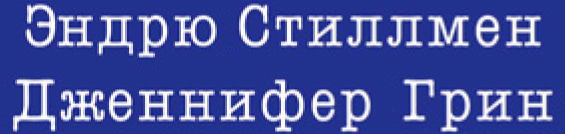


Эндрю Стиллмен
Дженнифер Грин

Включая C# 5.0, Visual Studio 2012 и .NET 4.5 Framework

Изучаем C#

3-е издание

Управляй данными с помощью абстрактных классов и наследования

Познай секреты абстрагирования и наследования

Изучай C# на забавных примерах

Узнай, как методы расширения облегчают жизнь программиста

Научись эффективно использовать обобщенные коллекции

O'REILLY®

ПИТЕР®

Head First C#

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

3-rd Edition

Andrew Stellman
Jennifer Greene

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Изучаем C#

Включая C# 5.0, Visual Studio 2012 и .NET 4.5 Framework

Как бы было хорошо
найти книгу по C#,
которая будет веселее визита
к зубному врачу и понятнее
налоговой декларации...
Наверное, об этом можно
только мечать...

3-е издание

Э. Стиллмен
Дж. Грин

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2014

ББК 32.973.2-018.1
УДК 004.43
С80

Стиллмен Э., Грин Дж.

С80 Изучаем C#. 3-е изд. — СПб.: Питер, 2014. — 816 с.: ил. — (Серия «Head First O'Reilly»).

ISBN 978-5-496-00867-9

В отличие от большинства книг по программированию, построенных на основе скучного изложения спецификаций и примеров, с этой книгой читатель сможет сразу приступить к написанию собственного кода на языке программирования C# с самого начала. Вы освоите минимальный набор инструментов, а далее примете участие в забавных и интересных программных проектах: от разработки карточной игры до создания серьезного бизнес-приложения. Второе издание книги включает последние версии C# 5.0, Visual Studio 2012 и .NET 4.5 Framework и будет интересно всем изучающим 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.

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

ББК 32.973.2-018.1
УДК 004.43

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.

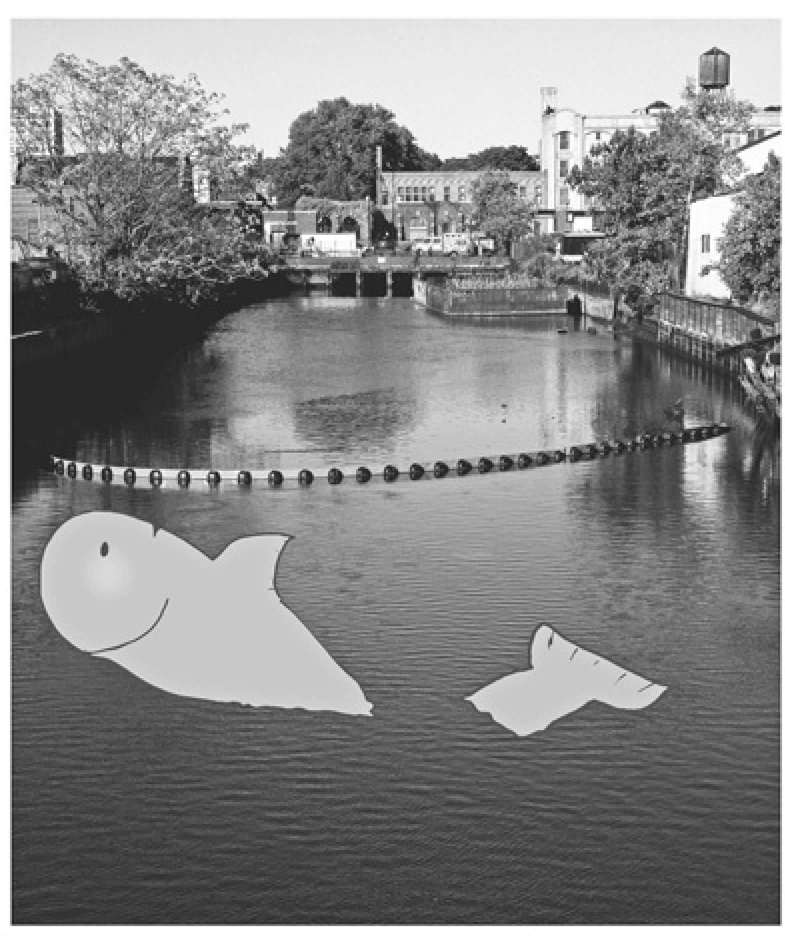
ISBN 978-1449343507 англ.

Authorized Russian translation of the English edition of Head First C#, 3rd Edition (ISBN 9781449343507) © 2013 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.

ISBN 978-5-496-00867-9

© Перевод на русский язык ООО Издательство «Питер», 2014
© Издание на русском языке, оформление ООО Издательство «Питер», 2014

*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*

Спасибо, что купили нашу книгу!
Мы писали ее с удовольствием
и надеемся, что вы получите
кайф от ее прочтения...

Эндрю

Автор этой фотографии (как и снимка канала Гованус) — Ниша Сондхе

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 инвестиционного 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 музыку и в компьютерные игры, практикует тайцзицюань и айкидо и заботится о своем карликовом шпице.

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 онлайновой 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.

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

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

Содержание (сводка)

	Введение	25
1	Начало работы с C#. *Быстро сделать что-то классное!*	37
2	Это всего лишь код. *Под покровом*	89
3	Объекты, по порядку стройся! *Приемы программирования*	137
4	Типы и ссылки. *10:00 утра. Куда подевались наши данные?*	175
5	Инкапсуляция. *Пусть личное остается… личным*	219
6	Наследование. *Генеалогическое древо объектов*	259
7	Интерфейсы и абстрактные классы. *Пусть классы держат обещания*	313
8	Перечисления и коллекции. *Большие объемы данных*	369
9	Чтение и запись файлов. *Прибереги последний байт для меня*	425
10	Приложения для магазина Windows на языке XAML. *Приложения следующего уровня*	479
11	Async, await и сериализация контрактов данных. *Позвольте вас прервать*	527
12	Обработка исключений. *Борьба с огнем надоедает*	561
13	Капитан Великолепный. *Смерть объекта*	601
14	LINQ: *Управляем данными*	639
15	События и делегаты. *Что делает ваш код, когда вы на него не смотрите*	691
16	Проектирование с шаблоном MVM. *Прекрасные изнутри и снаружи*	735
17	Бонусный проект! *Приложение Windows Phone*	797

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	26
Мы знаем, о чем вы думаете	27
Метапознание: наука о мышлении	29
Заставьте свой мозг повиноваться	31
Что вам потребуется	32
Научные редакторы	34
Благодарности	35

1

начало работы с C#

Быстро сделать что-то классное!

Хотите программировать действительно 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 сфокусироваться на достижении результата. Звучит заманчиво? Тогда переверните страницу и приступим к делу.

Зачем вам изучать C#	38
C#, IDE Visual Studio многое упрощает	39
Это вы делаете в Visual Studio	40
А это Visual Studio делает за вас	40
Инопланетяне атакуют!	44
Только ты можешь спасти Землю	45
Вот что вам нужно сделать	46
Начнем с пустого приложения	48
Настройка сетки	54
Добавим к сетке элементы управления	56
Меняем вид элементов управления	58
Элементы управления игрой	60
Игровое поле готово	65
Что дальше?	66
Добавляем метод, который что-то делает	67
Пишем код метода	68
Завершение метода и запуск программы	70
Управляющие игрой таймеры	74
Активация кнопки Start	76
Взаимодействие с игроком	78
Соприкосновение людей с врагами означает конец игры	80
В игру уже можно играть	81
Превратим врагов в пришельцев	82
Добавим заставку и значок приложения	83
Публикация приложения	84
Загрузка через удаленный отладчик	85
Удаленная отладка	86

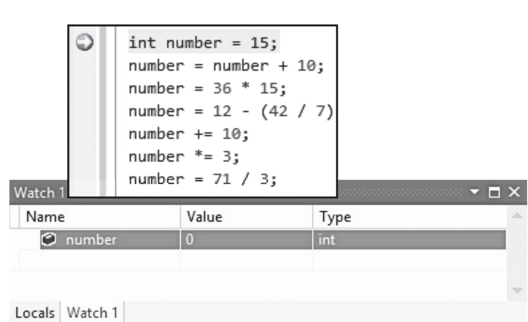
это всего лишь код

Под покровом

Вы — программист, а не просто пользователь IDE.

IDE может сделать за вас многое, но не всё. При написании приложений часто приходится решать повторяющиеся задачи. Пусть эту работу выполняет IDE. Вы же будете в это время думать над более глобальными вещами. Научившись писать код, вы получите возможность решить любую задачу.

Когда вы делаете это... 90
...IDE делает это. 91
Как рождаются программы 92
Писать код помогает IDE 94
Структура программы 96
Классы могут принадлежать одному пространству имен 101
Что такое переменные 102
Знакомые математические символы 104
Наблюдение за переменными в процессе отладки 105
Циклы 107
Оператор выбора 108
Приложение с нуля 109
Пусть каждая кнопка что-то делает 111
Проверка условий 112
Приложения для рабочего стола Windows 123
Перестроим приложение для рабочего стола Windows 124
Начало работы программы 128
Редактирование точки входа 130
Любые действия ведут к изменению кода 132

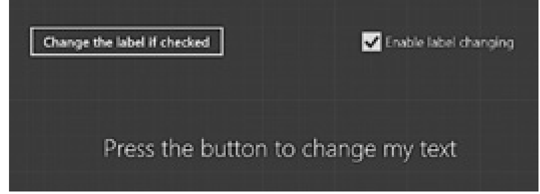


Для каждой программы вы определяете свое пространство имен, отделяя код от классов .NET Framework и Windows Store API.

Класс содержит **фрагмент** вашей программы (очень маленькие программы могут состоять из всего одного класса).

Класс включает один или несколько методов. Методы всегда **принадлежат какому-либо классу**. Методы, в свою очередь, состоят из операторов.

Пространство имен
Класс
Метод 1
оператор
оператор
Метод 2
оператор
оператор



3

объекты, по порядку стройся!

Приемы программирования

Каждая программа решает какую-либо проблему. Перед написанием программы нужно четко сформулировать, какую задачу она будет решать. Именно поэтому так полезны объекты. Ведь они позволяют структурировать код наиболее удобным образом. Вы же можете сосредоточиться на обдумывании путей решения, так как вам не нужно тратить время на написание кода. Правильное использование объектов позволяет получить интуитивно понятный код, который при необходимости можно легко отредактировать.

Что думает Майк о своих проблемах 138

Проблема Майка с точки зрения навигационной системы 139

Методы прокладки и редактирования маршрутов 140

Программа с использованием классов 141

Идея Майка 143

Объекты как способ решения проблемы 144

Возьмите класс и постройте объект 145

Экземпляры 146

Простое решение! 147

Создаем экземпляры! 153

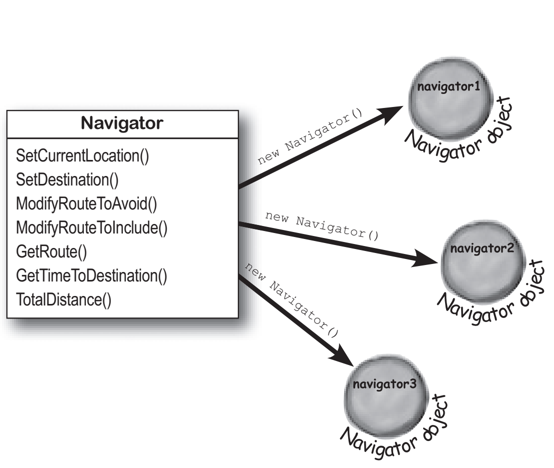
Спасибо за память 154

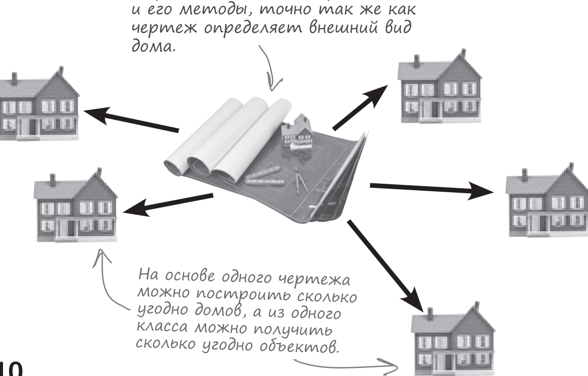
Что происходит в памяти программы 155

Выбор структуры класса при помощи диаграммы 160

Форма для взаимодействия с кодом 166

Более простые способы присвоения начальных значений 169





типы и ссылки

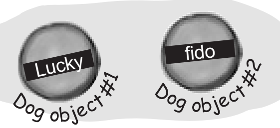
10:00 утра. Куда подевались наши данные?

Без данных программы бесполезны. Взяв информацию от пользователей, вы производите новую информацию, чтобы вернуть ее им же. Практически все в программировании связано с обработкой данных тем или иным способом. В этой главе вы познакомитесь с используемыми в C# типами данных, узнаете методы работы с ними и даже ужасный секрет объектов (только т-с-с-с… объекты — это тоже данные).

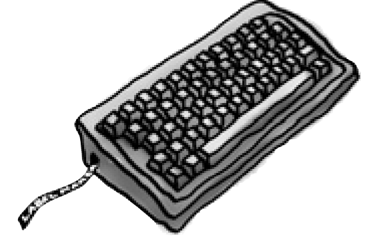
Тип переменной определяет, какие данные она может сохранять 176
Наглядное представление переменных 178
10 литров в 5-литровой банке 179
Приведение типов 180
Автоматическая коррекция слишком больших значений 181
Иногда приведение типов происходит автоматически 182
Аргументы метода должны быть совместимы с типами параметров 183
Отладка калькулятора расстояний 187
Комбинация с оператором = 188
Объекты тоже используют переменные 189
Переменные ссылочного типа 190
Ссылки подобны маркерам 191
При отсутствии ссылок объект превращается в мусор 192
Побочные эффекты множественных ссылок 193
Две ссылки — это ДВА способа редактировать данные объекта 198
Массив может состоять из ссылочных переменных 200
Добро пожаловать на распродажу сэндвичей от Джо! 201
Ссылки позволяют объектам обращаться друг к другу 203
Сюда объекты еще не отправлялись 204
Играем в печатную машинку 209
Элементы управления — это тоже объекты 213


```
Dog fido;
Dog lucky = new Dog();
```

```
fido = new Dog();
```




```
lucky = null;
```

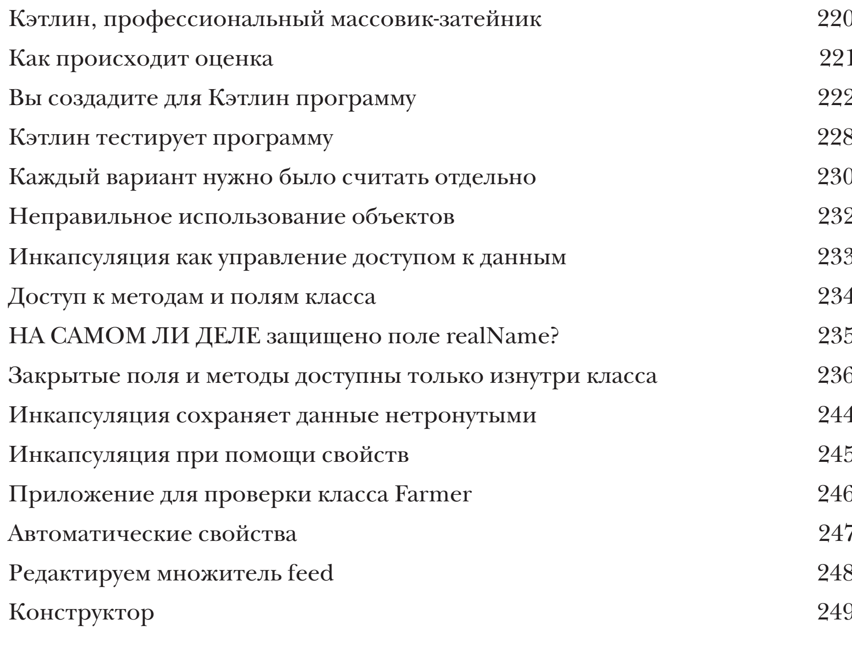


5

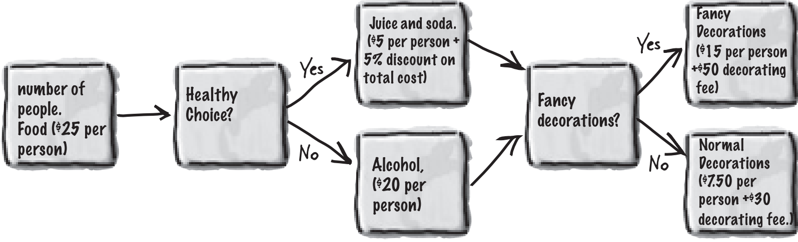
инкапсуляция

Пусть личное остается... личным

Вы когда-нибудь мечтали о том, чтобы вашу личную жизнь оставили в покое? Иногда объекты чувствуют то же самое. Вы же не хотите, чтобы посторонние люди читали ваши записки или рассматривали банковские выписки? Вот и объекты не хотят, чтобы другие объекты имели доступ к их полям. В этой главе мы поговорим об инкапсуляции. Вы научитесь закрывать объекты и добавлять методы, защищающие данные от доступа.



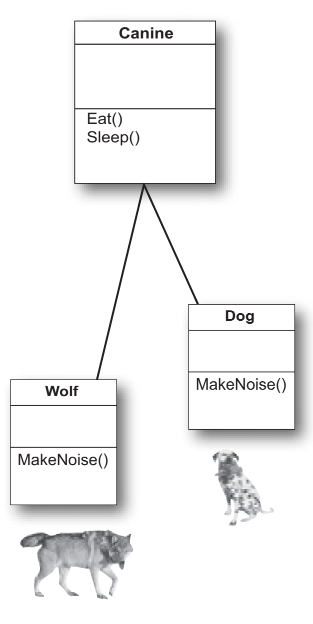
Кэтлин, профессиональный массовик-затейник 220
Как происходит оценка 221
Вы создадите для Кэтлин программу 222
Кэтлин тестирует программу 228
Каждый вариант нужно было считать отдельно 230
Неправильное использование объектов 232
Инкапсуляция как управление доступом к данным 233
Доступ к методам и полям класса 234
НА САМОМ ЛИ ДЕЛЕ защищено поле realName? 235
Закрытые поля и методы доступны только изнутри класса 236
Инкапсуляция сохраняет данные нетронутыми 244
Инкапсуляция при помощи свойств 245
Приложение для проверки класса Farmer 246
Автоматические свойства 247
Редактируем множитель feed 248
Конструктор 249

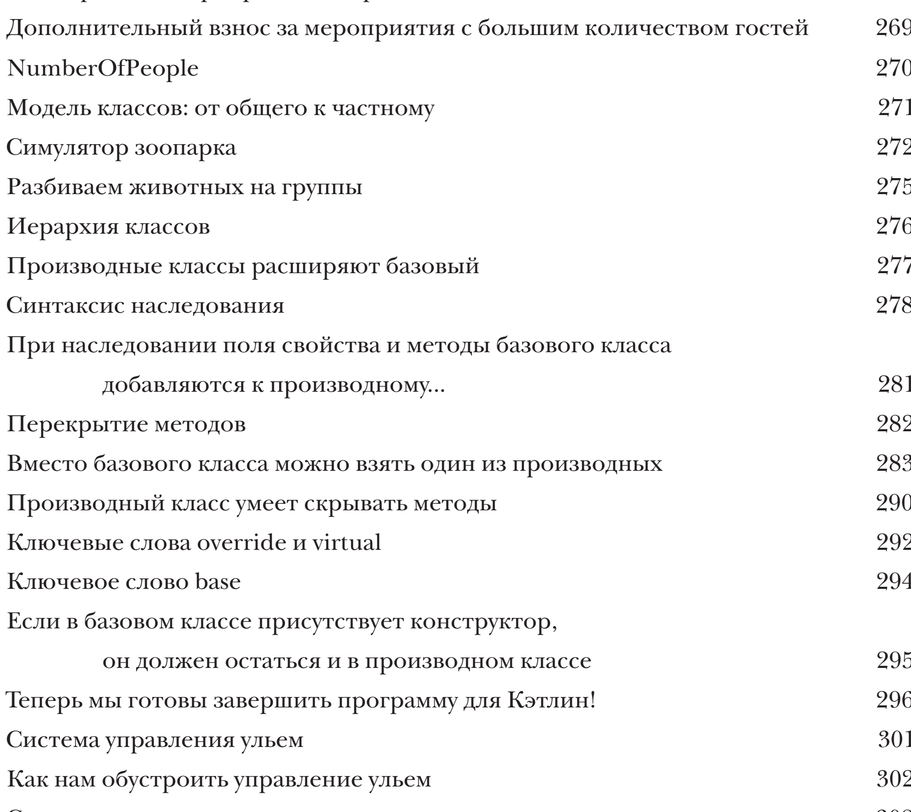


наследование

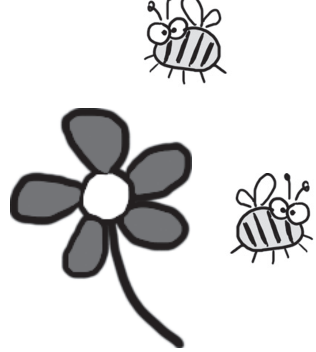
Генеалогическое древо объектов

Иногда люди хотят быть похожими на своих родителей.
Вы встречали объект, который действует почти так, как нужно? Думали ли вы о том, какое совершенство можно было бы получить, изменив всего несколько элементов? Именно по этой причине наследование является одним из самых мощных инструментов C#. В этой главе вы узнаете, как производный класс повторяет поведение родительского, сохраняя при этом гибкость редактирования. Вы научитесь избегать дублирования кода и облегчите последующее редактирование своих программ.





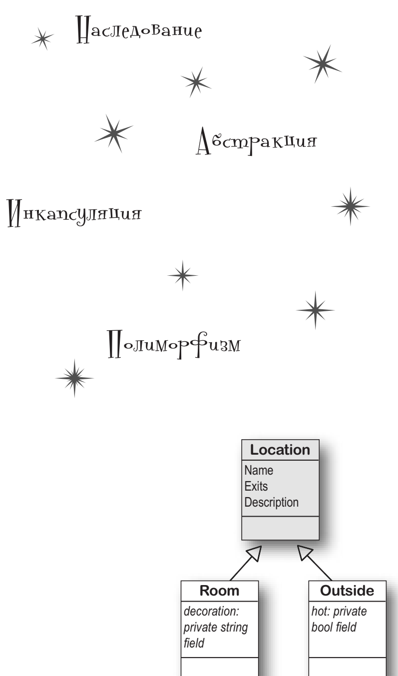
Организация дней рождения — это тоже работа Кэтлин 260
Нам нужен класс BirthdayParty 261
Планировщик мероприятий, версия 2.0 262
Дополнительный взнос за мероприятия с большим количеством гостей 269
NumberOfPeople 270
Модель классов: от общего к частному 271
Симулятор зоопарка 272
Разбиваем животных на группы 275
Иерархия классов 276
Производные классы расширяют базовый 277
Синтаксис наследования 278
При наследовании поля свойства и методы базового класса добавляются к производному... 281
Перекрытие методов 282
Вместо базового класса можно взять один из производных 283
Производный класс умеет скрывать методы 290
Ключевые слова override и virtual 292
Ключевое слово base 294
Если в базовом классе присутствует конструктор, он должен остаться и в производном классе 295
Теперь мы готовы завершить программу для Кэтлин! 296
Система управления ульем 301
Как нам обустроить управление ульем 302
Совершенствуем систему управления ульем при помощи наследования 308



7 интерфейсы и абстрактные классы

Пусть классы держат обещания

Действия значат больше, чем слова. Иногда возникает необходимость сгруппировать объекты по выполняемым функциям, а не по классам, от которых они наследуют. Здесь вам на помощь приходят интерфейсы — они позволяют работать с любым классом, отвечающим вашим потребностям. Но чем больше возможностей, тем выше ответственность, и если классы, реализующие интерфейс, не выполнят обязательств… программа компилироваться не будет.

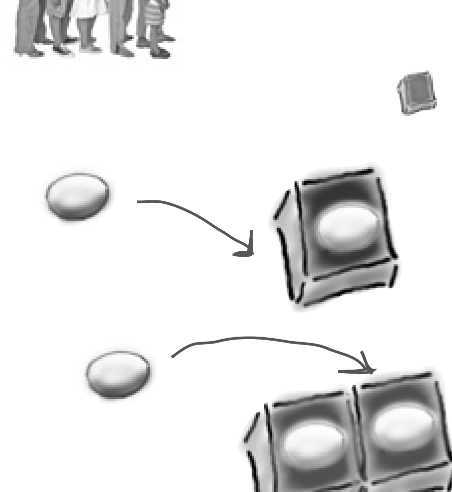


Вернемся к нашим пчелам 314
Классы для различных типов пчел 315
Интерфейсы 316
Ключевое слово interface 317
Экземпляр NectarStinger 318
Классы, реализующие интерфейсы, должны включать ВСЕ методы интерфейсов 319
Учимся работать с интерфейсами 320
Ссылки на интерфейс 322
Ссылка на интерфейс аналогична ссылке на объект 323
Интерфейсы и наследование 325
Кофеварка относится к Приборам 328
Восходящее приведение 329
Нисходящее приведение 330
Нисходящее и восходящее приведение интерфейсов 331
Изменение видимости при помощи модификаторов доступа 336
Классы, для которых недопустимо создание экземпляров 339
Абстрактный класс. Перепутье между классом и интерфейсом 340
Как уже было сказано, создавать экземпляры некоторых классов недопустимо 342
Абстрактный метод не имеет тела 343
Смертельным ромбом! 348
Различные формы объекта 351

перечисления и коллекции

8 Большие объемы данных

Пришла беда — отворяй ворота. В реальном мире данные, как правило, не хранятся маленькими кусочками. Данные поступают вагонами, штабелями и кучами. Для их систематизации нужны мощные инструменты, и тут вам на помощь приходят коллекции. Они позволяют хранить, сортировать и редактировать данные, которые обрабатывает программа. В результате вы можете сосредоточиться на основной идее программирования, оставив задачу отслеживания данных коллекциям.



Категории данных не всегда можно сохранять в переменных типа string 370
Перечисления 371
Присвоим числам имена 372
Создать колоду карт можно было при помощи массива... 375
Проблемы работы с массивами 376
Коллекции 377
Коллекции List 378
Динамическое изменение размеров 381
Обобщенные коллекции 382
Инициализаторы коллекций похожи на инициализаторы объектов 386
Коллекция уток 387
Сортировка элементов коллекции 388
Интерфейс IComparable<Duck> 389
Способы сортировки 390
Создадим экземпляр объекта-компаратора 391
Сложные схемы сравнения 392
Перекрытие метода ToString() 395
Обновим цикл foreach 396
Интерфейс IEnumerable<T> 397
Восходящее приведение с помощью IEnumerable 398
Создание перегруженных методов 399
Словари 405
Функциональность словарей 406
Практическое применение словаря 407
Дополнительные типы коллекций... 419
Очередь: первый вошел, первый вышел 420
Стек: последним вошел, первым вышел 421

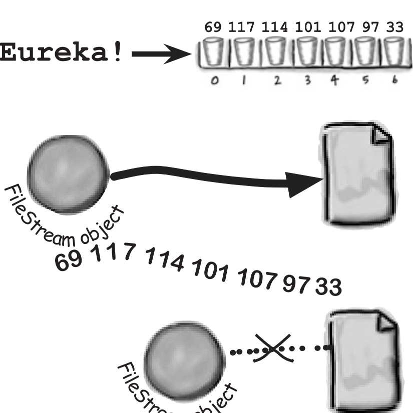
9

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 потоковыми классами .NET и узнаете о тайнах шестнадцатеричной и двоичной систем счисления.

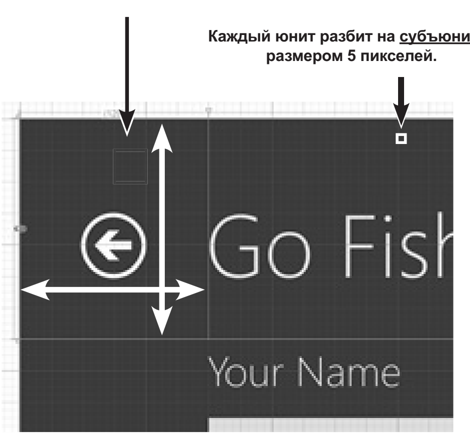


Для чтения и записи данных в .NET используются потоки 426
Различные потоки для различных данных 427
Объект FileStream 428
Трехшаговая процедура записи текста в файл 429
Чтение и запись при помощи двух объектов 433
Данные могут проходить через несколько потоков 434
Встроенные объекты для вызова стандартных окон диалога 437
Встроенные классы File и Directory 440
Открытие и сохранение файлов при помощи окон диалога 443
Интерфейс IDisposable 445
Операторы using как средство избежать системных ошибок 446
Проблемы на работе 447
Запись файлов сопровождается принятием решений 452
Оператор switch 453
Чтение и запись информации о картах в файл 454
Чтение карт из файла 455
Сериализации подвергается не только сам объект... 459
Сериализация позволяет читать и записывать объект целиком 460
Сериализуем и десериализуем колоду карт 462
.NET использует Unicode для хранения символов и текста 465
Перемещение данных внутри массива байтов 466
Класс BinaryWriter 467
Чтение и запись сериализованных файлов вручную 469
Найдите отличия и отредактируйте файлы 470
Сложности работы с двоичными файлами 471
Программа для создания дампа 472
Достаточно StreamReader и StreamWriter 473
Чтение байтов из потока 474

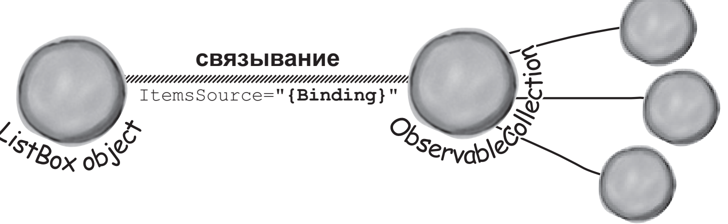
10 приложения для магазина windows на языке xaml

Приложения следующего уровня

Новые возможности разработки приложений. Работа с интерфейсом WinForms знакомит с важными понятиями C#, но и не только. В этой главе вы создадите приложения для магазина Windows на языке XAML, научитесь конструировать страницы для различных устройств, встраивать данные путем привязки, а затем освоите тайны XAML-страниц, исследуя XAML-объекты при помощи Visual Studio.



Брайан запускает Windows 8	480
Windows Forms использует граф объекта, настроенный IDE	486
Исследуем граф объекта в IDE	489
Применение XAML для создания UI-объектов	490
Переделаем форму Go Fish! в страницу Windows Store	492
Компоновка страницы начинается с элементов управления	494
Размеры столбцов и строк подстраиваются под страницу	496
Система сеток и компоновка страниц	498
Связывание страниц XAML с классами	504
Элементы XAML могут содержать... не только текст	506
Обновим меню при помощи связывания данных	508
Объявление объектов в XAML через статические ресурсы	514
Отображение объектов через шаблон данных	516
Интерфейс INotifyPropertyChanged	518
Предупредим MenuMaker об изменении свойства GeneratedDate	519



11 async, await и сериализация контрактов данных

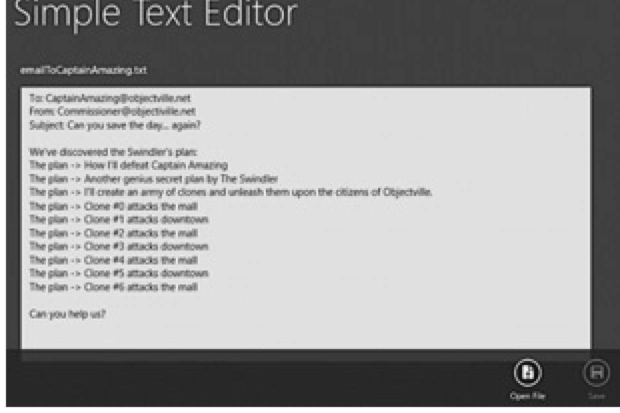
Позвольте вас прервать

Никто не любит ждать... особенно пользователи.

Компьютеры умеют решать несколько задач одновременно, почему бы не добавить вашим приложениям аналогичную возможность? В этой главе вы увеличите «отзывчивость» приложений при помощи асинхронных методов. Вы научитесь использовать встроенные средства выбора файлов и диалоговые окна с сообщениями, а также асинхронный файловый ввод и вывод. Скомбинировав это с сериализацией, вы получите технологию создания современных приложений.

У Брайана проблемы с файлами 528
Увеличение отзывчивости с оператором await 530
Чтение и запись файлов при помощи класса FileIO 532
Слегка усложненный текстовый редактор 534
Контракт данных — абстрактное определение данных вашего объекта 539
Асинхронные методы для поиска и открытия файлов 540
Граф объекта целиком сериализуется в XML 543
Сохраним объекты Guy в локальную папку приложения 544
Тестирование нашего сериализатора 548
Новый генератор оправданий для Брайана 550
Разделим страницу, оправдания и Excuse Manager 551
Главная страница генератора оправданий 552
Добавим панель приложения 553
Создание класса ExcuseManager 554
Добавление программного кода 556

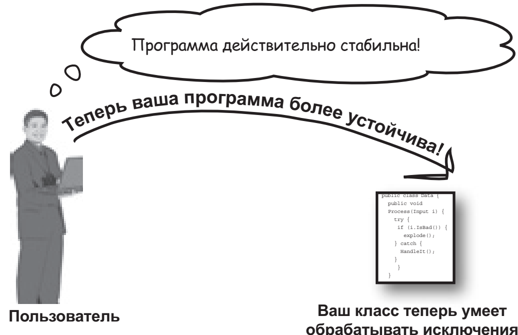

```
FileIO.
    AppendLinesAsync
    AppendTextAsync
    Equals
    ReadBufferAsync
    ReadLinesAsync
    ReadTextAsync
    ReferenceEquals
    WriteBufferAsync
    WriteBytesAsync
```

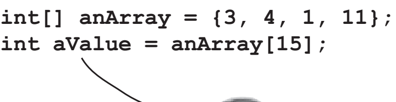


12 обработка исключений

Борьба с огнем надоедает

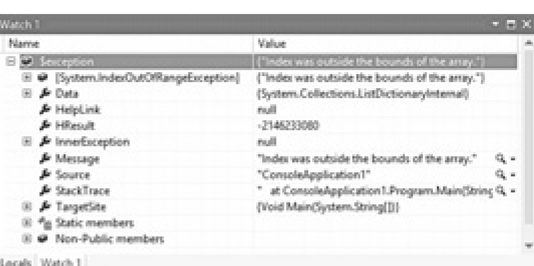
Программисты не должны уподобляться пожарным. Вы усердно работали, штудировали справочники и руководства и, наконец, достигли вершины: теперь вы главный программист. Но вам до сих пор продолжают звонить с работы по ночам, потому что программа упала или работает неправильно. Ничто так не выбивает из колеи, как необходимость устранять странные ошибки… но благодаря обработке исключений вы сможете написать код, который сам будет разбираться с возможными проблемами.





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

Exception object



Брайану нужны мобильные оправдания 562

Объект Exception 566

Код Брайана работает не так, как предполагалось 568

Исключения наследуют от объекта Exception 570

Работа с отладчиком 571

Поиск ошибки в приложении Excuse Manager с помощью отладчика 572

А код все равно не работает... 575

Ключевые слова try и catch 577

Вызов сомнительного метода 578

Результаты применения ключевых слов try/catch 580

Ключевое слово finally 582

Получение сведений о проблеме 587

Обработка исключений разных типов 588

Один класс создает исключение, другой его обрабатывает 589

Исключение OutOfHoney для пчел 590

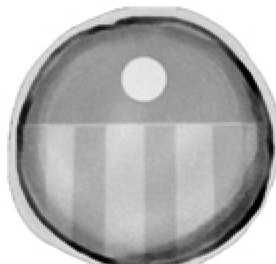
Наихудший вариант блока catch 596

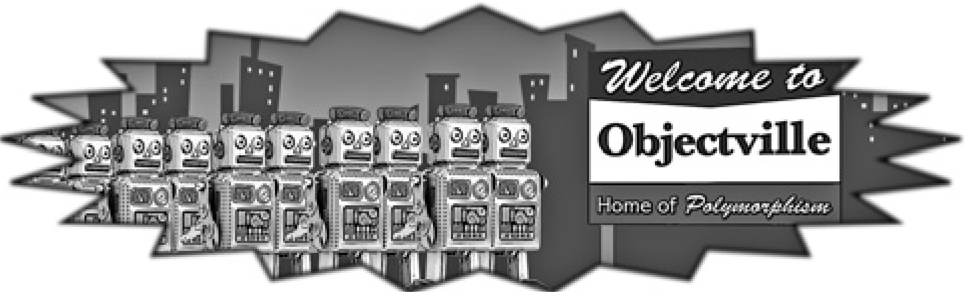
Временные решения 597

Краткие принципы обработки исключений 598

13 Капитан Великолепный: смерть объекта

Метод завершения объекта 608
Когда запускается метод завершения объекта 609
Явное и неявное высвобождение ресурсов 610
Возможные проблемы 612
Сериализуем объект в методе Dispose() 613
Структура напоминает объект... 617
...но объектом не является 617
Значения копируются, а ссылки присваиваются 618
Структуры — это значимые, а объекты — ссылочные типы 619
Сравнение стека и кучи 621
Необязательные параметры 626
Типы, допускающие значение null 627
Типы, допускающие значение null, увеличивают робастность программы 628
Что осталось от Великолепного 631
Методы расширения 632
Расширяем фундаментальный тип: string 634





14 LINQ

Управляем данными

Этот мир управляется данными... вам лучше знать, как в нем жить. Времена, когда можно было программировать днями и даже неделями, не касаясь множества данных, давно позади. В наши дни с данными связано все. Часто приходится работать с данными из разных источников и даже разных форматов. Базы данных, XML, коллекции из других программ... все это давно стало частью работы программиста на C#. В этом ему помогает LINQ. Эта функция не только упрощает запросы, но и умеет разбивать данные на группы и, наоборот, соединять данные из различных источников.

Джимми фанат комиксов... 640
...но его коллекция валяется по всему дому 641
Сбор данных из разных источников 642
Коллекции .NET уже настроены под LINQ 643
Простой способ сделать запрос 644
Сложные запросы 645
Идем на помощь Джимми 648
Начало работы над приложением для Джимми 650
Создание анонимных типов 653
Универсальность LINQ 656
Новые запросы 658
Сгруппируем результаты Джимми 664
Предложение join 667
Джимми изрядно сэкономил 668
Контекстное масштабирование 674
Усовершенствуем приложение Джимми 676
Вы осчастливили Джимми 681
Шаблон Split App 682

15 события и делегаты

Что делает код, когда вы на него не смотрите

Невозможно все время контролировать созданные объекты. Иногда что-то… происходит. И хотелось бы, чтобы объекты умели реагировать на происходящее. Здесь вам на помощь приходят события. Один объект их публикует, другие объекты на них подписываются, и система работает. А для контроля подписчиков вам пригодится метод обратного вызова.

Хотите, что объекты научились думать сами? 692
Как объекты узнают, что произошло? 692
События 693
Один объект инициирует событие, другой реагирует на него 694
Обработка события 695
Соединим все вместе 696
Автоматическое создание обработчиков событий 700
Обобщенный EventHandler 706
Все формы используют события 707
Несколько обработчиков одного события 708
Приложения для магазина Windows и события 710
Управление жизненным циклом приложения Джимми 711
Элементы XAML пользуются перенаправленными событиями 714
Исследуем перенаправленные события 715
Связь между издателями и слушателями 720
Делегат замещает методы 721
Делегаты в действии 722
Объект может подписаться на событие… 725
Обратный вызов 726
Обратный вызов как способ работы с делегатами 728
Обратные вызовы с командами класса MessageDialog 730
Настройка панели чудо-кнопок в Windows 732

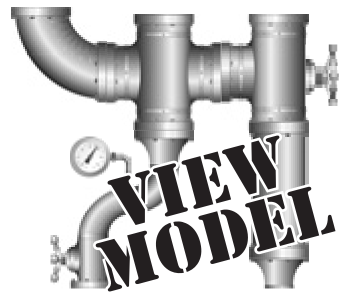
Border sets handled
Off
Grid sets handled
Off
Ellipse sets handled
Off
Rectangle sets handled
Off
The rectangle was pressed
The grid was pressed
The border was pressed
The panel was pressed
Update Rectangle IsHitTestVisible
New IsHitTestVisible value
On

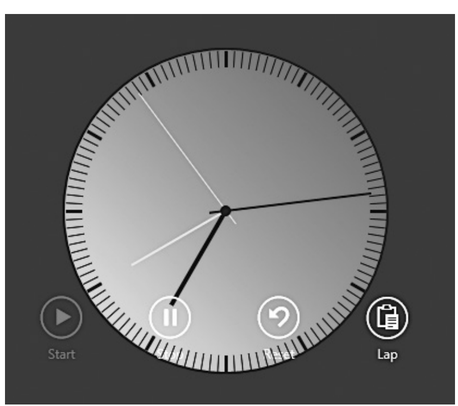
BallInPlay event
BallEventArgs
Pitcher object
Umpire object
Fan object

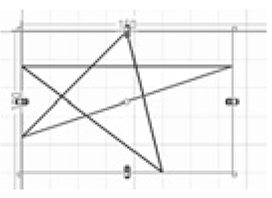
проектирование с шаблоном mvvm

Прекрасные изнутри и снаружи

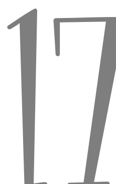
Для приложения важна не только визуальная привлекательность. Что первым делом приходит на ум, когда вы думаете о дизайне? Грамотная архитектура? Прекрасная компоновка? Эстетически приятный и хорошо работающий продукт? Все это относится и к вашим приложениям. В этой главе вы познакомитесь с шаблоном Model-View-ViewModel и узнаете, как получить хорошо спроектированные, слабо связанные приложения. Попутно мы поговорим об анимации и шаблонах элементов управления, о том, как при помощи преобразователей упростить связывание данных, и о том, как связать все воедино, заложив твердый фундамент C# под свою способность написать любое приложение.







