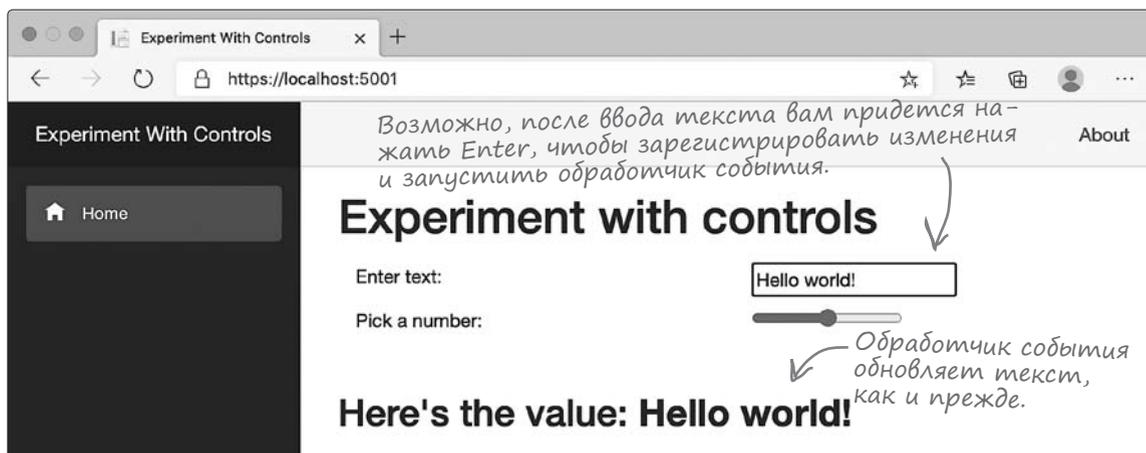
В страницу с верхним полем, размер которого составляет две единицы, добавляется еще одна строка.

```
<div class="container">
  <div class="row">
    <h1>Experiment with controls</h1>
  </div>
  <div class="row mt-2">
    <div class="col-sm-6">
      Enter text:
    </div>
    <div class="col-sm-6">
      <input type="text" placeholder="Enter text"
        @onchange="UpdateValue" />
    </div>
  </div>
  <div class="row mt-2">
    <div class="col-sm-6">
      Pick a number:
    </div>
```

Разметка текстового поля. Это элемент управления, который имеет тип `"text"` и использует такой же тег `@onchange`, как и ползунок. Также он содержит дополнительный атрибут для назначения текста заполнителя, так что до ввода текста элемент выглядит так:

Enter text

Снова запустите свое приложение — теперь в нем присутствует элемент текстового поля. Весь вводимый текст будет отображаться в нижней части страницы. Попробуйте изменить текст, сдвиньте ползунок, а затем снова измените текст. Значение в нижней части изменяется каждый раз, когда вы изменяете состояние элемента.



2 Добавьте метод — обработчик события, который принимает только числовые значения.

А если текстовое поле должно принимать только числовой ввод от пользователей? Добавьте следующий метод в код между фигурными скобками в нижней части страницы Razor:

```
private void UpdateNumericValue(ChangeEventArgs e)
{
    if (int.TryParse(e.Value.ToString(), out int result))
    {
        DisplayValue = e.Value.ToString();
    }
}
```

Вы узнаете все об `int.TryParse` позднее в этой книге — пока просто введите код точно в таком виде, в каком он приведен здесь.

Установите в методе точку прерывания, воспользуйтесь отладчиком и разберитесь в том, как она работает.

3 Измените текстовое поле для использования нового обработчика события.

Измените атрибут `@onchange` текстового поля, чтобы для него вызывался новый обработчик события:

```
<input type="text" placeholder="Enter text"
    @onchange="UpdateNumericValue" />
```

Теперь попробуйте ввести текст в текстовом поле — значение в нижней части страницы обновляется только в том случае, если введенный текст является целочисленным значением.



Упражнение

Элементы `Button` использовались в игре с поиском пар в главе 1. Ниже приведен фрагмент разметки HTML для добавления ряда кнопок на страницу — он очень похож на тот код, который мы использовали ранее. Ваша задача — **завершить этот код**, чтобы он добавлял шесть кнопок, а также **включить обработчик события в код С#**.

```
<div class="row mt-2">
    <div class="col-sm-6">Pick a number:</div>
    <div class="col-sm-6"><input type="range" @onchange="UpdateValue" /></div>
</div>
<div class="row mt-2">
    <div class="col-sm-6">Click a button:</div>
    <div class="col-sm-6 btn-group" role="group">
        {
            string valueToDisplay = $"Button #{buttonNumber}";
            <button type="button" class="btn btn-secondary"
                @onclick="() => ButtonClick(valueToDisplay)">
                @buttonNumber
            </button>
        }
    </div>
</div>
<div class="row mt-5">
    <h2>
        Here's the value: <strong>@DisplayValue</strong>
    </h2>
</div>
```

Замените этот прямоугольник строкой кода С#, которая размещает на странице шесть кнопок.

При щелчке на кнопке вызывается обработчик события с именем `ButtonClick`. Добавьте этот метод в нижнюю часть страницы — он состоит только из одной команды.



Упражнение
Решение

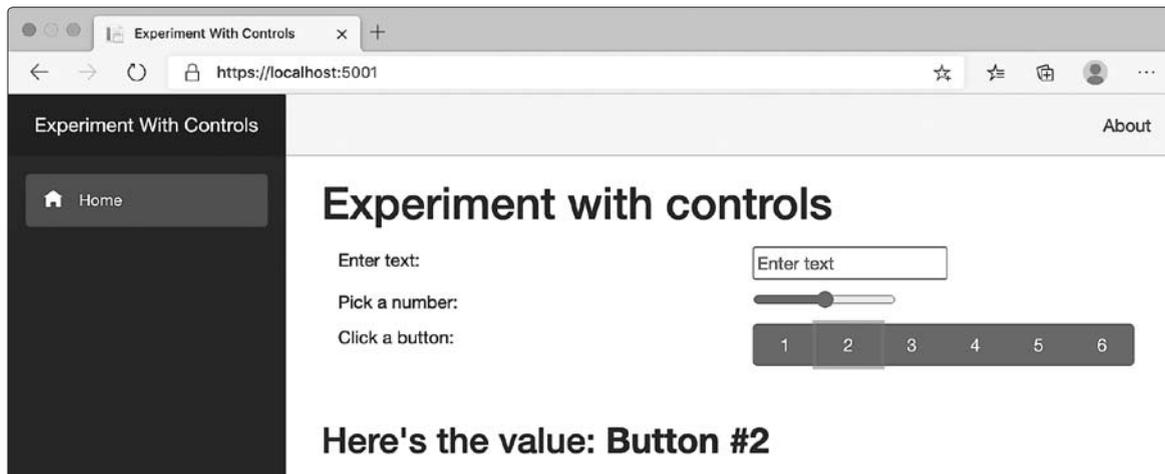
Ниже приведена строка кода, которая заставляет разметку Razor разместить шесть кнопок на странице. Это цикл for, и он работает точно так же, как цикл for, о котором вы узнали в главе 2.

```
<div class="row mt-2">
  <div class="col-sm-6">Pick a number:</div>
  <div class="col-sm-6"><input type="range" @onchange="UpdateValue" /></div>
</div>
<div class="row mt-2">
  <div class="col-sm-6">Click a button:</div>
  <div class="col-sm-6 btn-group" role="group">
    @for (var buttonNumber = 1; buttonNumber <= 6; buttonNumber++)
    {
      string valueToDisplay = $"Button #{buttonNumber}";
      <button type="button" class="btn btn-secondary"
        @onclick="() => ButtonClick(valueToDisplay)">
        @buttonNumber
      </button>
    }
  </div>
</div>
<div class="row mt-5">
  <h2>
    Here's the value: <strong>@DisplayValue</strong>
  </h2>
</div>
```

Цикл for, создающий кнопки, работает точно так же, как в игре с поиском пар, — код совпадает практически полностью. Кнопки объединены стилевым оформлением в группу (это делает btn-group) и окрашиваются в разные цвета (это делает btn-secondary).