Приложение для баскетбольного матча Head First	736
Как прийти к согласию?	737
Проектирование для связывания или для работы с данными?	738
Проектирование для связывания и для данных	739
Начнем строить приложение для баскетболистов	740
Ваши собственные элементы управления	743
Секундомер для судьи	751
MVVM как забота о состоянии приложения	752
Начало работы над слоем Model	753
События как сигнал об изменении состояния	754
Представление для простого секундомера	755
Слой ViewModel для секундомера	756
Завершение работы над секундомером	758
Автоматическое преобразование значений для связывания	760
Преобразователи работают с разными типами	762
Меняем вид кнопок	764
Визуальные состояния и элементы управления	768
Класс DoubleAnimation	769
Анимация параметров через анимацию объектов	770
Аналоговый секундомер на основе того же класса ViewModel	771
Экземпляры элементов UI	776
C# и «реальная» анимация	778
Элемент управления для анимации картинки	779
Пусть пчелы летают по странице	780
Связывание через ItemsPanelTemplate	783
Наши поздравления! (Но нам еще рано почивать на лаврах...)	796



бонусный проект!

Приложение Windows Phone

Вы уже можете создавать приложения Windows Phone. Классы, объекты, XAML, инкапсуляция, наследование, полиморфизм, LINQ, MVVM… у вас есть все, что нужно превосходным приложениям для магазина Windows и для рабочего стола. Но знали ли вы, что эти же инструменты позволяют создавать приложения Windows Phone? Это действительно так! И наш бонусный проект посвящен созданию игры для Windows Phone. Не волнуйтесь, если у вас отсутствует Windows Phone, вам поможет эмулятор. Так что начинаем!

Пчелы атакуют! 798

Перед тем как начать... 799

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#?

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.

Знаете другой язык программирования и теперь хотите изучить C#?

Вы уже пишете на C#, но хотите больше знать о XAML, шаблоне Model-View-ViewModel (MVVM) или проектировании приложений для магазина Windows?

Просто хотите попрактиковаться в написании кода?

Многие люди с аналогичными желаниями уже воспользовались нашей книгой для точно таких же целей!

Кому эта книга не подойдет?

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

1. Вы скучаете или раздражаетесь при мысли о том, что придется часто и много писать код?

2. Вы отличный программист на C++ или Java, которому нужен *справочник*?

3. Вы боитесь попробовать что-нибудь новое? Скорее пойдете к зубному врачу, чем наденете полосатое с клетчатым? Считаете, что техническая книга, в которой концепции C# изображены в виде человечков, серьезной быть не может?

...эта книга не для вас.

Чтобы читать эту книгу опыт программирования не нужен... достаточно вашего интереса! Тысячи новичков уже воспользовались предыдущим изданием нашей книги для изучения этого языка. На их месте можете быть и вы!

[Замечание от отдела продаж: «Вообще-то эта книга для любого, у кого есть деньги».]

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?»

Ваш мозг считает, что ЭТО важно.

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.

В наши дни вы вряд ли попадете на обед к тигру. Но наш мозг постоянно остается настороже. Просто мы об этом не знаем.

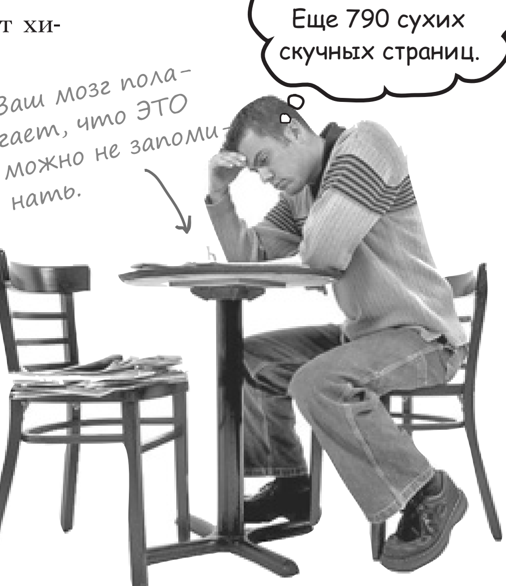
Как же наш мозг поступает со всеми обычными, повседневными вещами? Он всеми силами пытается оградиться от них, чтобы они не мешали его *настоящей* работе — запоминанию того, что действительно *важно*. Мозг не считает нужным сохранять скучную информацию. Она не через проходит фильтр, отсекающий «очевидно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

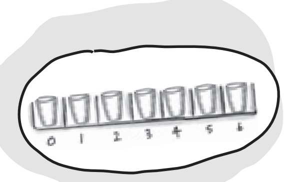
И тут возникает проблема: ваш мозг пытается оказать вам услугу. Он старается сделать так, чтобы на эту *очевидно* несущественную информацию не тратились драгоценные ресурсы. Их лучше потратить на что-нибудь важное. На тигров, например. Или на то, что к огню лучше не прикасаться. Или на то, что ни в коем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

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»:

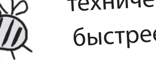


Наглядность. Графика запоминается гораздо лучше, чем обычный текст, и значительно повышает эффективность восприятия информации (до 89% по данным исследований). Кроме того, материал становится более понятным. **Текст размещается на рисунках**, к которым он относится, а не под ними или на соседней странице.

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?

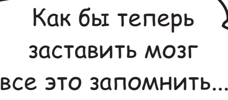
Активное участие читателя. Пока вы не начнете напрягать извилины, в вашей голове ничего не произойдет. Читатель должен быть заинтересован в результате; он должен решать задачи, формулировать выводы и овладевать новыми знаниями. А для этого необходимы упражнения и каверзные вопросы, в решении которых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



Обращение к эмоциям. Известно, что наша способность запоминать в значительной мере зависит от эмоционального сопереживания. Мы запоминаем то, что нам небезразлично. 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, — или когда вы понимаете, что разбираетесь в теме лучше, чем всезнайка Боб из технического отдела.

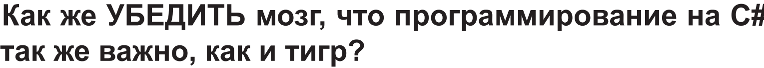
Метапознание: наука о мышлении



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, вы 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?

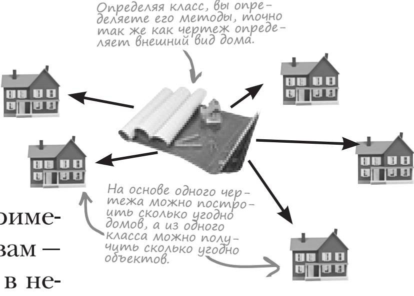
Есть способ медленный и скучный, а есть быстрый и эффективный. Первый основан на тупом повторении. Всем известно, что даже самую скучную информацию *можно* запомнить, если повторять ее снова и снова. При достаточном количестве повторений ваш мозг прикидывает: «*Вроде бы* несущественно, но раз одно и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

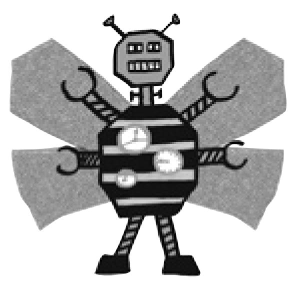
Вот что сделали МЫ



Мы использовали **рисунки**, потому что мозг лучше приспособлен для восприятия графики, чем текста. С точки зрения мозга рисунок сто́ит 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.

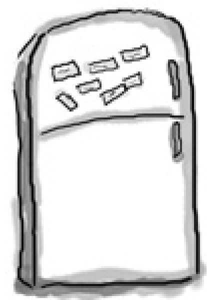
Мы совместили ***несколько стилей обучения***, потому что одни читатели любят пошаговые описания, другие стремятся сначала представить «общую картину», а третьим хватает фрагмента кода. Независимо от ваших личных предпочтений полезно видеть несколько вариантов представления одного и того же материала.

Мы постарались задействовать ***оба полушария вашего мозга***: это повышает вероятность усвоения материала. Пока одна сторона мозга работает, другая часто имеет возможность отдохнуть; это повышает 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*.



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.

Вырежьте и прикрепите на холодильник.

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* учения 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, когда ваш мозг начинает уставать. 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 программы!**

   Научиться программировать можно только одним способом: **писать код**. 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.

Что Вам потребуется

Все скриншоты были сделаны в Visual Studio 2012 Express Edition. Мы, конечно, будем обновлять скриншоты, но на сайте Microsoft обычно можно скачивать и более старые версии ПО.

Эта книга написана при помощи **Visual Studio Express 2012 для Windows 8** и **Visual Studio Express 2012 для Windows Desktop**. Именно в этих версиях Visual Studio были сняты все скриншоты, поэтому мы рекомендуем установить именно их. Если у вас установлена версия Visual Studio 2012 Professional, Premium, Ultimate или Test Professional, картинки будут отличаться (но это никак не скажется на выполнении представленных в книге упражнений).

НАСТРОЙКА VISUAL STUDIO 2012 EXPRESS EDITIONS

- Вы можете *бесплатно скачать* **Visual Studio Express 2012 для Windows 8** с сайта Microsoft. **http://www.microsoft.com/visualstudio/eng/downloads.** Она устанавливается так же, как и предыдущие версии.

- Затем вам придется установить **Visual Studio Express 2012 для Windows Desktop**.

Если у Вас нет Windows 8 или Вам не удается запустить Visual Studio 2012

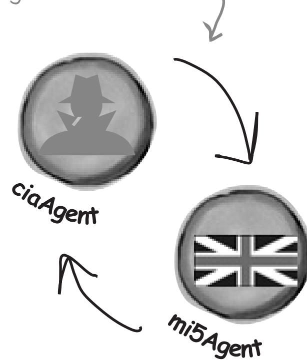
Многие упражнения в книге требуют Windows 8. Но мы понимаем, что среди читателей могут оказаться люди, использующие старые операционные системы, например Windows 2003 или Visual Studio 2010. **Им не стоит беспокоиться** — они смогут выполнить *почти* все упражнения:

- Упражнения в главах 3–9 не требуют Windows 8, для них подойдет Visual Studio 2010 (и даже 2008), просто вид вашего экрана будет отличаться от скриншотов в книге.
- Для остальной части книги вместо приложений Windows 8 **будут создаваться приложения Windows Presentation Foundation (WPF)**. Скачайте с сайта Head First Labs (http://headfirstlabs.com/hfcsharp) PDF-файл с инструкцией.

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.

Для наглядного представления сложных понятий в книге используются диаграммы.



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 некоторые — научиться применять ее на практике. Необязательными являются только «Ребусы в бассейне», но следует помнить, что они хорошо развивают логическое мышление.

Выполняйте все упражнения этого раздела.

Возьми в руку карандаш

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 ставит 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.

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

Упражнение

Код для упражнений из книги можно скачать здесь
`http://www.headfirstlabs.com/books/hfcsharp/`

Упражнения «Мозговой штурм» не имеют ответов.

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

А этим значком помечены дополнительные упражнения для любителей логических задач.

Научные редакторы

Лиза Кельнер

Ребекка Дан-Кран

Крис Барроус

Джонни Халиф

Дэвид Стерлинг

Здесь нет фотографий Джо Албахари, Джея Хилярда, Аяма Синга, Теодоры, Петера Ричи, Билла Метельски, Энди Паркера, Вейна Бредни, Дэйва Мэрдока, Бриджит-Жули Ландерс, Ника Палдино, Девида Стерлинго. Особая благодарность Алану Уетту и остальным читателям, которые нашли опечатки, просочившиеся в предыдущие издания.

Эта книга практически не содержит ошибок, и сказать за это спасибо нужно нашим научным редакторам. Мы безмерно благодарны им за проделанную работу — книга вышла бы с ошибками (включая пару довольно серьезных), если бы у нас не было самой лучшей команды в мире...

Прежде всего хотелось бы сказать большое спасибо **Лизе Кельнер** — это уже наша девятая (!) совместная книга, и именно она повышает читабельность конечного продукта. Спасибо, Лиза! Также хотелось бы поблагодарить **Криса Барроуса, Ребеку Дан-Кран** и **Дэвида Стерлинга** за множество технических рекомендаций, **Джо Албахари** и **Йона Скита** за тщательный и вдумчивый анализ первого издания книги и **Ника Паладино**, сделавшего анализ второго издания.

Крис Барроус работает в компании Microsoft над компиляторами C#. Его специализация — дизайн и реализация различных свойств языка C# 4.0.

Ребекка Дан-Кран — одна из основателей магазина программного обеспечения Semaphore Solutions, специализирующегося на приложениях .NET. Проживает в городе Виктория (Канада) с мужем Тобиасом, детьми Софией и Себастьяном, кошкой и тремя курами.

Дэвид Стерлинг последние три года работает в команде компиляторов Visual C#.

Джонни Халиф — главный архитектор и соучредитель Mural.ly (http://murally.com), проекта, позволяющего создавать виртуальные доски, помещая на них любое содержимое и систематизируя его нужным образом. Он является специалистом по облачным и высокомасштабируемым решениям. Еще он страстный бегун и спортивный болельщик.

Благодарности

Редактор

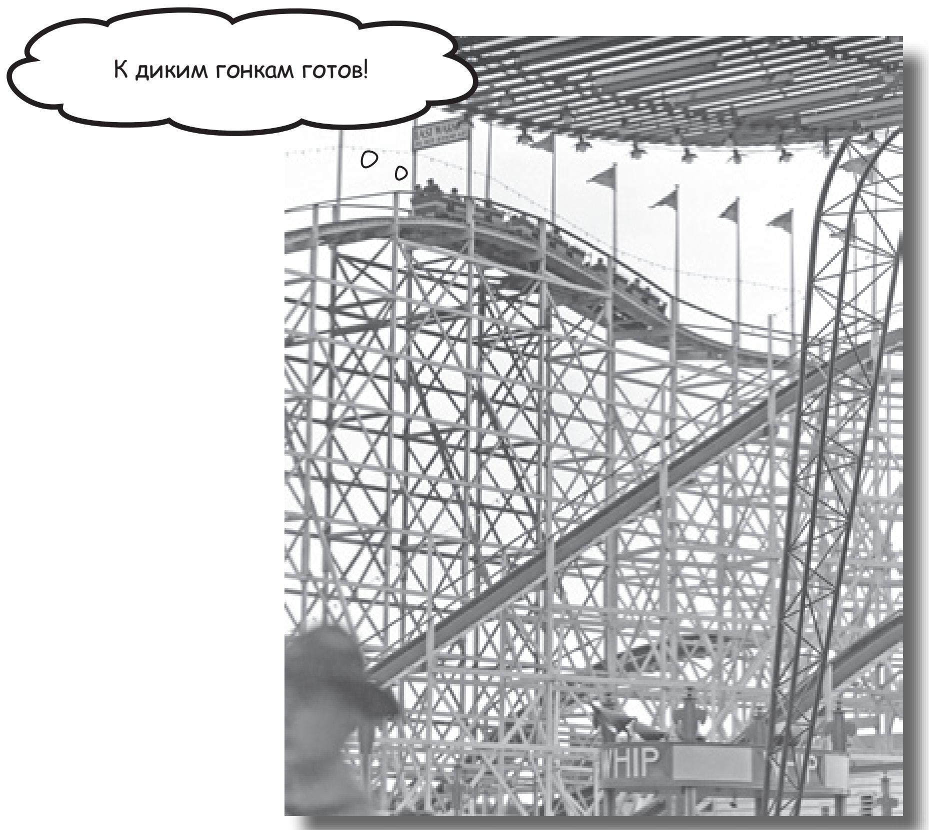
Мы хотим поблагодарить нашего редактора, **Кортни Нэш**, за работу над этой книгой.

Команда издательства O'Reilly

В издательстве O'Reilly очень много сотрудников, которых мы хотели бы поблагодарить, надеемся, что никого не забыли. Особая благодарность **Мелани Ярбрух**, **Эллен Траутман-Зайг**, **Рашель Монаган**, **Рону Билодо** и всем, кто помог подготовить эту книгу к печати. И как всегда, хотелось бы признаться в любви **Мэри Треслер** и сказать, что мы ждем случая поработать с ней снова! Жму руку нашим редакторам и друзьям **Энди Ораму**, **Майку Хендриксону**, **Лори Петрюки**, **Тиму О'Рейли** и **Сандерсу Кляйнфельду**. За то, что вы сейчас читаете эту книгу, следует поблагодарить самую лучшую команду рекламистов: **Марси Хэнон**, **Сару Пейтон** и остальных сотрудников из города Севастополь, штат Калифорния.

1 начало работы с c#

Быстро сделать что-то классное!



Хотите программировать действительно 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# и IDE Visual Studio облегчают и ускоряют процесс написания кода.

IDE (integrated development environment, интегрированная среда разработки) — это программа для редактирования кода, управления файлами и публикации проектов.

Задачи, которые за вас решает IDE

Чтобы поместить на форму кнопку, вам потребуются большие куски повторяющегося кода.

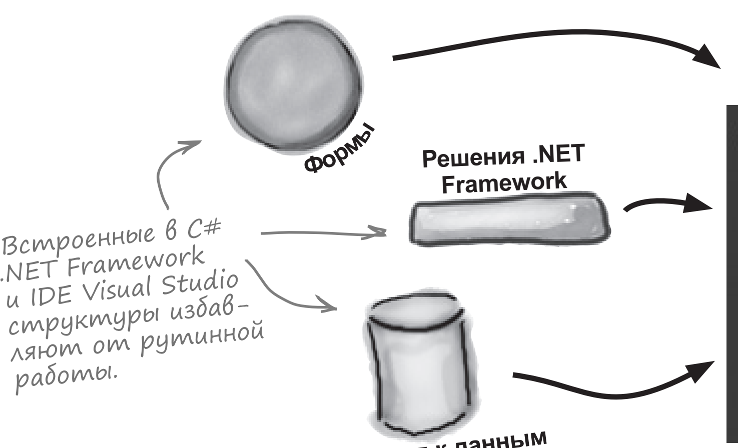
```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
namespace A_New_Program
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

```
private void InitializeComponent()
{
  this.button1 = new System.Windows.Forms.Button();
  this.SuspendLayout();
  //
  // button1
  //
  this.button1.Location = new System.Drawing.Point(105, 56);
  this.button1.Name = "button1";
  this.button1.Size = new System.Drawing.Size(75, 23);
  this.button1.TabIndex = 0;
  this.button1.Text = "button1";
  this.button1.UseVisualStyleBackColor = true;
  this.button1.Click += new System.EventHandler(this.button1_Click);
  //
  // Form1
  //
  this.AutoScaleDimensions = new System.Drawing.SizeF(8F, 16F);
  this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
  this.ClientSize = new System.Drawing.Size(292, 267);
  this.Controls.Add(this.button1);
  this.Name = "Form1";
  this.Text = "Form1";
  this.ResumeLayout(false);
}
```

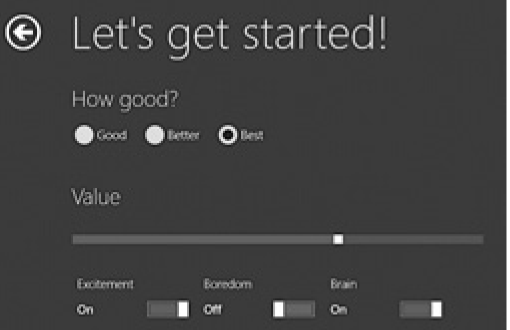
Этот код всего лишь добавляет на форму кнопку. Добавление других элементов может увеличить его в десятки раз.

Преимущества Visual Studio и C#

Язык C#, оптимизированный для программирования в Windows, вместе с Visual Studio позволяет сфокусироваться на непосредственных задачах.



Такое приложение не только лучше выглядит, но и быстрее создается.



C#, IDE Visual Studio многое упрощает

Язык C# и Visual Studio позволят без дополнительных усилий выполнять следующие задачи:

1. **Быстро создавать приложения.** Программировать на C# очень просто. Это мощный, легко осваиваемый язык, а Visual Studio позволяет автоматизировать большинство процессов.

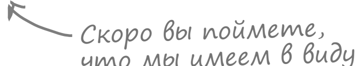
2. **Разрабатывать красивый пользовательский интерфейс.** При создании приложений на C# инструмент Visual Designer в Visual Studio превращает конструирование великолепного пользовательского интерфейса в одну из самых увлекательных задач. Вам больше не требуется тратить часы на написание с нуля графических элементов.

3. **Создавать восхитительные в визуальном отношении программы.** Комбинация C# с визуальным языком разметки XAML, предназначенным для проектирования пользовательских интерфейсов, дает один из самых эффективных инструментов для создания программ, которые выглядят так же великолепно, как работают.

4. **Фокусироваться на решении РЕАЛЬНЫХ задач.** За конечный результат работы, разумеется, отвечаете вы, и только вы. Но IDE позволяет концентрироваться на глобальных вещах, взяв на себя:

   - слежение за всеми проектами;
   - упрощенное редактирование кода;
   - отслеживание графики, аудиофайлов, значков и прочих ресурсов;
   - управление данными.

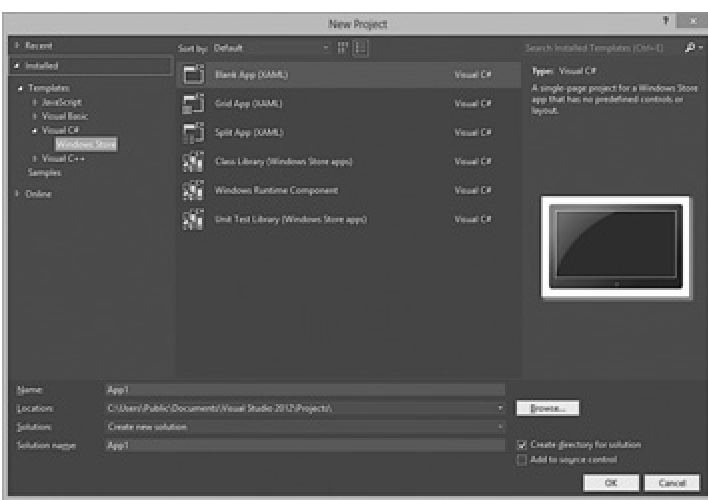
   Теперь вместо рутинного написания кода вы можете потратить время на **создание потрясающих программ**.



Это Вы делаете в Visual Studio

Если данный пункт меню отсутствует, возможно, вы работаете с Visual Studio 2012. Закройте IDE и запустите Visual Studio Express 2012 для Windows 8.

Запустите Visual Studio для Windows 8, если вы этого еще не сделали. Пропустите начальную страницу и выберите в меню **File** команду New Project. Раскройте раздел Visual C#, выделите строчку Windows Store и выберите вариант **Blank App (XAML)**. IDE создаст папку *Visual Studio 2012*, вложенную в папку *Documents*, и поместит ваше приложение в подпапку *Projects* (или в любую папку, которую вы укажете в поле Location).



Будьте осторожны!

В вашей IDE все может выглядеть иначе.

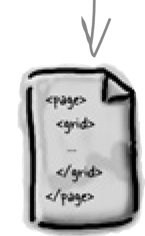
*На рисунке показано окно New Project в **Visual Studio Express Edition** для Windows 8. В версиях Professional и Team Foundation оно может выглядеть по-другому, но принцип один и тот же.*

А это Visual Studio делает за Вас

Команда Save All из меню File сохраняет все открытые файлы, в то время как команда Save — только файл, активный в данный момент.

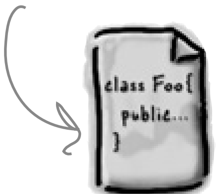
В момент сохранения проекта IDE создает набор файлов, в том числе файлы *MainPage.xaml*, *MainPage.Xaml.cs* и *App.xaml.cs* для новых проектов. Они добавляются в окно Solution Explorer и по умолчанию попадают в папку *Projects\App1\App1*.

Этот файл содержит XAML-код, определяющий пользовательский интерфейс главной страницы.



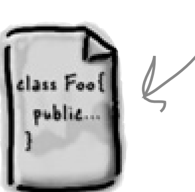
MainPage.xaml

Здесь находится код C#, управляющий поведением главной страницы.



MainPage.Xaml.cs

Код C# из этого файла запускается при загрузке или возобновлении работы приложения.

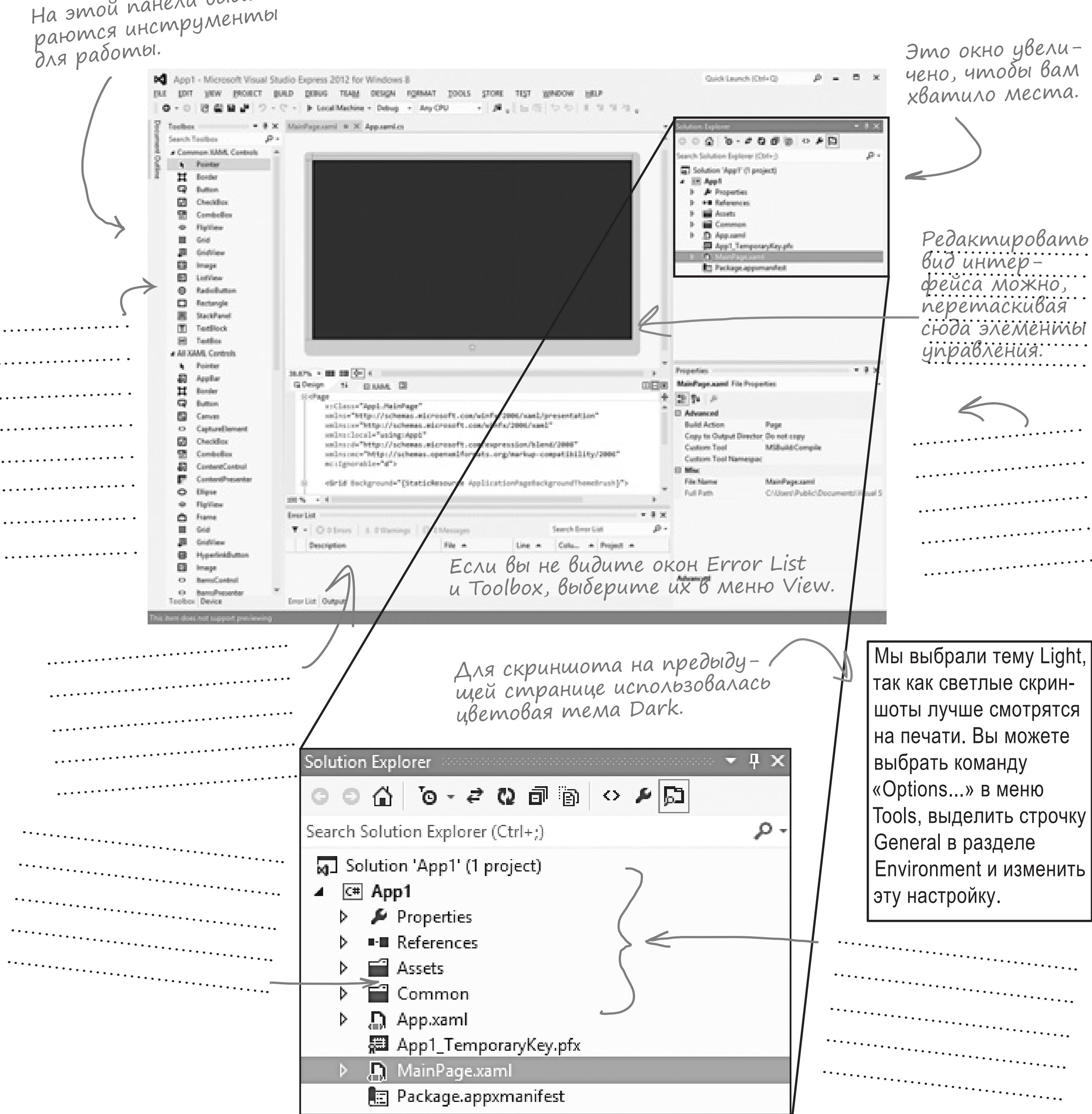


App.xaml.cs

Эти три файла создаются автоматически, как и другие файлы! Их можно увидеть в окне Solution Explorer.

Возьми в руку карандаш

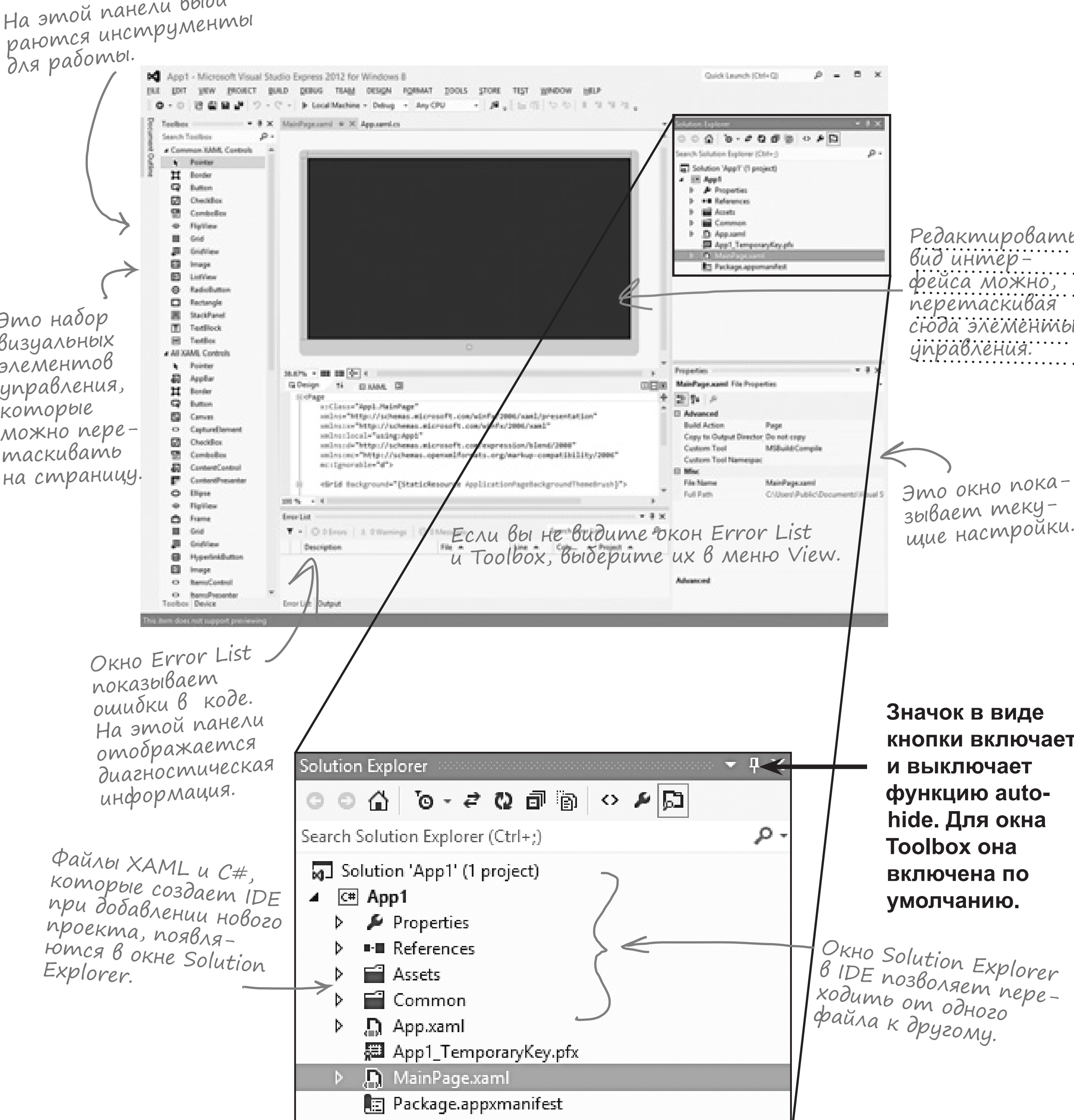
Для начала преобразуем окно программы в показанный ниже вид. Откройте окна Toolbox и Error List, **выбрав в меню View одноименные команды**. Затем выберите цветовую тему **Light** через меню **Options**. Вам должно быть уже известно назначение большинства окон и файлов. Заполните пустые строчки, указав, для чего предназначены соответствующие части IDE, как показано в примере.



Мы выбрали тему Light, так как светлые скриншоты лучше смотрятся на печати. Вы можете выбрать команду «Options...» в меню Tools, выделить строчку General в разделе Environment и изменить эту настройку.

Возьми в руку карандаш
Решение

Здесь вы можете сравнить свое описание назначения различных частей IDE Visual Studio C# с правильными ответами.



Часто Задаваемые Вопросы

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

О: Нет. IDE поможет вам в выборе начальных точек или изменении свойств элементов управления форм, но понять, какую работу должна выполнять программа и как достичь поставленной цели, можете только вы.

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

О: Его можно отредактировать. Если по умолчанию создан не тот код, который вам требуется, достаточно внести в него необходимые изменения — вручную или средствами пользовательского интерфейса IDE.

В: Я загрузил и установил бесплатную версию Visual Studio Express. Достаточно ли этого для выполнения упражнений из данной книги или мне потребуется купить другую версию приложения?

О: Все упражнения из этой книги выполняются в бесплатной версии Visual Studio (которую можно получить на сайте Microsoft). Различия между версиями Express, Professional и Team Foundation никак не повлияют на процесс написания программ на C# и создания полнофункциональных приложений.

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

О: XAML (X произносится как Z и рифмуется с «camel») — это **язык разметки**, позволяющий создавать пользовательские интерфейсы для приложений магазина Windows. Основой XAML является XML (о котором мы тоже поговорим), поэтому те, кому приходилось иметь дело с HTML, получат преимущество. Вот пример **тега** XAML, рисующего серый эллипс:

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

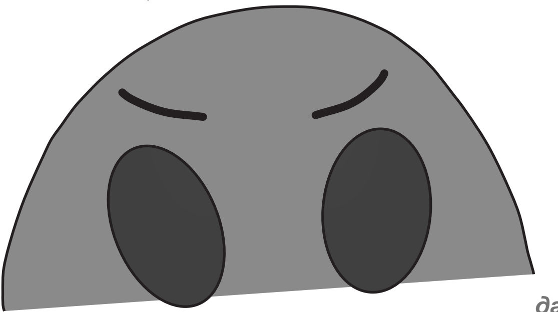
Это тег, на что указывает символ <, за которым следует слово ("Ellipse"). Все вместе составляет начальный тег. Тег Ellipse обладает тремя свойствами: одно задает серый цвет заливки, а два других указывают ширину и высоту. Завершает его символ `/>`, но некоторые теги XAML могут включать в себя другие теги. Данный тег можно превратить в **контейнерный,** заменив `/>` на `>`, добавив другие теги (которые в свою очередь могут содержать теги) и завершив конструкцию **конечным тегом**: `</Ellipse>`. По мере чтения вы узнаете, как работает XAML, и познакомитесь с различными тегами.

В: Окно IDE отличается от показанного на картинке в книге. Что делать?

О: Для перехода к настройкам по умолчанию выберите команду Reset Window Layout в меню Window, затем воспользуйтесь командами меню View→Other Windows, чтобы открыть недостающие окна.

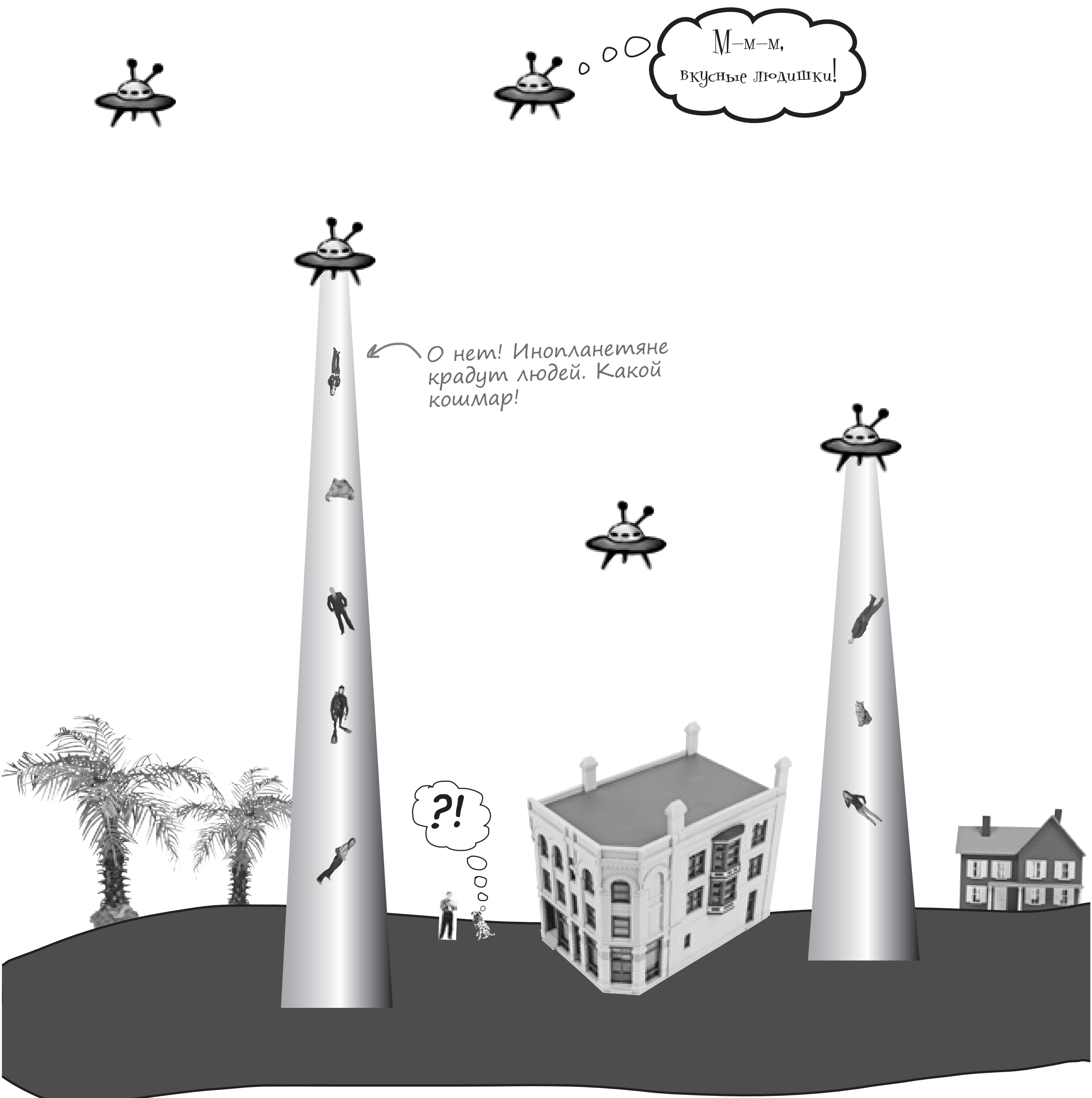
Visual Studio генерирует код, который можно использовать как основу для программы.

Но только <u>вы</u> отвечаете за корректную работу этой программы.



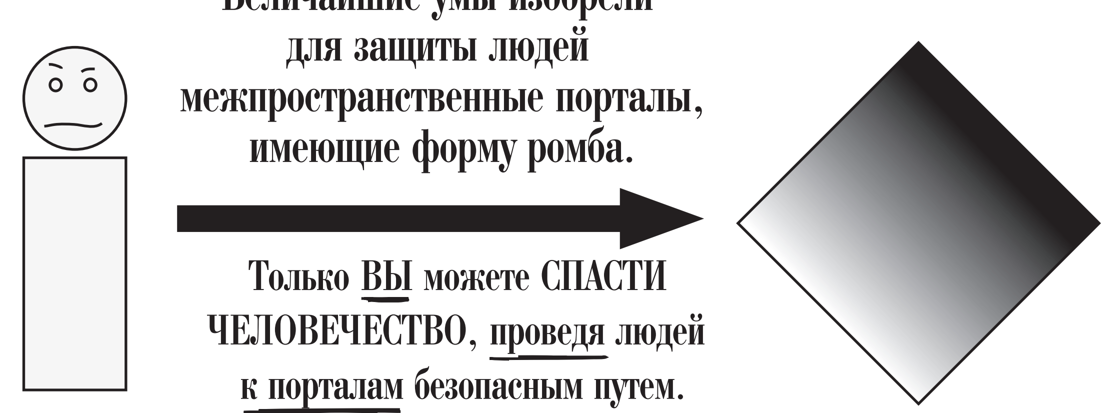
Инопланетяне атакуют!

Вот так новость: злобные инопланетяне начали настоящую охоту на планете Земля, похищая людей для гнусных и отвратительных гастрономических экспериментов. Никто этого не ожидал!



Только ты можешь спасти Землю

Ты — последняя надежда человечества! Людям планеты Земля нужно, чтобы ты **создал удивительное приложение C#**, которое скоординирует их действия по спасению от инопланетной угрозы. Готов принять вызов?



Вот что вам нужно сделать

Вам потребуется приложение с графическим интерфейсом, объектами и исполняемым файлом. На первый взгляд кажется, что предстоит большая работа, но все будет готово уже к концу этой главы. Кроме того, вы узнаете как спроектировать страницу в IDE и добавить к ней код C#.

Вот как выглядит структура нашего приложения:

Вы строите приложение, на главной странице которого будет множество элементов управления.

Для реализации игрового процесса приложение использует элементы управления.

Главная страница и контейнеры XAML

Элементы управления UI Windows

Main page

Grid

Canvas

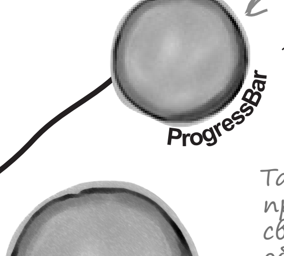
Rectangle

ProgressBar

StackPanel

Ellipse

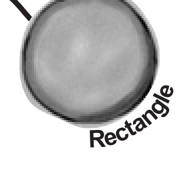
Rectangle



Через эти элементы управления в приложении реализуются цель, к которой перетаскиваются люди, и таймер.

Таймер Target проверяет свойства объекта ProgressBar для контроля времени.

Каждый человечек, которого нужно спасти, рисуется через объект StackPanel, состоящий из эллипса и прямоугольника.



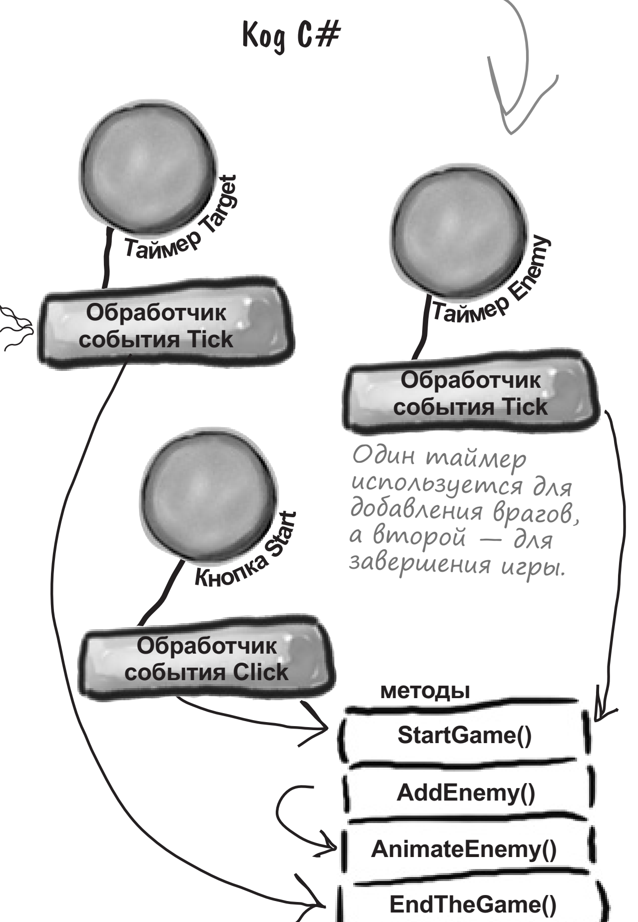
Для компоновки главной страницы применяется сетка. Игровой процесс будет происходить в центральной ячейке — для этого мы воспользуемся элементом Canvas.

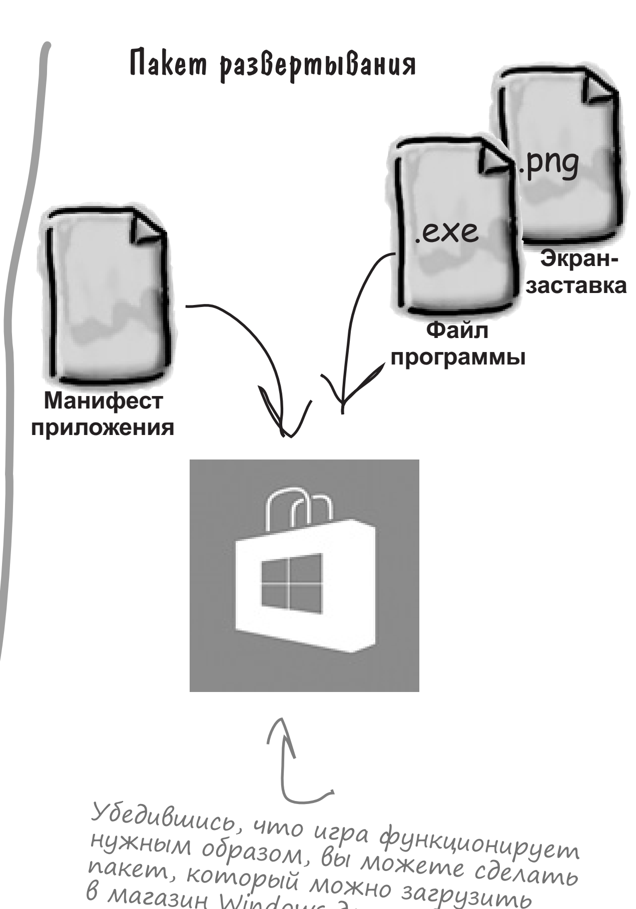
Для построения приложения используется код двух видов. Пользовательский интерфейс проектируется с помощью XAML (Extensible Application Markup Language). Затем добавляется код C#, заставляющий игру работать. С XAML вы подробно познакомитесь во второй половине книги.

Нет Windows 8?

Ничего страшного.

Проекты из первых двух глав и второй половины книги созданы в *Visual Studio 2012 для Windows 8*, но эта ОС установлена далеко не у всех читателей. К счастью, большинство приложений можно создать в Windows Presentation Foundation (WPF), совместимом с более ранними версиями Windows. Инструкцию на английском языке можно скачать с сайта *http://www.headfirstlabs.com/hfcsharp*...



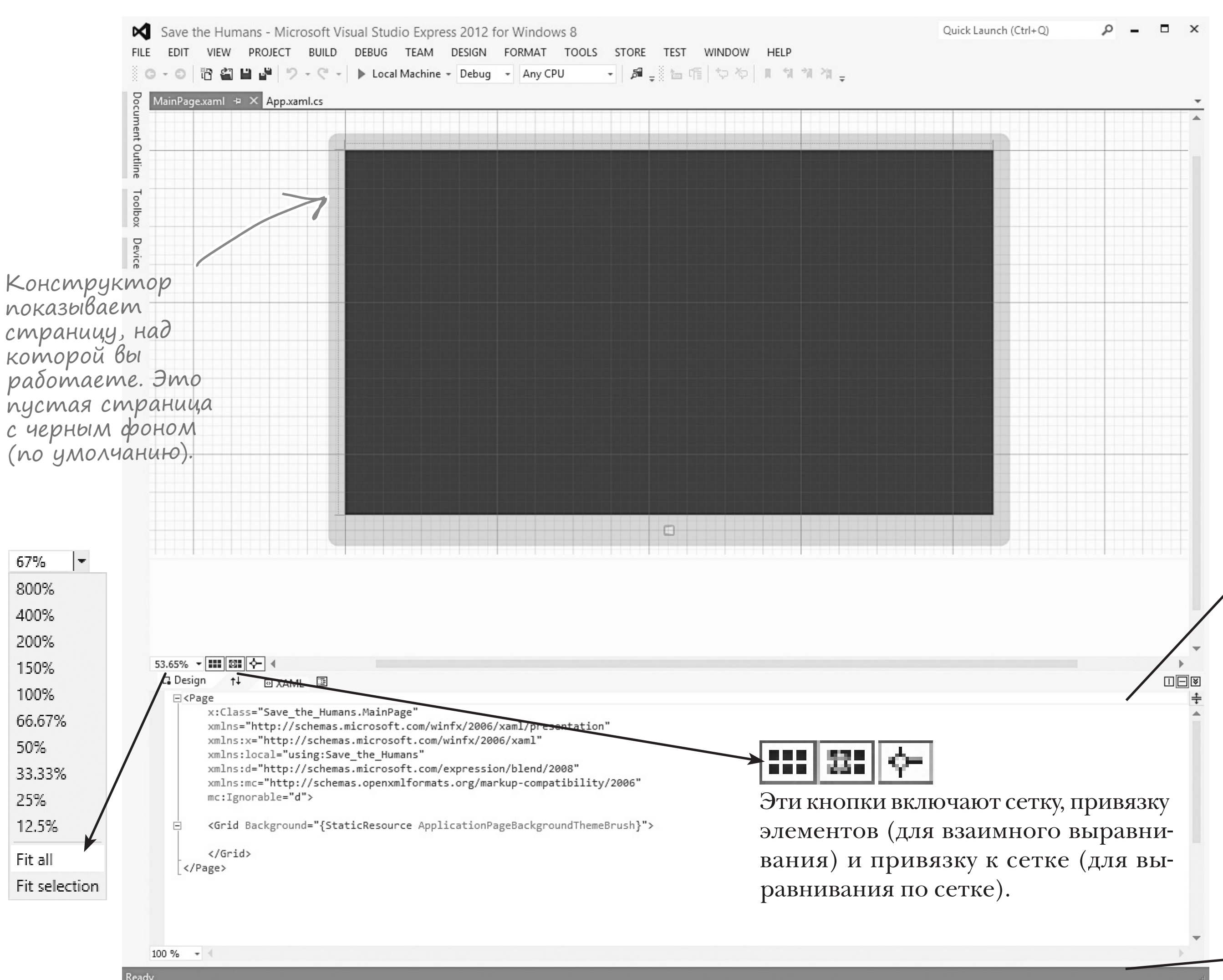


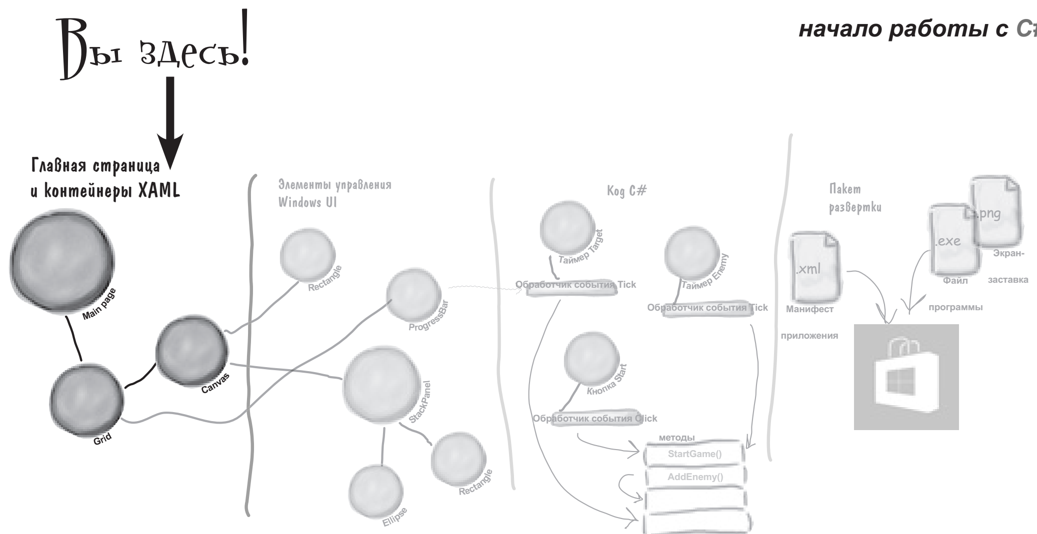
Начнем с пустого приложения

Все приложения начинаются с нового проекта. Выберите в меню File команду New Project. Выделите Visual C#→Window Store и укажите в качестве типа проекта **Blank App (XAML)**. Присвойте проекту имя **`Save the Humans`**.

Без расширения «.cs» ваш файл может случайно превратиться в программу JavaScript, Visual Basic или Visual C++. Чтобы сохранить проект «Save the Humans», удалите папку предыдущего проекта.

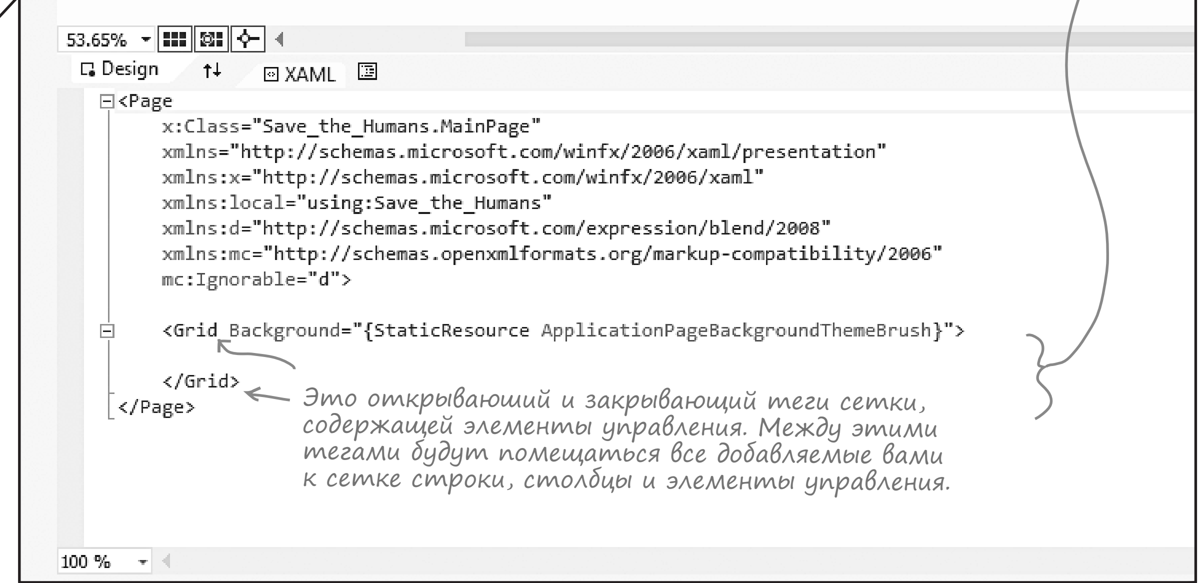
1. Начнем мы с окна **Designer**. Дважды щелкните на строчке *MainPage.xaml* в окне Solution Explorer. Найдите список масштабов в нижнем левом углу окна конструктора и выберите вариант «Fit all», чтобы уменьшить изображение.





В нижней части окна Designer находится код XAML. Как видите, ваша «пустая» страница не так уж пуста, на ней располагается **сетка XAML**. На страницах HTML и в документах Word аналогами сеток являются таблицы. Сетка позволяет создавать страницы, умеющие подстраиваться под размер и форму экрана.

Это код XAML для пустой сетки, которую для вас сгенерировала IDE. Через минуту мы добавим к ней строки и столбцы.



Эта часть проекта содержит шаги с (1) по (5).

Перевернем страницу и продолжим! →

Хотите изучить WPF? Без проблем!

Большинство приложений для магазина Windows в этой книге **можно построить в системе WPF** (Windows Presentation Foundation), совместимой с Windows 7 и более ранними версиями. Скачайте руководство по WPF (на английском языке) с нашего сайта: *http://headfirstlabs.com/hfcsharp*.

На следующих страницах вы познакомитесь с множеством функций Visual Studio, ведь мы будем использовать ее как обучающий инструмент. Именно с ее помощью вы освоите C#. Это самый эффективный способ запоминания информации!

② Странице понадобится заголовок, не так ли? А еще поля. Все это можно добавить с помощью XAML, хотя существует и более простой способ придать программе типичный для приложения магазина Windows вид.

В окне Solution Explorer щелкните правой кнопкой мыши на строчке ▷ MainPage.xaml и выберите команду Delete для **удаления страницы *MainPage.xaml***:

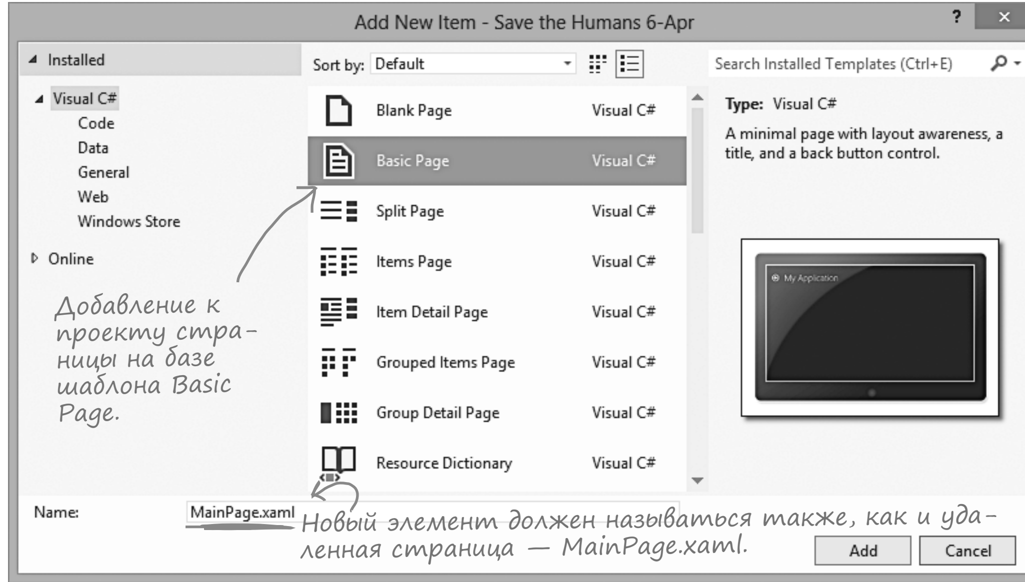
Если вы не видите окна Solution Explorer, выберите одноименную команду в меню View. Кроме того, команды меню Window позволяют сбросить макет окон.

При работе над приложением для магазина Windows главная страница часто заменяется одним из шаблонов Visual Studio.

Если при создании проекта вы дали ему имя, отличное от "Save the Humans", именно его вы увидите в окне Solution Explorer.

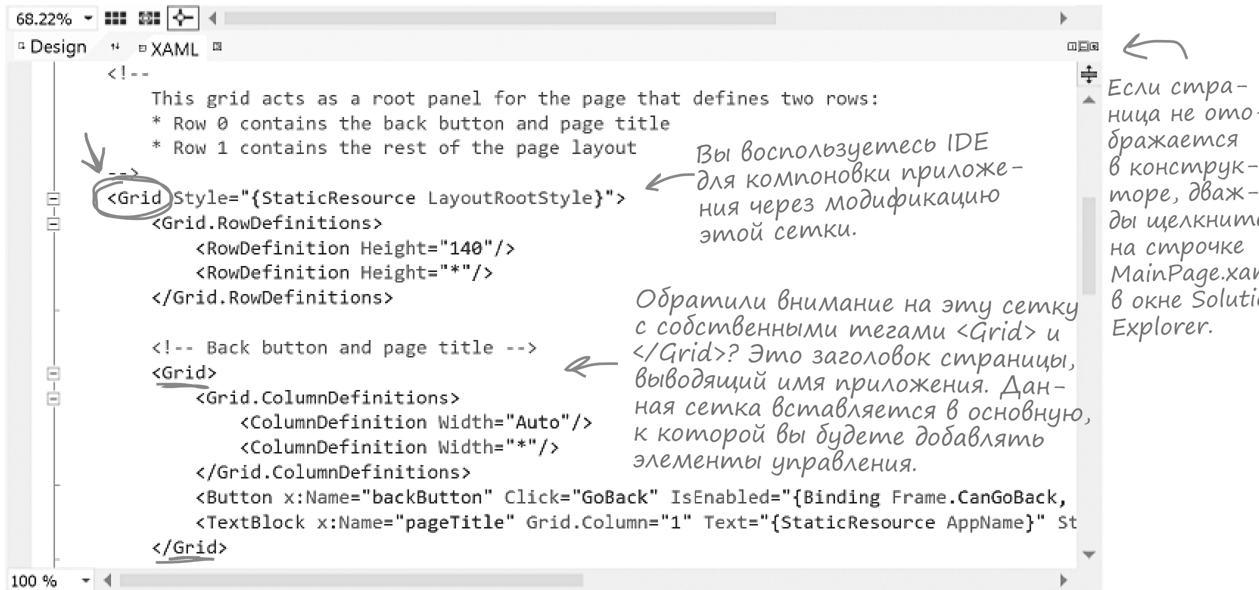
③ Теперь нужно заместить главную страницу. В окне Solution Explorer щелкните правой кнопкой мыши на строчке ◢ **Save the Humans** (это вторая сверху строка) и выберите в появившемся меню команду **Add→New Item...** :

IDE откроет окно Add New Item. Выберите вариант **Basic Page** и присвойте ему имя **MainPage.xaml**. Щелкните на кнопке **Add** для добавления замещающей страницы.



IDE предложит добавить отсутствующие файлы, **нажмите *Yes***. Дождитесь окончания загрузки. Могут появиться надписи Invalid Markup или Build the Project to update Design view. Выберите в меню Build команду **Rebuild Solution** для обновления окна конструктора.

Давайте исследуем добавленную страницу *MainPage.xaml file*. Промотайте вкладку XAML в окне конструктора до кода XAML, — это сетка, которая послужит основой вашей программы:



4. Нам требуется сетка из двух строк и трех столбцов (плюс строка заголовка, имеющаяся в шаблоне), с одной большой ячейкой для игровой области в центре. Наведите указатель мыши на границу, чтобы появились линия и треугольник:

Save the Humans - MainPage.xaml

My Application

Если вы не видите цифр 140 и 1*, щелкните рядом с границей левой кнопкой мыши.

Наведитесь на границу сетки, чтобы появились линия и оранжевый треугольник...

...и щелкните левой кнопкой мыши для создания нижнего края сетки.

После добавления строки цвет линии станет голубым, а рядом появится значение ее высоты. Рядом с центральной линией вместо 1* появится большая цифра со звездочкой.

125*

32*

Приложения магазина Windows должны корректно выглядеть на экранах любого размера и ориентации.

Использование сетки со столбцами и строками позволяет приложению автоматически корректировать свой вид.

Часто задаваемые вопросы

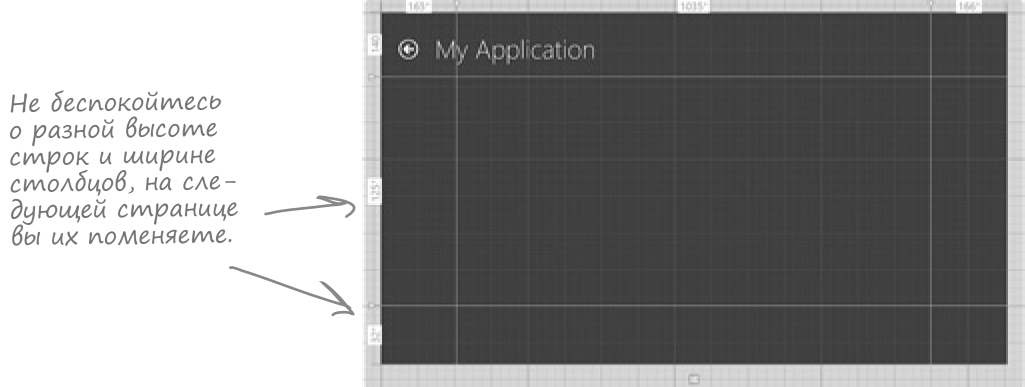
В: Мне кажется, что в сетке уже достаточное количество строк и столбцов. Что это за серые линии?

О: Серые линии — это направляющие, которые помогают при размещении на странице элементов управления. Их можно отключить и включить щелчком на кнопке ▦. Вы не увидите этих линий при запуске приложения вне Visual Studio. Линии же, которые создаются щелчком, меняют код XAML, что отражается на поведении приложения при компиляции и выполнении.

В: Подождите минутку. Я хотел изучать C#. Почему я трачу столько времени на XAML?

О: Дело в том, что приложения магазина Windows, написанные на C#, практически всегда начинаются с пользовательского интерфейса, который реализуется средствами XAML. Именно поэтому в Visual Studio добавлен редактор XAML, позволяющий создавать потрясающие интерфейсы. В книге вы узнаете, как при помощи C# строятся два других типа программ — настольные и консольные приложения. Для них вам не потребуется XAML, но эта информация даст вам более глубокое понимание принципов программирования на C#.

5 Теперь щелкните рядом с верхней границей шаблона, но создайте два столбца, — своего рода маленькие поля слева и справа. Точная высота строк и ширина столбцов не имеют значения. Через минуту мы все равно отредактируем этот параметр.



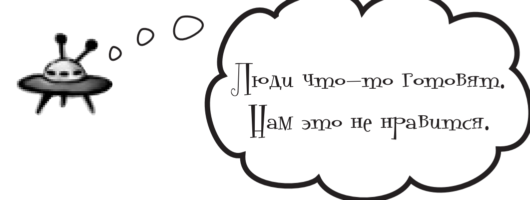
Когда все будет готово, найдите в окне XAML сетку, которую мы рассматривали на предыдущей странице. Теперь ширина столбцов и высота строк совпадают с цифрами в верхней и в боковой частях шаблона.