Следующий обработчик события добавляется в код в нижней части страницы. Он присваивает DisplayValue значение, переданное в параметре, при щелчке на кнопке:

```
private void ButtonClick(string displayValue)
{
  DisplayValue = displayValue;
}
```



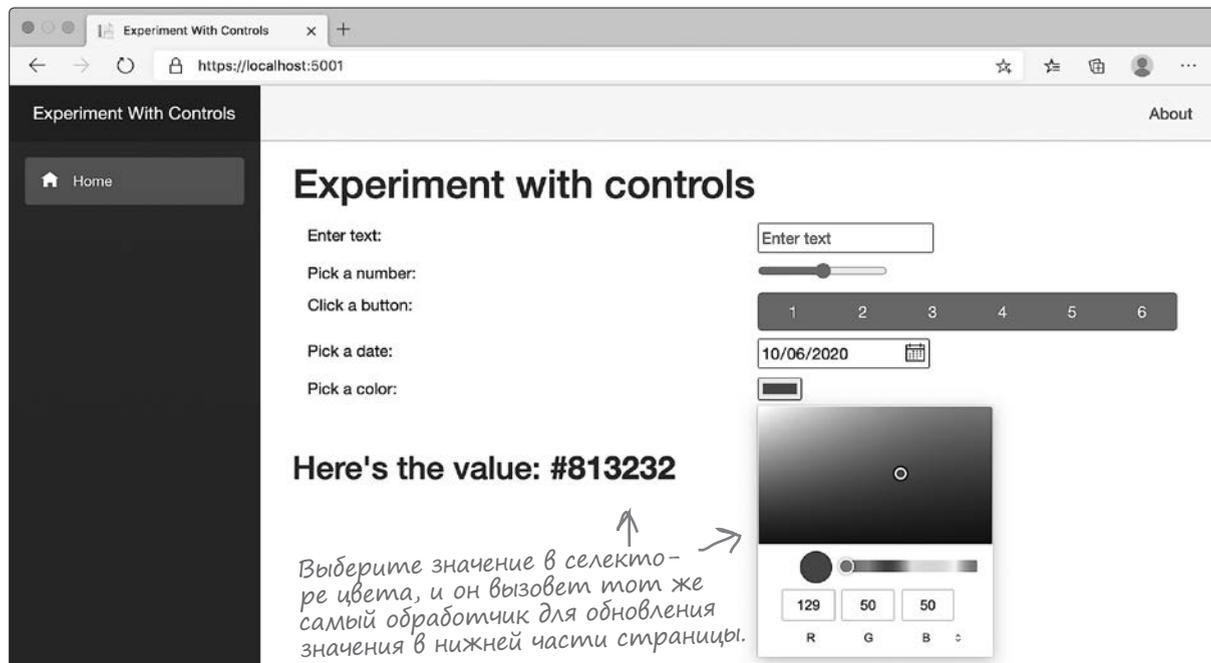
Добавление селекторов для выбора цвета и даты

Селекторы (pickers) — всего лишь еще одна разновидность элементов ввода данных. Селектор даты имеет тип "date", а селектор цвета имеет тип "color" — в остальном разметка HTML для этих элементов управления не отличается.

Измените свое приложение и добавьте в него селекторы даты и цвета. Ниже приведена разметка HTML — добавьте ее над тегом <div> для отображаемого значения:

```
<div class="row mt-2">
  <div class="col-sm-6">Pick a date:</div>
  <div class="col-sm-6">
    <input type="date" @onchange="UpdateValue" />
  </div>
</div>
<div class="row mt-2">
  <div class="col-sm-6">Pick a color:</div>
  <div class="col-sm-6">
    <input type="color" @onchange="UpdateValue" />
  </div>
</div>
<div class="row mt-5">
  <h2>Here's the value: @DisplayValue</h2>
</div>
</div>
```

Селекторы даты и цвета используют один обработчик события Change, так что вам вообще не придется изменять код для вывода цвета или даты, выбранных пользователем.



Проект подошел к концу — отличная работа! Вы можете вернуться к главе 2 в самом конце, где некто в кресле размышляет:
СТОЛЬКО РАЗНЫХ СПОСОБОВ ВЫБОРА ЧИСЕЛ! У МЕНЯ ПОЯВЛЯЕТСЯ НЕВЕРОЯТНО МНОГО ВАРИАНТОВ ПРИ РАЗРАБОТКЕ ПРИЛОЖЕНИЯ.

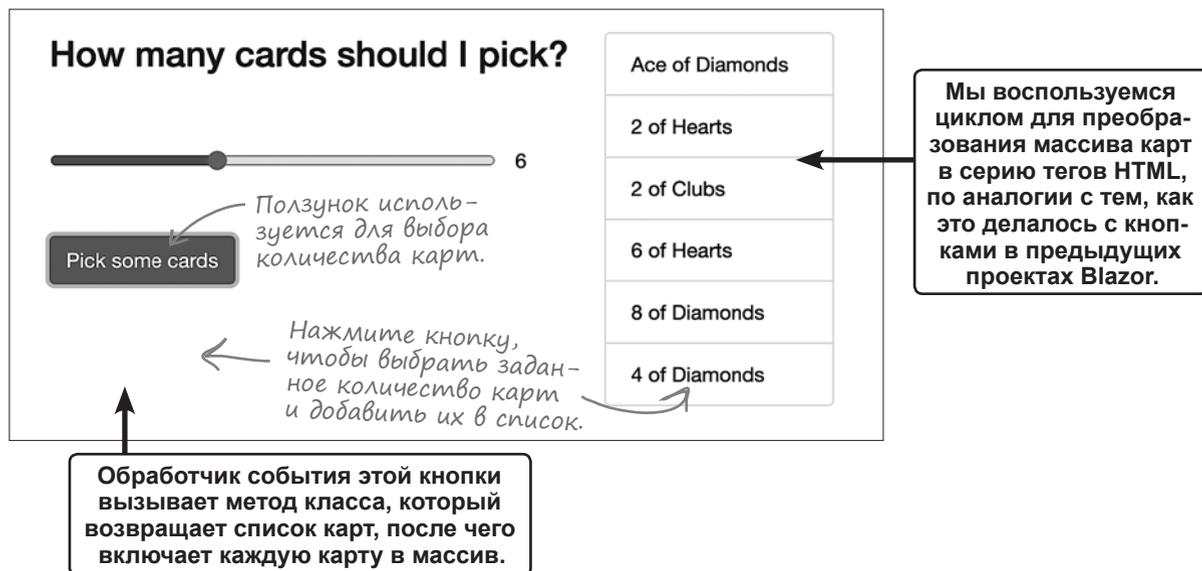
Из главы 3 Ориентируемся на объекты

Это Blazor-версия проекта настольного приложения Windows из главы 3.