```
Design    XAML
<!--
    This grid acts as a root panel for the page that defines two rows:
    * Row 0 contains the back button and page title
    * Row 1 contains the rest of the page layout
-->
<Grid Style="{StaticResource LayoutRootStyle}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="165*"/>
        <ColumnDefinition Width="1035*"/>
        <ColumnDefinition Width="166*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="140"/>
        <RowDefinition Height="125*"/>
        <RowDefinition Height="32*"/>
    </Grid.RowDefinitions>
100 %
```

Это ширина левого столбца, созданного на шаге 5, — она совпадает с шириной в конструкторе, так как IDE сгенерировала этот XAML-код для вас.

Строки и столбцы сетки добавлены!

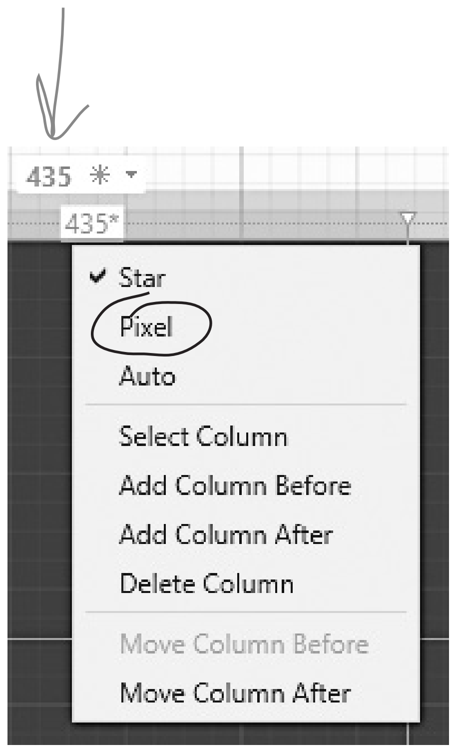
Сетки XAML являются **контейнерными элементами управления**. Строки и столбцы определяют ячейки, и каждая может содержать элементы XAML с кнопками, текстом и формами. Сетка хорошо подходит для компоновки страницы, ведь размер ее строк и столбцов может меняться в зависимости от размеров экрана.



Настройка сетки

Наше приложение должно работать на самых разных устройствах, и сетка позволяет этого добиться. Можно указать размер строк и столбцов в пикселях. А можно выбрать вариант **Star**, при котором размеры будут меняться пропорционально в зависимости от размера и ориентации экрана.

Меняя эту цифру, вы редактируете сетку и ее XAML-код.



1. **Установите ширину левого столбца.**
Наведите указатель на число над первым столбцом и дождитесь появления раскрывающегося списка. Выберите вариант Pixel, измените значение на `160`. Ширина столбца будет выглядеть так:
160

2. **Повторите для правого столбца и нижней строки.**
Задайте правому столбцу и нижней строке размер, выбрав вариант Pixel и введя в поле значение `160`.

Для фиксации ширины или высоты столбца или строки выбирайте вариант Pixel. Вариант Star позволяет строкам и столбцам менять размер вместе с сеткой. Эта настройка в конструкторе изменяет параметры Width и Height в коде XAML. Удаление свойства Width или Height аналогично присвоению значения 1*.

Расслабьтесь

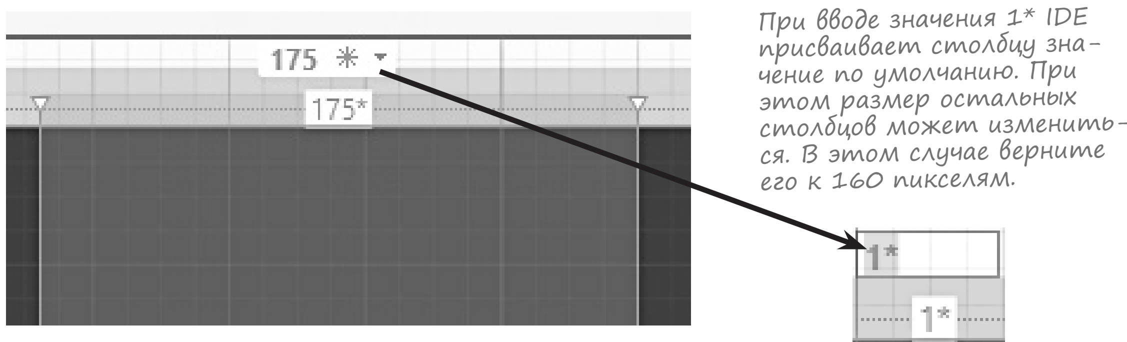
Не страшно, если вы *пока* не умеете проектировать приложения!

О том, как проектировать приложения, мы поговорим позже. Пока же мы показываем вам процесс создания игры. К концу книги вы будете знать, зачем нужно то или иное действие!

3 Центральному столбцу и центральной строке присвойте размер по умолчанию 1* (если это еще не сделано).

Щелкните на цифре над центральным столбцом и введите 1. В меню должен быть выбран вариант Star. Посмотрите на остальные столбцы и убедитесь, что IDE не поменяла их размер. Если же это случилось, снова введите значение `160`.

XAML и C# чувствительны к регистру! Поэтому копируйте код внимательно.



4 Посмотрите на код XAML!

Щелчком выделите сетку и посмотрите в окно XAML на полученный код.

```
<!--
    This grid acts as a root panel for the page that defines two rows:
    * Row 0 contains the back button and page title
    * Row 1 contains the rest of the page layout
-->
<Grid Style="{StaticResource LayoutRootStyle}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="160"/>
        <ColumnDefinition/>
        <ColumnDefinition Width="160"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="140"/>
        <RowDefinition/>
        <RowDefinition Height="160"/>
    </Grid.RowDefinitions>
```

Строка <Grid ... > показывает, что весь идущий следом код является частью сетки.

Так задается столбец для сетки XAML. Мы добавили три столбца и три строки, а значит, три тега ColumnDefinition и три тега RowDefinition.

Верхний ряд высотой 140 пикселей является частью добавленного нами шаблона Basic Page.

Для задания свойств Width и Height мы использовали раскрвающиеся списки столбцов и строк.

Через минуту мы добавим к сетке элементы управления, которые появятся после определения строк и столбцов.

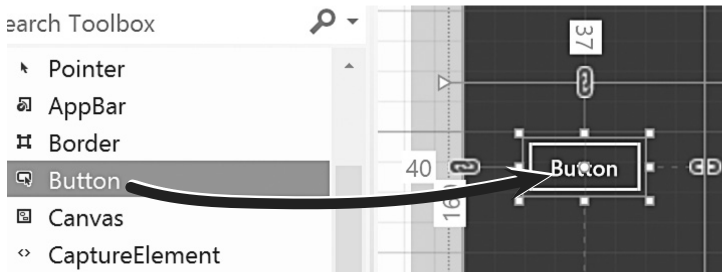
Добавим к сетке элементы управления

Если панель элементов отсутствует, откройте ее командой меню View и закрепите, щелкнув на средней кнопке.

Toolbox
Search Toolbox
Common XAML Controls
Pointer
Border
Button
CheckBox
ComboBox
FlipView
Grid
GridView
Image
ListView
RadioButton
Rectangle
StackPanel
TextBlock
TextBox

Рассматривая приложения, вы обращали внимание на все эти кнопки, текст, картинки, индикаторы, ползунки, раскрывающиеся списки и меню? Они называются **элементами управления**, и пришло время добавить их к нашей сетке.

1 Раскройте раздел Common XAML Controls панели элементов и перетащите элемент Button в **нижнюю левую ячейку** сетки.



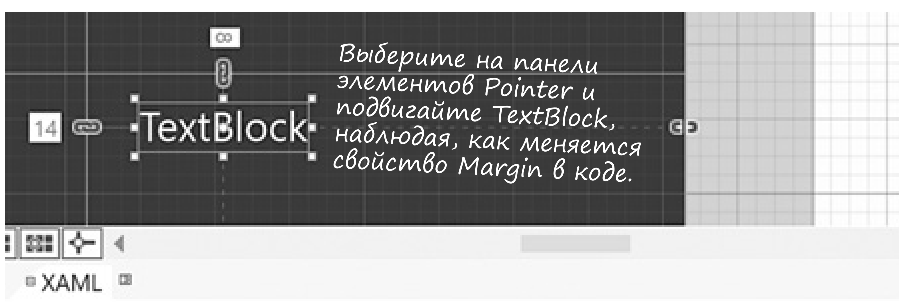
Посмотрите, какой **тег XAML** сгенерировала IDE. Величина полей у вас может отличаться, как и порядок перечисления свойств.

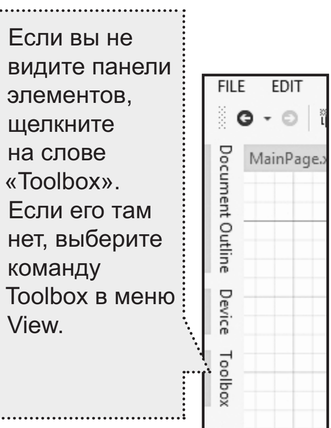
С открывающего тега начинается XAML-код кнопки.

Это свойства. Каждое из них имеет имя, за которым после знака равенства следует значение.

```
<Button Content="Button" HorizontalAlignment="Left"
        Margin="60,72,0,0" Grid.Row="2" VerticalAlignment="Top"/>
```

2 Перетащите элемент TextBlock в **нижнюю правую ячейку** сетки. Попробуйте понять, в какой столбец и строку попал элемент управления.

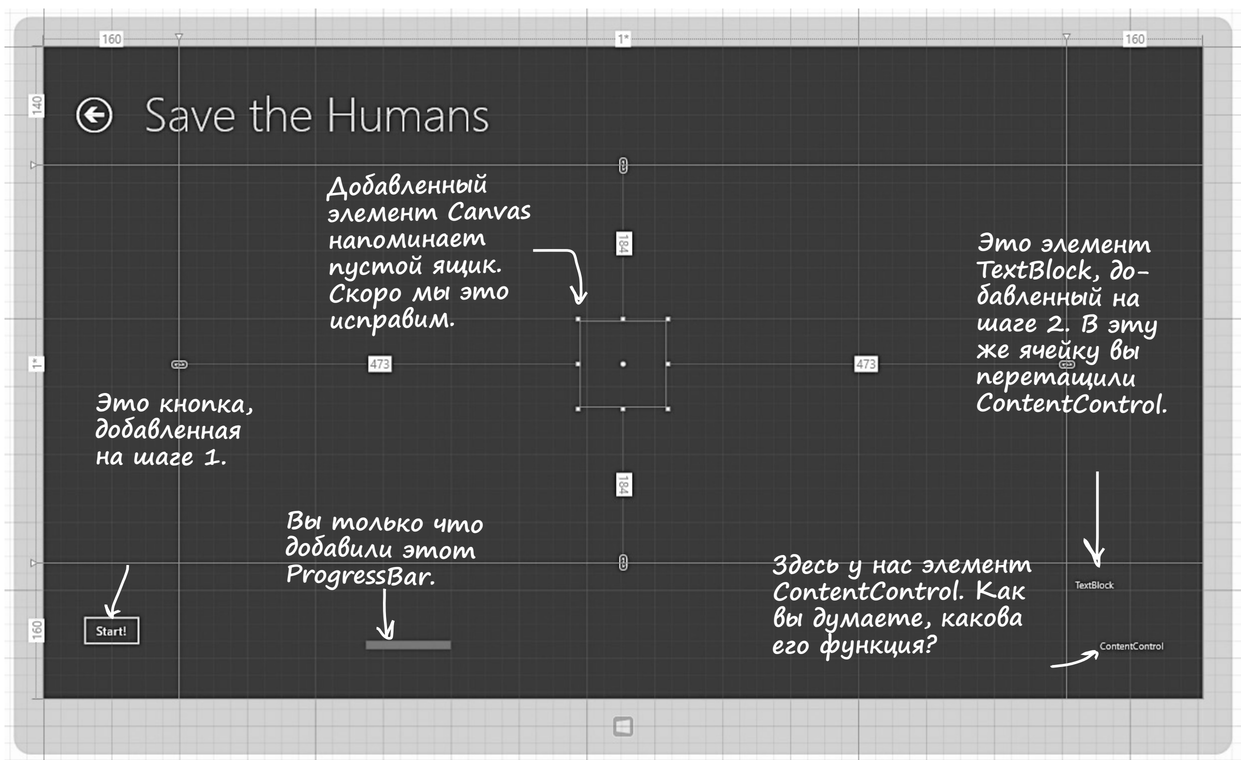





```
<TextBlock Grid.Column="2" HorizontalAlignment="Left"
           Margin="14,8,0,0" Grid.Row="2" TextWrapping="Wrap"
           Text="TextBlock" VerticalAlignment="Top"/>
```

Для простоты чтения мы добавили переносы строк. Вы тоже можете это сделать. Попробуйте!

(3) Раскройте раздел ◢ All XAML Controls панели элементов и перетащите в нижнюю центральную ячейку элемент ProgressBar, в нижнюю правую — ContentControl (он должен оказаться **ниже** вставленного в ячейку элемента TextBlock), а в центральную — Canvas. Теперь страница снабжена элементами управления (не обращайте внимания на их расположение, этим мы сейчас займемся):

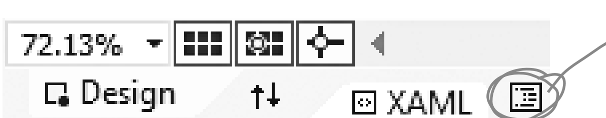


(4) У нас выделен добавленный последним элемент Canvas. (Если это не так, выделите его.) Посмотрите в окно XAML:

```
<Canvas Grid.Column="1" Grid.Row="1" HorizontalAlignment="Left" Height="100"...
```

Там показан тег XAML для элемента Canvas. Он начинается с `<Canvas` и заканчивается `/>`. Между ними располагаются свойства `Grid.Column="1"` (элемент в центральном столбце) и `Grid.Row="1"` (элемент в центральной строке). Последовательно выделяйте элементы управления *в сетке и в окне XAML.*

При перетаскивании элемента с панели в конструктор автоматически генерируется XAML-код.



Щелкните на кнопке. Откроется окно Document Outline. Можете угадать его назначение? Мы скоро займемся этим окном.

Для завершения редактирования пользуйтесь клавишей Escape. Это работает и для других действий.

Меняем вид элементов управления

Visual Studio дает вам контроль над элементами управления. Их вид и даже поведение можно редактировать в окне **Properties**.

Меняем текст кнопки.

Щелкните правой кнопкой мыши на кнопке в ячейке сетки и выберите команду **Edit Text**. Измените текст на: `Start!` и посмотрите на XAML-код кнопки:

```
<Button Content="Start!" HorizontalAlignment="Left" VerticalAlignment="Top" ...
```

При редактировании текста кнопки обновляется свойство Content.

Воспользуйтесь полем Name, чтобы поменять название элемента на startButton.

2 Модифицируем кнопку в окне Properties.

Выделите кнопку и посмотрите на расположенное в нижнем правом углу IDE окно Properties. Присвойте элементу имя `startButton` и расположите его по центру ячейки. Завершив преобразования, **щелкните на элементе правой кнопкой мыши и выберите команду View Source** для перехода к тегу `<Button>` в окне XAML.

Возможно, вам понадобится раскрыть разделы Common и Layout.

Эти маленькие квадратики показывают, было ли задано свойство. Пустой квадратик означает, что значение по умолчанию не редактировалось.

После выбора команды "Edit Text" для редактирования текста на кнопке обновляется свойство Content.

Кнопки и присваивают свойствам HorizontalAlignment и VerticalAlignment значение "Center" и центрируют элемент в ячейке.

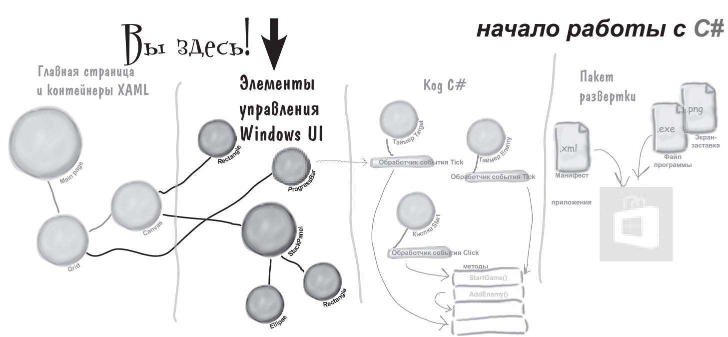
При перетаскивании кнопки на страницу IDE пользовалась свойством Margin для помещения элемента в определенную точку ячейки. Щелкните на квадратике ■ и выберите в меню команду Reset, чтобы обнулить поля.

```
<Button x:Name="startButton"
        Content="Start!"
        Grid.Row="2"
        HorizontalAlignment="Center"
        VerticalAlignment="Center"/>
```

Вернитесь в окно XAML, чтобы увидеть обновленный код!

Свойства могут быть перечислены и в другом порядке.

Последнее действие можно отменить командой Edit→Undo (или нажать Ctrl-Z). При выделении неверного элемента достаточно выбрать в меню Edit команду Select None или нажать Escape. Если элемент находится внутри контейнера StackPanel или Grid, это приведет к выделению контейнера, и нажатие Escape придется повторить.



3 Меняем текст заголовка.

Щелкните правой кнопкой мыши на заголовке страницы («My Application») и выберите команду View Source для перехода к коду XAML текстового блока. Найдите в окне XAML свойство `Text`:

```
Text="{StaticResource AppName}"
```

Минуточку! Тут нет слов «My Application»!

Шаблон Blank Page использует для отображаемого в верхней части страницы имени **статического ресурса** `AppName`. Найдите в XAML-коде раздел `<Page.Resources>`, содержащий следующий код:

```
<x:String x:Key="AppName">My Application</x:String>
```

Вставьте вместо «My Application» название нашего приложения:

```
<x:String x:Key="AppName">Save the Humans</x:String>
```

Сверху появится корректный текст:

Save the Humans

На черную кнопку пока не обращайте внимания. О ней и о статических ресурсах мы поговорим в главе 14.

4 Меняем текст и стиль TextBlock.

Воспользуйтесь командой Edit Text и задайте для элемента TextBlock текст `Avoid These`. Затем щелкните на элементе правой кнопкой мыши, выберите Edit Style, затем Apply Resource и **SubheaderTextStyle**, чтобы увеличить текст.

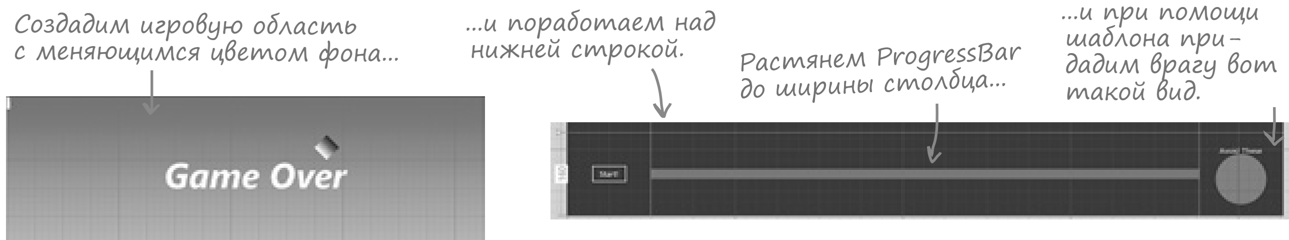
5 Объединяем элементы TextBlock и ContentControl.

Элемент TextBlock должен располагаться вверху ячейки, а элемент ContentControl — внизу. **Одновременно выделите оба элемента и щелкните правой кнопкой мыши**. Выберите в меню команду Group Into, а затем — StackPanel. В форме появится новый элемент управления **StackPanel**.

Подобно Grid и Canvas он служит контейнером для других элементов, поэтому в форме его не видно. Так как элемент TextBlock располагался сверху, а ContentControl — снизу, то StackPanel создается в виде вертикальной строки. Выделите этот элемент, щелкните правой кнопкой мыши и выберите команду Reset Layout, а затем All для быстрого сброса свойств. Таким же образом сбросьте макеты TextBlock и ContentControl. Для выравнивания последнего элемента по вертикали и горизонтали укажите значение Center.

Элементы управления игрой

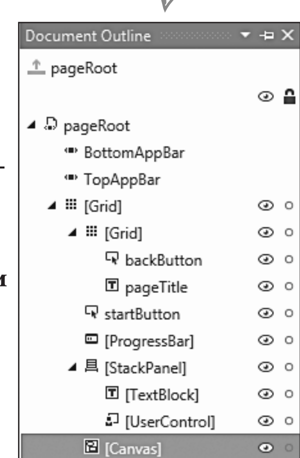
Элементы управления отвечают не только за вид заголовка и подсказок. Именно они являются двигателем игры. Поэтому добавим элементы, с которыми будет взаимодействовать пользователь:



1 Обновляем управляющий элемент ProgressBar.
Щелкните правой кнопкой мыши на элементе ProgressBar в нижней центральной ячейке, выберите команду **Reset Layout**, а затем **All**, чтобы вернуть всем свойствам значения по умолчанию. В поле `Height` в разделе Layout окна Properties введите **`20`**. IDE сначала уберет из кода XAML все связанные с компоновкой свойства, а затем добавит параметр `Height`:

```
<ProgressBar Grid.Column="1" Grid.Row="2" Height="20"/>
```

Открыть окно Document Outline можно командой View→Other Windows.



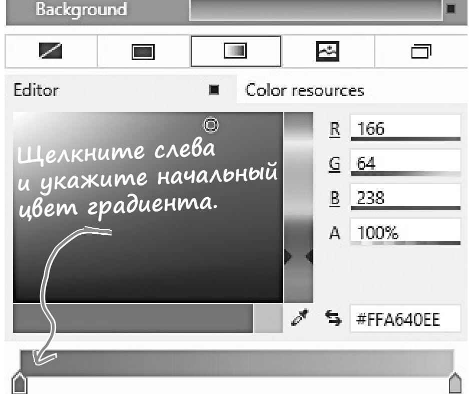
2 Превращаем элемент Canvas в игровую область.
Перетащенный в центральную ячейку элемент Canvas невидим, но найти его просто. Щелкните на кнопке [значок] над окном XAML и откройте окно **Document Outline**. Теперь остается только выделить [Canvas].

Убедитесь, что элемент Canvas выделен, и **воспользуйтесь полем Name** в окне Properties, чтобы присвоить ему имя `playArea`.

Сразу после этого надпись [Canvas] в окне Document Outline сменится надписью playArea.

Document Outline

Открыть окно Document Outline можно щелчком на расположенной сбоку вкладке.



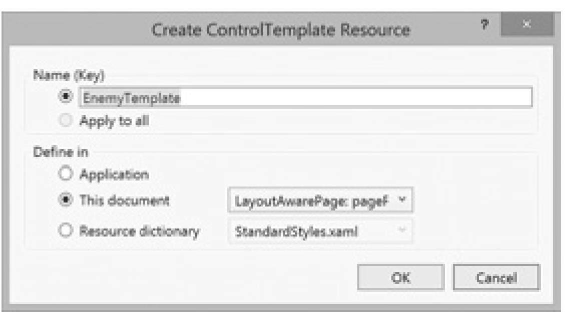
Закройте Document Outline. При помощи кнопок [значок] и [значок] в окне Properties укажите вертикальное и горизонтальное выравнивание как Stretch, сбросьте значения параметра `Margin` и щелкните на кнопке [значок] для параметров `Width` и `Height`, чтобы присвоить им значение Auto. В поле Column введите 0, а в расположенное рядом поле ColumnSpan — 3.

В разделе **Brush** окна Properties щелкните на [значок], чтобы сделать **градиентную заливку**. Выберите начальный и конечный цвета градиента в редакторе цветов.

3 Создаем шаблон врага.

В процессе игры на экране появится множество врагов, и все они должны выглядеть одинаково. К счастью, в XAML существуют **шаблоны**, позволяющие создать набор одинаковых элементов управления. Щелкните правой кнопкой мыши на строке ContentControl в окне Document Outline и выберите команду **Edit Template**, а затем **Create Empty...** Присвойте шаблону имя `EnemyTemplate`, и он добавится к XAML-коду.

Нам придется поработать «вслепую»: конструктор не показывает шаблон, пока мы не добавим элемент управления и не укажем его размеры. Не волнуйтесь, ошибочное действие всегда можно отменить.



Если выделение с сетки было снято, его можно вернуть в окне Document Outline.

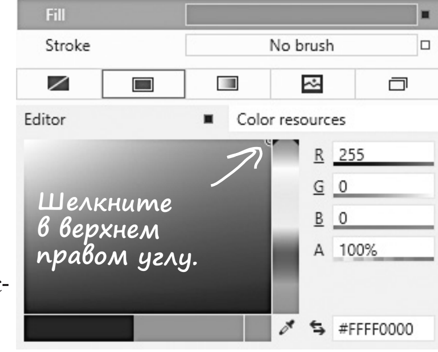
Сейчас выделен созданный шаблон. Сверните окно Document Outline, чтобы оно не закрывало Toolbox. *Шаблон* ***невидим****, но это мы скоро исправим. Если вы случайно сняли с шаблона выделение,* ***его можно вернуть****, открыв окно Document Outline, щелкнув правой кнопкой мыши на строке Content Control и выбрав команду Edit Template→Edit Current.*

Щелчок в поле конструктора до появления эллипса снимет выделение с шаблона. Постарайтесь этого не допустить.

4 Редактируем шаблон врага.

Добавим к шаблону красную окружность:

- Дважды щелкните в панели элементов на строчке Ellipse.
- Присвойте свойствам `Height` и `Width` эллипса значение `100`, после чего он появится в ячейке.
- Сбросьте свойства `HorizontalAlignment`, `VerticalAlignment` и `Margin`, щелкнув на квадратиках рядом с соответствующими полями и выбрав команду Reset.
- В разделе Brush выберите одноцветную кисть.
- Щелкните в верхней части цветовой полосы, а затем в верхнем правом углу квадрата, чтобы сделать эллипс красным.



Теперь XAML код для элемента ContentControl выглядит так:

```
<ContentControl Content="ContentControl"  HorizontalAlignment="Center"
                VerticalAlignment="Center" Template="{StaticResource EnemyTemplate}"/>
```

Посмотрите на код в окне XAML и найдите место, где определяется шаблон EnemyTemplate. Он должен находиться сразу под ресурсом AppName.

5 Редактируем элементы управления StackPanel и TextBlock в окне Document Outline.

Вернитесь в окно Document Outline (если вы видите наверху EnemyTemplate (ContentControl Template), щелкните на кнопке). Выделите элемент StackPanel, убедитесь, что выравнивание по горизонтали и вертикали имеет значение center, и уберите поля. То же самое проделайте с элементом TextBlock.

Компоновка формы почти закончена! Еще немного на следующей странице...

❻ Добавляем человека.

Добавить человека можно двумя способами – как описано в следующих трех абзацах или вставкой четырех строчек XAML-кода. Выбор за вами!

Выделите элемент Canvas и дважды щелкните на строчке Ellipse в разделе **All XAML Controls** панели элементов. Снова выделите элемент Canvas и дважды щелкните на строчке Rectangle. Поверх эллипса на холсте появится прямоугольник. Перетащите его вниз.

Удерживая клавишу Shift, щелкните на эллипсе, чтобы выделить оба элемента. Затем щелкните правой кнопкой мыши и выберите команды **Group Into** и **StackPanel**. Выделите эллипс, одноцветной кистью выберите белый цвет и присвойте свойствам `Width` и `Height` значение 10. Затем выделите прямоугольник, сделайте его белым и присвойте свойствам `Width` и `Height` значения 10 и 25.

В окне Document Outline выделите Stack Panel (в окне Properties должно быть написано Type StackPanel). Кнопками ☒ присвойте параметрам `Width` и `Height` значение Auto. В поле Name введите `human`. В результате будет сгенерирован XAML-код:

```
<StackPanel x:Name="human" Orientation="Vertical">
    <Ellipse Fill="White" Height="10" Width="10"/>
    <Rectangle Fill="White" Height="25" Width="10"/>
</StackPanel>
```

Можно ввести этот код непосредственно в окно XAML перед тегом </Canvas>. Так вы укажете, что человек должен быть частью холста.

В окне Document Outline должен появиться новый элемент управления:

Ваш XAML-код может задать свойство Stroke. Попробуйте догадаться сами, как его добавить и удалить.

❼ Добавляем текст Game Over.

При завершении игры должно появляться сообщение «Game Over». Для этого добавим элемент TextBlock, выберем шрифт и зададим имя:

- Выделите элемент Canvas и перетащите туда TextBlock с панели элементов.
- В поле Name окна Properties введите имя `gameOverText`.
- В разделе Text выберите шрифт Arial Black, сделайте его размер равным 100 px и нажмите кнопки Bold и Italic.
- Перетащите элемент TextBlock в центр холста.
- Измените текст надписи на `Game Over`.

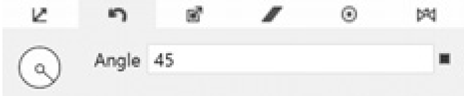
При перемещении элемента по холсту меняются его свойства Left и Top. Редактируя эти свойства, вы можете двигать элемент.

8 Добавляем портал, к которому нужно будет перетаскивать людей.

На холст осталось добавить всего один элемент: портал, к которому игрок будет перетаскивать людей. (Он может располагаться в произвольном месте элемента Canvas.)

Выделите элемент Canvas и перетащите на него элемент Rectangle. Кнопкой ▣ из раздела Brushes создайте градиентную заливку. Параметрам `Height` и `Width` присвойте значение **`50`**.

Превратим прямоугольник в ромб. Для этого в разделе Transform окна Properties щелкнем на кнопке ↶ и зададим угол **`45`**.



В поле Name окна Properties введите имя `target`.

Поздравляем — главная страница нашего приложения готова!

После того как в процессе создания пользовательского интерфейса вы настроили ряд различных свойств, вам известно назначение некоторых элементов управления. Укажите, за что отвечает каждое свойство и в каком разделе окна Properties оно находится.

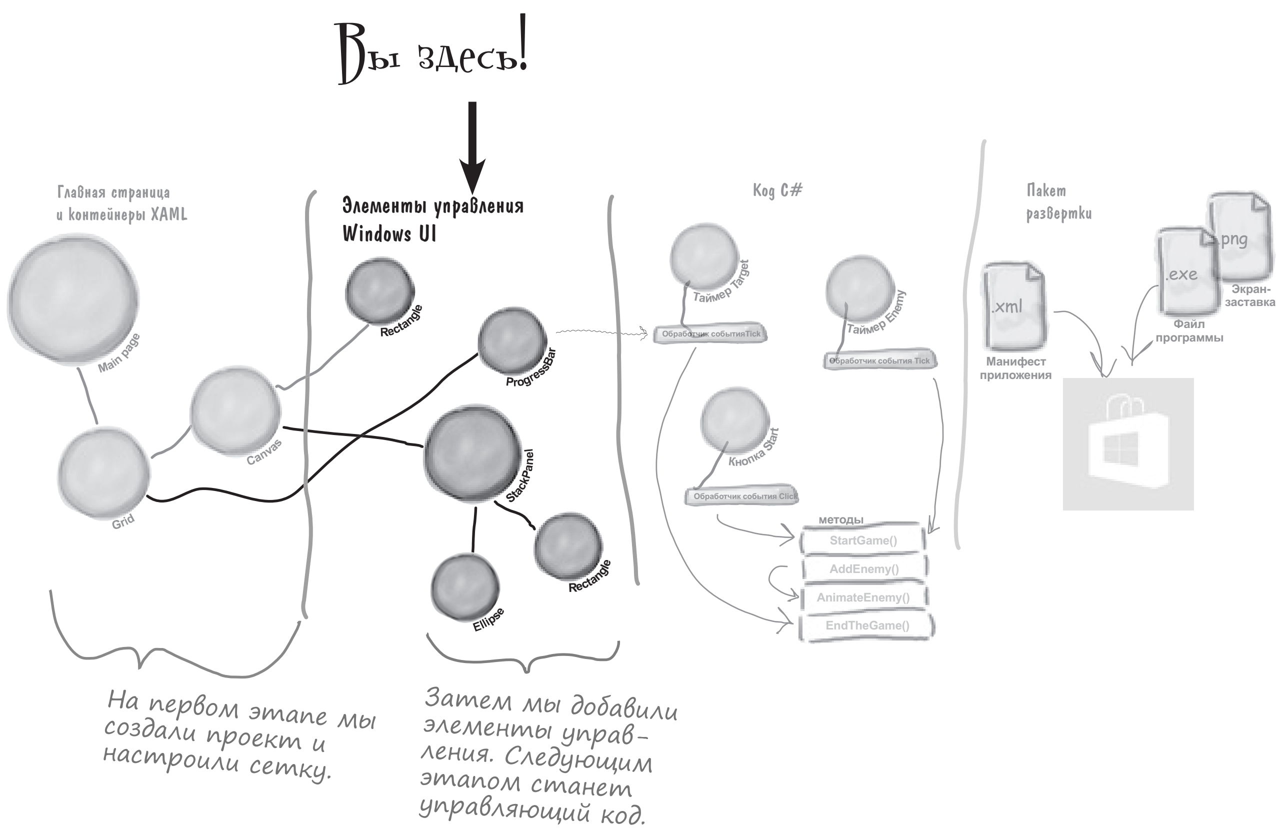
Свойство XAML	**В каком разделе окна Properties находится**	**Функция**
`Content`	Наверху	Задает высоту элемента
	Brush	Задает угол поворота элемента
`Height`	Appearance	
`Rotation`	Common	Используется в коде C# для манипуляции элементом
`Fill`	Layout	Задает цвет элемента
`x:Name`	Transform	Применяется для редактирования текста внутри элемента управления

Решение на странице 73 →

Подсказка: для поиска свойств используйте поле Search окна Properties, но не забывайте, что некоторые свойства присущи не всем типам элементов.

Игровое поле готово

Теперь, когда вы настроили сетку, которая служит основой страницы, и добавили элементы управления, можно приступать к написанию кода.



Visual Studio предоставляет инструментарий для компоновки страницы, по сути, помогая писать код XAML. Но отвечаете за результат вы, и только вы!

Что дальше?

Наступает самое интересное — добавление кода, который заставит игру работать. Сначала мы оживим врагов, затем дадим игроку возможность взаимодействовать с игрой и, наконец, улучшим внешний вид игрового интерфейса.

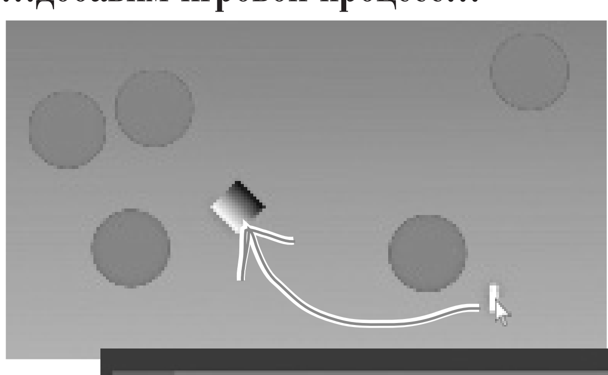
Анимируем врагов...

Первым делом напишем код C#, заставляющий врагов после щелчка на кнопке Start перемещаться в игровой области.

Большинство программистов пишут код по кускам: переходят к следующему фрагменту, только убедившись в корректной работе предыдущего. Именно так мы будем строить нашу программу. Начнем с метода `AddEnemy()`**, добавляющего к элементу Canvas анимированного врага. Мы свяжем его с кнопкой Start , нажатие которой будет заполнять игровое пространство перемещающимися врагами. Это послужит основой для остальной игры.**

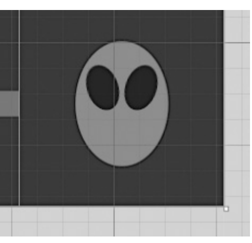
...добавим игровой процесс...

Чтобы игра заработала, нужно заставить индикатор начать обратный отсчет, людей — двигаться, а игру — завершаться при захвате человечка или по таймеру.

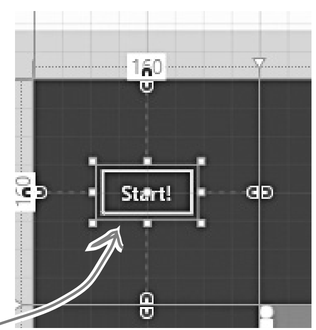


Шаблон позволяет рисовать врагов в виде красных кружков. Теперь мы обновим его, придав врагам вид злых инопланетян.

...и придадим ему интересный вид.



Добавляем метод, который что-то делает



Пора приступать к коду C#. Первым делом мы добавим **метод**, а IDE сгенерирует нам соответствующий код.

При редактировании страницы двойной щелчок на любом из ее элементов автоматически добавляет код к проекту. Дважды щелкните на кнопке Start в конструкторе. IDE добавит код, выполняющийся при щелчке игрока на кнопке. Вот как он выглядит:

IDE создала этот метод после двойного щелчка на кнопке Button. Он будет запускаться при щелчке на кнопке Start! в работающем приложении.

```
private void startButton_Click(object sender, RoutedEventArgs e)
{

}
```

```
Click="startButton_Click"
```

IDE добавила строку в код XAML. Попробуйте ее найти. О том, что это такое, мы поговорим в главе 2.

Используйте IDE для создания собственного метода

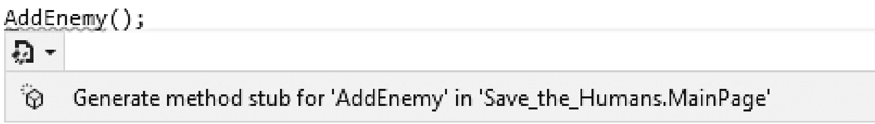
Введите между скобками { } текст со скобками и точкой с запятой:

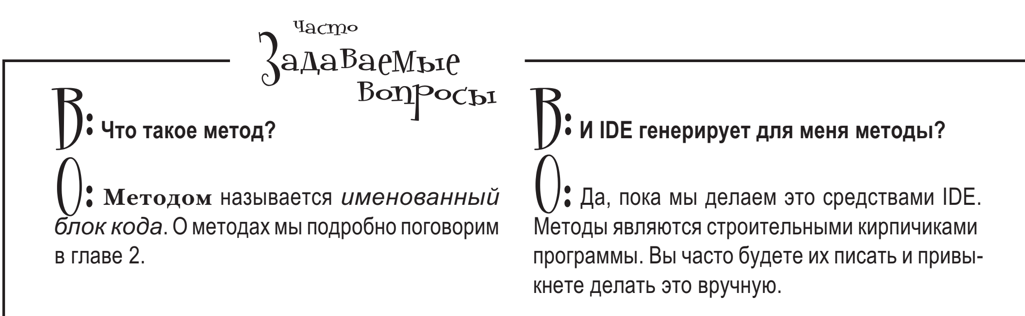
```
private void startButton_Click(object sender, RoutedEventArgs e)
{
    AddEnemy();
}
```

Красное подчеркивание указывает на наличие проблемы, а синий прямоугольник говорит о том, что существует решение.

Обратили внимание на красную линию под введенным текстом? Это IDE указывает, что что-то идет не так. Щелчок на линии приводит к появлению синего прямоугольника. Так IDE сообщает о том, что может помочь вам в решении возникшей проблемы.

Наведите указатель на прямоугольник и щелкните на значке. Вам предложат сгенерировать заглушку метода. Щелкните на этой строке, чтобы посмотреть, что получится!





Часто задаваемые вопросы

В: Что такое метод?

О: **Методом** называется *именованный блок кода*. О методах мы подробно поговорим в главе 2.

В: И IDE генерирует для меня методы?

О: Да, пока мы делаем это средствами IDE. Методы являются строительными кирпичиками программы. Вы часто будете их писать и привыкнете делать это вручную.

Пишем код метода

Пришло время заставить программу ***что-то сделать***, и у нас есть замечательная начальная точка. IDE сгенерировала **заглушку метода**: вам остается только добавить код.

Копируйте код C# аккуратно.

Добавляя к программе код C#, следите за регистром букв, скобками, запятыми и точками с запятой. Если пропустить хотя бы один знак, программа не будет работать!

1. Удалите сгенерированное IDE содержимое метода.

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

Выделите и удалите эту строку. Исключения мы будем изучать в главе 12.

2. Введите в тело метода слово Content. IDE откроет так называемое окно **IntelliSense** со списком возможных вариантов. Выберите строку `ContentControl`.

```
private void AddEnemy()
{
    Content
}
```

- _contentLoaded
- Content
- ContentControl
- ContentPresenter
- ContentProperty
- ContentThemeTransition
- HorizontalContentAlignment
- HorizontalContentAlignmentProperty
- ScrollContentPresenter

3. Завершите ввод первой строки кода. После слова new снова появится окно IntelliSense:

```
private void AddEnemy()
{
    ContentControl enemy = new ContentControl();
}
```

Эта строчка создает новый объект ContentControl. С объектами и ключевым словом new мы познакомимся в главе 3, а ссылочные переменные, такие как enemy, рассмотрим в главе 4.

4 Перед вводом остальной части метода AddEnemy() следует добавить кое-что в верхнюю часть файла. Найдите строку **public sealed partial class MainPage** и после скобки ({) введите:

```
/// <summary>
/// A basic page that provides characteristics common to most applications.
/// </summary>
public sealed partial class MainPage : Save_the_Humans.Common.LayoutAwarePage
{
    Random random = new Random();
```

Это поле. О том, как оно работает, вы узнаете в главе 4.

5 Завершите добавление метода. При этом появится ряд красных подчеркиваний. Линия под AnimateEnemy() исчезнет после генерации заглушки метода.

Видите красную линию под словом playArea? Вернитесь в редактор XAML и проверьте, что элементу Canvas присвоено имя playArea.

```
private void AddEnemy()
{
    ContentControl enemy = new ContentControl();
    enemy.Template = Resources["EnemyTemplate"] as ControlTemplate;
    AnimateEnemy(enemy, 0, playArea.ActualWidth - 100, "(Canvas.Left)");
    AnimateEnemy(enemy, random.Next((int)playArea.ActualHeight - 100),
        random.Next((int)playArea.ActualHeight - 100), "(Canvas.Top)");
    playArea.Children.Add(enemy);
}
```

Эта строка добавляет новый элемент enemy в коллекцию Children. С коллекциями вы познакомитесь в главе 8.

Для переходов между XAML и C# пользуйтесь вкладками в верхней части окна.

MainPage.xaml | MainPage.xaml.cs

6 При помощи кнопки сгенерируйте заглушку метода AnimateEnemy(). Добавятся четыре **параметра**: enemy, p1, p2 и p3. Отредактируйте верхнюю строку метода, изменив p1 на **from**, p2 — на **to**, а p3 — на **propertyToAnimate**. Затем поменяйте тип int на тип **double**.

```
private void AnimateEnemy(ContentControl enemy, int p1, double p2, string p3)
{
    throw new NotImplementedException();
}
```

О методах и параметрах речь пойдет в главе 2.

```
private void AnimateEnemy(ContentControl enemy, double from, double to, string propertyToAnimate)
```

Иногда IDE генерирует заглушки с типом «int». В данном случае его нужно поменять на «double». Типы данных будут рассматриваться в главе 4.

Перевернем страницу и запустим программу! →

Завершение метода и запуск программы

Программа практически готова! Осталось закончить метод AnimateEnemy(). Не паникуйте, если что-то не сработает. Возможно, вы пропустили запятую или скобки. При написании программ за этим нужно тщательно следить!

Будьте осторожны!

Красными линиями IDE указывает на проблемы.

Ничего страшного, если вы все еще видите красное подчеркивание! Вероятно, в код вкралось несколько опечаток. Эта глава была протестирована множеством пользователей, и текст кода проверен.

1. **Добавим в верхнюю часть файла оператор using.** В верхней части файла находятся несколько сгенерированных строк, начинающихся с using. Добавьте к этому списку еще одну строку:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

using Windows.UI.Xaml.Media.Animation;
```

Подобные операторы позволяют использовать код из библиотек .NET, прилагаемых к C#. О них мы поговорим в главе 2.

Эта строка заставляет работать следующий фрагмент кода. Для ее корректного написания можно воспользоваться IntelliSense. Не забудьте поставить в конце точку с запятой.

Оператор using позволяет воспользоваться кодом из .NET Framework для анимации врагов.

2. **Добавим код, заставляющий врагов перемещаться.** Добавим код к сгенерированной страницей раньше заглушке метода AnimateEnemy(). После этого враг начнет прыгать по игровому полю.

Инициализаторы объектов будут рассматриваться в главе 4.

```
private void AnimateEnemy(ContentControl enemy, double from, double to, string propertyToAnimate)
{
    Storyboard storyboard = new Storyboard() { AutoReverse = true, RepeatBehavior = RepeatBehavior.Forever };
    DoubleAnimation animation = new DoubleAnimation()
    {
        From = from,
        To = to,
        Duration = new Duration(TimeSpan.FromSeconds(random.Next(4, 6)))
    };
    Storyboard.SetTarget(animation, enemy);
    Storyboard.SetTargetProperty(animation, propertyToAnimate);
    storyboard.Children.Add(animation);
    storyboard.Begin();
}
```

Об анимации речь пойдет в главе 16.

Этот код перемещает врага по playArea. Меняя значения 4 и 6, можно ускорять и замедлять это движение.

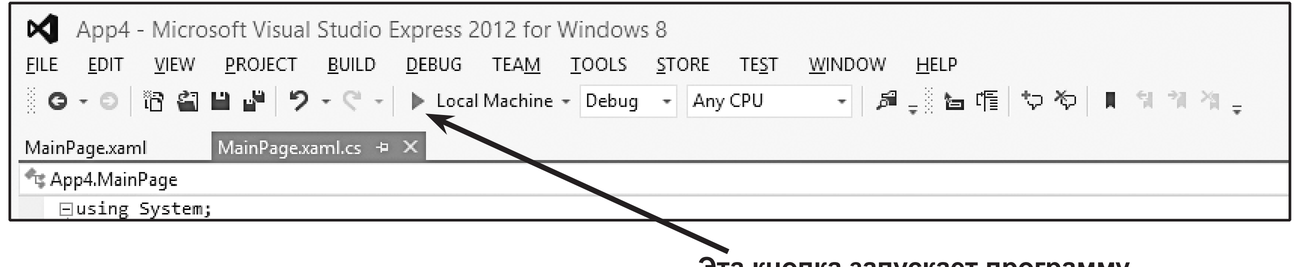
3. **Проверяем код.** В идеале окно Error List должно остаться пустым. Если это не так, дважды щелкните по ошибке. IDE установит указатель мыши в точку предполагаемой проблемы.

Если вы не видите окна Error List, выберите в меню View одноименную команду. Об использовании этого окна при отладке пойдет речь в главе 2.

Подсказка: вернуть перемещенные окна в положение по умолчанию можно командой Reset Window Layout из меню Window.

4 Запуск программы.

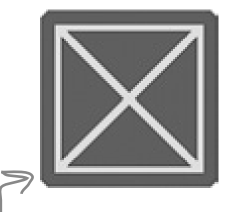
Найдите в верхней части IDE кнопку ▶. Именно она запускает программу.



Эта кнопка запускает программу.

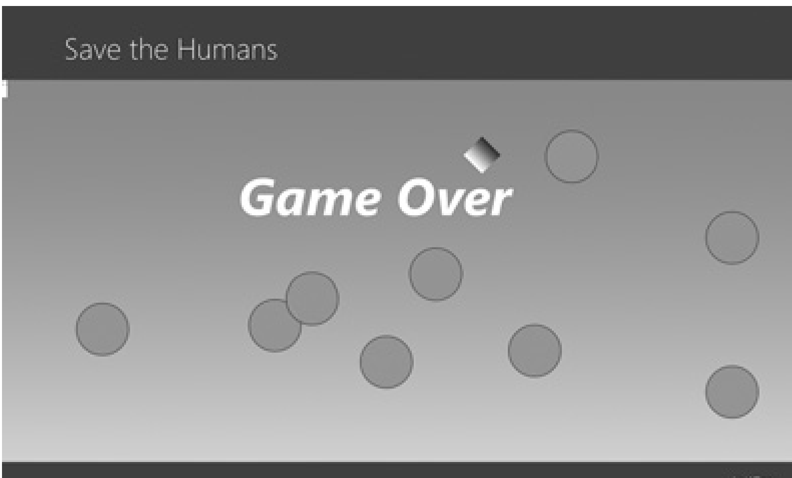
5 Программа заработала!

На несколько секунд на экране появится X, а затем вы увидите главную страницу. Щелкните на кнопке «Start!». При каждом щелчке по игровому полю прыгает круг.



Большой «X» — заставка. Собственную заставку вы сделаете к концу главы.

Классная штука! И вы создали ее быстро, как и было обещано. Но многое еще осталось несделанным.



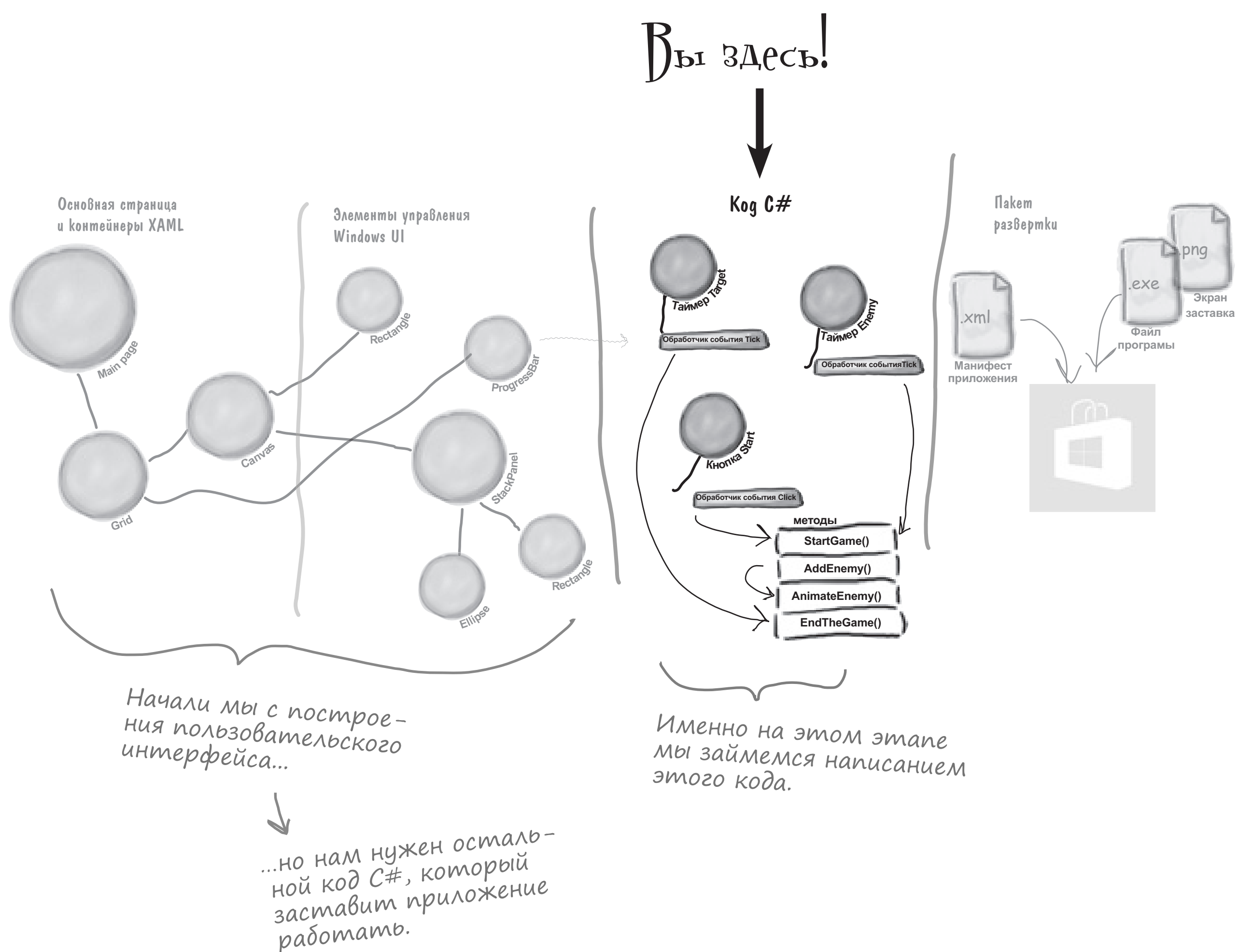
Если враги не прыгают или выходят за границы игровой области, проверьте код. Возможно, вы пропустили скобку или ключевое слово.

6 Останавливаем программу.

Нажмите Alt-Tab для возвращения в IDE. Вместо кнопки ▶ на панели инструментов появится панель ⏸ ■ ↻. Щелкните на квадратике, чтобы остановить работу программы.

Итак, о наших успехах

Поздравляем! Вы написали работающую программу. До полноценной игры еще далеко, но начало положено. Еще раз посмотрим, что мы сделали.



Visual Studio может сгенерировать код, но перед началом работы вы должны знать, какой результат хотите получить

Это ответ на упражнение со страницы 64. В книге вы найдете ответы на задачи и упражнения, но далеко не всегда они находятся на следующей странице.

решение

После того как в процессе создания пользовательского интерфейса вы настроили ряд различных свойств, вам известно назначение некоторых элементов управления. Укажите, за что отвечает каждое свойство и в каком разделе окна Properties оно находится.

Свойство XAML

Content

Height

Rotation

Fill

x:Name

В каком разделе окна Properties находится

At the top

Brush

Appearance

Common

Layout

Transform

Функция

Задает высоту элемента

Задает угол поворота элемента

Используется в коде C# для манипуляции определенным элементом

Задает цвет элемента

Применяется для редактирования текста внутри элемента управления

Помните, как мы присвоили параметру Name элемента Canvas значение "playArea"? Это определило свойство "x:Name" в XAML-коде данного элемента. Через минуту оно нам потребуется для работы с элементом Canvas.

Управляющие игрой таймеры

Добавим к заготовке элементы игрового процесса. Пока игрок перетаскивает человека в укрытие, на поле должны появляться новые враги, а индикатор должен показывать оставшееся время. Мы реализуем это при помощи **таймеров**.

1. **Добавим еще несколько строк к коду C#.**
   Перейдите в верхнюю часть файла, куда мы добавляли строку `Random`, и напечатайте:

   Редактируемый файл MainPage.Xaml.cs содержит код класса MainPage. О классах мы поговорим в главе 3.

```
/// <summary>
/// A basic page that provides characteristics common to most applications.
/// </summary>
public sealed partial class MainPage : Save_the_Humans.Common.LayoutAwarePage
{
    Random random = new Random();
    DispatcherTimer enemyTimer = new DispatcherTimer();
    DispatcherTimer targetTimer = new DispatcherTimer();
    bool humanCaptured = false;
```

Добавьте еще три строки. Это поля, которые мы будем рассматривать в главе 4.

2. **Добавляем метод для одного из таймеров.**
   Найдите этот фрагмент кода:

```
public MainPage()
{
    this.InitializeComponent();
}
```

Тик Тик Тик

Поместите курсор после точки с запятой, два раза нажмите Enter и введите `enemyTimer.` (не забудьте точку). Сразу после ввода точки появится окно IntelliSense. Выберите `Tick`. После ввода += IDE откроет окошко:

```
enemyTimer.Tick +=
```

enemyTimer_Tick; (Press TAB to insert)

Нажмите клавишу Tab для перехода к следующему окошку:

```
enemyTimer.Tick +=enemyTimer_Tick;
```

Press TAB to generate handler 'enemyTimer_Tick' in this class

Еще раз нажмите Tab. Вот код, сгенерированный IDE:

```
public MainPage()
{
    this.InitializeComponent();

    enemyTimer.Tick += enemyTimer_Tick;
}

void enemyTimer_Tick(object sender, object e)
{
    throw new NotImplementedException();
}
```

Сгенерированный метод называется обработчиком события. Обработчики событий будут рассматриваться в главе 15.

Таймеры отсчитывают время, снова и снова вызывая методы. Один таймер будет добавлять врага каждые несколько секунд, а второй — завершать игру.

Когда пишешь про метод, нужно добавить скобки ().

3 Завершаем метод MainPage().

Добавим еще один обработчик события Tick ко второму таймеру. Это пара строк кода. Вот как должны выглядеть готовый метод MainPage() и два метода, которые сгенерирует IDE:

Мозговой штурм

Сейчас кнопка Start добавляет на игровое поле прыгающих врагов. Что нужно сделать, чтобы она вместо этого запускала игру?

```
public MainPage()
{
    this.InitializeComponent();

    enemyTimer.Tick += enemyTimer_Tick;
    enemyTimer.Interval = TimeSpan.FromSeconds(2);

    targetTimer.Tick += targetTimer_Tick;
    targetTimer.Interval = TimeSpan.FromSeconds(.1);
}

void targetTimer_Tick(object sender, object e)
{
    throw new NotImplementedException();
}

void enemyTimer_Tick(object sender, object e)
{
    throw new NotImplementedException();
}
```

После окончания игры попробуйте поменять эти цифры. Как это повлияет на игровой процесс?

IDE сгенерировала эти строки как заглушки, когда вы нажимали Tab для добавления обработчиков события Tick. Вы замените их кодом, который будет запускаться при каждом отсчете таймера.

4 Добавляем метод EndTheGame().

В новом методе targetTimer_Tick() удалите сгенерированную строку и добавьте следующий код. Окно IntelliSense может оказаться не совсем корректным:

```
void targetTimer_Tick(object sender, object e)
{
    progressBar.Value += 1;
    if (progressBar.Value >= progressBar.Maximum)
        EndTheGame();
}
```

Попытается ли IDE исправить букву P в слове progressBar? Ближайшим вариантом замены является тип элемента управления.

Если вы случайно закрыли вкладку конструктора с кодом XAML, дважды щелкните на строчке MainPage.xaml в окне Solution Explorer.

Обратили внимание на ошибку в слове progressBar? Мы сделали ее намеренно (и не сожалеем!), чтобы показать, что происходит при попытке воспользоваться элементом без имени или с неправильно написанным именем. Вернитесь к коду XAML, найдите добавленный элемент ProgressBar и присвойте ему имя progressBar.

Теперь вернитесь в окно кода и сгенерируйте заглушку метода EndTheGame(), как вы это делали для метода AddEnemy(). Получится следующий код:

```
private void EndTheGame()
{
    if (!playArea.Children.Contains(gameOverText))
    {
        enemyTimer.Stop();
        targetTimer.Stop();
        humanCaptured = false;
        startButton.Visibility = Visibility.Visible;
        playArea.Children.Add(gameOverText);
    }
}
```

Если gameOverText подчеркивается как ошибка, значит, вы не задали имя элемента TextBlock. Вернитесь и сделайте это сейчас.

Этот метод завершает игру, останавливая таймеры, делая видимой кнопку Start и добавляя в игровую область текст GAME OVER.

Активация кнопки Start

Помните, как мы заставили Start бросать круги на Canvas? Исправим ситуацию, ведь эта кнопка должна запускать игру.

> **Готово к употреблению**
>
> **Мы заставляем вас вводить много кода.**
>
> К концу книги вы будете знать, зачем нужен этот код. Более того, вы научитесь писать его самостоятельно.
>
> А пока вам нужно следить за правильностью ввода каждой строки и точностью соблюдения инструкций. Это даст вам возможность привыкнуть к процессу набора кода и почувствовать плюсы и минусы IDE.
>
> Если что-то пойдет не так, вы можете скачать *MainPage.xaml* и *MainPage.Xaml.cs* или скопировать и вставить код XAML и C# каждого метода: *http://www.headfirstlabs.com/hfcsharp.*

1 Пусть кнопка Start запускает игру.
Найдите код, который заставляет кнопку Start добавлять очередного врага. Отредактируйте его следующим образом:

```
private void startButton_Click(object sender, RoutedEventArgs e)
{
    StartGame();
}
```

Изменив эту строку, вы заставите кнопку Start запускать игру вместо добавления врага на playArea Canvas.

2 Добавим метод StartGame().
Сгенерируйте заглушку `StartGame()`, а затем вставьте в нее следующий код:

```
private void StartGame()
{
    human.IsHitTestVisible = true;
    humanCaptured = false;
    progressBar.Value = 0;
    startButton.Visibility = Visibility.Collapsed;
    playArea.Children.Clear();
    playArea.Children.Add(target);
    playArea.Children.Add(human);
    enemyTimer.Start();
    targetTimer.Start();
}
```

Свойство IsHitTestVisible рассматривается в главе 15.

Не забыли ли вы присвоить элементу Rectangle имя target, а элементу StackPanel имя human? Можно вернуться на несколько страниц назад и убедиться, что у всех элементов управления корректные имена.

3 Создадим таймер и начнем добавлять врагов.
Найдите сгенерированный IDE метод `enemyTimer_Tick()` и замените его содержимое на:

```
void enemyTimer_Tick(object sender, object e)
{
    AddEnemy();
}
```

Привыкнув вводить код, вы будете легко находить пропущенные скобки, точки с запятой и т. п.

Видите в окне Error List странные ошибки? Одна пропущенная запятая может стать причиной появления двух, трех и более ошибок. Не тратьте время, пытаясь отследить каждую опечатку! Посетите веб-страницу Head First Labs, чтобы скопировать корректный код и вставить его в программу.

http://www.headfirstlabs.com/hfcsharp/

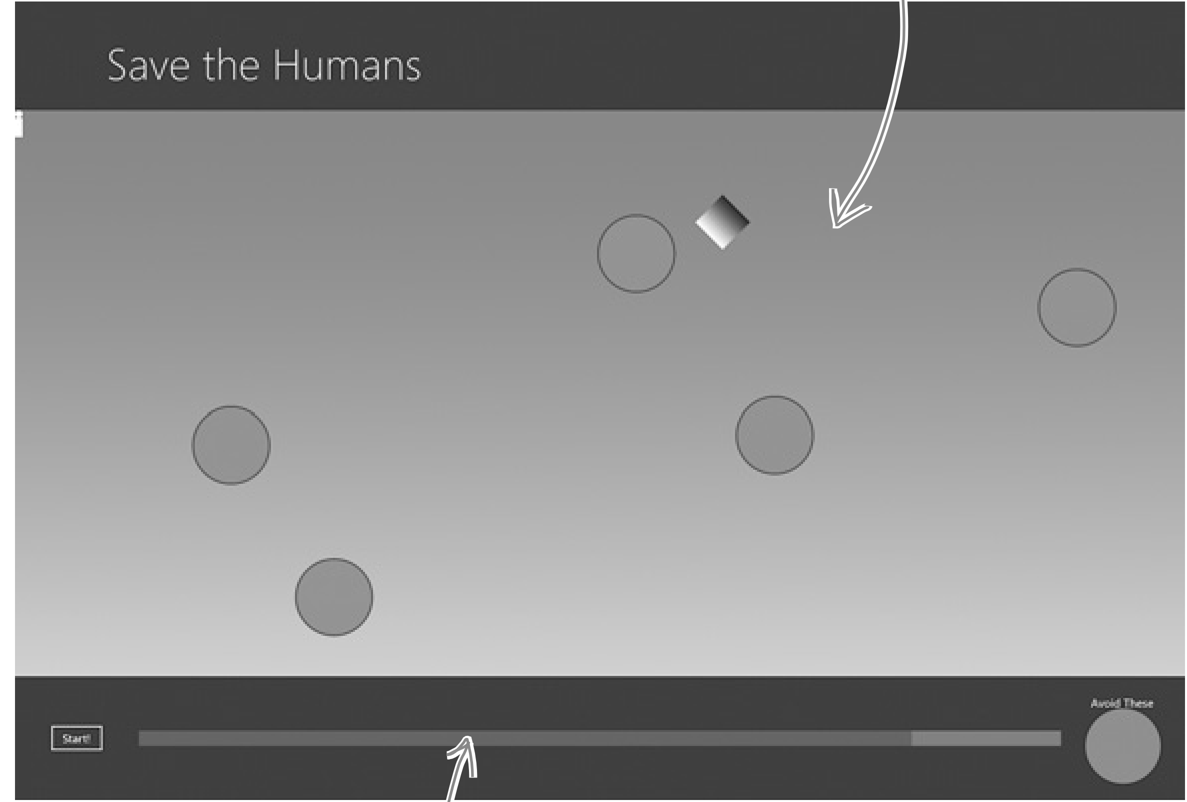
Посмотрим на свои успехи

Запустим игру еще раз, чтобы оценить результат.