В середине главы 3 была приведена Windows-версия проекта для выбора карт. Мы воспользуемся Blazor, чтобы построить веб-версию того же приложения.

Построение Blazor-версии приложения для выбора карт

В следующем проекте мы построим приложение Blazor с именем PickACardBlazor. В нем ползунок будет использоваться для выбора количества карт; карты будут выводиться в списке. Вот как это выглядит:



Повторное использование класса CardPicker в приложении Blazor

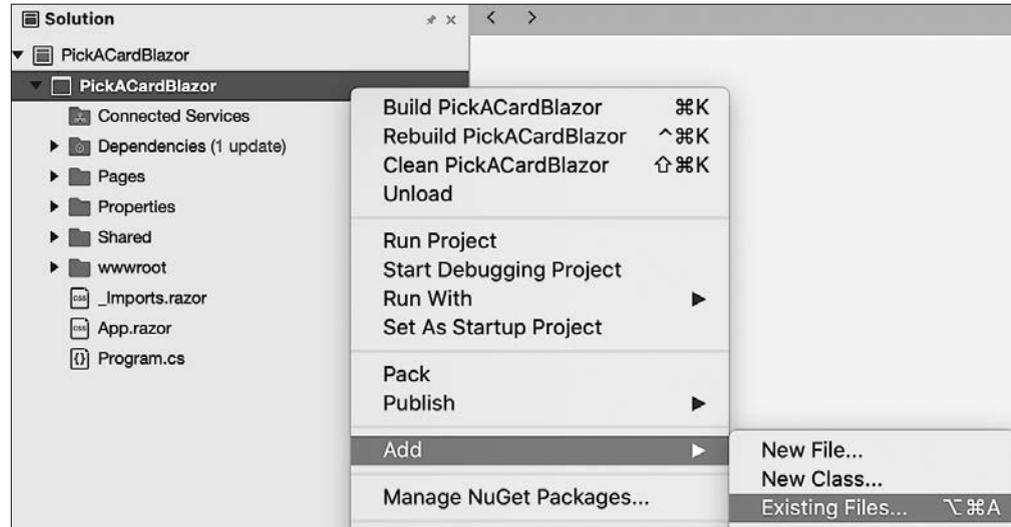
Если вы написали класс для одной программы, часто бывает возможно использовать то же поведение в другой программе. Одно из главных преимуществ классов как раз и заключается в том, что они упрощают **повторное использование** кода. Давайте создадим для приложения красивый новый интерфейс, но сохраним прежнее поведение за счет повторного использования класса CardPicker.

- ① **Создайте новый проект Blazor WebAssembly App с именем PickACardRazor**
Выполните точно такие же действия, какие были выполнены при создании игры с поиском пар в главе 1:
 - ★ Откройте Visual Studio и создайте новый проект.
 - ★ Выберите **Blazor WebAssembly App**, как и в предыдущих приложениях Blazor.
 - ★ Присвойте приложению имя **PickACardBlazor**. Visual Studio создаст проект.

2

Добавьте класс CardPicker, созданный для проекта консольного приложения.

Щелкните правой кнопкой мыши на имени проекта и выберите команду меню **Add>>Existing Files...**:



Перейдите в папку с консольным приложением и щелкните на файле *CardPicker.cs*, чтобы добавить его в проект. Visual Studio спросит, что вы хотите сделать: скопировать, переместить или создать ссылку на файл. Прикажите Visual Studio **скопировать файл**. Теперь в вашем проекте должна появиться копия файла *CardPicker.cs* из консольного приложения.

3

Измените пространство имен для класса CardPicker.

Сделайте двойной щелчок на файле *CardPicker.cs* в окне Solution. В нем остается пространство имен из консольного приложения. **Измените пространство имен** и приведите его в соответствие с именем проекта:

```
PickSomeCards(int numberofCards)
using System;
namespace PickACardBlazor
{
```

Вы изменяете пространство имен в файле *CardPicker.cs*, чтобы оно совпало с пространством, использованным при создании файлов в новом проекте. Это необходимо для использования класса *CardPicker* в коде нового проекта. Например, открыв файл *Program.cs*, вы увидите, что он принадлежит к тому же пространству имен.

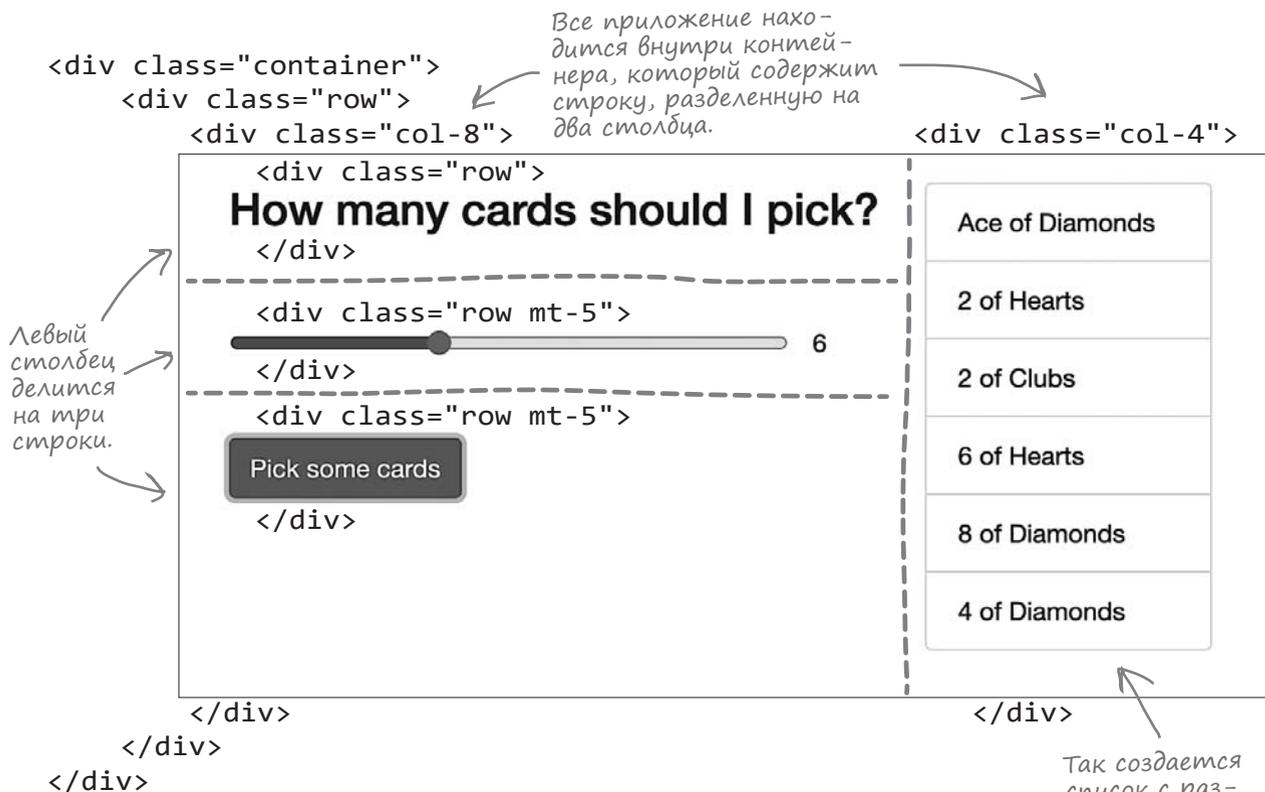
Теперь класс *CardPicker* должен принадлежать пространству имен *PickACardBlazor*:

```
namespace PickACardBlazor
{
    class CardPicker
    {
```