Внимание! Разведчики сообщают, что люди строят оборонительные сооружения!

Кнопка Start! после нажатия исчезает, игровое поле очищается, а индикатор начинает отсчитывать время.

Игровая зона медленно начинает заполняться прыгающими врагами.



По окончании установленного времени игра завершается, и появляется надпись Game Over.

Индикатор должен закрашиваться медленно, а новый враг должен появляться каждые две секунды. Если что-то работает не так, убедитесь, что вы добавили в метод MainPage() все строки.

Как вы думаете, что следует сделать, чтобы заставить работать все остальное?

Ответ на следующей странице!

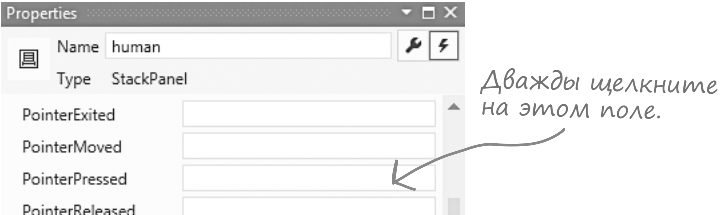
Взаимодействие с игроком

Перед тем как приступить к редактированию кода, остановите приложение и вернитесь в IDE.

У нас есть человек, которого игрок должен перетащить к порталу, и портал, который должен почувствовать приближение человека. Добавим код, реализующий подобное поведение.

1. Вернитесь в конструктор XAML и в окне Document Outline выделите элемент `human` (это контейнер StackPanel, состоящий из элементов Circle и Rectangle). Затем в окне Properties нажмите кнопку ⚡ для перехода к списку обработчиков событий. Найдите строку PointerPressed и дважды щелкните на пустом поле.

Подробно обработчики событий в окне Properties будут изучаться в главе 4.



Дважды щелкните на этом поле.

В окне Document Outline могут быть свернуты [Grid], playArea и другие строки. В этом случае просто раскройте их для поиска элемента human.

Теперь посмотрим, что IDE добавила к XAML-коду элемента StackPanel:

```
<StackPanel x:Name="human" Orientation="Vertical" PointerPressed="human_PointerPressed">
```

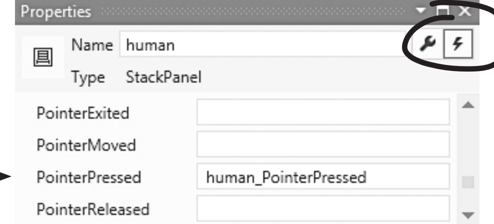
Одновременно была сгенерирована заглушка метода. Щелкните правой кнопкой мыши на названии `human_PointerPressed` в коде XAML и выберите команду “Navigate to Event Handler” для перехода к коду C# :

```
private void human_PointerPressed(object sender, PointerRoutedEventArgs e)
{

}
```

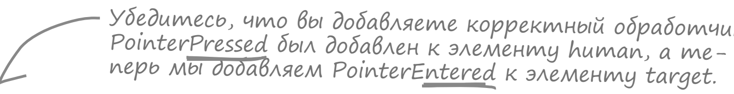
2. Введите это в код C#:

```
private void human_PointerPressed(object sender, PointerRoutedEventArgs e)
{
    if (enemyTimer.IsEnabled)
    {
        humanCaptured = true;
        human.IsHitTestVisible = false;
    }
}
```

Эти кнопки окна Properites позволяют переходить от списка свойств к списку обработчиков событий.



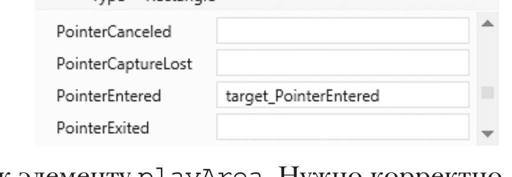
Если вернуться в конструктор и выделить элемент StackPanel, вы увидите, что IDE вставила имя нового метода обработчика событий. Аналогичным способом вы добавите и другие обработчики событий.



3 В окне Document Outline выделите прямоугольник с именем `target` и через окно Properties добавьте обработчик события `PointerEntered`. Вот код этого метода:

```
private void target_PointerEntered(object sender, PointerRoutedEventArgs e)
{
    if (targetTimer.IsEnabled && humanCaptured)
    {
        progressBar.Value = 0;
        Canvas.SetLeft(target, random.Next(100, (int)playArea.ActualWidth - 100));
        Canvas.SetTop(target, random.Next(100, (int)playArea.ActualHeight - 100));
        Canvas.SetLeft(human, random.Next(100, (int)playArea.ActualWidth - 100));
        Canvas.SetTop(human, random.Next(100, (int)playArea.ActualHeight - 100));
        humanCaptured = false;
        human.IsHitTestVisible = true;
    }
}
```

Когда окно Properties отображает список обработчиков событий, двойной щелчок на пустом поле рядом с именем обработчика приводит к добавлению заглушки метода.



Окно Properties следует вернуть в режим отображения свойств.

4 Добавим еще два обработчика событий, на этот раз к элементу `playArea`. Нужно корректно выбрать строку `[Grid]` в окне Document Outline (вам нужна дочерняя сетка, имя которой находится ниже основной) и присвоить элементу имя `grid`. Затем добавим обработчики:

```
private void playArea_PointerMoved(object sender, PointerRoutedEventArgs e)
{
    if (humanCaptured)
    {
        Point pointerPosition = e.GetCurrentPoint(null).Position;
        Point relativePosition = grid.TransformToVisual(playArea).TransformPoint(pointerPosition);
        if ((Math.Abs(relativePosition.X - Canvas.GetLeft(human)) > human.ActualWidth * 3)
        || (Math.Abs(relativePosition.Y - Canvas.GetTop(human)) > human.ActualHeight * 3))
        {
            humanCaptured = false;
            human.IsHitTestVisible = true;
        }
        else
        {
            Canvas.SetLeft(human, relativePosition.X - human.ActualWidth / 2);
            Canvas.SetTop(human, relativePosition.Y - human.ActualHeight / 2);
        }
    }
}
```

Здесь очень много скобок! Будьте крайне внимательны.

Эти полоски являются логическим оператором. С ним вы познакомитесь в главе 2.

Вы можете варьировать точность, заменяя 3 на большее или меньшее число.

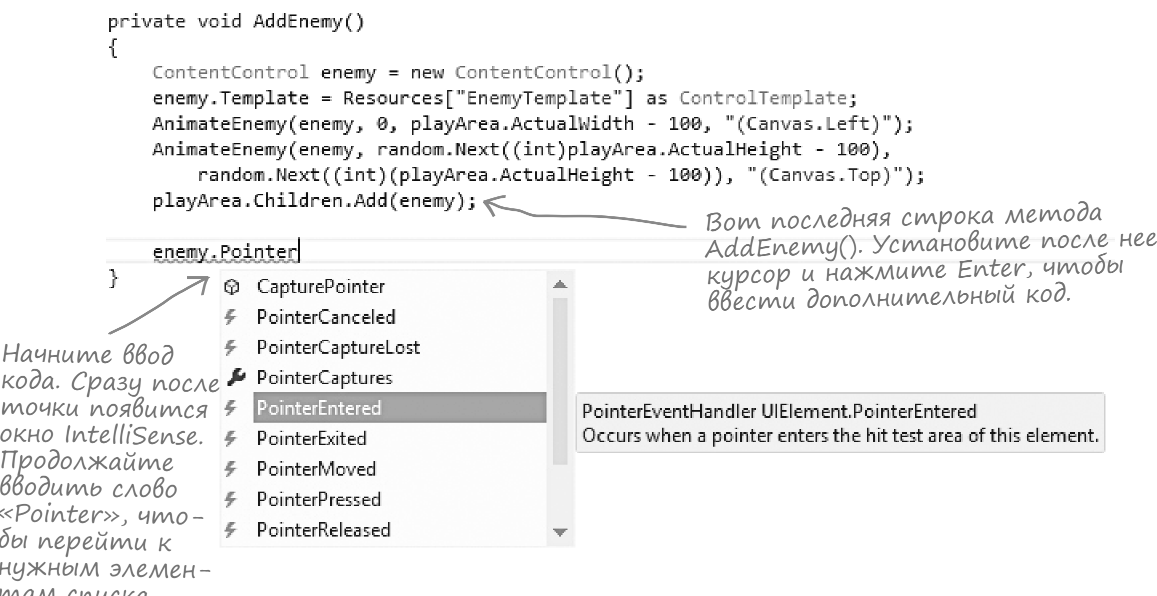
```
private void playArea_PointerExited(object sender, PointerRoutedEventArgs e)
{
    if (humanCaptured)
        EndTheGame();
}
```

Properties
Name playArea
Type Canvas
PointerExited playArea_PointerExited
PointerMoved playArea_PointerMoved
PointerPressed
PointerReleased

Убедитесь, что вы связали код с корректным обработчиком событий! Не перепутайте их.

Соприкосновение людей с врагами означает конец игры

Если при перетаскивании человек касается врага, игра должна завершиться. Добавим соответствующий код. Вернитесь к методу `AddEnemy()` и добавьте в конце еще одну строку. Выберите из списка IntelliSense вариант `enemyPointer.PointerEntered`:



Выберите в списке вариант `PointerEntered`. (Если вы ошиблись, удалите и снова поставьте точку для повторного вызова окна IntelliSense.)

Уже знакомым способом добавьте обработчик события. Напечатайте += и нажмите Tab:

```
enemy.PointerEntered +=
```

enemy_PointerEntered; (Press TAB to insert)

О том, как функционируют обработчики событий, вы узнаете в главе 15.

Снова нажмите клавишу Tab, чтобы сгенерировать заглушку для обработчика событий:

```
enemy.PointerEntered += enemy_PointerEntered;
```

Press TAB to generate handler 'enemy_PointerEntered' in this class

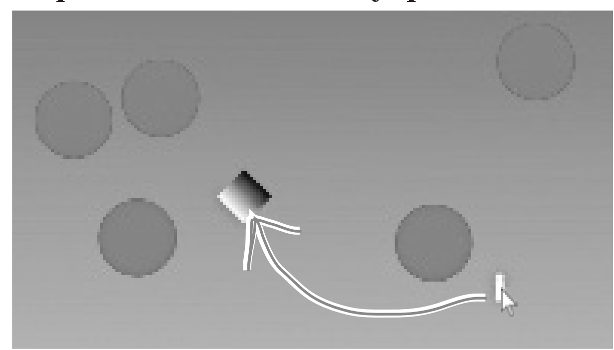
Теперь остается ввести код в сгенерированный для вас IDE метод:

```
void enemy_PointerEntered(object sender, PointerRoutedEventArgs e)
{
    if (humanCaptured)
        EndTheGame();
}
```

В игру уже можно играть

Запустим игру, она почти готова! При нажатии кнопки Start из игровой области исчезают враги, остается только человек и спасительный портал. Человека нужно провести к порталу, пока индикатор отсчитывает время. На первый взгляд все просто, но задача усложняется по мере заполнения экрана враждебно настроенными пришельцами!

Перетащи человека в укрытие!

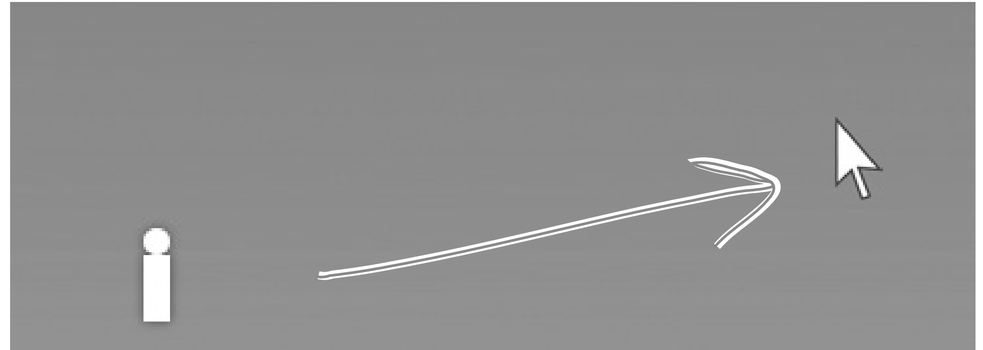


Пришельцы следят только за движущимися людьми, поэтому игра завершается только при столкновении человека с пришельцем. Если отпустить человека, пришельцы его не заметят.

Найдите в коде свойство IsHitTestVisible объекта human. При значении on для человека наступает событие PointerEntered, так как элемент управления StackPanel оказывается между противником и указателем мыши.

Перетащите человека в укрытие, пока еще есть время...

...но двигаясь слишком быстро, его можно потерять!



Превратим врагов в пришельцев

Красные круги не кажутся пугающими. К счастью, они созданы на основе шаблона, который можно обновить.

Вместо свойств видите события?

Окно Properties переключается с отображения свойств на отображение событий выделенного элемента управления двумя кнопками.

1. В окне Document Outline щелкните правой кнопкой мыши на строке ContentControl, выберите Edit Template, а затем Edit Current. Вы увидите шаблон. Отредактируйте код эллипса, задав ширину `75` и цвет `Gray`. Затем напишите `Stroke="Black"` для добавления черного контура и сбросьте выравнивание по горизонтали и вертикали. Вот что должно получиться в итоге (можно удалить все дополнительные свойства, появившиеся в процесс работы):

```
<Ellipse Fill="Gray" Height="100" Width="75" Stroke="Black" />
```

2. С панели инструментов перетащите еще один элемент Ellipse поверх существующего. Измените его цвет на черный, сделайте ширину равной `25`, а высоту `35`. Выравнивание и поля задайте следующим образом:

Придать эллипсу нужную форму можно при помощи мыши и кнопок со стрелками. Попробуйте скопировать эллипс и поместить его поверх другого командами Copy и Paste из меню Edit.

3. Кнопкой ▰ в разделе Transforms окна Properties добавьте скос Skew:

4. Перетащите на существующий эллипс еще один элемент Ellipse с панели элементов. Присвойте цвету значение `Black`, задайте ширину `25`, высоту `35`. Выравнивание и поля настройте следующим образом:

и добавьте следующий скос:

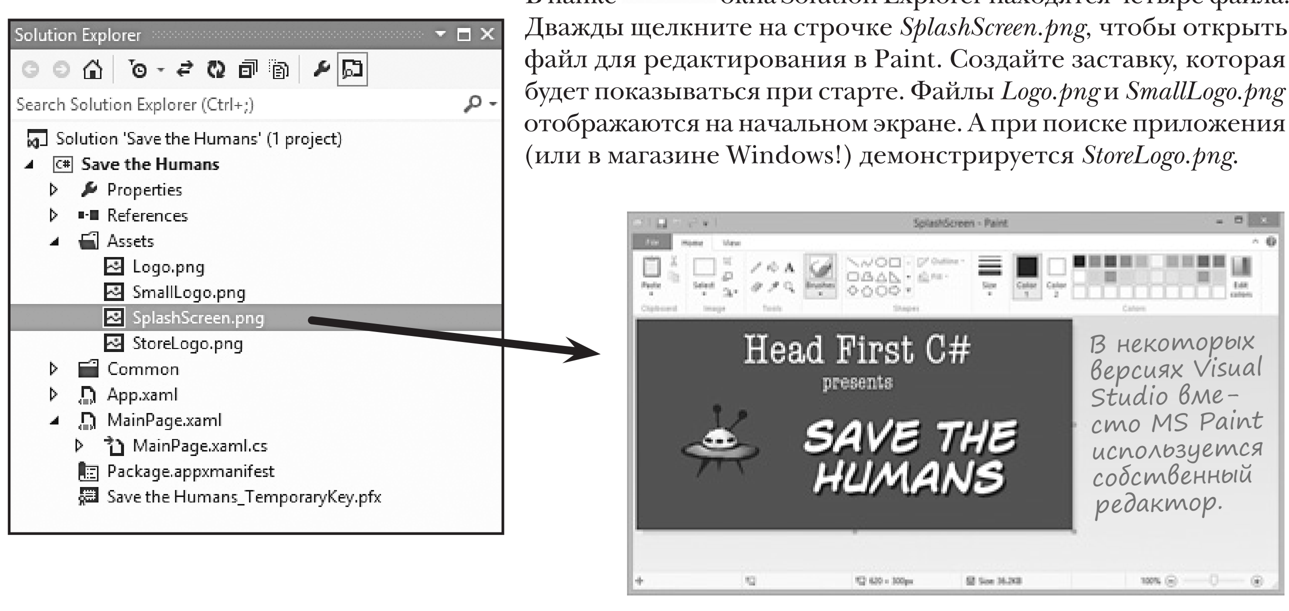
Теперь враги больше похожи на страшных пришельцев.

Добавим заставку и значок приложения

Большой крест, появляющийся при запуске программы, — это заставка. Давайте все это отредактируем.

Не можете нарисовать собственную заставку? Скачайте наш вариант:
http://www.headfirstlabs.com/hfcsharp

В папке Assets окна Solution Explorer находятся четыре файла. Дважды щелкните на строчке *SplashScreen.png*, чтобы открыть файл для редактирования в Paint. Создайте заставку, которая будет показываться при старте. Файлы *Logo.png* и *SmallLogo.png* отображаются на начальном экране. А при поиске приложения (или в магазине Windows!) демонстрируется *StoreLogo.png*.



В некоторых версиях Visual Studio вместо MS Paint используется собственный редактор.

```
<ControlTemplate x:Key="EnemyTemplate" TargetType="ContentControl">
    <Grid>
        <Ellipse Fill="Gray" Stroke="Black" Height="100" Width="75"/>
        <Ellipse Fill="Black" Stroke="Black" Height="35" Width="25"
                 HorizontalAlignment="Center" VerticalAlignment="Top"
                 Margin="40,20,70,0" RenderTransformOrigin="0.5,0.5">
            <Ellipse.RenderTransform>
                <CompositeTransform SkewX="10"/>
            </Ellipse.RenderTransform>
        </Ellipse>
        <Ellipse Fill="Black" Stroke="Black" Height="35"  Width="25"
                 HorizontalAlignment="Center" VerticalAlignment="Top"
                 Margin="70,20,40,0" RenderTransformOrigin="0.5,0.5">
            <Ellipse.RenderTransform>
                <CompositeTransform SkewX="-10"/>
            </Ellipse.RenderTransform>
        </Ellipse>
    </Grid>
</ControlTemplate>
```

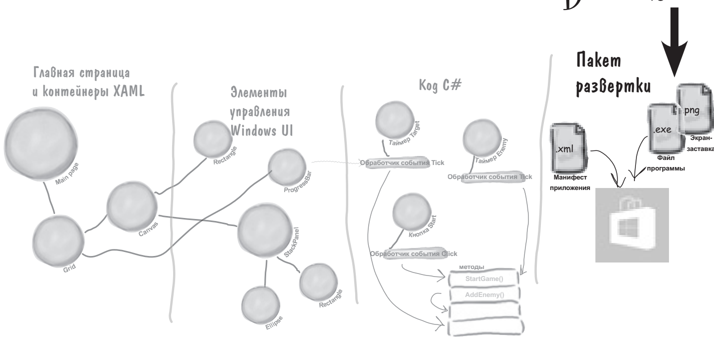
Это обновленный код XAML для нового шаблона врага.

Придумайте свои версии человечка, портала, игровой области и инопланетянина.

Остался всего один шаг... Поиграть!

И не забудьте посмотреть на результаты своего труда со стороны!

Публикация приложения



Приложение получилось достойным! Теперь нужно сделать его доступным. Публикация в магазине Windows покажет его множеству пользователей.

Вот как это выглядит:

1. Откройте учетную запись разработчика магазина Windows.

2. Укажите название приложения и возрастной рейтинг, добавьте описание и выберите бизнес-модель: приложение может быть бесплатным, платным или с поддержкой рекламы.

3. Протестируйте приложение при помощи Windows App Certification Kit.

4. **Отправьте приложение в магазин!** Как только оно будет принято, его смогут скачать миллионы пользователей по всему миру.

STORE TEST WINDOW HELP

- Open Developer Account...
- Reserve App Name...
- Acquire Developer License...
- Edit App Manifest
- Associate App with the Store...
- Capture Screenshots...
- Create App Packages...
- Upload App Packages...

Меню Store в IDE содержит все необходимые для публикации приложения инструменты.

В некоторых версиях Visual Studio команды для работы с магазином Windows расположены не в отдельном меню Store верхнего уровня, а в меню Project.

На протяжении всей книги мы часто будем ссылаться на MSDN (Microsoft Developer Network). Этот великолепный ресурс поможет вам узнать много нового.

Подробнее узнать о магазине Windows можно здесь:
http://msdn.microsoft.com/ru-ru/library/windows/apps/jj657972.aspx

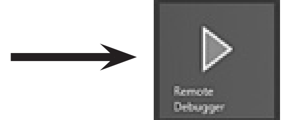
Загрузка через удаленный отладчик

Иногда нужно загрузить приложение на удаленной машине без его публикации в магазине. Это называется **загрузкой неопубликованных приложений** и проще всего реализуется путем установки на удаленном компьютере компонента **Visual Studio Remote Debugger**.

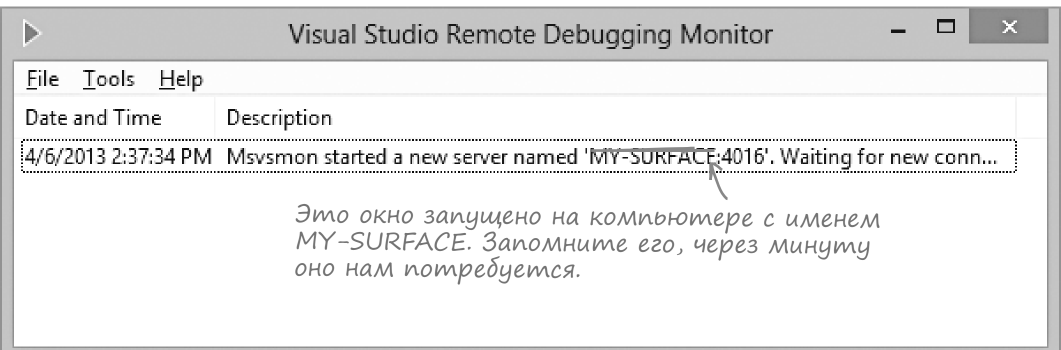
На момент написания книги это был "Remote Tools for Visual Studio 2012 Update 2," но вам может быть доступна новая версия.

Вот каким образом осуществляется загрузка через Remote Debugger:

- Убедитесь в наличии на удаленной машине системы Windows 8.
- С удаленной машины перейдите в центр скачивания Microsoft (*http://www.microsoft.com/ru-ru/download/default.aspx*) и выполните поиск "Remote Tools for Visual Studio 2012."
- Скачайте установщик под нужную вам архитектуру (x86, x64, ARM) и запустите его.
- Зайдите на страницу Start и загрузите Remote Debugger.



- Если конфигурация сети потребует изменений, появится соответствующий мастер. После запуска вы увидите окно Visual Studio Remote Debugging Monitor:



- Теперь на удаленном компьютере работает Visual Studio Remote Debugging Monitor, ожидающий соединения со стороны Visual Studio на машине разработчика.

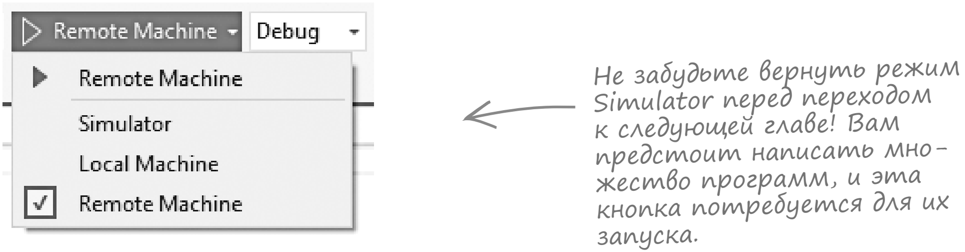
Если из-за нестандартных сетевых настроек вы не можете запустить удаленный отладчик, попробуйте воспользоваться инструкциями отсюда:
http://msdn.microsoft.com/ru-ru/library/vstudio/bt727f1t.aspx

Перевернем страницу и запустим приложение на чужом компьютере!

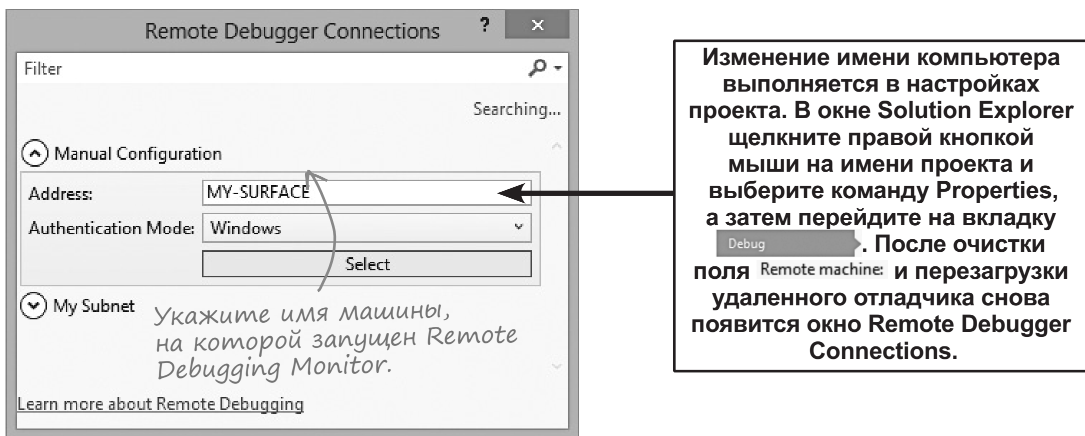
Удаленная отладка

Теперь, когда на чужом компьютере запущен монитор удаленной отладки, можно загрузить приложение Visual Studio для установки и последующего запуска. После этого вы сможете запускать его со страницы Start в любое время.

1 **В раскрывающемся списке Debug выбираем "Remote Machine".**
Раскрывающийся список Debug может заставить IDE запустить программу на чужой машине. Внимательно посмотрите на кнопку ▶ Local Machine ▾. Обратите внимание на раскрывающийся список (▾) и выберите вариант Remote Machine:



2 **Запустим программу на удаленной машине.**
Теперь запустите программу, нажав кнопку ▶. IDE откроет окно с вопросом, на какой машине следует это сделать. Если машина не будет распознана автоматически, ее имя можно ввести вручную:

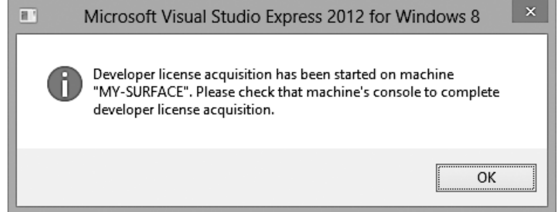


3 Ввведем свои учетные данные.

Вам будет предложено указать имя пользователя и пароль на удаленной машине. Этого можно избежать, отключив авторизацию в окне Remote Debugging Monitor (но это не очень хорошая идея, так как в этом случае кто угодно сможет удаленно запускать на вашей машине программы!).

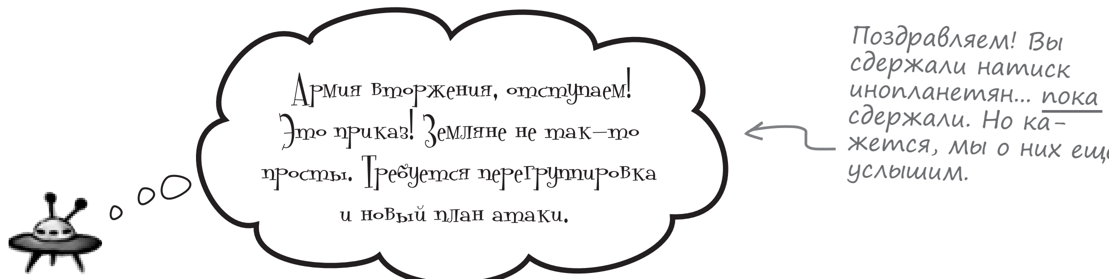
4 Получаем лицензию разработчика.

При установке Visual Studio вы уже получили бесплатную лицензию разработчика от Microsoft. Она требуется и при загрузке неопубликованных приложений. К счастью, монитор удаленной загрузки вызывает мастера, который сделает все автоматически.



5 Теперь... спасем людей!

После завершения настройки программа запустится на удаленной машине. Так как это неопубликованное приложение, загруженное с удаленного доступа, его повторный запуск происходит со страницы Windows Start. Поздравляем! Вы создали первое приложение для магазина Windows и загрузили его на другой компьютер!



Под покровом

Вы — программист, а не просто пользователь IDE.

IDE может сделать за вас многое, но не всё. При написании приложений часто приходится решать ***повторяющиеся задачи***. Пусть эту работу выполняет IDE. Вы же будете в это время думать над более глобальными вещами. Научившись ***писать код***, вы получите возможность решить любую задачу.

Когда вы делаете это...

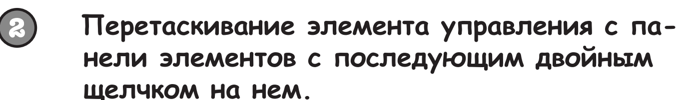
IDE — инструмент мощный. Но это всего лишь *инструмент*. При каждом вашем действии он автоматически создает код. Но это хорошо только при выполнении **шаблонных** задач или в случае, когда код может быть использован без дополнительных изменений.

Посмотрим на действия IDE при разработке типичного приложения.

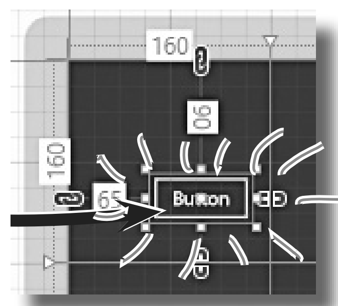
Все эти задачи являются стандартными и требуют шаблонного кода. Поэтому их можно отдать на откуп IDE.

1 **Создание приложения Windows Store.**
IDE позволяет создавать несколько видов приложений. Пока мы сконцентрируемся на приложениях Windows Store, а приложения других типов будут рассмотрены в следующей главе.

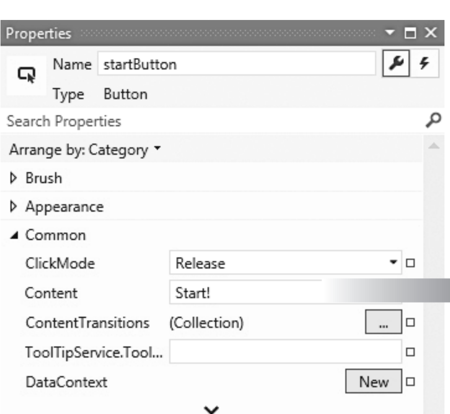
В главе 1 мы создавали пустой проект Windows Store. Это заставило IDE создать пустую страницу и добавить ее к новому проекту.



2 **Перетаскивание элемента управления с панели элементов с последующим двойным щелчком на нем.**
Функционал страницы реализуется при помощи элементов управления. В этой главе мы познакомимся с различными аспектами языка C# на примере элемента управления Button.



3 **Задание свойств страницы.**
Окно Properties представляет собой мощный инструмент, позволяющий менять атрибуты практически всего: визуальных и функциональных свойств элементов управления и даже параметры самого проекта.



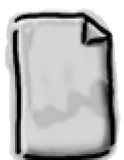
В IDE окно Properties предоставляет легкий способ автоматически отредактировать нужный фрагмент кода XAML в файле MainPage.xaml, позволяя сильно экономить время. Оно открывается клавиатурной комбинацией Alt-Enter.

...IDE делает это.

Любые ваши действия приводят к изменениям кода, а значит, и файлов, которые содержат этот код. Иногда редактируются всего несколько строчек, а иногда система создает новые файлы.

Эти файлы были созданы из предустановленного шаблона, содержащего базовый код создания и отображения страницы.

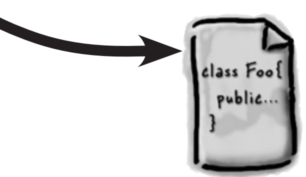
1 **...IDE создает для проекта файлы и папки.**



2 **... IDE добавляет к файлу `MainPage.xaml` код кнопки и метод `MainPage.xaml.cs`, который запускается при каждом щелчке на этой кнопке.**

```
private void startButton_Click(object sender, RoutedEventArgs e)
{

}
```

IDE умеет добавлять пустой метод обработки щелчков на кнопке. Но IDE не знает, что должно находиться внутри этого метода. Это должны написать вы.



3 **...IDE открывает файл `MainPage.xaml` и обновляет строку кода XAML.**

IDE входит в этот файл...

```
<Button x:Name="startButton"
                 Content="Start!"
                 HorizontalAlignment="Center"
                 VerticalAlignment="Center" Click="startButton_Click"/>
```

MainPage.xaml

...и обновляет XAML-код.

Как рождаются программы

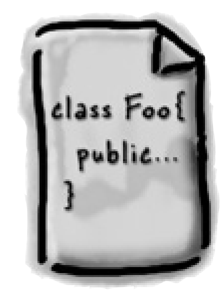
Программа на C# может начинаться как набор операторов в различных файлах, но в конце должна получиться программа, работающая на вашем компьютере. Вот как это происходит.

Любая программа начинается с файлов, текста кода

Как вы уже видели, IDE сохраняет вашу программу в файлы. Эти файлы можно открыть и найти все добавленные к проекту элементы: формы, ресурсы, код и прочее.

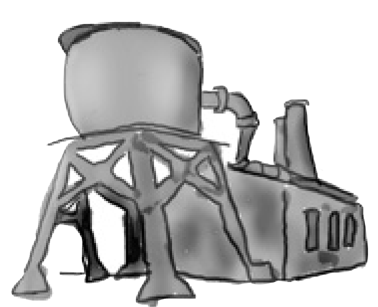
IDE можно представить как хороший редактор файлов. Он автоматически делает отступы, выделяет ключевые слова цветом, закрывает скобки и даже предполагает, что вы напишете в следующую секунду. Другими словами, именно IDE редактирует файлы, содержащие вашу программу.

IDE связывает все файлы программы в **решение** путем создания файла (`.sln`) и папки, в которой оказываются все материалы. Решение содержит список всех входящих в проект файлов (оно имеет расширение `.csproj`), последние же, в свою очередь, включают в себя список всех файлов, связанных с программой. В этой книге мы будем строить решения на основе всего одного проекта, но окно Solution Explorer позволяет работать и с другими проектами.



Программы можно писать даже в Блокноте (Notepad), но это очень долго.

Создание исполняемого файла



Выбор команды Build Solution (меню Build) начинает **компиляцию** вашей программы. Запускается **компилятор**, то есть инструмент, читающий код программы и преобразующий его в **исполняемый файл**. Этот файл с расширением *.exe* представляет собой программу, которую запускает Windows. Он создается внутри папки *bin*, вложенной в свою очередь в папку проекта. При публикации решения исполняемый файл (вместе с остальными нужными файлами) копируется в пакет, который можно отправить в магазин Windows или загрузить без публикации.

В ответ на команду Start Debugging (меню Debug) IDE компилирует программу и запускает полученный файл на выполнение. Существуют и более совершенные инструменты **отладки**, то есть запуска программы с возможностью прерывания, чтобы вы могли понять механизм происходящего.

Инструменты от .NET Framework

Сам по себе C# не более чем язык, не умеющий ничего *делать*. Но на помощь приходит **.NET Framework**. Все, что вы перетаскаваете с панели элементов, является частью библиотеки инструментов, классов, методов и др. В нее входят как визуальные элементы управления XAML, так и другие полезные вещи, например DispatcherTimer, давший жизнь нашему проекту *Save the Humans*.

API или прикладной интерфейс программирования представляет собой набор инструментов для доступа и управления системой. API встроены во многие системы, но особенно они важны в таких операционных системах, как Windows.

Все использованные элементы управления являются частью **.NET для приложений магазина Windows**, содержащей API с сетками, кнопками, страницами и остальными элементами. Но начиная с главы 3 несколько глав мы посвятим теме приложений для рабочего стола на базе **.NET для рабочего стола Windows** (или WinForms). Эти инструменты создают приложения из окон с различными формами, флажками, кнопками и списками. Они могут рисовать, читать и записывать в файлы, управлять коллекциями... словом, выполнять рутинную работу. И все это требуется приложениям магазина Windows! К концу книги вы узнаете, чем отличается выполнение определенных задач в приложениях разных типов. Именно понимание этих вещей превращает *хорошего* программиста в *отличного*.

Как в Windows Runtime, так и в .NET Framework инструменты распределены по **пространствам имен**. Вы видели их в верхней части кода, в строках "using". В пространстве имен `Windows.UI.Xaml.Conrols` содержатся кнопки, флажки и прочие элементы управления. При создании нового проекта для магазина Windows IDE добавит нужные файлы, чтобы сформировать страницу, а в верхней части этих файлов можно будет видеть строки "`using Windows.UI.Xaml.Controls;`".

Обзор .NET для приложений магазина Windows можно почитать тут:
http://msdn.microsoft.com/ru-ru/library/windows/apps/br230302.aspx

Ваша программа работает в CLR

Основой всех программ в Windows 8 является модель программирования Windows Runtime. Но между Windows Runtime и вашей программой существует дополнительный «слой» **общеязыковой исполняющей среды**, или CLR (Common Language Runtime). Раньше (еще до изобретения C#) писать программы было намного сложнее, так как приходилось иметь дело с аппаратным обеспечением, заниматься низкоуровневым программированием. CLR, или **виртуальная машина,** является своего рода «переводчиком» между программой и компьютером, на котором она работает.

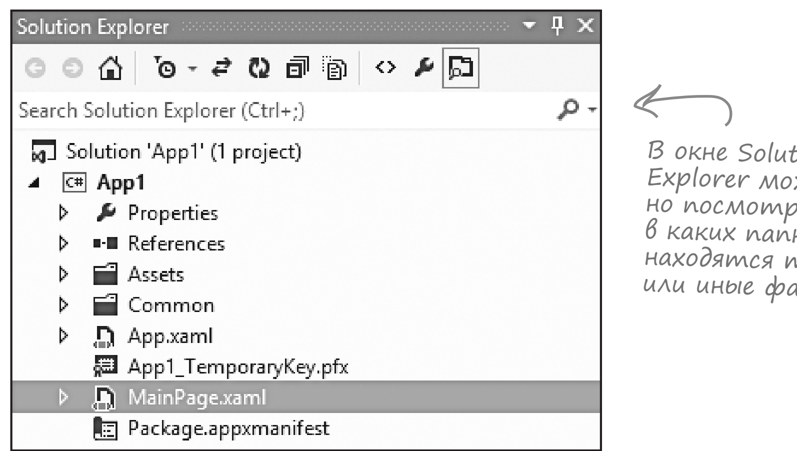
На данном этапе достаточно запомнить, что CLR автоматически запускает ваши программы. Более подробно она будет рассмотрена позднее.

Функции CLR будут рассмотрены позднее. Пока упомянем только, что она отвечает за работу с памятью компьютера, определяя, что программа закончила работать с конкретными данными, и избавляясь от них. Некоторые программисты привыкли самостоятельно работать с памятью, но без этого можно обойтись, переложив заботу на CLR.

Писать код помогает IDE

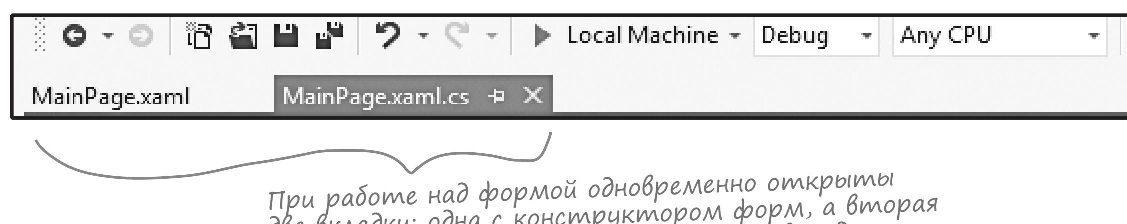
С частью возможностей IDE вы уже познакомились. Здесь же мы более детально рассмотрим некоторые инструменты.

- **Окно Solution Explorer содержит список всех частей проекта**
  Вам предстоит переключаться с одного класса на другой, и проще всего это делать, используя окно Solution Explorer. Вот вид этого окна после создания программы обмена контактными данными:



В окне Solution Explorer можно посмотреть, в каких папках находятся те или иные файлы.