Поздравляем, вы повторно использовали класс *CardPicker*! Класс должен появиться в окне Solution, и вы можете использовать его в коде приложения Blazor.

Страница состоит из строк и столбцов

В приложениях Blazor из глав 1 и 2 для создания строк и столбцов использовалась разметка HTML, и новое приложение делает то же самое. На следующей диаграмме представлена структура вашего приложения:



Следующий код генерирует список карт в правом столбце. Он использует цикл `foreach` (вроде того, который использовался в игре с поиском пар) для создания списка из массива с именем `pickedCards`:

```

<div class="col-4">
  <ul class="list-group">
    @foreach (var card in pickedCards)
    {
      <li class="list-group-item">@card</li>
    }
  </ul>
</div>

```

```

<div class="col-4">
  <ul>
    <li>Ace of Diamonds</li>
    <li>2 of Hearts</li>
    <li>2 of Clubs</li>
    <li>6 of Hearts</li>
    <li>8 of Diamonds</li>
    <li>4 of Diamonds</li>
  </ul>
</div>

```

Список начинается с тега `<ul class="list-group">` и завершается тегом `` (сокращение `ul` означает «нумерованный список» (Unnumbered List)). Каждый элемент списка начинается с тега `<li class="list-group-item">` и завершается тегом ``.

Ползунок использует связывание данных для обновления переменной

Код в нижней части страницы начинается с переменной с именем numberOfCards:

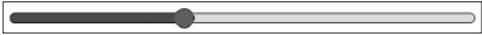
```
@code {
    int numberOfCards = 5;
```

Для обновления numberOfCards *можно* использовать обработчик события, но в Blazor существует более эффективный механизм **связывания данных**, позволяющий настроить элементы ввода для автоматического обновления данных C# и автоматической вставки значений из кода C# в страницу.

Ниже приведена разметка HTML для заголовка, ползунка и текста с текущим значением:

```
<div class="row">
    <h3>How many cards should I pick?</h3>
</div>
<div class="row mt-5">
    <input type="range" class="col-10 form-control-range"
        min="1" max="15" @bind="numberOfCards" />
    <div class="col-2">@numberOfCards</div>
</div>
```

How many cards should I pick?



6

Присмотритесь повнимательнее к атрибутам тега `input`. Атрибуты `min` и `max` ограничивают ввод значениями от 1 до 15. Атрибут `@bind` настраивает связывание данных, так что при каждом изменении состояния ползунка Blazor автоматически обновляет numberOfCards.

За тегом `input` следует фрагмент `<div class="col-2">@numberOfCards</div>` — эта разметка добавляет текст (`ml-2` означает расширение левого поля). Здесь также используется связывание данных, но в другом направлении: при каждом обновлении поля numberOfCards Blazor автоматически обновляет текст внутри тега `div`.



Упражнение

Мы предоставили почти все необходимое для добавления разметки HTML и кода в файл `Index.razor`. Удастся ли вам собрать все воедино, чтобы веб-приложение заработало?

Шаг 1: Завершение разметки HTML

Первые четыре строки файла `Index.razor` идентичны первым четырем строкам приложения `ExperimentWithControlBlazor` из главы 2. Следующие две строки HTML приведены в верхней части снимка экрана, где мы объясняем, как работают строки и столбцы. Единственная разметка, которую мы еще не приводили, относится к кнопке — вот она:

```
<button type="button" class="btn btn-primary"
    @onclick="UpdateCards">Pick some cards</button>
```

Когда вы введете этот код, IDE может добавить разрыв строки после открывающего и перед закрывающим тегом.

Шаг 2: Завершение кода

Мы предоставили вам начало раздела `@code` в нижней части страницы, в нем присутствует поле `int` с именем numberOfCards.

- Добавьте поле со строковым массивом: `pickedCards: string[] pickedCards = new string[0];`
- Добавьте обработчик события `UpdateCards`, вызываемый нажатием кнопки. Он вызывает `CardPicker.PickSomeCards` и присваивает результат полю `pickedCards`.



Упражнение
Решение

Ниже приведен полный код файла *Index.razor*. Вы также можете повторить действия из проекта *ExperimentWithControlsBlazor*, чтобы удалить лишние файлы и обновить меню навигации.

@page "/"

```
<div class="container">
  <div class="row">
    <div class="col-8">
      <div class="row">
        <h3>How many cards should I pick?</h3>
      </div>
      <div class="row mt-5">
        <input type="range" class="col-10 form-control-range"
          min="1" max="15" @bind="numberOfCards" />
        <div class="col-2">@numberOfCards</div>
      </div>
      <div class="row mt-5">
        <button type="button" class="btn btn-primary"
          @onclick="UpdateCards">
          Pick some cards
        </button>
      </div>
    </div>
    <div class="col-4">
      <ul class="list-group">
        @foreach (var card in pickedCards)
        {
          <li class="list-group-item">@card</li>
        }
      </ul>
    </div>
  </div>
</div>
```

Ползунок и следующий за ним текст размещаются в столбцах своей маленькой строки.

Когда вы щелкаете на кнопке, ее обработчик события *Click* с именем *UpdateCards* присваивает массиву *pickedCards* новый набор случайных карт. Сразу же после изменения в происходящее вмешивается механизм связывания данных Blazor, и цикл *foreach* автоматически выполняется заново.

numberOfCards и pickedCards — особые разновидности переменных, называемые полями. Вы узнаете о них в главе 3.

Обработчик события *Click*, связанный с кнопкой, вызывает метод *PickSomeCards* в классе *CardPicker*, написанном ранее в этой главе.

```
@code {
  int numberOfCards = 5;

  string[] pickedCards = new string[0];

  void UpdateCards()
  {
    pickedCards = CardPicker.PickSomeCards(numberOfCards);
  }
}
```

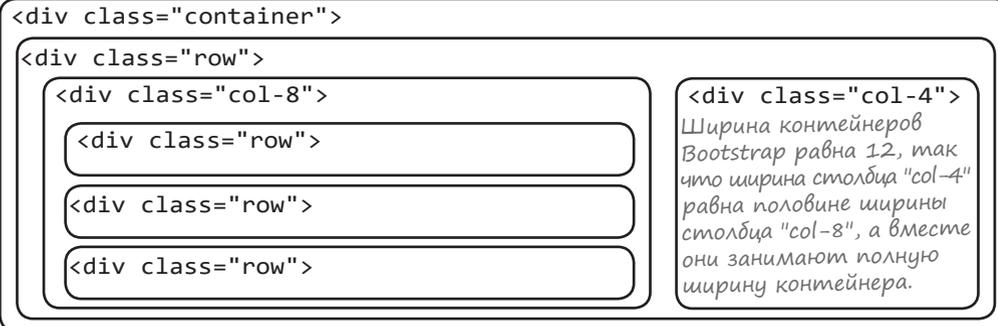


Ваши веб-приложения Blazor используют Bootstrap для формирования макета.