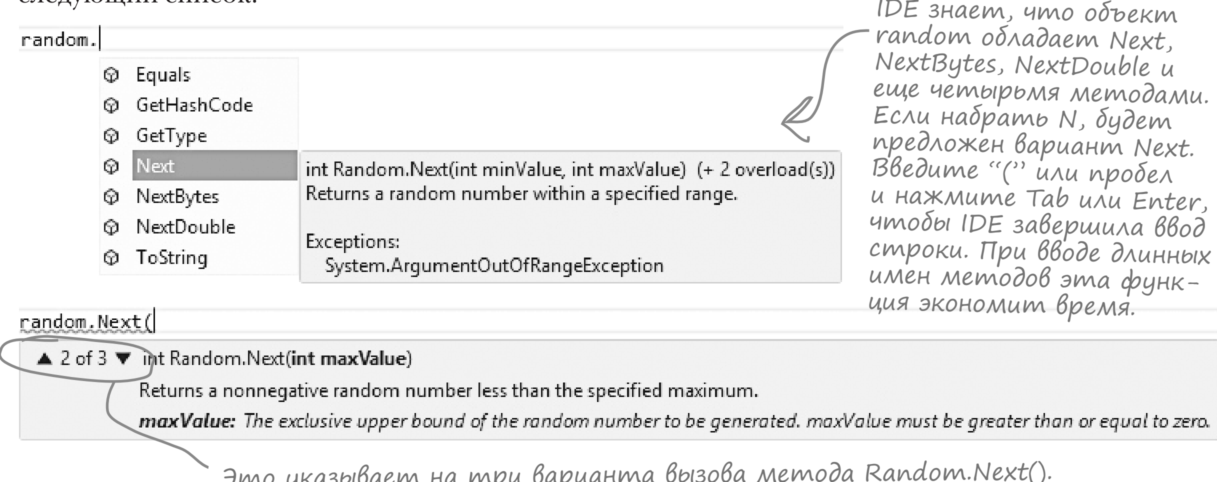
- **Вкладки позволяют легко перейти от одного файла к другому**
  Так как типичная программа состоит из множества файлов, приходится работать с несколькими файлами одновременно. Каждый из них имеет собственную вкладку в верхней части окна редактирования кода. Несохраненные файлы помечены значком (*).



При работе над формой одновременно открыты две вкладки: одна с конструктором форм, а вторая с кодом. Для быстрого перехода между вкладками пользуйтесь комбинацией клавиш **Ctrl-Tab**.

IDE помогает писать код

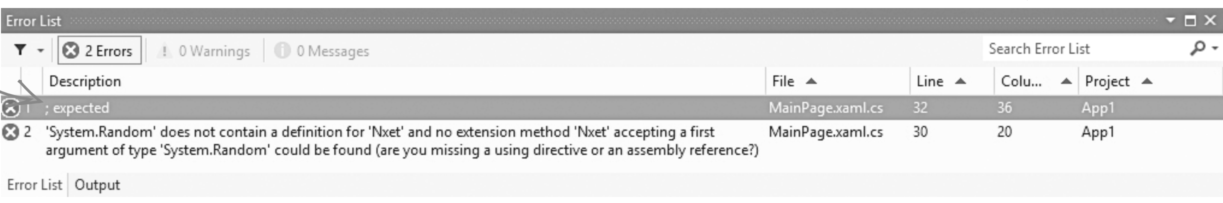
Обратили внимание на окно, появляющееся при вводе кода? Это крайне полезная функция IntelliSense, предлагающая доступные способы завершения вводимой строки. Если ввести `random` и поставить точку, вам предложат следующий список:



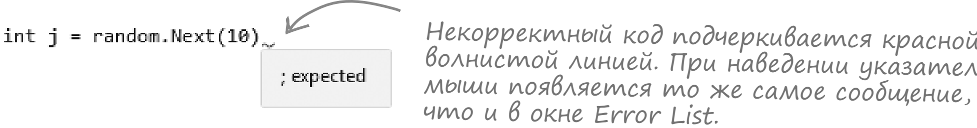
Если выбрать Next и ввести `(`, IntelliSense предложит вариант завершения строки.

Окно Error List показывает ошибки компиляции

Вы даже не представляете, как легко сделать опечатку при наборе кода на C#! К счастью, существует инструмент, позволяющий решить эту проблему. При построении решения всё мешающее компиляции будет перечислено в окне Error List, расположенном в нижней части экрана:



Для перехода к проблемному коду дважды щелкните на ошибке:



Структура программы

Код всех программ на C# структурирован одинаково. Везде для простоты управления кодом применяются **пространства имен**, **классы** и **методы**.

Для каждой программы определяется свое пространство имен, чтобы отделить код от классов .NET Framework и Windows Store API.

Класс содержит **фрагмент** вашей программы (очень маленькие программы могут состоять из одного класса).

Класс включает один или несколько методов. Методы всегда **принадлежат какому-либо классу**. Методы, в свою очередь, состоят из операторов.

Порядок методов в файле класса не имеет значения: метод 2 можно легко поместить перед методом 1.

Внимательно рассмотрим код

Откройте файл *MainPage.xaml.cs* проекта Save the Humans, чтобы как следует рассмотреть код.

1 **Файл кода начинается с перечисления инструментов .NET Framework.**
В верхней части любого файла программы находится набор строк с оператором `using`. Они указывают, с какой частью .NET Framework или Windows Store API будет работать программа. Чтобы воспользоваться классами из других пространств имен, нужно указать их с помощью оператора `using`. Приложения часто задействуют инструменты .NET Framework и Windows Store API, поэтому в верхней части файла страницы можно увидеть много операторов `using`.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
```

Подобные строки находятся в верхней части любого файла с кодом. Они указывают C#, какие классы из .NET Framework следует использовать. Каждый такой оператор указывает программе, что классы в определенном файле .cs будут пользоваться классами определенного пространства имен .NET Framework (System) или Windows Store API.

В принципе, без оператора `using` можно обойтись, если пользоваться полными именами. В приложении Save the Humans есть строка:

```
using Windows.UI.Xaml.Media.Animation;
```

Превратите ее в комментарий, поставив в начале `//`, и изучите ошибки. Одну из них можно исправить. Найдите слово `Storyboard`, которое IDE подчеркивает как ошибку, и вместо него напишите `Windows.UI.Xaml.Media.Animation.Storyboard` (но чтобы программа заработала, следует убрать добавленный комментарий).

2 **Программы C# используют классы**

Любая программа на C# использует классы. Класс может делать что угодно, но большинство классов заточено на определенное действие. При создании программы добавляется отображающий страницу класс `MainPage`.

```
namespace Save_the_Humans
{
  public sealed partial class MainPage : Page
  {
```

При присвоении программе имени Save the Humans IDE создала пространство имен Save_the_Humans (пробелы при этом автоматически заменяются нижним подчеркиванием), добавив ключевое слово namespace. Все находящееся в скобках является частью пространства имен Save_the_Humans.

Это класс MainPage, который содержит код, заставляющий вашу страницу работать. IDE формирует его, получая команду создать проект Windows Store.

3 **Классы содержат методы**

Для выполнения различных действий классы используют методы. Методы производят некое действие на основе входных данных. Данные в метод передаются при помощи параметров. Именно параметры влияют на поведение метода. Некоторые методы возвращают значение. Ключевое слово `void` перед именем метода означает, что он не возвращает никаких данных.

Обращайте внимание на пары скобок. Среди них могут попадаться вложенные.

```
void startButton_Click(object sender, object e)
{
    StartGame();
}
```

*Эта строка **вызывает метод** StartGame(), в создании которого вам помогла IDE, когда вы попросили добавить заглушку метода.*

*У этого метода два **параметра**: sender и e.*

4 **Каждый оператор выполняет всего одно действие**

При заполнении метода `StartGame()` вы добавили набор **операторов**. Именно операторы составляют тело любого метода. При вызове метода сначала выполняется первый оператор, потом следующий и т. д. При достижении конца списка или оператора `return` метод завершает работу, и программа продолжает работу после вызвавшего метод оператора.

```
    private void StartGame()
    {
        human.IsHitTestVisible = true;
        humanCaptured = false;
        progressBar.Value = 0;
        startButton.Visibility =
                    Visibility.Collapsed;
        playArea.Children.Clear();
        playArea.Children.Add(target);
        playArea.Children.Add(human);
        enemyTimer.Start();
        targetTimer.Start();
    }
  }
}
```

*Это **метод** StartGame(), вызываемый при нажатии игроком кнопки Start.*

*Метод StartGame() содержит девять **операторов**. После каждого оператора ставится точка с запятой.*

Для улучшения читабельности строки можно разрывать. При построении программы это игнорируется.

Это закрывающаяся фигурная скобка в самом низу вашего файла MainPage.xaml.cs.

Часто задаваемые вопросы

В: Зачем нужны все эти фигурные скобки?

О: В C# фигурные скобки объединяют операторы в блоки. Скобки всегда используются парами. IDE помогает в поиске пар: достаточно щелчком выделить любую скобку, и вторая скобка будет выделена автоматически.

В: Почему при попытке запуска программы в окне Error List появляются сообщения об ошибках? Я думал, это происходит только при выполнении команды «Build Solution».

О: При выборе команды Start Debugging или при нажатии кнопки запуска программы первым делом сохраняются все файлы вашего решения и происходит попытка их компиляции. А в процессе компиляции кода, вне зависимости от того, при запуске или при построении решения она выполняется, IDE отображает все найденные ошибки в окне Error List.

Большинство ошибок, проявляющихся при попытке запуска программы, отображаются в окне Error List и выделяются красным подчеркиванием в коде.

IDE помогает писать корректный код.

Раньше для редактирования кода программисты пользовались простыми текстовыми редакторами, например Блокнотом. (Хорошо, если в редакторе были реализованы, например, поиск и замена или переход к строке с определенным номером нажатием ^G.) Для построения, запуска, отладки и публикации кода приходилось прибегать к приложениям командной строки.

За прошедшие годы Microsoft (и множество других компаний и индивидуальных разработчиков) изобрела ряд полезных вещей, таких как выделение ошибок, IntelliSense, WYSIWYG-редактирование, автоматическая генерация кода и многое другое.

В процессе эволюции Visual Studio превратилась в один из наиболее продвинутых инструментов для редактирования. И к счастью для вас, это еще и замечательный инструмент для изучения C# и разработки приложений.

Определите соответствие описаний и фрагментов кода. (Некоторые из них вам незнакомы, попробуйте угадать их функцию!)

```
myGrid.Background =
    new SolidColorBrush(Colors.Violet);
```

Задает свойства элемента управления TextBlock

```
// Цикл выполняется три раза
```

Ничего не делает — это комментарий, добавленный для объяснения смысла кода тем, кто будет читать программу

```
public sealed partial class MainPage : Page
{
   private void InitializeComponent()
   {
   :
   }
}
```

Отключает кнопку развертки ( ) в строке заголовка окна `Form1`

```
helloLabel.Text = "hi there";
helloLabel.FontSize = 24;
```

Специальный вид комментариев, которым IDE объясняет назначение целого блока кода

```
/// <summary>
/// При щелчке на кнопке появляется
/// изображение Rover
/// </summary>
```

Меняет фоновый цвет элемента управления Grid с именем `myGrid`

```
partial class Form1
{
   :
   this.MaximizeBox = false;
   :
}
```

Метод, который выполняется при отображении программой главной страницы

Вот правильные ответы на вопрос о назначении различных блоков кода. Сравните со своим вариантом!

```
myGrid.Background =
    new SolidColorBrush(Colors.Violet);
```

```
// Цикл выполняется три раза
```

```
public sealed partial class MainPage : Page
{
  private void InitializeComponent()
  {
   :
  }
}
```

```
helloLabel.Text = "hi there";
helloLabel.FontSize = 24;
```

```
/// <summary>
/// При щелчке на кнопке появляется
/// изображение Rover
/// </summary>
```

```
partial class Form1
{
   :
  this.MaximizeBox = false;
   :
}
```

Задает свойства элемента управления TextBlock

Ничего не делает — это комментарий, добавленный для объяснения смысла кода тем, кто будет читать программу

Отключает кнопку развертки ( ) в строке заголовка окна `Form1`

Окно? Не страница? Здесь речь идет о приложении для рабочего стола с формами и окнами, о которых мы еще поговорим.

Специальный вид комментариев, которым IDE объясняет назначение целого блока кода

Меняет фоновый цвет элемента управления Grid с именем `myGrid`

Метод, который выполняется при отображении программой главной страницы

Классы могут принадлежать одному пространству имен

Рассмотрим два файла с классами из программы PetFiler2 (Домашний любимец). В них содержатся три класса: Dog (Собака), Cat (Кошка) и Fish (Рыбка). Так как все они принадлежат пространству имен PetFiler2, операторы в методе Dog.Bark() (Собака лает) могут вызывать операторы Cat.Meow() (Кошка мяукает) и Fish.Swim() (Рыбка плавает). Распределение различных имен пространств и классов по файлам не влияет на действия, выполняемые после запуска.

Ключевое слово public означает, что методы этого класса доступны из любого другого класса.

SomeClasses.cs

```
namespace PetFiler2 {

  class Dog {

    public void Bark() {
      // Здесь операторы
    }

  }

  partial class Cat {

    public void Meow() {
      // Еще операторы
    }

  }
}
```

MoreClasses.cs

```
namespace PetFiler2 {

  class Fish {

    public void Swim() {
      // операторы
    }

  }

  partial class Cat {

    public void Meow() {
      // операторы
    }

  }
}
```

Классы из одного пространства имен могут «видеть» друг друга, даже находясь в разных файлах.

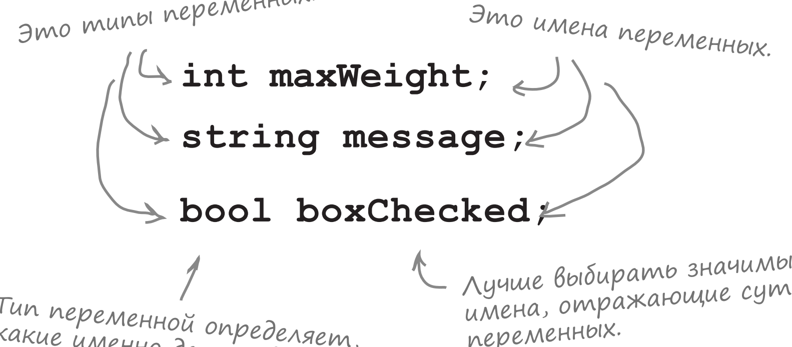
Для помещения класса в разные файлы пользуйтесь ключевым словом "partial". При написании кода в процессе чтения данной книги вы вряд ли будете к этому прибегать, но так поступает IDE, разбивая страницу на два файла и помещая код XAML в файл MainPage.xaml, а код C# — в файл MainPage.xaml.cs.

Что такое переменные

Все программы работают с данными. Данные могут быть представлены в виде документа, графического фрагмента, видеоигры или мгновенного сообщения. Для их хранения программа использует **переменные**.

Объявление переменных

Объявить (declare) переменную — значит указать программе ее *тип* и *имя*. Благодаря типам невозможно скомпилировать программы в случае, когда вы сделали ошибку и пытаетесь сделать нечто лишенное смысла, например вычесть Fido из 48353.



Будьте осторожны!

Знаете ли вы другие языки программирования?

Если да, то вам, скорее всего, уже известна большая часть этого материала. Но выполнить упражнения все равно имеет смысл, так как не исключено, что C# чем-то отличается от известных вам языков.

Переменные меняются

В процессе работы программы любой переменной может быть присвоено произвольное значение. То есть значения переменных ***меняются***. Это ключевая идея любой программы. К примеру, если переменной `myHeight` было присвоено значение 63:

```
int myHeight = 63;
```

как только имя `myHeight` появится в коде, C# заменит его значением 63. Представим, что позднее ему было присвоено значение 12:

```
myHeight = 12;
```

Теперь C# будет заменять параметр `myHeight` на число 12, несмотря на то что имя переменной не изменилось.

Для работы с числами, текстом, булевыми значениями и любым другим видом данных используйте переменные.

Присвоение значений

Поместите в программу эти операторы:

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

При попытке запустить программу IDE откажется компилировать код. Компилятор проверил ваши переменные и обнаружил, что им не присвоено никакого значения. Чтобы избежать подобных ошибок, имеет смысл комбинировать оператор объявления переменной с оператором присвоения значения:

Отсутствие у переменных значения является препятствием для компиляции. Этой ошибки легко избежать, объединив в один оператор объявление переменной и присвоение ей значения.

Значение переменной всегда можно изменить. Поэтому присвоение начальных значений не создает никаких неудобств.

Некоторые типы переменных

Тип переменной определяет, какие именно данные в ней можно хранить. Подробно типы будут рассматриваться в главе 4, а пока запомните три наиболее используемых типа. Переменные типа `int` сохраняют целые числа, переменные типа `string` — текст, а переменные типа `bool` — логические значения true/false.

> пе-ре-мен-ный, прил.
> умеющий меняться или приспосабливаться
> *Предсказывать погоду было бы намного проще, если бы метеорологам не приходилось принимать во внимание такое количество переменных.*

Знакомые математические символы

Для программиста слово string почти всегда означает «строка текста», а сокращение int указывает на целое число.

Числа, хранящиеся в переменных, можно складывать, вычитать, умножать и делить. В этом вам помогут **операторы (operators)**. Часть из них вам уже знакома. Рассмотрим код, решающий несложную математическую задачку:

```
int number = 15;
number = number + 10;
number = 36 * 15;
number = 12 - (42 / 7);
number += 10;
number *= 3;
number = 71 / 3;

int count = 0;
count ++;
count --;

string result = "hello";
result += " again " + result;
output.Text = result;
result = "the value is: " + count;
result = "";

bool yesNo = false;
bool anotherBool = true;
yesNo = !anotherBool;
```

Мы объявили переменную number целочисленного типа и присвоили ей значение 15. Затем прибавили 10, в результате чего она получила значение 25.

Третий оператор присваивает переменной number значение 36 * 15 = 540. Следующий оператор присваивает новое значение 12 – (42 / 7) = 6.

Оператор += означает, что к значению переменной number нужно прибавить 10. Так как ее значение сейчас равно 6, добавив 10, вы получите 16.

Оператор *= означает, что вы должны умножить текущее значение переменной на 3, в итоге получаем 48.

Если71 разделить на 3, получится 23.666666... но результат деления целых чисел также должен быть целым, поэтому 23.666... округляется до 23.

Целочисленные переменные используются в счетчике в сочетании с операторами ++ и --. Инкремент ++ увеличивает значения на 1, а декремент — уменьшает на 1.

Элемент управления TextBlocke вызывает окно с текстом «hello again hello».

Оператор + в данном случае соединяет вместе две строки. При этом числа автоматически преобразовываются к типу string.

Пустые кавычки обозначают строку, не содержащую символов.

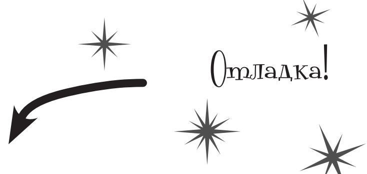
Переменные типа bool принимают значения true или false. Оператор ! обозначает отрицание и меняет значение с true на false и обратно.

Расслабьтесь

Не волнуйтесь, если вы не смогли запомнить всё.

Эта информация будет часто повторяться в книге.

Наблюдение за переменными в процессе отладки



Отладчик позволяет понять, как работает программа. Рассмотрим код с предыдущей страницы.

1. **Создадим для магазина Windows новый проект типа Blank App (XAML).** Перетащите на страницу элемент TextBlock и назовите его `output`. Затем добавьте кнопку Button и дважды щелкните на ней для добавления метода `Button_Click()`. IDE автоматически откроет редактор кода. Введите в метод код с предыдущей страницы.

2. **Вставим в первую строку точку останова.** Щелкните правой кнопкой мыши на первой строке кода (`int number = 15;`) и выберите в меню Breakpoint команду Insert Breakpoint. (Также можно выделить строку и воспользоваться командой Debug→Toggle Breakpoint или нажать клавишу F9.)

Chapter 2 - Program 1

MainPage.xaml.cs — Chapter_2__Program_1.MainPage — OnNavigatedTo(NavigationEventArgs e)

```
/* Double-clicking on the Button in the designer caused it to
 * create the empty Button_Click() method.
 */

private void Button_Click(object sender, RoutedEventArgs e)
{
    int number = 15;      // There's a breakpoint on this line
    number = number + 10;
    number = 36 * 15;
    number = 12 - (42 / 7);
    number += 10;
    number *= 3;
    number = 71 / 3;

    int count = 0;
    count++;
    count--;

    string result = "hello";
    result += " again " + result;
    output.Text = result;
    result = "the value is: " + count;
    result = "";

    bool yesNo = false;
    bool anotherBool = true;
    yesNo = !anotherBool;
}
```

150 %

Комментарии, начинающиеся с двух и более косых черт или заключенные между / и */ в IDE выделены зеленым. Компилятор их игнорирует, поэтому там можно писать что угодно.*

После вставки точки останова строка кода выделяется красным, а на полях рядом с ней появляется красная точка.

При отладке кода программа прекращает свою работу в точке останова и дает возможность проверить и отредактировать значения переменных.

Создание нового проекта Blank App заставит IDE создать проект с чистой страницей. Можно присвоить ей имя UseTheDebugger. В процессе чтения книги вам предстоит создать много программ, и к некоторым из них вы можете захотеть вернуться позже.

Переверните страницу и продолжим!

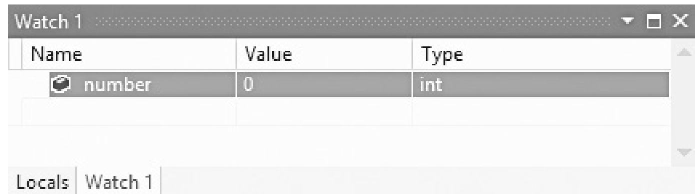
3 Отладка программы.
Щелкните на кнопке Start Debugging (или нажмите клавишу F5) (Можно использовать команду Start Debugging, меню Debug.) Программа начнет работу и откроет страницу.

При отладке приложения для магазина Windows для возвращения в отладчик можно нажать клавишу с логотипом Windows и букву D. На устройствах с сенсорными экранами следует провести пальцем слева направо. Приостановить или прекратить работу отладчика позволяет панель Debug или команды меню.

4 Переход к точке останова.
Как только программа дойдет до точки останова, IDE автоматически вызовет редактор кода и выделит нужную строчку желтым цветом.

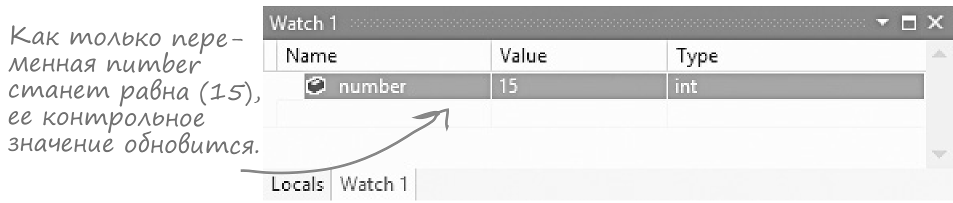
```
int number = 15;
number = number + 10;
number = 36 * 15;
number = 12 - (42 / 7)
number += 10;
number *= 3;
number = 71 / 3;
```

5 Контрольное значение переменной `number`.
Щелкните правой кнопкой мыши на переменой number и выберите в меню команду Add Watch. На панели в нижней части IDE откроется окно Watch:



Функция Watch помогает отслеживать значения переменных на каждом этапе выполнения кода. Вы оцените ее удобство по мере усложнения ваших программ.

6 Обход кода.
Нажмите F10, чтобы обойти эту строку. (Можно также выбрать в меню Debug команду Step Over или щелкнуть на кнопке Step Over панели инструментов Debug.) Выделенная строка будет выполнена, и значение переменной `number` станет равным 15. Теперь желтым будет выделена следующая строка кода:



В процессе отладки достаточно навести на переменную указатель мыши, чтобы появилось всплывающее окно со значением этой переменной.

7 Продолжение работы программы
Для возвращения в обычный режим выполнения программы нажмите F5 или выберите в меню Debug команду Continue.

Циклы

В большинстве сложных программ одни и те же действия повторяются больше одного раза. В такой ситуации на помощь приходят **циклы (loops)** — они заставляют программу выполнять набор операторов — до тех пор, пока указанное вами условие не примет значение `true` (или `false`!).

```
while (x > 5)
{
    x = x - 3;
}
```

Вот почему так важны переменные логического типа.

В цикле while операторы внутри фигурных скобок выполняются до тех пор, пока условие имеет значение true.

СОВЕТ: Скобки

Наличие непарных скобок является препятствием к построению программы. К счастью, достаточно навести указатель мыши на скобку, и IDE тут же выделит ее «вторую половину»:

```
bool test;
while (test == true)
{
    // Contents of the loop
}
```

Цикл for состоит из трех операторов. Первый задает начало цикла. Третий оператор цикла будет выполняться, пока соблюдается условие, заданное вторым оператором.

```
for (int i = 0; i < 8; i = i + 2)
{
    // Все находящееся в этих скобках
    // запускается 4 раза
}
```

Написание простого цикла при помощи фрагментов кода

Через минуту вам предстоит написать программу. Эту задачу можно ускорить средствами IDE. Если после ввода ключевого слова `for` дважды нажать кнопку Tab, нужный код появится автоматически. При вводе имени переменной оно автоматически появится во всем фрагменте. Повторное нажатие кнопки Tab перемещает курсор на переменную length.

Количество «прогонов» цикла зависит от значения параметра length (длина). Этот параметр может быть как константой, так и переменной.

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

Достаточно изменить имя переменной здесь, и оно автоматически изменится во всем фрагменте кода.

Оператор выбора

Операторы if/else инициируют действия при выполнении или невыполнении **заданных условий**. В большинстве случаев речь идет о проверке равенства. Для этого используется оператор ==. Не путайте его с оператором (=), который присваивает переменным определенное значение.

```
string message = "";

if (someValue == 24)
{
    message = "Значение 24.";
}
```

Оператор if начинается с проверки условия.

Оператор в фигурных скобках выполняется только при соблюдении условия.

Для сравнения двух параметров используется оператор в виде двойного знака равенства.

```
if (someValue == 24)
{
    // Количество операторов в скобках
    // может быть произвольным
    message = "Значение 24.";
} else {
    message = "Значение отлично от 24.";
}
```

При соблюдении заданного условия выполняется первый оператор, в противном случае — второй.

Будьте осторожны!

Следите за количеством знаков равенства в операторе!

Оператор, состоящий из одного знака (=), присваивает переменной значение, а из двух знаков (==) — сравнивает две переменные. Это очень распространенная ошибка. Если вы видите сообщение «невозможно преобразовать тип 'int' в тип 'bool'», скорее всего, именно эта ошибка является его причиной.

Выберите для этого проекта запоминающееся имя, так как мы к нему еще вернемся.

Приложение с нуля

Эти кроссовки намекают, что вам предстоит писать код самостоятельно.

Упражнение

В любой программе всю работу совершают операторы. Вы уже видели, как они вписываются в страницу. А теперь пришло время разобраться во всех строках кода. Создайте **новый проект для магазина Windows Store типа Blank App**. Но вместо удаления сформированной на основе шаблона страницы *MainPage.xaml* воспользуемся IDE, чтобы добавить к сетке три строки и два столбца и поместить в ячейки четыре элемента Button и один TextBlock.

Построй эту страницу

Сетка страницы состоит из трех строк и двух столбцов. В определениях строк высота задана как `1*`, что создает `<RowDefinition/>`, не обладающий свойствами. Аналогичная ситуация с высотой столбцов.

На странице четыре элемента Button, распределенных по ячейкам. Через свойство `Content` поменяйте их текст на Show a message, If/else, Another conditional test и A loop.

MainPage.xaml

Show a message

If/else

Another conditional test

A loop

Каждая кнопка отцентрирована в своей ячейке. Для указания местоположения пользуйтесь свойствами Grid.Row и Grid.Column (по умолчанию они равны 0).

Здесь находится элемент TextBlock. Вы не увидите его, пока не присвоите ему какой-либо текст. Он расположен по центру нижней строки. Для объединения столбцов параметру ColumnSpan присвоили значение 2.

В нижней ячейке находится элемент TextBlock с именем `myLabel`. С свойству `Style` присвойте значение `BodyTextStyle`.

Используйте свойство `x:Name`, чтобы дать кнопкам имена `button1`, `button2`, `button3` и `button4`. Дважды щелкните на каждой, чтобы добавить метод-обработчик события.

Для этого потребуется команда Edit Style из контекстного меню. Но если вам сложно выделить элемент управления TextBlock, щелкните правой кнопкой на одноименной строке в окне Document Outline.

Упражнение
Решение

Вот решение. Вы написали то же самое? Отличаются только расстановка переносов строки и порядок свойств? Тогда все в порядке!

Многие программисты не пользуются IDE для создания кода XAML, они пишут его вручную. Если бы мы попросили вас написать код XAML без помощи IDE, смогли бы вы сделать это?

```
<Page
    x:Class="Chapter_2___Program_2.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Chapter_2___Program_2"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>

        <Button x:Name="button1" Content="Show a message"
                HorizontalAlignment="Center" Click="button1_Click"/>

        <Button x:Name="button2" Content="If/else" HorizontalAlignment="Center"
                Grid.Column="1" Click="button2_Click"/>

        <Button x:Name="button3" Content="Another conditional test" HorizontalAlignment="Center"
                Grid.Row="1" Click="button3_Click"/>

        <Button x:Name="button4" Content="A loop" HorizontalAlignment="Center"
                Grid.Column="1" Grid.Row="1" Click="button4_Click"/>

        <TextBlock x:Name="myLabel" HorizontalAlignment="Center" VerticalAlignment="Center"
                   Grid.Row="2" Grid.ColumnSpan="2" Style="{StaticResource BodyTextStyle}"/>
    </Grid>
</Page>
```

Это теги <Page> и <Grid>, которые IDE сгенерировала при создании пустого приложения.

Это определение строк и столбцов: три строки и два столбца.

При двойном щелчке на каждой кнопке IDE генерирует метод, в названии которого после имени кнопки следует _Click.

Эта кнопка из второй строки и второго ряда, поэтому свойства равны 1.

Как вы думаете, почему левому столбцу и верхней строке соответствует номер 0, а не 1? Почему для верхней левой ячейки можно не указывать `Grid.Row` и `Grid.Column`?

Пусть каждая кнопка что-то делает

Вот как должна работать наша программа. Нажатие каждой кнопки обновляет сообщение в TextBlock (который мы назвали myLabel). Для этого мы добавим код к каждому из четырех обработчиков методов, сгенерированных IDE. Приступим!

Эта надпись является командой открыть IDE и выполнить действия. Мы указываем, что делать и куда смотреть, чтобы извлечь максимум знаний.

1 **Пусть кнопка button1 обновляет метку.**
В код метода `button1_Click()` введите представленный ниже фрагмент. Это даст вам понять, зачем нужен каждый из операторов и каким образом получается именно такой результат:

```
name is Quentin
x is 51
d is 1.5707963267949
```

Вот код для этой кнопки:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    // это комментарий
    string name = "Quentin";
    int x = 3;
    x = x * 17;
    double d = Math.PI / 2;
    myLabel.Text = "name is " + name
        + "\nx is " + x
        + "\nd is " + d;
}
```

x — это переменная. “int” указывает, что она целая, а дальше ей присваивается значение 3.

Эта строка вывода: обновленный текст в элементе TextBlock с именем myLabel.

Объект PI относится к встроенному классу Math из пространства имен System, поэтому в верхней части файла есть строка **using System**.

К счастью, эту строку сгенерирует для вас IDE.

\n — это **esc-последовательность**, добавляющая в текст элемента TextBlock символ переноса строки.

Несколько советов

- Не забудьте, что после оператора должна стоять точка с запятой:

```
name = "Joe";
```

- К коду можно добавить комментарий, поставив две косые черты:

```
// это комментарий
```

- Переменные объявляются указанием **имени** и **типа** (типы будут рассмотрены в главе 4):

```
int weight;
// это целое
```

- Код классов и методов помещается в фигурные скобки:

```
public void Go() {
    // ваш код
}
```

- В большинстве случаев допустимы дополнительные пробелы:

```
int j     =      1234  ;
```

это то же самое, что и:

```
int j = 1234;
```

Запустите программу и убедитесь, что ваши результаты совпадают со скриншотом на этой странице.

Завершение программы на следующей странице!

Проверка условий

Рассмотрим работу условного оператора **if/else**.

Проверка при помощи операторов

Вы уже познакомились с оператором ==, проверяющим, равны ли друг другу две переменные. Существуют и другие операторы сравнения:

- ★ Оператор != в отличие от оператора == принимает значение true, если переменные **не равны**.
- ★ Операторы > и < выявляют, какая переменная больше.
- ★ Операторы ==, !=, > и < называются **операторами сравнения**. С их помощью осуществляется **проверка условий.**
- ★ Операторы сравнения можно комбинировать при помощи оператора && (И) и оператора || (ИЛИ). К примеру, условие i равно 3 или j меньше 5 записывается как `(i == 3) || (j < 5)`.

Использование условного оператора для сравнения двух чисел называется проверкой условия.

Для возвращения в режим редактирования кода остановите отладку программы. Это можно сделать при помощи команды Stop Debugging из меню Debug.

2 Задание переменной и проверка ее значения
Ниже вы видите код для второй кнопки. Он содержит оператор if/else, проверяющий, равна ли целочисленная **переменная** x десяти.

```
private void button2_Click(object sender, RoutedEventArgs e)
{
    int x = 5;
    if (x == 10)
    {
        myLabel.Text = "x must be 10";
    }
    else
    {
        myLabel.Text = "x isn't 10";
    }
}
```

Объявим переменную x и присвоим ей значение 5. Затем проверим, равна ли она 10.

x isn't 10

Это результат работы программы. Отредактируйте код таким образом, чтобы сообщение изменилось на x must be 10.

3 Проверка еще одного условия

А этот результат дает третья кнопка. Теперь измените значение someValue с 4 на 3. Элемент TextBlock обновится дважды, но так быстро, что вы этого не заметите. Поместите на первый оператор точку останова и выполните пошаговый проход, переходя к приложению и обратно нажатием Alt-Tab, чтобы убедиться в обновлении элемента TextBlock.

this line runs no matter what

В этой строке проверяется, равна ли переменная someValue трем и содержит ли переменная name значение «Joe».

```
private void button3_Click(object sender, RoutedEventArgs e)
{
    int someValue = 4;
    string name = "Bobbo Jr.";
    if ((someValue == 3) && (name == "Joe"))
    {
        myLabel.Text = "x is 3 and the name is Joe";
    }
    myLabel.Text = "this line runs no matter what";
}
```

4 Добавление цикла

Ниже показан код для последней кнопки. Он содержит два цикла: цикл **while**, реализующий операторы в скобках, пока выполняется заданное условие и цикл **for**. Посмотрим, как это работает.

```
private void button4_Click(object sender, RoutedEventArgs e)
{

    int count = 0;

    while (count < 10)
    {
        count = count + 1;
    }

    for (int i = 0; i < 5; i++)
    {
        count = count - 1;
    }

    myLabel.Text = "The answer is " + count;
}
```

Цикл работает, пока значение переменной count меньше 10.

Это условие проверяется перед началом работы цикла. Цикл запускается при соблюдении условия.

Здесь переменной, используемой для подсчета проходов цикла, присваивается начальное значение.

Оператор, при каждом проходе цикла увеличивающий значение i на 1.

Перед тем как щелкнуть на кнопке попытайтесь определить, какое значение будет получено на выходе. Проверьте правильность своих выводов!

Возьми в руку карандаш

Попрактикуйтесь в проверке условий циклов. Посмотрите на код и напишите пояснения к каждой его строке.

Заполните остальные строки по аналогии.

```
int result = 0; // переменная, в которую будет записан результат
int x = 6; // объявим переменную x и    присвоим значение 6
while (x > 3) {
  // операторы будут выполняться, пока ..............................
  result = result + x; // прибавим x ..............................
  x = x - 1; // вычтем ..............................
}
for (int z = 1; z < 3; z = z + 1) {
  // начнем цикл с ..............................
  // цикл работает, пока ..............................
  // на каждом этапе ..............................
  result = result + z; // ..............................
}
// Следующий оператор вызывает окно диалога с текстом
// ..............................
myLabel.Text ="Результат равен " + result;
```

Еще о проверке условий

Проверка условий осуществляется при помощи операторов сравнения:

```
x < y (меньше чем)
x > y (больше чем)
x == y (равно)
```

Эти операторы используются чаще всего.

Подождите! А что случится с циклом, если я напишу условие, которое никогда не выполняется?

Значит, цикл никогда не закончится!

Результатом каждой проверки условия является значение **true** или **false**. В первом случае цикл выполняется еще раз. Рано или поздно вы должны получить другой результат, и тогда цикл закончится.

В программах иногда используются бесконечные циклы.

Возьми в руку карандаш

Вот несколько циклов. Какие из них являются бесконечными? Сколько раз выполняются остальные циклы?

Цикл #1

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

Цикл #2

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

В цикле for сначала выполняется проверка условия, а потом итератор.

Цикл #3

```
int j = 2;
for (int i = 1; i < 100;
     i = i * 2)
{
   j = j - i;
   while (j < 25)
   {
      j = j + 5;
   }
}
```

Сколько раз будет выполнен этот оператор? (j = j - i;)

Цикл #4

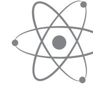
```
while (true) { int i = 1;}
```

Цикл #5

```
int p = 2;
for (int q = 2; q < 32;
     q = q * 2)
{
   while (p < q)
   {
      p = p * 2;
   }
   q = p - q;
}
```

Сколько раз будет выполнен этот оператор? (q = p - q;)

*При повторении итератора p = p * 2 помните, что начальное значение p равно 2.*



Мозговой штурм

Подумайте, в какой ситуации может понадобиться бесконечный цикл?

Возьми в руку карандаш
Решение

Вот как следовало закончить строчки комментариев к программе, иллюстрирующей работу циклов и проверку условий.

```
int result = 0; // переменная, в которую будет записан результат
int x = 6; // объявим переменную x и присвоим значение 6
while (x > 3) {
  // операторы будут выполняться, пока x больше 3
  result = result + x; // прибавим x к переменной result
  x = x - 1; // вычтем 1 из переменной x
}
for (int z = 1; z < 3; z = z + 1) {
  // начнем цикл с объявления переменной z и присвоения ей значения 1
  // цикл работает, пока z меньше 3
  // на каждом этапе значение переменной z увеличивается на 1
  result = result + z; // переменная z прибавляется к переменной result
}
// Следующий оператор вызывает окно диалога с текстом
// Результат равен 18
myLabel.Text = ("Результат равен" + result);
```

Этот цикл выполняется дважды: сначала при z, равном 1, затем при z, равном 2. Как только z получит значение 3, условие перестанет соблюдаться и цикл остановится.

Возьми в руку карандаш
Решение

Сравните свои ответы на вопрос, сколько раз будет выполнен тот или иной цикл, с правильными ответами.

Цикл #1
Будет выполнен один раз.

Цикл #2
Бесконечный цикл

Цикл #3
Будет выполнен семь раз.

Цикл #4
Еще один бесконечный цикл.

Цикл #5
Будет выполнен восемь раз.

Попробуйте решить задачу номер пять. Это отличная возможность поработать с отладчиком! Сделайте оператор q = p - q точкой останова и проверьте, как изменяются значения переменных p и q с помощью окна Watches.

Часто задаваемые вопросы

В: Всегда ли оператор принадлежит к классу?

О: Да. Все операторы принадлежат к определенным классам, а все классы в свою очередь принадлежат пространствам имен. При задании свойств элементов в конструкторе может показаться, что некоторые операторы находятся вне классов. Но внимательное рассмотрение кода показывает, что на самом деле это не так.

В: Существуют ли пространства имен, которыми нельзя пользоваться? А как насчет тех, которые я *обязан* использовать?

О: Да, некоторых названий лучше избегать. К примеру, пространство имен System зарезервировано для Windows Store API и .NET Framework. Именно здесь находятся такие инструменты, как позволяющий управлять последовательностями данных System.Linq и предназначенный для работы с файлами и потоками данных System.IO. Но по большей части вы можете называть пространства имен как вам нравится. Или отдать это на откуп IDE, которая автоматически будет присваивать имена, взяв за основу название программы.

В: А все-таки, зачем нужно ключевое слово partial?

О: Оно позволяет распределять код одного класса между разными файлами. При создании страницы IDE сохраняет редактируемый код в один файл (MainPage.xaml), а автоматически модифицируемый код — в другой файл (MainPage.xaml.cs). При этом если объявить в верхней части файла пространство имен, в него попадут все файлы, перечисленные в расположенных ниже фигурных скобках. В одном файле могут находиться объекты из разных пространств имен и классов. Но об этом мы поговорим в следующей главе.

В: Что происходит с кодом, который был автоматически создан IDE, если воспользоваться командой Undo?

О: Попробуйте, и вы сами найдете ответ на этот вопрос! Перетащите на форму кнопку и поменяйте ее свойства. А затем воспользуйтесь командой отмены. Вы увидите, что в простых случаях IDE просто возвращает вас в предыдущую точку. Но в более сложных случаях, например при работе с базой данных, может появиться предупреждение, что вы вносите изменения, отменить которые IDE не сможет. Впрочем, в упражнениях к этой книге данная ситуация исключена.

В: Насколько внимательно нужно относиться к автоматически создаваемому коду?

О: Относитесь к нему достаточно внимательно. Нужно понимать соответствие между кодом и вашими действиями, чтобы при необходимости иметь возможность отредактировать его вручную. Впрочем, в подавляющем большинстве случаев все необходимые изменения можно проделать с помощью IDE.

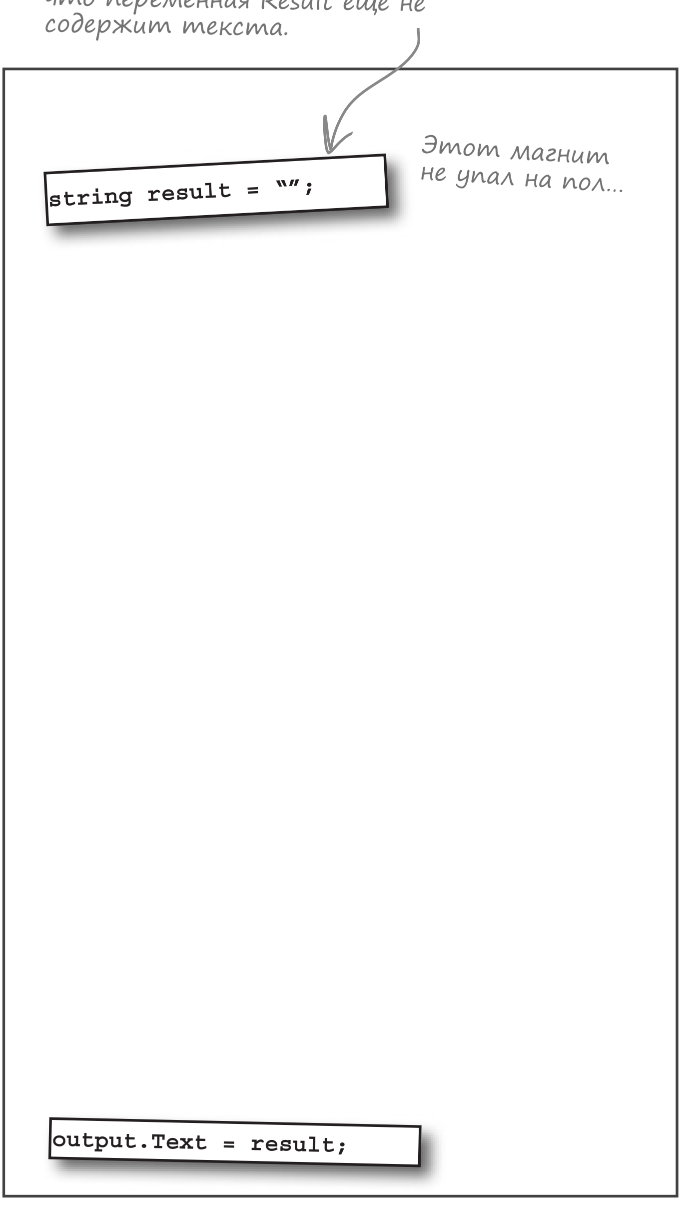
КЛЮЧЕВЫЕ МОМЕНТЫ

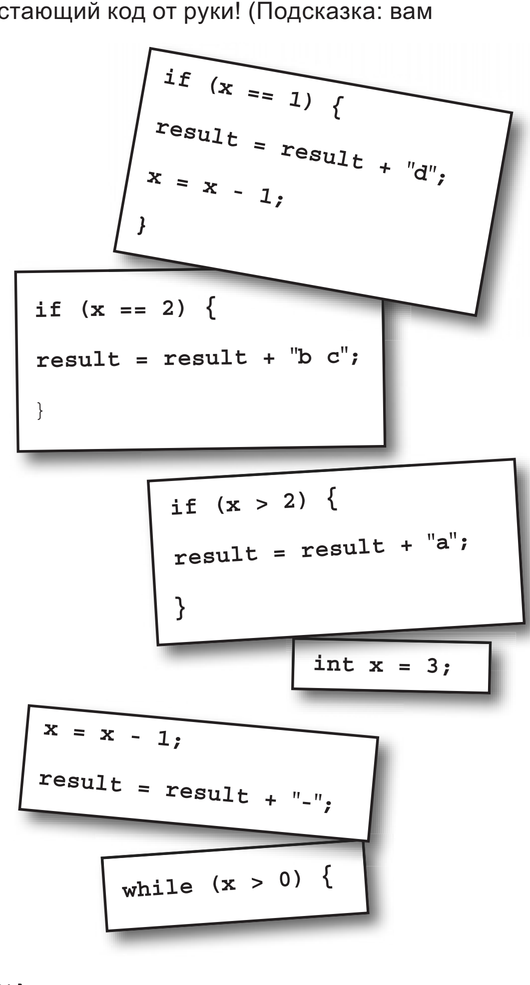
- Вашу программу заставляют работать операторы, которые всегда принадлежат к какому-либо классу. Класс же в свою очередь находится в каком-то пространстве имен.
- В конце операторов стоит знак (`;`).
- В ответ на ваши действия Visual Studio автоматически добавляет в программу код.
- Блоки кода, такие как классы, цикл `while`, операторы if/else и многие другие виды операторов, заключаются в фигурные скобки `{ }`.
- Результатом проверки условия является значение `true` или `false`. С его помощью определяется, будет ли закончен цикл, а также какой именно блок кода будет реализован в операторе if/else.
- Для хранения данных используются переменные. Оператор = присваивает переменной значение, а оператор == сравнивает значения двух переменных.
- Цикл `while` выполняет все операторы в фигурных скобках, пока результатом *проверки условия* является значение `true`.
- Как только проверка условия дает значение `false`, цикл `while` прекращает работу, и программа переходит к оператору, расположенному сразу после цикла.

Магниты с кодами

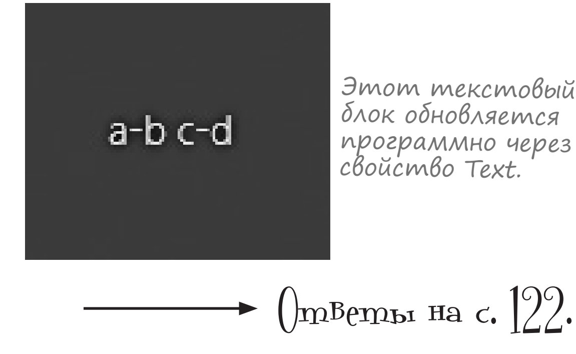
Мы поместили блоки с фрагментами кода на C# на магниты для холодильника. Составьте из них работающую программу, результатом которой является окно с сообщением. Некоторые магнитики упали на пол, и они слишком малы, чтобы их поднимать, поэтому просто впишите недостающий код от руки! (Подсказка: вам понадобится всего пара фрагментов!)

Пустыми кавычками обозначена пустая строка. Это означает, что переменная Result еще не содержит текста.





Результат:



Этот текстовый блок обновляется программно через свойство Text.

Ответы на с. 122.

Вам придется выполнить много подобных упражнений. Ответы можно найти на следующих страницах. Если вы не знаете, как решить задачу, не стесняйтесь подсмотреть.

В процессе чтения книги вам предстоит создать много разных приложений, и все их нужно будет как-то именовать. Сейчас мы рекомендуем имя "PracticeUsingIfElse". Созданные в одной главе программы желательно класть в одну и ту же папку.

Упражнение

Попрактикуемся в использовании оператора if/else. Сможете создать программу?

Постройте страницу.
Это сетка с двумя строками и двумя столбцами.

Если создать две строки и присвоить одной из них высоту 1, покажется, что она пропала, так как ее размер станет очень маленьким. Но она вернется, если сделать высоту второй строки равной 1*.*

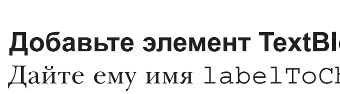
Добавьте кнопку и флажок.
Элементу Button присвойте имя `changeText`, а элементу CheckBox — имя `enableCheckbox`. Командой Edit Text задайте текст (клавиша Escape завершит редактирование). Щелкните на каждом элементе правой кнопкой мыши и выберите Reset Layout→All, а также убедитесь, что параметры VerticalAlignment и HorizontalAlignment имеют значение `Center`.

Добавьте элемент TextBlock.
Дайте ему имя `labelToChange`, а свойство `Grid.Row` сделайте равным `"1"`.

Элемент TextBlock должен показать это сообщение, если пользователь щелкает на кнопке, НЕ УСТАНОВИВ флажок.
Проверяем, установлен ли флажок:

```
enableCheckbox.IsChecked == true
```

Редактирование текста отключено



Если условие **НЕ** выполняется, программа должна выполнить два оператора:

```
labelToChange.Text = "Text changing is disabled";
labelToChange.HorizontalAlignment = HorizontalAlignment.Center;
```

Подсказка: этот код нужно поместить в блок else.

Если флажок УСТАНОВЛЕН, сделайте так, чтобы при щелчке слева TextBlock показывал текст Left, а справа — Right.

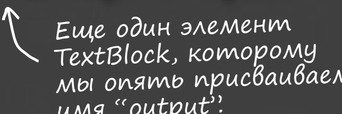
Если свойство `Text` метки имеет значение `"Right"`, его следует изменить на `"Left"`, поменяв свойство `HorizontalAlignment` на `HorizontalAlignment.Left`. В противном случае поменяйте текст на `"Right"`, а свойство `HorizontalAlignment` — на `HorizontalAlignment.Right`. В результате при нажатии кнопки текст будет меняться с одного на другой, но только если пользователь установил флажок.

Ребус в бассейне

Возьмите фрагменты кода из бассейна и поместите их на пустые строчки. Каждый фрагмент можно использовать один раз. В бассейне есть и лишние фрагменты. Нужно составить компилируемую и работающую программу. Имейте в виду, что задача не так проста, как кажется на первый взгляд!

```
int x = 0;
string poem = "";

while ( __________ ) {

  _____________________________
  if ( x < 1 ) {
    ___________________________
  }
  _____________________________

  if ( __________ ) {

    ___________________________

    ___________
  }
  if ( x == 1 ) {

    ___________________________
  }
  if ( ___________ ) {

    ___________________________
  }

  ____________
}
__________________
```

Результат:



Еще один элемент TextBlock, которому мы опять присваиваем имя "output".

Такие задачки иногда будут встречаться в книге. Любители логических загадок должны их оценить. Но даже если вы не относитесь к их числу, попробуйте найти решение.

Каждый фрагмент кода можно использовать только один раз!

```
x > 0
x < 1
x > 1
x > 3
x < 4

x = x + 1;
x = x + 2;
x = x  - 2;
x = x  - 1;

poem = poem + "  ";
poem = poem + "a";
poem = poem + "n";
poem = poem + "an";

output.Text = poem;

poem = poem + "noys  ";
poem = poem + "oise  ";
poem = poem + "  oyster ";
poem = poem + "annoys";
poem = poem + "noise";
```

Ответ на с. 122.

Вот как выглядит наша программа с операторами if/else

Как обычно для облегчения восприятия мы добавили переносы строк.

Код XAML для сетки:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Button x:Name="changeText" Content="Change the label if checked"
            HorizontalAlignment="Center" Click="changeText_Click"/>

    <CheckBox x:Name="enableCheckbox" Content="Enable label changing"
              HorizontalAlignment="Center" IsChecked="true" Grid.Column="1"/>

    <TextBlock x:Name="labelToChange" Grid.Row="1" TextWrapping="Wrap"
               Text="Press the button to set my text"
               HorizontalAlignment="Center" VerticalAlignment="Center"
               Grid.ColumnSpan="2"/>
</Grid>
```

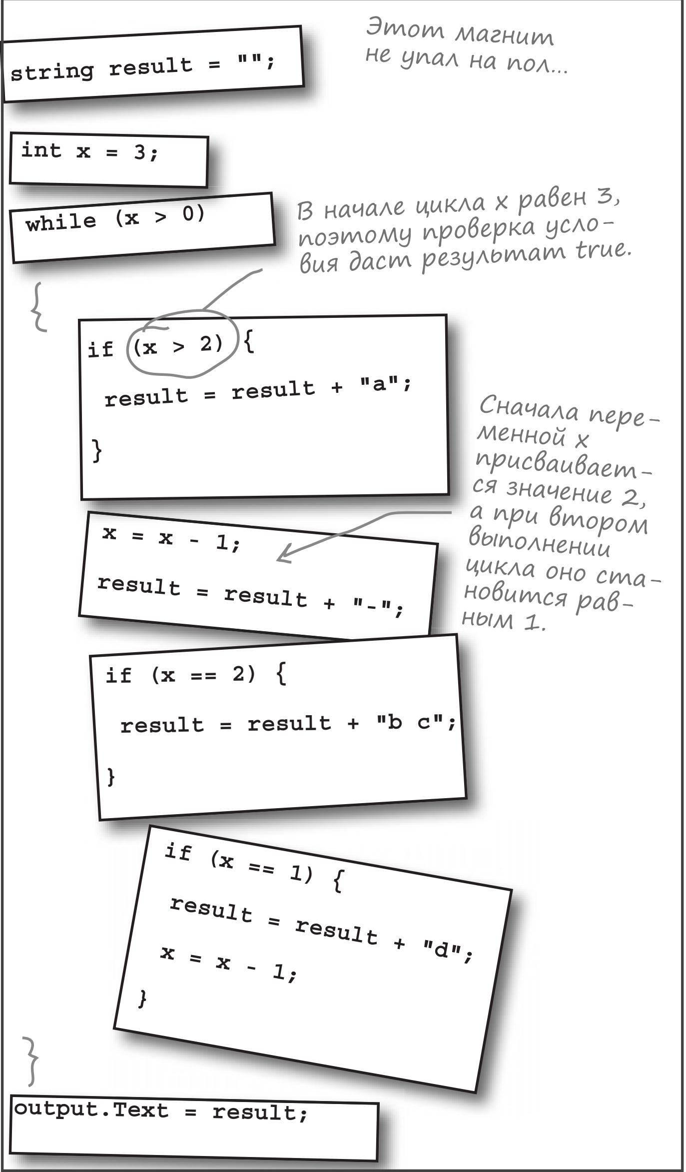
Если вы дважды щелкнули на кнопке в окне конструктора, пока ей еще не присвоено имя, обработчик события Click получит имя Button_Click_1(), а нам требуется changeText_Click().

А это код C# для метода обработчика события кнопки:

```
private void changeText_Click(object sender, RoutedEventArgs e)
{
    if (enableCheckbox.IsChecked == true)
    {
        if (labelToChange.Text == "Right")
        {
            labelToChange.Text = "Left";
            labelToChange.HorizontalAlignment = HorizontalAlignment.Left;
        }
        else
        {
            labelToChange.Text = "Right";
            labelToChange.HorizontalAlignment = HorizontalAlignment.Right;
        }
    }
    else
    {
    labelToChange.Text = "Text changing is disabled";
    labelToChange.HorizontalAlignment = HorizontalAlignment.Center;
    }
}
```

Магниты с кодами. Решение

Ну что ж, пришло время посмотреть, как же правильно расположить магниты с фрагментами кода, упавшие на пол с холодильника.



Ребус в бассейне. Решение

```
int x = 0;
string poem = "";

while ( x < 4 ) {

  poem = poem + "a";
  if ( x < 1 ) {
    poem = poem + " ";
  }
  poem = poem + "n";

  if ( x > 1 ) {

    poem = poem + " oyster";

    x = x + 2;
  }
  if ( x == 1 ) {

    poem = poem + "noys ";
  }
  if ( x < 1 ) {

    poem = poem + "oise ";
  }

  x = x + 1;
}
output.Text = poem;
```

Видите другое решение? Проверьте его в IDE и посмотрите, как оно работает! Подсказка: существует еще одно решение.

Если вы готовы к трудностям, попробуйте найти оба решения! Одно из них сохраняет порядок следования фрагментов. Если вы нашли именно этот вариант, попробуйте понять, почему наш код работает.

Приложения для рабочего стола Windows

Windows 8 познакомила нас с приложениями для магазина Windows и совершенно новым способом применения программного обеспечения. Но Visual Studio можно использовать и для построения **приложений для рабочего стола Windows** в виде окон на рабочем столе.

We'll use Visual Studio for Windows Desktop to build programs that run in windows on your Windows 8 desktop.

Не понимаю. Зачем мне изучать разные способы построения программ?

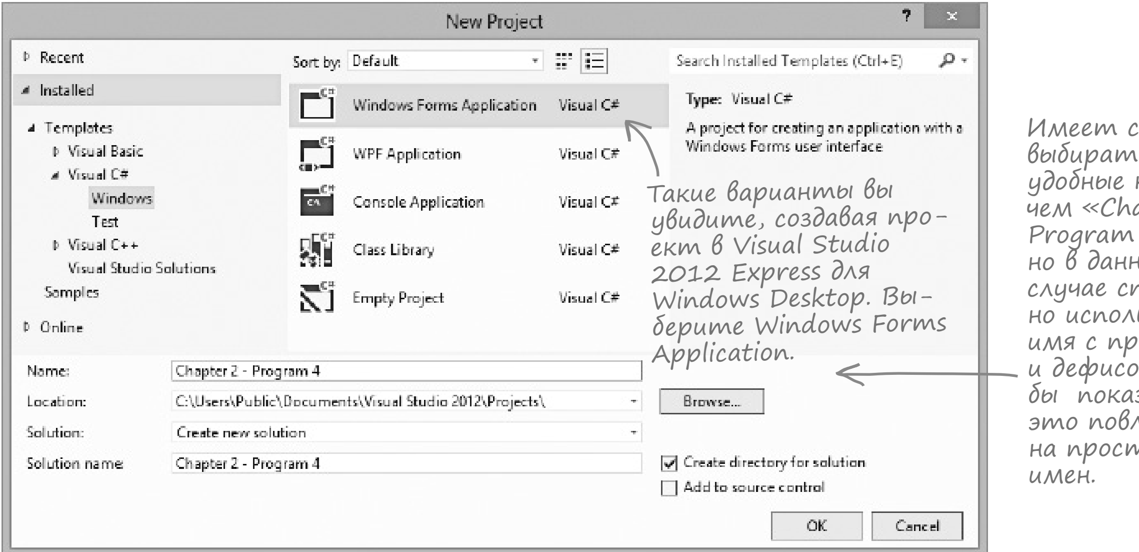
Приложения для рабочего стола Windows как обучающий инструмент

В нескольких следующих главах мы будем пользоваться Visual Studio для рабочего стола Windows и только потом вернемся к приложениям для магазина Windows. Дело в том, что первые во многом проще вторых. Они не настолько красивы, не интергируются с Windows 8 и не предоставляют такого удобного пользовательского интерфейса, как приложения для магазина Windows. Но для эффективного построения последних вам нужно познакомиться с рядом важных фундаментальных понятий. И проще всего начать, программируя для рабочего стола Windows. Сразу после этого мы вернемся к построению приложений для магазина Windows.

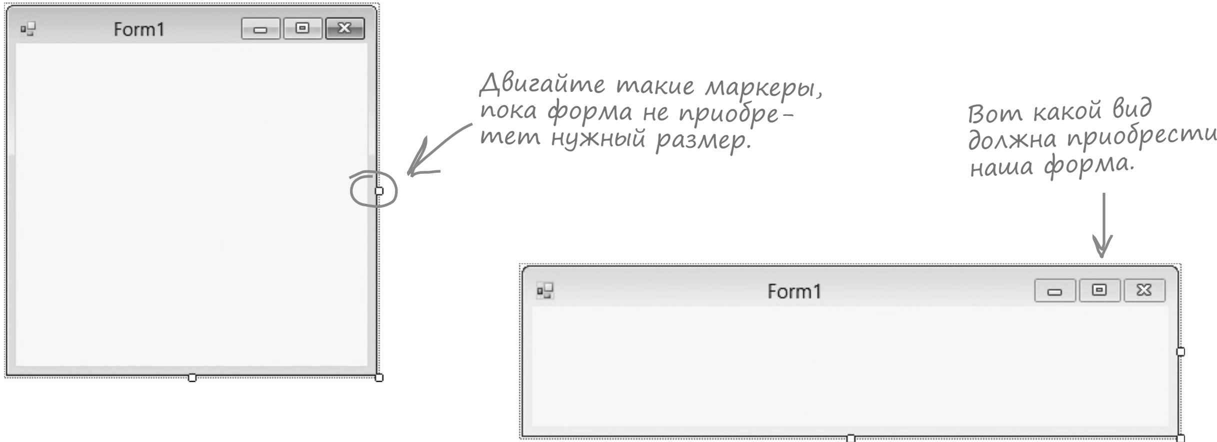
Изучая программирование для рабочего стола Windows, вы увидите альтернатиные способы реализации многих вещей. Это позволяет лучше усвоить самое важное. Переверните страницу, чтобы понять, что мы имеем в виду...

Перестроим приложение для рабочего стола Windows

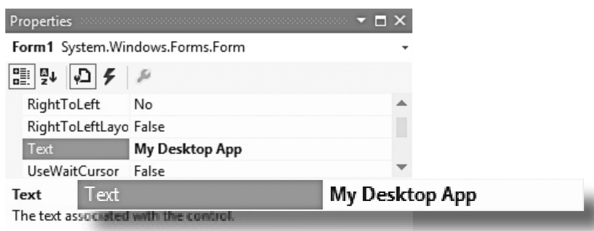
Запустите Visual Studio 2012 для рабочего стола Windows и создайте новый проект. На этот раз вам предлагается другое меню. Раскройте разделы Visual C# и Windows и **выберите Windows Forms Application**.



1. **Приложения Windows Forms начинаются с формы с редактируемыми размерами.** Приложение Windows Forms в конструкторе представляется в виде окна. Сделаем его размером 500×130. Найдите на границе формы маркер и перетащите его. Обратите внимание, как при этом меняются цифры в строке состояния IDE. Двигайте его, пока не увидите в строке состояния значение 500 x 130 .



Меняем заголовок формы.
По умолчанию в заголовке формы написано «Form1». Выделите форму щелчком и поменяйте свойство `Text` в окне Properties.



Добавим кнопку, флажок и метку.
Перетащите с панели элементов на форму элементы Button, CheckBox и Label.

Корректный выбор Visual Studio

Пользователям версии Express Visual Studio 2012 нужно установить два варианта приложения. Мы пользовались Visual Studio 2012 для Windows 8 при создании приложений для магазина Windows. А теперь нам нужен ***Visual Studio 2012 для Windows Desktop****. К счастью, обе версии Express можно бесплатно скачать на сайте Microsoft.*

Панель элементов открывается командой «Toolbox» из меню View или щелчком на вкладке Toolbox. Ее можно закрепить, щелкнув на кнопке в виде булавки (⊟) в правом верхнем углу окна Toolbox, или перетащить в сторону, превратив в плавающее окно.

Разделительные линии помогают указать положение элемента.

На следующей странице мы займемся окном Properties для изменения текста и задания состояния элемента CheckBox. Попробуйте самостоятельно понять, как это делается!

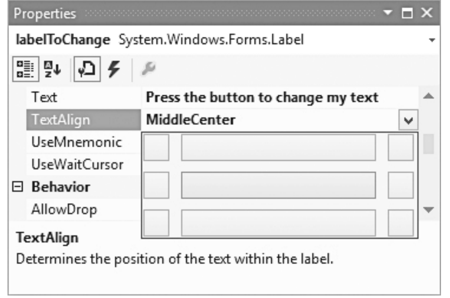
IDE помогает выравнивать элементы управления, отображая в процессе их перетаскивания направляющие линии.

Подсказка: для придания элементу Label нужного рамера используйте AutoSize.

Настраиваем элементы управления в окне Properties.

Щелчком выделите элемент Button и в окне Properties задайте свойство Text:

Измените свойство Text элементов CheckBox и Label в соответствии со снимком экрана на следующей странице и присвойте свойству Checked элемента CheckBox значение True. Затем выделите элемент Label и присвойте свойству TextAlign значение MiddleCenter. Кроме того, **дайте элементам имена**: кнопке — Button changeText, флажку — CheckBox enableCheckbox, а метке — Label labelToChange. Внимательно посмотрите на приведенный ниже код и определите, каким образом эти имена там используются.



Свойство AutoSize элемента Label поменяйте на False. По умолчанию метки подгоняют свой размер под текст. Убрав у параметра AutoSize значение true, мы сделаем видимыми манипуляторы, чтобы **растянуть метку на ширину окна**.

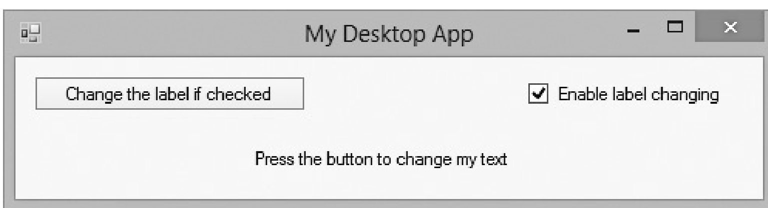
Добавим к кнопке метод обработчика событий.

Дважды щелкните на кнопке, чтобы добавить метод обработчика событий. Вот его код:

```
Form1.cs
Chapter_2__Program_4.Form1          Form1()

namespace Chapter_2___Program_4
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void changeText_Click(object sender, EventArgs e)
        {
            if (enableCheckbox.Checked == true)
            {
                if (labelToChange.Text == "Right")
                {
                    labelToChange.Text = "Left";
                    labelToChange.TextAlign = ContentAlignment.MiddleLeft;
                }
                else
                {
                    labelToChange.Text = "Right";
                    labelToChange.TextAlign = ContentAlignment.MiddleRight;
                }
            }
            else
            {
                labelToChange.Text = "Text changing is disabled";
                labelToChange.TextAlign = ContentAlignment.MiddleCenter;
            }
        }
    }
}
121 %
```

При двойном щелчке на кнопке IDE сгенерировала обработчик события и присвоила ему имя changeText_Click(), в соответствии с именем кнопки changeText.

Это код для метода обработчика события. Внимательно его прочитайте. Можете найти его отличия от аналогичного кода в предыдущем упражнении?



Отладим программу в IDE.

Затем IDE построит программу и запустит ее, открыв окно. Попробуйте щелкать на кнопке и менять состояние флажка.

My Desktop App
Change the label if checked
Enable label changing
Right

My Desktop App
Change the label if checked
Enable label changing
Left

My Desktop App
Change the label if checked
Enable label changing
Text changing is disabled

При установленном флажке метка показывает текст Left или Right с соответствующим выравниванием. Если же флажок снят, сообщение выводится по центру.

Флажок включает и отключает изменения текста метки.

Возьми в руку карандаш

Опишите назначение различных строк кода, как показано в примере.

```
using System;
using System.Linq;
using System.Text;
using System.Windows.Forms;
```

Оператор “using” добавляет в классы C# методы из других пространств имен

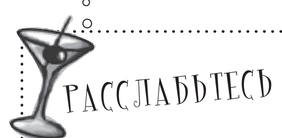
```
namespace SomeNamespace
{
    class MyClass {
        public static void DoSomething() {
            MessageBox.Show("Здесь будет сообщение");
        }
    }
}
```

Вы еще не встречались с функцией MessageBox, но она часто используется в приложениях для рабочего стола. И подобно большинству классов и методов, она носит значимое имя.

Решение на странице 131

Начало работы программы

При создании нового приложения Windows Forms IDE добавляет файл Program.cs. Дважды щелкните на его имени в окне Solution Explorer. Файл содержит класс Program, обладающий методом Main(). Этот метод представляет собой **точку входа**, то есть именно отсюда программа начинает свою работу.



Приложения для рабочего стола выглядят по-другому, и это хорошо для обучения.

Приложения для рабочего стола Windows выглядят примитивнее приложений для магазина Windows, потому что для них сложнее создать усовершенствованный интерфейс пользователя. Но на данном этапе это хорошо, так как ничто не будет отвлекать вас от изучения ключевых понятий C#. А когда мы вернемся к приложениям для магазина Windows, вам будет проще.

Этот код, автоматически созданный в упражнении из предыдущей главы, вы найдете в файле Program.cs.

Код под увеличительным стеклом

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms; ❶

namespace ❷ Chapter_2___Program_4
{
    static ❸ class Program
    {
        /// <summary>
        /// Точка входа в приложение.
        /// </summary>
        [STAThread] ❺
        static void Main()
        {
            Application.EnableVisualStyles();
          ❹ Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

Имя этого пространства имен IDE сгенерировала, взяв за основу имя проекта. В данном случае проект назывался "Chapter 2 - Program 4". Имя с пробелами и дефисом было выбрано намеренно, чтобы продемонстрировать, как IDE превращает их в нижние подчеркивания.

Начинающиеся со слэшей строки являются комментариями, и их можно добавлять куда угодно. Компилятор их просто не видит.

Программа начинает работу с точка входа.

Этот оператор создает и отображает форму Contacts, а также завершает программу при закрытии формы.

Первая часть имени класса или метода называется <u>объявлением</u>.

На стадии начального знакомства с кодом вам требуется понять, на что следует обращать внимание.

Существует несколько тонких моментов в разработке приложений. Вы еще развлечетесь с ними на следующих страницах. Но большую часть вашей работы будет составлять перетаскивание элементов управления и редактирования кода C#.

1. **Встроенные функции C# и .NET.**
Подобные строки находятся в верхней части почти всех файлов классов C#. `System.Windows.Forms` — это **пространство имен**. Строка `using System.Windows.Forms` дает программе доступ ко всем объектам этого пространства, в данном случае к визуальным элементам — кнопкам и формам.

Постепенно ваши программы будут содержать все больше пространств имен.

Без строчки using вам придется в явном виде вводить System.Windows. Forms при обращении к объекту из этого пространства имен.

2. **Выбор пространства имен для кода.**
IDE называет созданное пространство имен в соответствии с именем проекта. Именно к этому пространству относится весь код.

Пространства имен позволяют использовать одни и те же имена в различных программах, при условии, что программы не принадлежат к одному пространству.

3. **Код принадлежит к конкретному классу.**
В вашей программе этот класс называется `Program`. Он содержит код запуска программы и код вызова формы Form 1.

В одном пространстве имен может находится несколько классов.

4. **Наш код содержит один метод, состоящий из нескольких операторов.**
Внутри любого метода может находиться произвольное количество операторов. В нашей программе именно операторы вызывают форму.

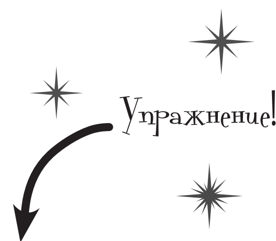
Технически программа может иметь несколько методов Main(), нужно только указать, какой из них будет точкой входа.

5. **Точка входа.**
Каждая программа на C# **должна** иметь только один метод с названием `Main`. Именно он выполняется первым. C# проверяет классы на его наличие, пока не находит строчку `static void Main()`. После этого выполняется первый и все следующие за ним операторы.

Любое приложение на C# должно иметь единственный метод Main. Он является точкой входа для вашего кода.

При запуске кода метод Main( ) выполняется ПЕРВЫМ.

Редактирование точки входа



В программе главное — точка входа. При этом не имеет значения, к какому классу принадлежит содержащий ее метод и какие действия производит. Нет ничего таинственного в том, как это работает. Вы можеет проверить это самостоятельно, изменив точку входа.

1. Вернитесь к программе, которую мы только что написали. В файле `Program.cs` присвойте методу `Main` имя `NotMain` и **попробуйте построить и запустить** программу. Что произойдет?

2. Создадим новую точку входа. **Добавьте класс** с именем `AnotherClass.cs`. Для этого щелкните правой кнопкой мыши на имени файла в окне Solution Explorer и выберите команду Add>>Class... IDE добавит в программу класс `AnotherClass.cs`. После этого код примет вид:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Chapter_2___Program_4
{
    class AnotherClass
    {
    }
}
```

В файл были добавлены четыре стандартные строчки с оператором `using`.

Этот класс находится в том же пространстве имен.

Имя присваивается классу автоматически (на основе имени файла).

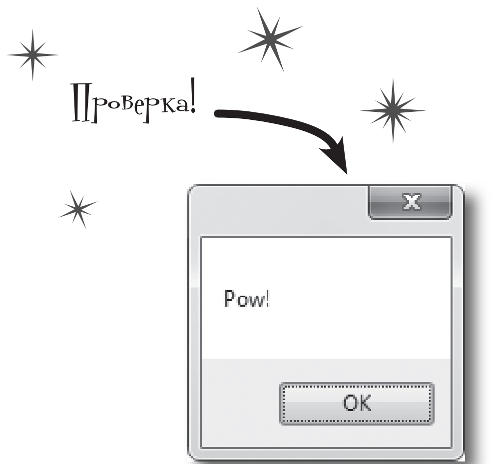
3. Добавьте в верхнюю часть строку **`using System.Windows.Forms;`**
Не забудьте, что в конце строки должна стоять точка с запятой!

4. Добавьте этот метод к классу **`AnotherClass`**, написав его внутри фигурных скобок:

```
class AnotherClass
{
    public static void Main()
    {
        MessageBox.Show("Pow!");
    }
}
```

*Класс **MessageBox** принадлежит пространству имен System.Windows.Forms, поэтому на шаге #3 вы и добавили оператор using. Метод **Show ()** является частью класса MessageBox.*

В C# регистр букв имеет значение! Обращайте внимание на прописные и строчные буквы.



Приложения для рабочего стола используют метод MessageBox.Show() для открытия окон с сообщениями и предупреждениями.

Что же произошло?

Теперь вместо приложения программа вызывает вот такое окно диалога. Переопределив метод `Main()`, вы указали новую точку входа. Поэтому программа первым делом выполняет оператор `MessageBox.Show()`. Данный метод больше ничего не содержит, поэтому после щелчка на кнопке OK программа завершит свою работу.

5. Верните программу в исходное состояние.

Подсказка: Достаточно изменить пару строк в двух файлах.

Возьми в руку карандаш — Решение

Вот правильные варианты ответа на вопрос о строках.

```
using System;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace SomeNamespace
{
    class MyClass {
        public static void DoSomething() {
            MessageBox.Show("Здесь будет собщение");
        }
    }
}
```

Оператор using добавляет в класс методы из других пространств имен.

Мы указываем, к какому классу принадлежит этот фрагмент кода.

Внутри этого класса находится метод DoSomething, вызывающий объект MessageBox.

Этот оператор вызывает окно с текстовым сообщением.

Любые действия ведут к изменению кода

Посмотрим, каким образом IDE пишет код. Откройте Visual Studio и **создайте новый проект Windows Forms Application**.

1 Просмотр кода.
Откройте файл `Form1.Designer.cs`, но не в конструкторе форм. Для этого щелкните правой кнопкой мыши на имени файла в окне Solution Explorer и выберите команду View Code. Посмотрите на объявление класса `Form1`

```
partial class Form1
```

Обратите внимание на ключевое слово partial. О том, что оно означает, мы поговорим далее.

2 Добавление к форме элемента PictureBox.
Привыкайте работать с множеством вкладок. Вернитесь в окно Solution Explorer и откройте конструктор форм двойным щелчком на имени файла `Form1.cs`. **Перетащите на форму элемент PictureBox**.

Чтобы выбрать изображение для элемента PictureBox, выделите его и щелкните на ссылке “Choose Image...” в окне Properties. Откроется окно, в котором можно будет указать, какой файл нужно загрузить.

Выберите “Local resource” и щелкните на кнопке Import, чтобы открыть окно для поиска файла с изображением.

3 Просмотр созданного конструктором для элемента PictureBox кода.
Вернитесь на вкладку Form1.Designer.cs. Найдите вот эту строчку:

Щелкните на квадратике со знаком «плюс».

Щелчок на этом квадратике раскроет код. Найдите в нем следующие строки:

После двойного щелчка на строке Form1.resx в окне Solution Explorer вы увидите импортированное изображение. IDE присвоила ему имя “pictureBox1.Image” и сгенерировала код для его загрузки в элемент PictureBox и последующего отображения.

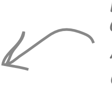
```
//
// pictureBox1
//
this.pictureBox1.Image = ((System.Drawing.Image)(resources.GetObject("pictureBox1.Image")));
this.pictureBox1.Location = new System.Drawing.Point(416, 160);
this.pictureBox1.Name = "pictureBox1";
this.pictureBox1.Size = new System.Drawing.Size(141, 147);
this.pictureBox1.TabIndex = 0;
this.pictureBox1.TabStop = false;
```

Ваши значения параметров Location и Size будут отличаться от указанных в книге.

Стоп! Что же это значит?

В верхней части кода вы увидите строки:

```
/// <summary>
/// Обязательный метод для поддержки конструктора - не изменяйте
/// содержимое данного метода при помощи редактора кода.
/// </summary>
```



Чаще всего комментарии помечаются двойным слэшем (//). Но IDE иногда может поставить три слэша.

Для детей нет ничего притягательнее, чем табличка с надписью «Не трогать!» Не отрицайте, вам же тоже хочется... поэтому отредактируем содержимое данного метода, воспользовавшись редактором кода! **Добавьте к форме кнопку button1 и выполните следующие действия:**

1. **Отредактируйте свойство Text кнопки button1. Как вы думаете, что изменится в окне Properties?**
   Сделайте и посмотрите, что получится! Затем вернитесь в конструктор форм и проверьте свойство Text. Оно изменилось?

2. **Воспользуйтесь окном Properties для редактирования свойства Name какого-нибудь объекта.**
   В окне Properties это свойство находится вверху и называется «(Name)». Какие изменения произойдут с кодом? Обратите внимание на комментарии.

3. **Присвойте свойству Location значение (0,0). Измените параметр Size, увеличив размер кнопки.**
   У вас получилось?

4. **Вернитесь в конструктор кода и произвольным образом измените свойство BackColor.**
   Внимательно посмотрите на код `Form1.Designer.cs`. Появились ли в нем новые строчки?

Для просмотра результата редактирования не нужно сохранять и запускать программу. Достаточно перейти на вкладку Form1.cs [Design].

Для изменения сгенерированного конструктором кода формы проще воспользоваться IDE. Но при этом любые внесенные вами в IDE изменения влияют на код вашего проекта.

Часто задаваемые вопросы

В: Объясните, пожалуйста, еще раз, что такое точка входа.

О: Программа состоит из множества операторов, но они не могут выполняться одновременно. Операторы принадлежат разным классам. Как же при запуске программы определить, какой оператор выполнять первым? Компилятор просто не будет работать при отсутствии метода Main(), который и называется точкой входа. Первый оператор этого метода и будет выполняться самым первым.

Анимировать приложения для рабочего стола не так просто, как приложения для магазина Windows. Тем не менее это возможно! И в доказательство построим кое-что **яркое**! Начните с создания нового проекта Windows Forms Application.

1 Вот форма, которую нужно получить

FlashyThing

Go!

Увеличьте кнопку, перемещая угловой маркер.

Переменная, объявленная внутри цикла for (**int c** = 0; ...), действует только внутри этого цикла. Поэтому при наличии двух циклов for loops, использующих одну переменную, нужно объявлять ее в каждом из них. Если переменная c уже объявлена вне циклов, в них ее использовать нельзя.

2 Сделайте фоновый цвет броским!

Пусть результатом щелчка на кнопке станет цикличное изменение фонового цвета! Создайте цикл, в котором значение переменной **c** меняется от 0 до 253. Вот как выглядит код:

```
this.BackColor = Color.FromArgb(c, 255 - c, c);

Application.DoEvents();
```

Попробуйте убрать эту строчку, и вы увидите, что форма перестанет обновляться, так как цикл еще не закончен и перейти к процедуре обновления невозможно.

Метод DoEvents () принудительно обновляет форму на каждом витке цикла, но это «обходной» путь, который не следует использовать в серьезных программах.

Удиви меня цветом!

В .NET множество предустановленных цветов, таких как Blue (синий) или Red (красный), но вы можете задавать и свои собственные варианты при помощи метода Color.FromArgb(). Достаточно указать три значения: процент красного, процент зеленого и процент синего цвета.

3 Замедлите процесс

Чтобы цвета менялись медленнее, после строки `Application.DoEvents()` напишите:

```
System.Threading.Thread.Sleep(3);
```

Этот оператор добавляет задержку 3 миллисекунды. Он находится в библиотеке .NET и принадлежит пространству имен **System.Threading**.

Напоминаем, что приложения Windows Forms создаются в Visual Studio для Windows Desktop.

4. **Сглаживание процесса.**
Пусть цветовая гамма возвращается к изначальному цвету. Для этого добавьте еще один цикл, в котором переменная **c** будет меняться уже от 254 до 0. В фигурные скобки поставьте блок кода из предыдущего цикла.

5. **Сделайте процесс непрерывным.**
Поместите два цикла внутрь третьего, который будет выполняться бесконечно. В итоге щелчок на кнопке будет приводить к непрерывному изменению фонового цвета. (Подсказка: цикл `while` является бесконечным, если проверка условия дает результат `(true)`!)

Цикл, находящийся внутри другого цикла, называется вложенным.

Ой-е-ей! Программа не останавливается!

После запуска этой программы остановить ее работу, просто закрыв окно, уже невозможно! Для прекращения ее работы воспользуйтесь, к примеру, командой Stop Debugging из меню Debug.

6. **Остановите ее!**
Сделаем так, чтобы цикл, добавленный на предыдущем шаге, останавливался при закрытии формы. Замените первую строчку на:

```
while (Visible)
```

Запустите программу и щелкните на крестике в правом верхнем углу формы. Окно закроется, и программа остановится. Пусть и с задержкой в несколько миллисекунд.

Проверку видимости объекта можно не писать в форме — Visible == true. Достаточно первой половины выражения.

Visible имеет значение true, пока отображаются форма или элемент управления. Если присвоить этому параметру значение false, форма или элемент управления тут же исчезнут.

Подсказка: **&&** означает «И». Именно этот оператор позволяет соединить вместе несколько условий. Результат будет истинным только при одновременном выполнении этих условий.

Можете ли вы объяснить эту задержку и переписать код таким образом, чтобы возвращение в режим редактирования происходило сразу после закрытия формы?

Упражнение Решение

Построим что-то **яркое!**

Добавляя этот метод, IDE поставила дополнительные пробелы перед фигурными скобками. Иногда для экономии места эти скобки могут располагаться на одной строке с оператором — в C# такая форма записи вполне допустима.

Иногда в разделе «Решение» приводится не весь код программы, а только те фрагменты, которые требовалось отредактировать.

Крайне важно, чтобы ваш код могли легко читать другие пользователи. Но мы намеренно показываем вам разные варианты, так как вы должны привыкнуть читать код, написанный в разном стиле.

```
private void button1_Click(object sender, EventArgs e) {
    while (Visible) {
        for (int c = 0; c < 254 && Visible; c++) {
            this.BackColor = Color.FromArgb(c, 255 - c, c);
            Application.DoEvents();
            System.Threading.Thread.Sleep(3);
        }
        for (int c = 254; c >= 0 && Visible; c--) {
            this.BackColor = Color.FromArgb(c, 255 - c, c);
            Application.DoEvents();
            System.Threading.Thread.Sleep(3);
        }
    }
}
```

Внешний цикл выполняется, пока открыта форма.

Мы написали && Visible вместо && Visible == true. Это все равно что сказать «если это видимо» вместо «если это верно, что это видимо». Обе формулировки означают одно и то же.

Оба цикла меняют цвет формы, но циклы работают в разные стороны.

Оператор && позволяет прервать цикл for, как только параметр Visible примет значение false.

Помните вопрос, как убрать задержку с возвращением в режим редактирования после закрытия окна?

Задержка возникает из-за невозможности проверить значение параметра `Visible` до завершения цикла `for`. Поэтому к проверке условия добавили код **`&& Visible`**.

Любую задачу программирования можно решить более чем одним способом, так что попробуйте написать свой вариант кода, взяв за основу циклы while вместо циклов for.

3 объекты, по порядку стройся!