Приложение выглядит неплохо! Отчасти это связано с тем, что оно использует **Bootstrap** – бесплатный фреймворк с открытым кодом для создания веб-страниц, которые автоматически адаптируются к изменению размеров экрана и хорошо работают на мобильных устройствах.

За сценой

Макет со строками и столбцами, управляющий макетом приложения, происходит непосредственно из Bootstrap. Ваше приложение использует атрибут `class` (не имеющий никакого отношения к классам C#) для того, чтобы получить доступ к средствам формирования макета Bootstrap.



Поэкспериментируйте: попробуйте заменить `col-8` и `col-4` на `col-6`, чтобы столбцы имели одинаковые размеры. Что произойдет, если выбрать числа, которые в сумме не дают 12?

Bootstrap также помогает со стиливым оформлением элементов. Попробуйте удалить атрибут `class` из тегов `button`, `input`, `ul` или `li` и снова запустите приложение. Оно продолжает работать точно так же, но выглядит иначе – элементы потеряли часть своего оформления. Попробуйте удалить все атрибуты `class` – строки и столбцы исчезают, но приложение все еще работает.

За дополнительной информацией о Bootstrap обращайтесь по адресу <https://getbootstrap.com>.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Классы содержат методы, а методы состоят из команд, выполняющих действия. В хорошо спроектированных классах используются содержательные имена методов.
- Некоторые методы имеют **возвращаемый тип**, который задается в объявлении метода. Метод с объявлением, начинающимся с ключевого слова `int`, возвращает значение `int`. Пример команды, возвращающей значение `int`: `return 37;`
- Если метод имеет возвращаемый тип, он **должен** содержать команду `return` со значением, соответствующим возвращаемому типу. Таким образом, если в объявлении метода указан строковый возвращаемый тип, этот метод должен содержать команду `return`, которая возвращает строку.
- Как только в методе будет выполнена команда `return`, программа возвращается к команде, из которой был вызван метод.
- Не все методы имеют возвращаемый тип. Метод с объявлением, начинающимся с `public void`, ничего не возвращает. При этом для выхода из метода `void` может использоваться команда `return`: `if (finishedEarly) { return; }`
- Разработчики часто хотят **повторно использовать** один код в разных программах. Классы помогают расширить возможности повторного использования кода.

Проект подошел к концу — отличная работа! Вы можете вернуться к главе 3 и продолжить с раздела с заголовком: «Прототипы Анны выглядят замечательно...»

Из главы 3 Типы и ссылки

В конце главы 4 был приведен проект для Windows. Мы построим Blazor-версию этого проекта.

Добро пожаловать в заголовку эконо-класса «У неторопливого Джо»!

«У неторопливого Джо» подают сэндвичи. У него есть мясо, гора хлеба и больше приправ, чем вы можете себе представить. Но вот меню у него нет! Сможете ли вы построить программу, которая генерирует *случайное* новое меню на каждый день? Да, вы определенно можете это сделать... при помощи **нового приложения Blazor WebAssembly App**, массивов и пары полезных приемов.

Сделайте это!

1 Добавьте в проект новый класс MenuItem с набором полей.

Взгляните на диаграмму класса. Он содержит шесть полей: экземпляр Random, три массива для хранения различных частей сэндвича и строковые поля с описаниями и ценой. Поля-массивы используют **инициализаторы коллекций**: вы определяете элементы массива, заключая их в фигурные скобки.

```
class MenuItem
{
    public Random Randomizer = new Random();
    public string[] Proteins = { "Roast beef", "Salami", "Turkey",
        "Ham", "Pastrami", "Tofu" };
    public string[] Condiments = { "yellow mustard", "brown mustard",
        "honey mustard", "mayo", "relish", "french dressing" };
    public string[] Breads = { "rye", "white", "wheat", "pumpernickel", "a roll" };

    public string Description = "";
    public string Price;
}
```

MenuItem
Randomizer
Proteins
Condiments
Breads
Description
Price
Generate

2 Добавьте метод Generate в класс MenuItem.

Этот метод использует уже хорошо знакомый вам метод Random.Next для выбора случайных элементов из массивов в полях Proteins, Condiments и Breads и их конкатенации в строку:

```
public void Generate()
{
    string randomProtein = Proteins[Randomizer.Next(Proteins.Length)];
    string randomCondiment = Condiments[Randomizer.Next(Condiments.Length)];
    string randomBread = Breads[Randomizer.Next(Breads.Length)];
    Description = randomProtein + " with " + randomCondiment + " on " + randomBread;

    decimal bucks = Randomizer.Next(2, 5);
    decimal cents = Randomizer.Next(1, 98);
    decimal price = bucks + (cents * .01M);
    Price = price.ToString("c");
}
```

Метод Generate вычисляет случайную цену в диапазоне от 2.01 до 5.97 преобразованием двух случайных int в decimal. Внимательно присмотритесь к последней строке — она возвращает к последней строке — она возвращает price.ToString("c"). Параметр метода ToString определяет формат. В данном случае формат "c" приказывает ToString отформатировать значение с локальной денежной единицей: в США будет вводиться знак \$, в Великобритании — £, в Европе — €, и т. д.

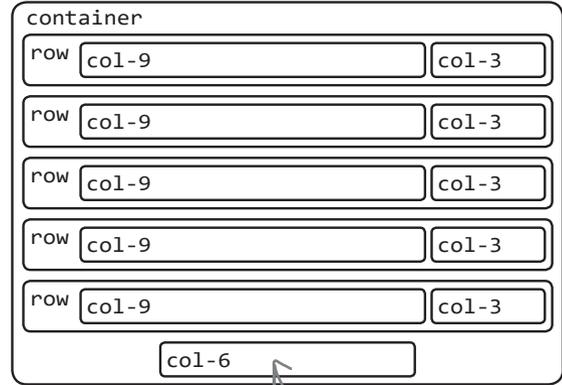
3 Добавьте макет страницы в файл `Index.razor`.

Меню страницы состоит из серии строк Bootstrap, по одной для каждого пункта меню. Каждая строка содержит два столбца: `col-9` с описанием элемента меню и `col-3` с ценой. Внизу добавлена последняя строка со столбцом `col-6`, выровненным по центру.

```
@page "/"

<div class="container">
  @foreach (MenuItem menuItem in menuItems)
  {
    <div class="row">
      <div class="col-9">
        @menuItem.Description
      </div>
      <div class="col-3">
        @menuItem.Price
      </div>
    </div>
  }
  <div class="row justify-content-center">
    <div class="col-6">
      <strong>Add guacamole for @guacamolePrice</strong>
    </div>
  </div>
</div>

@code {
  MenuItem[] menuItems = new MenuItem[5];
  string guacamolePrice;
}
```



Нижняя строка содержит один столбец, ширина которого составляет половину от ширины контейнера. Этой строке назначен класс `justify-content-center`, который обеспечивает выравнивание нижней строки по центру страницы.



Упражнение

Добавьте раздел `@code` в конец файла `Index.razor`. Он добавляет пять объектов `MenuItem` в поле `menuItems`, а также инициализирует поле `guacamolePrice`.

Шаг 1: Добавьте метод `OnInitialized`

Вы уже использовали метод `OnInitialized` для случайной перестановки животных в игре с поиском пар. Добавьте следующую строку кода:

```
protected override void OnInitialized()
```

Шаг 2: Замените тело метода `OnInitialized` кодом создания объектов `MenuItem`

IDE автоматически заполняет тело метода (`base.OnInitialized()`), как это было сделано при создании игры с поиском пар. Удалите эту команду. Замените ее кодом, инициализирующим поля `menuItems` и `guacamolePrice`.

- Напишите цикл `for`, который добавляет пять объектов `MenuItem` в поле с массивом `menuItems` и вызывает их методы `Generate`.
- Присвойте полю `Breads` новый массив строк:

```
new string[] { "plain bagel", "onion bagel",
              "pumpernickel bagel", "everything bagel" }
```
- Создайте новый экземпляр `MenuItem`, вызовите его метод `Generate` и используйте поле `Price` для задания `guacamolePrice`.



Упражнение Решение

Ниже приведен полный код файла `Index.razor`. Все до `string guacamolePrice`; соответствует коду, который я вам предоставил, а вам было предложено заполнить остаток блока `@code`.

```
@page "/"
```

```
<div class="container">
  @foreach (MenuItem menuItem in menuItems)
  {
    <div class="row">
      <div class="col-9">
        @menuItem.Description
      </div>
      <div class="col-3">
        @menuItem.Price
      </div>
    </div>
  }
  <div class="row justify-content-center">
    <div class="col-6">
      <strong>Add guacamole for @guacamolePrice</strong>
    </div>
  </div>
</div>
```

Salami with brown mustard on pumpernickel	\$4.89
Tofu with relish on pumpernickel	\$3.22
Turkey with french dressing on a roll	\$4.13
Tofu with yellow mustard on onion bagel	\$2.08
Pastrami with mayo on onion bagel	\$4.30
Add guacamole for \$2.42	

```
@code {
  MenuItem[] menuItems = new MenuItem[5];
  string guacamolePrice;

  protected override void OnInitialized()
  {
    for (int i = 0; i < 5; i++)
    {
      menuItems[i] = new MenuItem();
      if (i >= 3)
      {
        menuItems[i].Breads = new string[] {
          "plain bagel",
          "onion bagel",
          "pumpernickel bagel",
          "everything bagel"
        };
      }
      menuItems[i].Generate();
    }

    MenuItem guacamoleMenuItem = new MenuItem();
    guacamoleMenuItem.Generate();
    guacamolePrice = guacamoleMenuItem.Price;
  }
}
```

Страница использует эти два поля для связывания данных. Поле `menuItems` используется для генерирования пяти строк, тогда как поле `guacamolePrice` содержит цену для пункта меню в нижней части таблицы.

Значение присваивается напрямую элементу массива. Также можно было создать отдельную переменную для создания нового экземпляра `MenuItem` и присвоить ее элементу массива в конце цикла `for`.

Не забудьте вызвать метод `Generate`; в противном случае поля `MenuItem` будут пустыми, а страница останется в основном пустой.



Под увеличительным стеклом

Метод `Randomizer.Next(7)` получает случайное число, меньшее 7. `Breads.Length` возвращает количество элементов в массиве `Breads`. Следовательно, `Randomizer.Next(Breads.Length)` возвращает случайное число, большее или равное нулю, но меньшее количества элементов в массиве `Breads`.

`Breads [Randomizer . Next (Breads . Length)]`

`Breads` — строковый массив с пятью элементами, пронумерованными от 0 до 4. Таким образом, `Breads[0]` содержит "rye", а `Breads[3]` содержит "a roll".

Я обедаю только «У Неторопливого Джо»!



Если ваш компьютер достаточно производительен, возможно, ваша программа не столкнется с этой проблемой. Но если запустить ее на более медленном компьютере, вы наверняка увидите ее.

4

Запустите программу и просмотрите сгенерированное меню.

Э-э-э... Что-то не так. Все цены в новом меню одинаковы, и пункты меню выглядят странно: первые три позиции совпадают, потом следующие две, и начинаются они одинаково. Что происходит?

Оказывается, класс `.NET Random` на самом деле является генератором псевдослучайных чисел; это означает, что он по математической формуле генерирует последовательность чисел, удовлетворяющих некоторым статистическим критериям случайности. Такие числа достаточно хороши для любых приложений, которые строим мы с вами (только не используйте их в системах безопасности, зависящих от действительно случайных чисел!). Вот почему метод называется `Next` — он выдает следующее число в последовательности. Формула начинается со специального значения, которое называется «затравкой», — это значение используется для вычисления следующего значения в серии. Когда вы создаете новый экземпляр `Random`, системные часы используются для получения «затравки» формулы, но вы можете задать собственное значение. Попробуйте ввести в интерактивном окне `C#` вызов `new Random(12345).Next()`; несколько раз. Тем самым вы приказываете создать новый экземпляр `Random` с одним начальным значением (12345), поэтому метод `Next` будет каждый раз давать одно и то же «случайное» число. Когда вы видите, что несколько разных экземпляров `Random` дают одно и то же значение, это объясняется тем, что они инициализировались практически в один момент времени, показания системных часов не изменились, поэтому все они использовали одно начальное значение. Как решить проблему? Используйте один экземпляр `Random` и объявите поле `Randomizer` статическим, чтобы все команды `MenuItem` совместно использовали один экземпляр `Random`:

```
public static Random Randomizer = new Random();
```

Снова запустите программу — на этот раз меню станет случайным.

Проект подошел к концу! Вы можете вернуться к разделу «Ключевые моменты» в самом конце главы 4.

Salami with brown mustard on pumpernickel	\$4.89
Salami with brown mustard on pumpernickel	\$4.89
Salami with brown mustard on pumpernickel	\$4.89
Salami with brown mustard on everything bagel	\$4.89
Salami with brown mustard on everything bagel	\$4.89
Add guacamole for \$2.42	

Почему пункты меню и цены остаются одинаковыми?

Salami with brown mustard on pumpernickel	\$2.54
Roast beef with mayo on a roll	\$2.59
Salami with honey mustard on a roll	\$3.81
Salami with french dressing on plain bagel	\$4.52
Turkey with yellow mustard on everything bagel	\$2.67
Add guacamole for \$2.76	



За описаниями проектов для глав 5 и 6 обращайтесь на страницу GitHub

Возможно, вы заметили, что проекты глав 5 и 6 получились более длинными и сложными, чем проекты предшествующих глав. Мы хотим предоставить наилучшее возможное качество обучения, но для этого потребуются больше страниц, чем мы могли бы выделить в этом приложении! По этой причине проекты глав 5 и 6 «Visual Studio для пользователей Mac» доступны для загрузки в формате PDF на странице GitHub: <https://github.com/head-first-csharp/fourth-edition>.



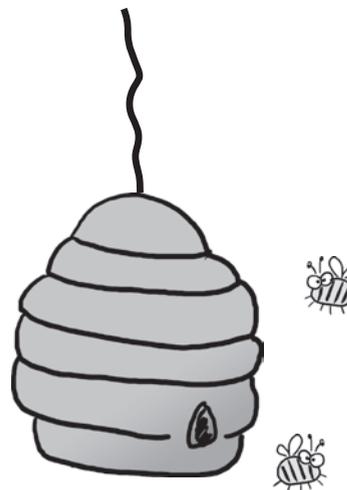
Превосходно! Но я тут задумался... Как вы считаете, можно ли построить более наглядное приложение для вычисления повреждений?

Да! Мы можем построить приложение Blazor, использующее тот же класс.

В главе 5 мы создали для Оуэна консольное приложение для вычисления повреждений в ролевой игре. Теперь мы можем повторно использовать тот же класс для проекта в веб-приложении Blazor.

Система управления ульем

Проект из главы 6 был серьезным бизнес-приложением. *Пчелиной матке нужна ваша помощь!* Улей вышел из-под контроля, и ей нужна программа, которая поможет управлять предприятием по производству меда. В улье полно рабочих пчел, масса невыполненной работы, но каким-то образом руководительница забыла, какая пчела чем занимается, и хватит ли у нее пчелиных сил для выполнения всех незавершенных задач. Вам предложено построить систему управления ульем, которая будет заниматься отслеживанием рабочих и порученных им заданий.



Все остальные проекты настольных приложений Windows и проекты Blazor, следующие после главы 6, доступны для загрузки в формате PDF. Поищите их на странице GitHub!

Ката программирования для опытных и/или нетерпеливых

Ката программирования



У вас уже есть опыт работы на другом языке программирования? Возможно, в таком случае **ката программирования** станут эффективной, результативной и приятной альтернативой для чтения книги «от корки до корки».

«Ката» — японское слово, которое означает «форма» или «путь». Оно используется во многих боевых искусствах для описания метода тренировки, основанного на многократной отработке серий движений или приемов. Многие разработчики применяют эту концепцию для отработки своих навыков программирования посредством написания конкретных программ, чаще более одного раза. *Многие опытные разработчики, желающие освоить C#, применяют ката программирования как альтернативный путь изучения этих глав*. Этот метод работает так:

- ◆ Начиная новую главу, **пролистайте или бегло просмотрите ее**, пока не найдете первый элемент ката программирования (см. инструкции ниже). Прочитайте ближайшую врезку «Ключевые моменты», чтобы увидеть, какой материал рассматривался в этом разделе.
- ◆ В описаниях ката программирования приводятся **инструкции по выполнению конкретного упражнения** — обычно проект из области программирования. Попробуйте выполнить проект, возвращаясь к предшествующим разделам (особенно врезкам «Ключевые моменты») за дополнительной информацией.
- ◆ Если вы **зайдете в тупик** при выполнении упражнения, значит, вы столкнулись с той областью, в которой C# сильно отличается от уже известных вам языков. Вернитесь к предыдущему элементу ката программирования (или к началу главы для первого) и начните читать книгу последовательно до точки, в которой вы застряли.
- ◆ В **лабораторных работах Unity** вы тренируетесь в написании кода C# на примере разработки 3D-игр на базе Unity. Эти лабораторные работы необязательны для метода ката программирования, но мы настоятельно рекомендуем выполнять их. Кроме того, это очень увлекательный и по-настоящему интересный способ отточить новообретенные навыки C#.

Ката — в боевых искусствах или в программировании — предназначены для многократного повторения. Таким образом, если вы действительно хотите закрепить C# у себя в голове, после завершения главы вернитесь к ней, найдите разделы с ката программирования и выполните их снова. **В главе 1 ката программирования отсутствуют.** Даже опытному разработчику стоит полностью прочитать эту главу и выполнить все проекты и упражнения, даже те, которые выполняются «на бумаге». В главе представлены некоторые фундаментальные идеи, которые будут развиваться в остальных главах. Кроме того, если у вас есть опыт работы на другом языке программирования и вы собираетесь выбрать путь ката программирования, **посмотрите видеоролик Патрисии Аас (Patricia Aas)**, посвященный изучению C# как второго (или пятнадцатого) языка: https://bit.ly/cs_second_language.

Это необходимо для пути ката программирования.

↑
Вы обладаете серьезным опытом работы на другом языке программирования и хотите изучить C# для работы или собственного удовольствия? Поищите разделы ката программирования, начиная с главы 2: они формируют альтернативный путь изучения, идеально подходящий для более опытных (или нетерпеливых!) разработчиков. Ознакомьтесь с этим роликом, чтобы понять, подойдет ли этот путь лично вам.

Глава 2: Знакомство с C#



В главе 2 наши читатели знакомятся с основными концепциями C#: распределением кода по пространствам имен, классами, методами и командами, а также некоторыми базовыми синтаксическими конструкциями. Глава завершается проектом построения простого пользовательского интерфейса для ввода: для читателей, работающих на платформе Windows, это настольное приложение WPF; для читателей macOS — веб-приложение Blazor (см. приложение «*Visual Studio для пользователей Mac*»). Такие проекты включаются в большинство глав для того, чтобы продемонстрировать читателю другие подходы к решению аналогичных задач, что может сильно пригодиться при изучении нового языка.

Для начала просмотрите всю главу и прочитайте все разделы «Ключевые моменты». Просмотрите все примеры кода. Что-то показалось знакомым? В таком случае вы готовы приступить к некоторым ката.

Ката №1: Найдите элемент `Console.WriteLine` в разделе с заголовком «Генерирование нового метода для работы с переменными». Он станет вашей отправной точкой. Добавьте код в этот раздел и в следующий раздел «Добавление кода с использованием операторов» в консольное приложение, созданное в главе 1. Воспользуйтесь отладчиком Visual Studio для пошагового выполнения кода — в следующем разделе показано, как это делается.

Ката №2: В следующих разделах приводятся примеры команд `if` и циклов, которые вы должны добавить в свое приложение и отладить.

Все выглядит логично? В таком случае вы готовы взяться за проект WPF в конце главы или за проект Blazor в приложении «*Visual Studio для пользователей Mac*». Если вы уверенно обращаетесь с Visual Studio и можете сориентироваться в IDE для создания, выполнения или отладки консольного приложения .NET Core, значит, вы готовы для...

Глава 3: Объекты

В этой главе представлены основные концепции классов, объектов и экземпляров. В первой половине мы работаем со статическими методами или полями (т. е. принадлежащими самому типу, а не его конкретному экземпляру); во второй половине будут создаваться экземпляры классов. Если вы выполняете — и понимаете — эти ката, можете спокойно переходить к следующей главе.

Ката № 1: Первый проект этой главы — простая программа, генерирующая случайные игральные карты. Найдите первый элемент `Console.WriteLine` (рядом с заголовком «Создание консольного приложения PickRandomCards»). Это ваша отправная точка. Создайте класс и используйте его в простой программе.

Ката № 2: Проработайте материал следующих разделов и завершите класс `CardPicker`. Этот класс будет повторно использован в настольном приложении WPF или в веб-приложении Blazor.

Ката № 3: Пролитайте или бегло просмотрите главу до раздела «Возьми в руку карандаш», в котором создаются объекты `Clown`. Введите код, добавьте комментарии для ответов, а затем выполните в пошаговом режиме.

Ката № 4: Ближе к концу главы найдите заголовок «Классы, парни и деньги». Сразу же после него приведен класс с именем `Guu`, за которым следует упражнение. Выполните его, чтобы построить простое приложение.

Ката № 5: Выполните следующее упражнение, в котором класс `Guu` повторно используется в простой игре со ставками. Предложите как минимум один способ улучшения игры: предоставьте игроку возможность выбирать разные шансы и ставки, включите поддержку нескольких игроков и т. д.

В процессе работы над программами обращайтесь особое внимание на места, в которых вы могли бы задать себе следующие вопросы (ответы на них приводятся в последующих главах, но если вы уделите им внимание сейчас, это поможет вам быстрее освоить C#):

- ♦ Нет ли в C# какой-либо разновидности команды `switch`?
- ♦ Разве некоторые разработчики не считают наличие нескольких команд `return` в методе или функции плохой практикой?
- ♦ Почему мы используем `return` для выхода из цикла? Нет ли в C# более удобного способа выхода из цикла без возврата управления из метода?

Глава 4: Типы и ссылки



Эта глава посвящена типам и ссылкам в C#. Прочитайте несколько первых разделов и убедитесь в том, что достаточно знаете о разных типах: скажем, у чисел с плавающей точкой есть некоторые странности, о которых необходимо знать. Просмотрите диаграммы и проверьте свое понимание того, как работают ссылки, потому что эти особенности их поведения будут использоваться в книге. Все усвоили? Хорошо, переходите к трем ката. Если вы справитесь с ними без особых затруднений, можно смело переходить к следующей главе.

Ката № 1: Первый проект этой главы — программа для расчета характеристик персонажа в ролевой игре. Начните с заголовка «Поможем Оуэну в экспериментах с характеристиками». Вы должны найти и исправить как синтаксическую ошибку, так и ошибку в логике, не заглядывая в ответ.

Ката № 2: Займитесь проектом в упражнении, которое начинается со слов: «Создайте программу с классом Elephant». Когда упражнение будет выполнено, просмотрите весь материал вплоть до заголовка «Объекты используют ссылки для взаимодействия друг с другом».

Ката № 3: Выполните проект под заголовком «Добро пожаловать в забегаловку эконо-класса „У неторопливого Джо“!». Несколько вопросов, которые стоит принять во внимание (ответы на них даются позднее в книге):

- ♦ Различаются ли в C# типы значений (такие, как double и int) и ссылочные типы (такие, как Elephant, MenuItem и Random)?
- ♦ Можно ли как-то управлять временем уничтожения сборщиком мусора тех объектов, на которые не осталось ни одной ссылки?
- ♦ Как в IDE отслеживать и различать конкретные экземпляры?

Размышления над этими вопросами помогут вам быстрее освоить C#.

Глава 5: Инкапсуляция

В этой главе речь пойдет об инкапсуляции — в C# под этим термином подразумевается ограничение доступа к некоторым компонентам классов, чтобы предотвратить их некорректное использование внешними объектами. Если вы справитесь с последним ката (где требуется исправить ошибку, внешнюю в первом), не подсматривая в решение, можете смело переходить к следующей главе.

Ката № 1: Первый проект этой главы — программа для расчета повреждений в ролевой игре. Он начинается от начала главы и следует до врезки «Великий сыщик». Убедитесь в том, что вы понимаете, почему программа не работает. Бегло просмотрите упражнение в конце главы. Если вы справитесь с ним, не подглядывая в ответ, значит, вы *уже* готовы перейти к следующей главе.

Ката № 2: Выполните проект из упражнения, которое начинается со слов: «Чтобы немного потренироваться в использовании ключевого слова **private**, мы создадим маленькую игру “Больше-меньше”». Убедитесь в том, что вы понимаете, как в нем используются ключевые слова **const**, **public** и **private**. Не забудьте о дополнительной задаче.

Ката № 3: Реализуйте маленький проект из врезки «Будьте осторожны!», которая начинается со слов: **«Инкапсуляция — не то же самое, что безопасность. Приватные поля не защищены»**.

Прочитайте следующие разделы: «Свойства упрощают инкапсуляцию», «Автоматически реализуемые свойства упрощают ваш код» и «Использование приватного set-метода для создания свойств, доступных только для чтения».

Ката № 4: Выполните последнее упражнение в этой главе, в котором инкапсуляция применяется для исправления класса SwordDamage.

Глава 6: Наследование



Темой этой главы является наследование; в C# это означает создание классов, которые повторно используют, расширяют и изменяют поведение других классов. Если вы пошли по пути ката программирования, весьма вероятно, что вы уже знаете какой-нибудь язык с наследованием. Эти ката дают представление о синтаксисе наследования в C#.

Ката № 1: Первый проект этой главы расширяет калькулятор повреждений из главы 4. Наследование в нем *не используется* — он всего лишь показывает начинающим программистам, в чем ценность наследования. Также в нем представлен синтаксис команды C# `switch`. Вероятно, вы легко справитесь с этим упражнением.

Прежде чем выполнять остальные ката, просмотрите следующие разделы: «*В любом месте, где может использоваться базовый класс, вместо него можно использовать один из subclasses*» и «*Некоторые компоненты реализуются только в subclasses*». Затем просмотрите раздел «*Subclass может скрывать методы базового класса*» вместе со всеми подразделами.

Ката № 2: Вернитесь и выполните упражнение, которое начинается со слов: «Немного потренируемся в расширении базовых классов». После него с базовым синтаксисом наследования в C# проблем быть не должно.

Ката № 3: Выполните проект из раздела «*Если базовый класс содержит конструктор, ваш subclass должен его вызвать*».

Ката № 4: Выполните упражнение после раздела «*Пора сделать приложение для Оуэна*». Упражнение состоит из двух частей: в первой, «бумажной», вам будет предложено заполнить диаграмму классов, а во второй — написать код для ее реализации.

Если вы завершили первые четыре ката без особых трудностей, можно двигаться дальше. Тем не менее **мы настоятельно рекомендуем построить систему управления ульем** в конце главы. На самом деле это довольно занятный проект, из которого можно извлечь ряд полезных уроков об игровой динамике, а преобразование пошаговой игры в игру в реальном времени всего несколькими строками кода — в самом конце главы.

Если у вас возникнут затруднения, будет наиболее эффективно переключиться на последовательное чтение главы от последнего успешно завершеного ката.

Поздравляем тех, кто выбрал короткий путь по книге. Если вы добрались до главы 6, следуя по пути «Ката программирования», значит, вы уже полностью готовы к главе 7.