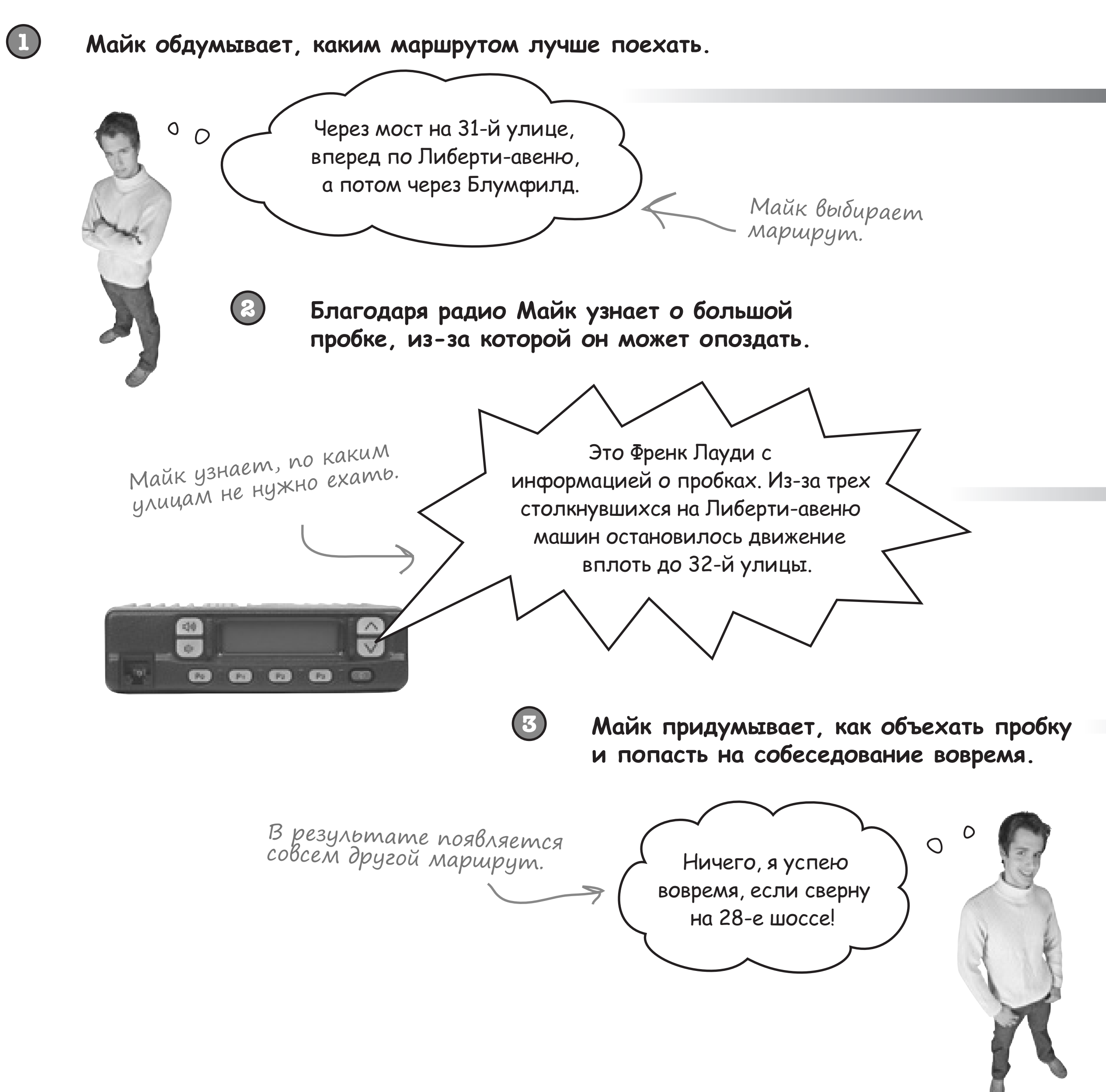
Приемы программирования

Каждая программа решает какую-либо проблему.

Перед написанием программы нужно четко сформулировать, какую *задачу* она будет решать. Именно поэтому так полезны ***объекты***. Ведь они позволяют структурировать код наиболее удобным образом. Вы же можете сосредоточиться на *обдумывании путей решения*, так как вам не нужно тратить время на написание кода. Правильное использование объектов позволяет получить *интуитивно понятный* код, который при необходимости можно легко отредактировать.

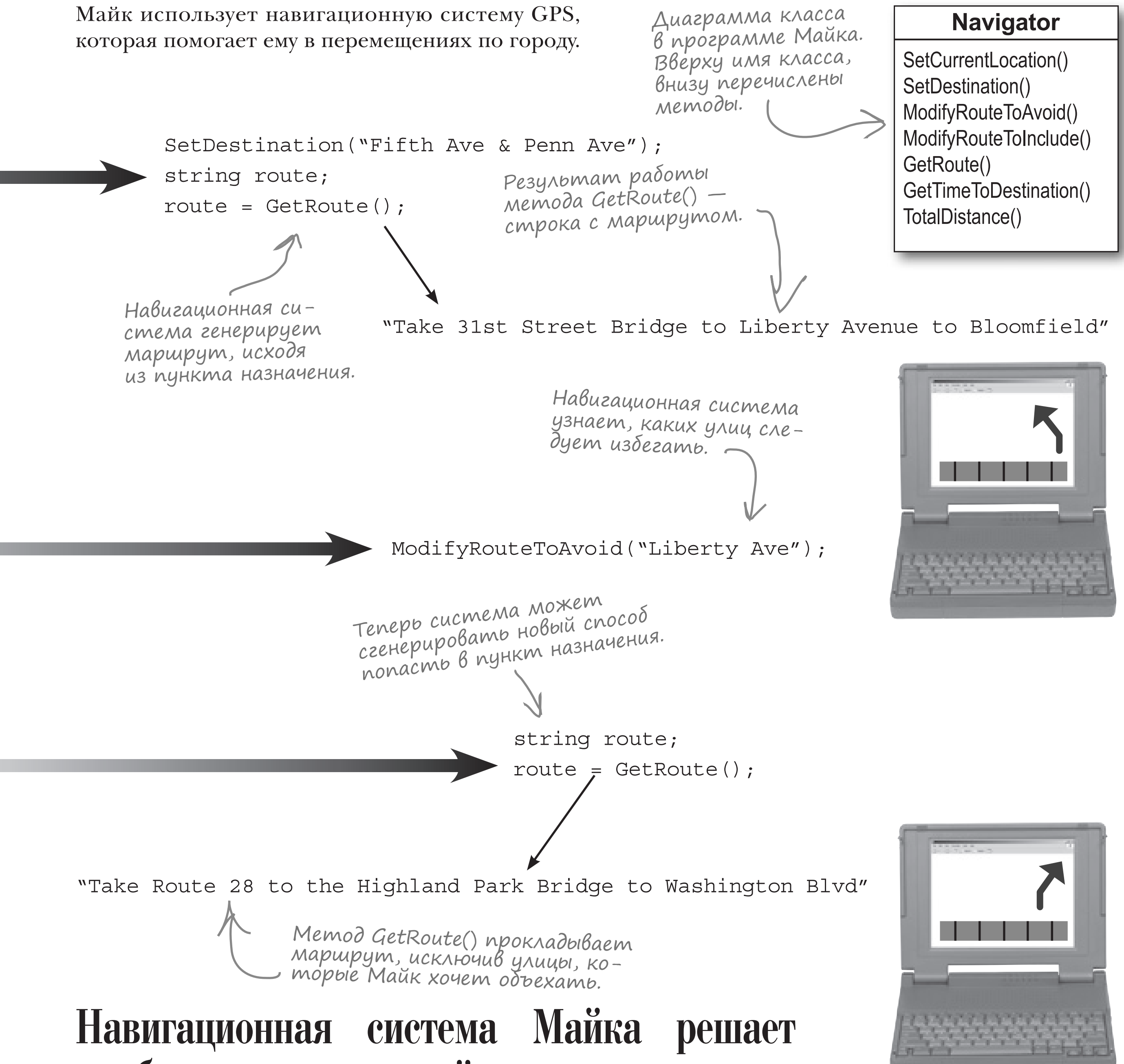
Что думает Майк о своих проблемах

Майк — программист в поисках новой работы. Ему не терпится показать, как хорошо он пишет программы на C#, но сначала ему нужно попасть на собеседование. А он опаздывает!



Проблема Майка с точки зрения навигационной системы

Майк использует навигационную систему GPS, которая помогает ему в перемещениях по городу.



Навигационная система Майка решает проблему с прокладкой маршрута так же, как это сделал бы он сам.

Методы прокладки и редактирования маршрутов

Класс Navigator содержит методы, которые выполняют все действия. В отличие от уже знакомых вам методов button_Click() они решают другую задачу: прокладывают маршрут по городу. Именно поэтому Майк поместил эти методы в единый класс и присвоил ему имя Navigator.

Для определения маршрута сначала вызывается метод SetDestination(), указывающий конечную точку, затем применяется метод GetRoute(), выводящий маршрут в виде символьной строки. Если маршрут требуется изменить, на помощь приходит метод ModifyRouteToAvoid(), позволяющий избежать определенных улиц. Затем метод GetRoute() выводит новый вариант маршрута.

Майк выбирает для методов значимые имена.

```
class Navigator {
    public void SetCurrentLocation(string locationName) { ... }
    public void SetDestination(string destinationName) { ... };
    public void ModifyRouteToAvoid(string streetName) { ... };
    public string GetRoute() { ... };
}
```

Это **тип** возвращаемого методом GetRoute() значения. Если указан тип **void**, метод не возвращает значение.

```
string route =
        GetRoute();
```

Методы, возвращающие значение

Методы состоят из операторов. Некоторые из них выполняют все входящие операторы и заканчивают работу. Другие же **возвращают какое-то значение**. Это значение принадлежит к определенному типу (например, string или int).

Оператор **return** прерывает работу метода. Если метод не возвращает значение, тип возвращаемого значения объявляется как void, присутствие этого оператора является необязательным. Но если метод возвращает значение, без оператора return не обойтись.

Вот пример метода, возвращающего значение типа int. Метод использует два параметра для вычисления результата.

```
public int MultiplyTwoNumbers(int firstNumber, int secondNumber) {
      int result = firstNumber * secondNumber;
      return result;
}
```

Оператор return возвращает значение вызвавшему метод оператору.

Оператор вызывает метод, перемножающий два числа. Возвращаемое значение принадлежит к типу int:

```
int myResult = MultiplyTwoNumbers(3, 5);
```

В методы можно подставлять не только константы, но и переменные.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Классы состоят из методов, которые в свою очередь состоят из операторов. Осмысленный выбор методов позволяет получить удобный для работы класс.
- Некоторые методы могут **возвращать значение**. Тип этого значения нужно объявлять. Например, метод, объявленный как `public int`, возвращает целое число. Пример такого оператора: `return 37;`
- Метод, возвращающий значение, **обязан** включать в себя оператор `return`. Если в объявлении метода указано `public string`, значит, оператор `return` возвращает значение типа `string`.
- После оператора `return` программа возвращает управление оператору, вызывающему метод.
- Метод, при объявлении которого было указано `public void`, не возвращает значение. Но оператор `return` может использоваться для прерывания такого метода: `if (finishedEarly) { return; }`.

Программа с использованием классов

Привяжем форму к классу и сделаем так, чтобы принадлежащая форме кнопка вызывала метод этого класса.

1. Создайте новый проект Windows Forms Application. В окне Solution Explorer щелкните правой кнопкой мыши на имени проекта и выберите в появившемся меню команду Add>>Class... Назовите файл `Talker.cs`, при этом класс автоматически получит имя `Talker`. В IDE появится новая вкладка с именем `Talker.cs`.

2. Сверху вставьте строчку `using System.Windows.Forms`, а затем введите код самого класса:

```
class Talker {
    public static int BlahBlahBlah(string thingToSay, int numberOfTimes)
    {
        string finalString = "";

        for (int count = 1; count <= numberOfTimes; count++)
        {
            finalString = finalString + thingToSay + "\n";
        }
        MessageBox.Show(finalString);
        return finalString.Length;
    }
}
```

Оператор объявляет переменную finalString и присваивает ей нулевое значение (пустую строку).

К переменной finalString добавляется значение переменной thingToSay и знак переноса строки (\n).

Метод BlahBlahBlah() возвращает значения типа int. Свойство .Length можно добавить к любой строке и узнать ее длину.

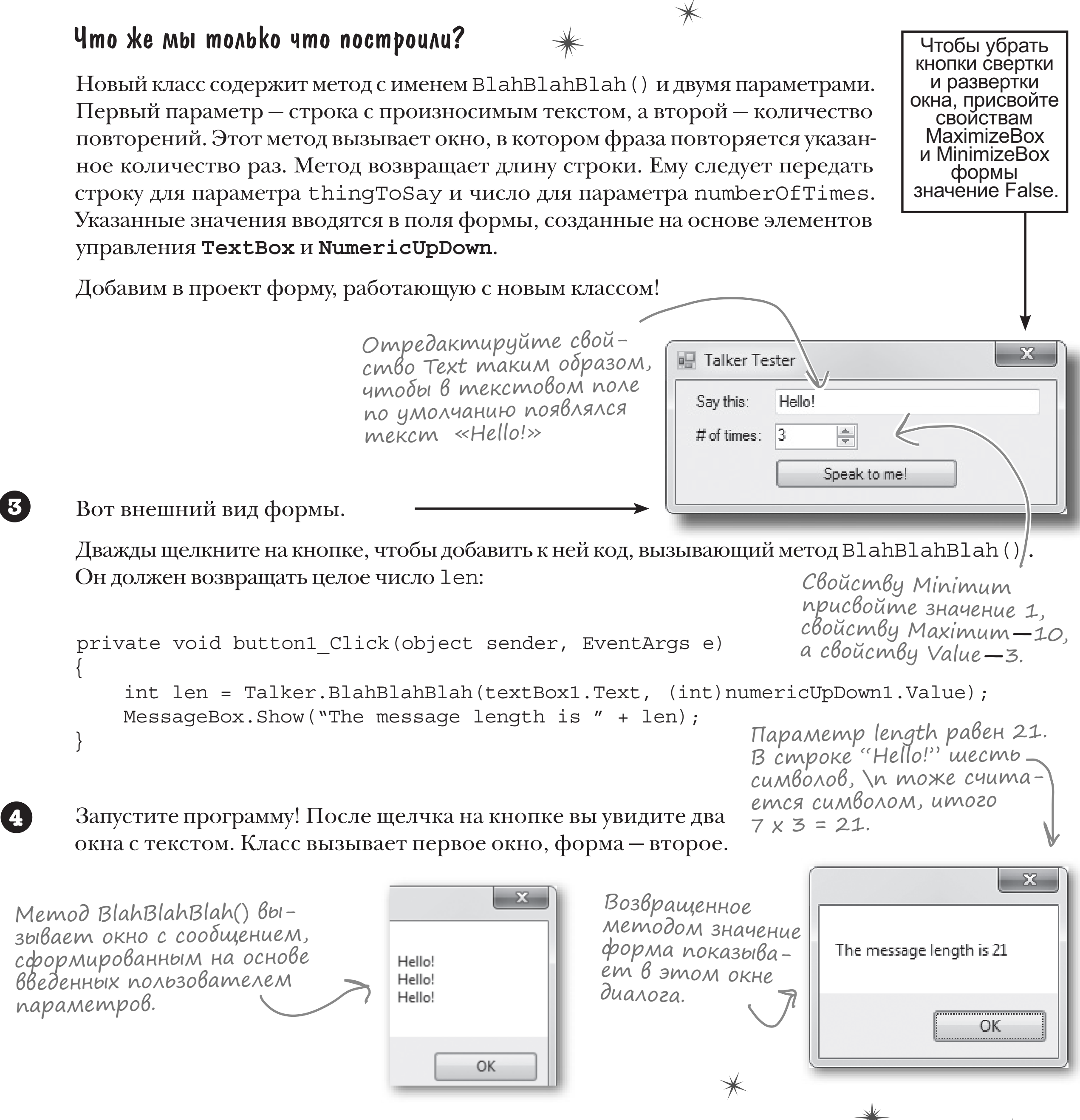
Свойством Length обладают все строки. Знак переноса (\n) считается за один символ.

Переверните страницу и продолжим!

Что же мы только что построили?

Новый класс содержит метод с именем `BlahBlahBlah()` и двумя параметрами. Первый параметр — строка с произносимым текстом, а второй — количество повторений. Этот метод вызывает окно, в котором фраза повторяется указанное количество раз. Метод возвращает длину строки. Ему следует передать строку для параметра `thingToSay` и число для параметра `numberOfTimes`. Указанные значения вводятся в поля формы, созданные на основе элементов управления **`TextBox`** и **`NumericUpDown`**.

Добавим в проект форму, работающую с новым классом!



3 Вот внешний вид формы.

Дважды щелкните на кнопке, чтобы добавить к ней код, вызывающий метод `BlahBlahBlah()`. Он должен возвращать целое число `len`:

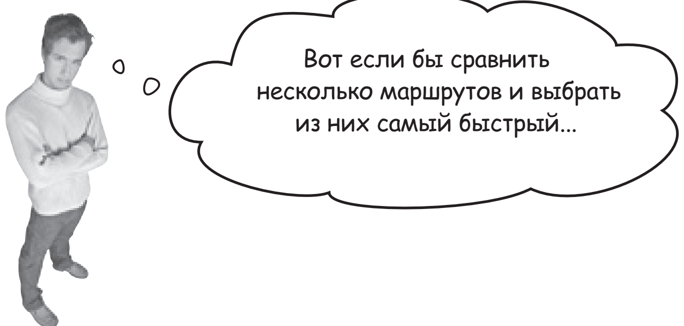
```
private void button1_Click(object sender, EventArgs e)
{
    int len = Talker.BlahBlahBlah(textBox1.Text, (int)numericUpDown1.Value);
    MessageBox.Show("The message length is " + len);
}
```

4 Запустите программу! После щелчка на кнопке вы увидите два окна с текстом. Класс вызывает первое окно, форма — второе.

Методы добавленного к проекту класса можно использовать и в других классах.

Идея Майка

Собеседование прошло замечательно! Но утренние пробки заставили Майка задуматься об усовершенствовании его навигационной программы.



Почему бы не создать три класса Navigator...

Майк *может* скопировать код класса `Navigator` и вставить его в другие классы. В результате программа получит возможность одновременно сохранять три маршрута.

Это **диаграмма классов**. В ней перечисляются все входящие в класс методы. Это наглядное представление выполняемых классом функций.

Navigator
SetCurrentLocation()
SetDestination()
ModifyRouteToAvoid()
ModifyRouteToInclude()
GetRoute()
GetTimeToDestination()
TotalDistance()

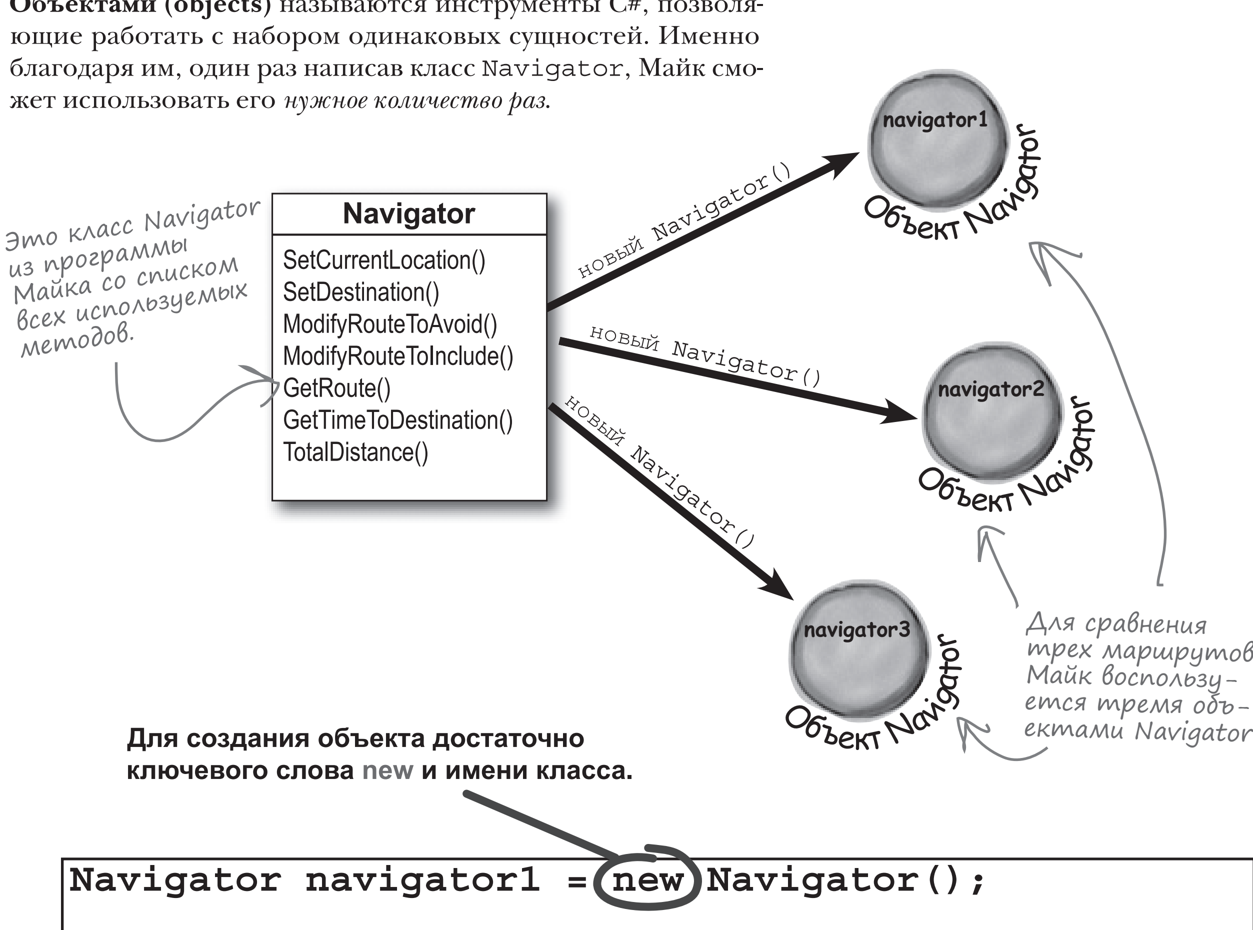
Navigator2
SetCurrentLocation()
SetDestination()
ModifyRouteToAvoid()
ModifyRouteToInclude()
GetRoute()
GetTimeToDestination()
TotalDistance()

Navigator3
SetCurrentLocation()
SetDestination()
ModifyRouteToAvoid()
ModifyRouteToInclude()
GetRoute()
GetTimeToDestination()
TotalDistance()

Именно так! Управление тремя копиями одного кода — непростая задача. Но большинство задач, с которыми вам придется столкнуться, требуют неоднократного использования ***одного*** элемента. В данном случае это один маршрутизатор. Нужно сделать так, чтобы редактирование любого фрагмента кода сопровождалось внесением аналогичных изменений во все его копии.

Объекты как способ решения проблемы

Объектами (objects) называются инструменты C#, позволяющие работать с набором одинаковых сущностей. Именно благодаря им, один раз написав класс `Navigator`, Майк сможет использовать его *нужное количество раз.*



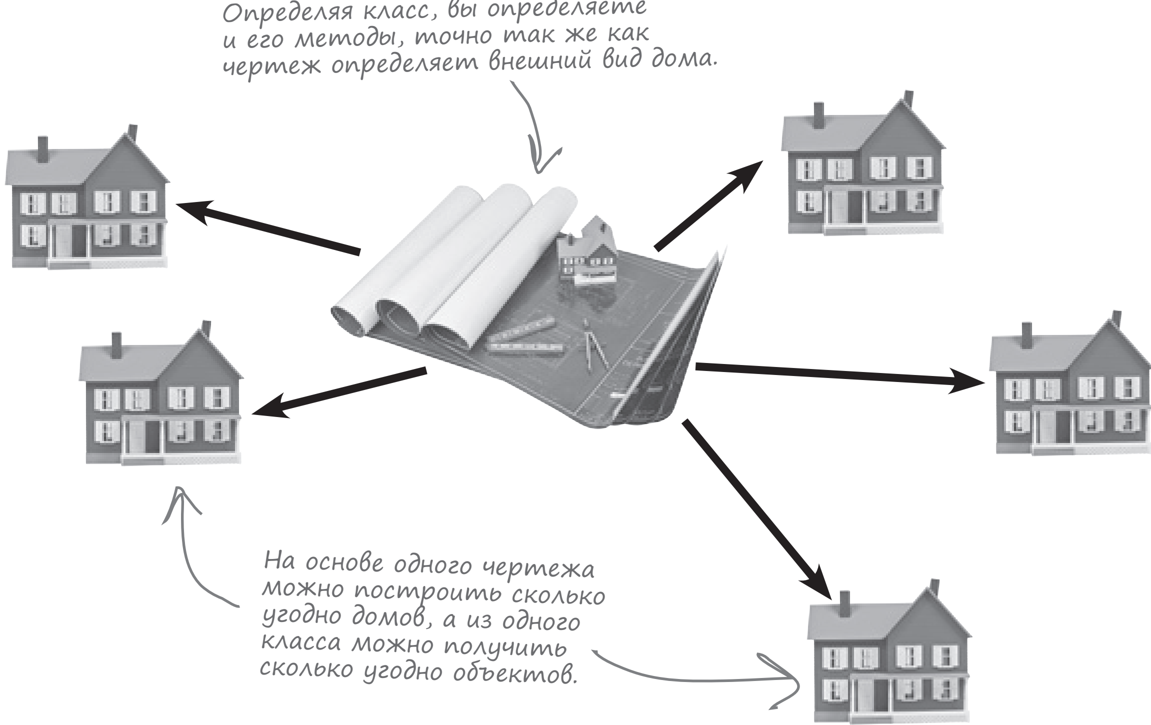
Для создания объекта достаточно ключевого слова new и имени класса.

```
Navigator navigator1 = new Navigator();
navigator1.SetDestination("Fifth Ave & Penn Ave");
string route;
route = navigator1.GetRoute();
```

Объект уже можно использовать! Так как он получен из класса, то содержит все входившие в класс методы.

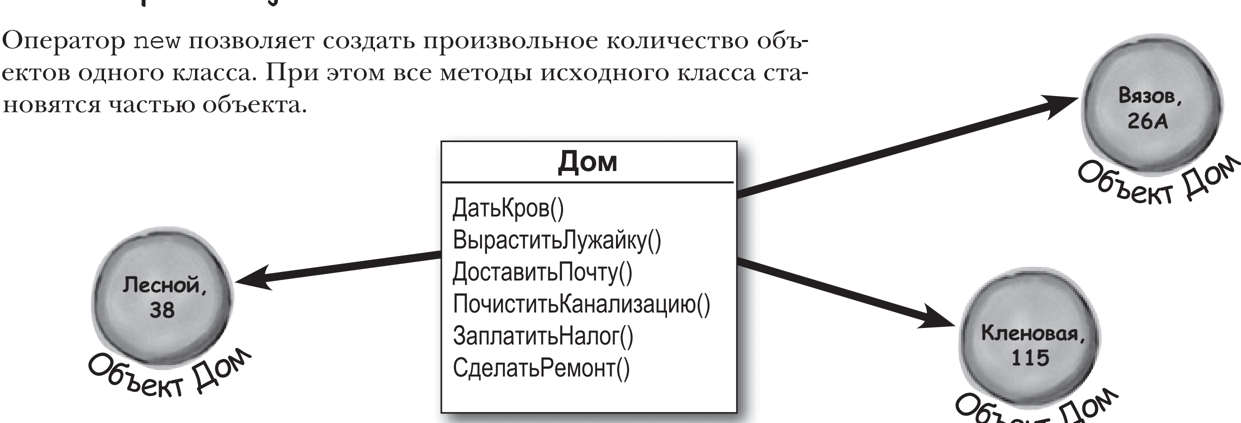
Возьмите класс и постройте объект

Класс можно представить как копию объекта. Скажем, для построения пяти одинаковых домов в коттеджном поселке архитектору не нужно рисовать пять одинаковых чертежей. Для выполнения задачи вполне достаточно одного.



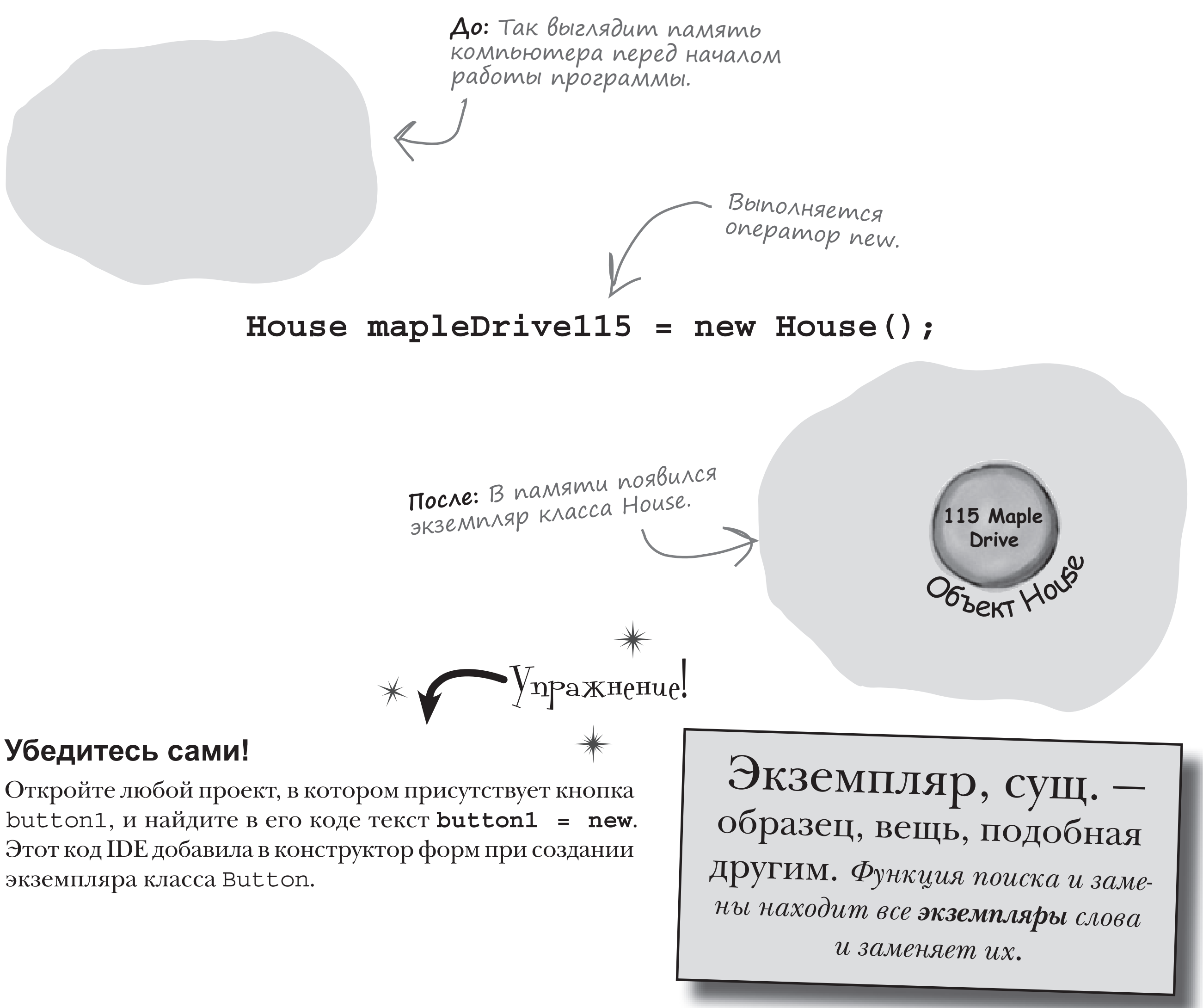
Объект берет методы из класса

Оператор new позволяет создать произвольное количество объектов одного класса. При этом все методы исходного класса становятся частью объекта.



Экземпляры

Все элементы окна Toolbox являются классами: класс `Button`, класс `TextBox`, класс `Label` и т. п. При перетаскивании на форму кнопки из окна Toolbox автоматически создается экземпляр класса `Button`, которому присваивается имя `button1`. Перетаскивание второй кнопки приводит к появлению второго экземпляра с именем `button2`. Каждый экземпляр имеет собственные свойства и методы. Но при этом все кнопки работают одинаково, так как они были созданы из одного класса.



Убедитесь сами!

Откройте любой проект, в котором присутствует кнопка `button1`, и найдите в его коде текст **`button1 = new`**. Этот код IDE добавила в конструктор форм при создании экземпляра класса `Button`.

Экземпляр, сущ. — образец, вещь, подобная другим. *Функция поиска и замены находит все **экземпляры** слова и заменяет их.*

Простое решение!

Майк придумал новую программу сравнения маршрутов, которая ищет кратчайший путь при помощи объектов. Вот как она создавалась.

GUI — это сокращение от Graphical User Interface (графический интерфейс пользователя).

1. Майк добавил к GUI текстовое поле — `textBox1`, содержащее информацию о **пункте назначения**. Затем создал поле `textBox2` для ввода названия улицы, которую **нельзя** включать в маршрут; и поле `textBox3`, содержащее информацию о еще одной улице, которую **желательно** объехать.

2. Он создал объект `Navigator` и указал пункт назначения.

Navigator
SetCurrentLocation() SetDestination() ModifyRouteToAvoid() ModifyRouteToInclude() GetRoute() GetTimeToDestination() TotalDistance()

navigator1 5,6 км Объект Navigator

Это экземпляр класса Navigator.

```
string destination = textBox1.Text;

Navigator navigator1 = new Navigator();

navigator1.SetDestination(destination);

route = navigator1.GetRoute();
```

3. Он добавил второй объект `Navigator` с именем `navigator2` и вызвал метод `SetDestination()` этого объекта для задания пункта назначения, после чего вызвал метод `ModifyRouteToAvoid()` (Редактировать избегаемые улицы).

4. Третий объект `Navigator` называется `navigator3`. Майк указал пункт назначения и вызвал метод `ModifyRouteToInclude()` (Редактировать включаемые в маршрут улицы)

Параметрами методов SetDestination(), ModifyRouteToAvoid() и ModifyRouteToInclude() являются переменные типа string.

Создание нового объекта на основе класса называется созданием экземпляра класса.

5. Теперь Майк может вызвать общий для всех объектов метод `TotalDistance()` (Общее расстояние) и определить самый короткий маршрут. И для этого ему понадобился один фрагмент кода, а не три!

Эй, подождите-ка! Данной вами информации недостаточно для того, чтобы написать навигационную программу!

Это действительно так. Программа построения маршрутов очень сложна. Но сложные программы работают по тому же принципу, что и простые. Навигационную программу Майка мы привели только как пример использования объектов на практике.

Теория и практика

Наша книга построена по следующему принципу. Вводится некое понятие, для демонстрации которого используются картинки и небольшие фрагменты кода. На данном этапе вы должны всего лишь понять новую информацию. Процедура написания рабочих программ будет рассматриваться позднее.

```
House mapleDrive115 = new House();
```

Знакомясь с новыми понятиями (например, с объектами), внимательно смотрите на картинки и код.

115 Maple Drive

Объект House

У вас будет достаточно возможностей применить полученные знания на практике. Иногда это выполнение письменных упражнений, одно из которых вы найдете на следующей странице. В других случаях вам сразу будет предложено написать код. Такая комбинации теории с практикой является самым эффективным способом запоминания новой информации.

Если у вас сложности с выполнением упражнения, можно сразу посмотреть решение. Его можно даже скачать с сайта Head First Labs.

Упражняясь в написании кодов...

...желательно помнить следующее:

- Сделать опечатку очень легко. При этом даже одна незакрытая скобка является препятствием к построению программы.
- ***Лучше*** подсмотреть решение, чем долго мучиться. Мучения отбивают желание учиться.
- Код, который вы найдете в этой книге, протестирован и работает в Visual Studio 2012! Но от опечаток никто не застрахован (можно перепутать 1 и букву L нижнего регистра).

Возьми в руку карандаш

Попробуйте вслед за Майком написать код для объектов `Navigator` и вызова их методов.

```
string destination = textBox1.Text;
string route2StreetToAvoid = textBox2.Text;
string route3StreetToInclude = textBox3.Text;
```

Здесь Майк задавал пункт назначения и улицы, которых следует избегать.

```
Navigator navigator1 = new Navigator();
navigator1.SetDestination(destination);
int distance1 = navigator1.TotalDistance();
```

А здесь мы создаем объект navigator, указываем пункт назначения и определяем расстояние.

1. Создайте объект **navigator2**, укажите пункт назначения, вызовите метод **ModifyRouteToAvoid()**, а затем воспользуйтесь методом **TotalDistance()** для вычисления переменной **distance2**.

Navigator navigator2 = ...

navigator2...

navigator2...

int distance2 = ...

2. Создайте объект **navigator3**, укажите пункт назначения, вызовите метод **ModifyRouteToInclude()**, а затем воспользуйтесь методом **TotalDistance()** для вычисления целой переменной **distance3**.

....

....

....

....

Встроенный в .NET Framework метод Math.Min() сравнивает два числа и возвращает меньшее. Именно с его помощью Майк нашел самый короткий путь.

```
int shortestDistance = Math.Min(distance1, Math.Min(distance2, distance3));
```

Возьми в руку карандаш
Решение

Вот как правильно создать объекты `Navigator` и вызвать их методы.

```
string destination = textBox1.Text;
string route2StreetToAvoid = textBox2.Text;
string route3StreetToInclude = textBox3.Text;
```

Здесь Майк задавал пункт назначения и улицы, которых следует избегать.

```
Navigator navigator1 = new Navigator();
navigator1.SetDestination(destination);
int distance1 = navigator1.TotalDistance();
```

А здесь мы создаем объект navigator, указываем пункт назначения и определяем расстояние.

1. Создайте объект **navigator2**, укажите пункт назначения, вызовите метод **ModifyRouteToAvoid()**, а затем воспользуйтесь методом **TotalDistance()** для вычисления переменной **distance2**.

```
Navigator navigator2 = new Navigator()
navigator2. SetDestination(destination);
navigator2. ModifyRouteToAvoid(route2StreetToAvoid);
int distance2 = navigator2.TotalDistance();
```

2. Создайте объект **navigator3**, укажите пункт назначения, вызовите метод **ModifyRouteToInclude()**, а затем воспользуйтесь методом **TotalDistance()** для вычисления переменной **distance3**.

```
Navigator navigator3 = new Navigator()
navigator3.SetDestination(destination);
navigator3.ModifyRouteToInclude(route3StreetToInclude);
int distance3 = navigator3.TotalDistance();
```

Встроенный в .NET Framework метод Math.Min() сравнивает два числа и возвращает меньшее. Именно с его помощью Майк нашел самый короткий путь.

```
int shortestDistance = Math.Min(distance1, Math.Min(distance2, distance3));
```

Я написал уже несколько новых классов, но ни разу не воспользовался ключевым словом new! Получается, я могу вызывать методы, не создавая объекты?

Да! И именно поэтому в методах использовалось ключевое слово `static`.

Еще раз посмотрим на объявление класса `Talker`:

```
class Talker
{
  public static int BlahBlahBlah(string thingToSay, int numberOfTimes)
  {
    string finalString = "";
```

При вызове метода не создавался `новый` экземпляр `Talker`. Вы написали только:

```
Talker.BlahBlahBlah("Hello hello hello", 5);
```

Именно так вызываются `статические` методы, с которыми вы работали до сих пор. Если убрать модификатор `static` из объявления метода `BlahBlahBlah()`, вызов метода окажется уже невозможным без создания экземпляра `Talker`. Впрочем, это единственное отличие статических методов. Вы можете передавать им параметры, они возвращают значения и принадлежат определенным классам.

Модификатором `static` можно отметить **целый класс**. Все входящие в этот класс методы также **должны быть** `статическими`. Добавив в статический класс нестатический метод, вы сделаете компиляцию невозможной.

В: Слово «статический» ассоциируется у меня с вещами, которые не меняются. Означает ли это, что нестатические методы могут меняться, а статические нет?

О: Нет. Единственным отличием статического метода от нестатического является невозможность создавать его экземпляры. Слово «статический» в данном случае не следует воспринимать слишком буквально.

В: То есть я не смогу пользоваться классом, не создав экземпляр объекта?

О: Создание экземпляров является обязательным условием для работы с нестатическими классами. Для статических классов это не требуется.

В: Почему не сделать статическими все методы?

О: При наличии объектов, отслеживающих данные, например экземпляров класса `Navigator`, каждый из которых отслеживал свой маршрут, для работы с этими данными можно использовать собственные методы экземпляра. Скажем, при вызове метода `ModifyRouteToAvoid()` для экземпляра `navigator2` менялся только маршрут номер два. На маршруты экземпляров `navigator1` и `navigator3` эта процедура никак не влияла. Именно поэтому программа Майка могла работать с тремя маршрутами одновременно.

В: А как именно экземпляры отслеживают данные?

О: Переверните страницу — и узнаете!

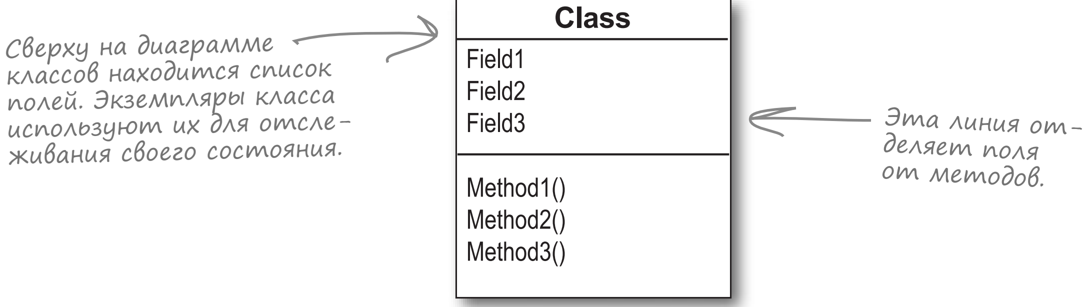
Поля

Редактирование расположенного текста на кнопке осуществляется при помощи свойства `Text`. При внесении подобных изменений в конструктор добавляется следующий код:

```
button1.Text = "Текст для кнопки";
```

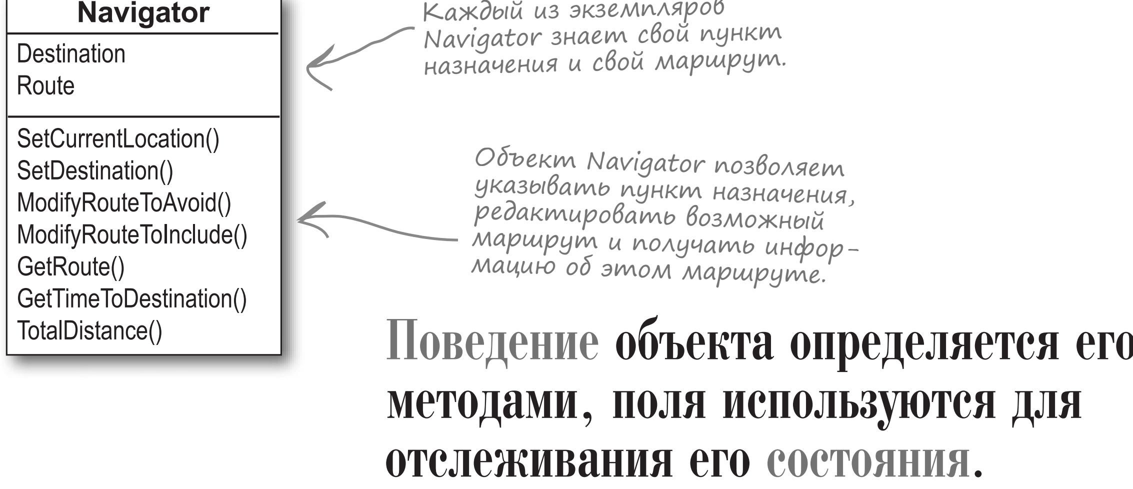
*Технически вы задаете **свойство**. Свойства очень похожи на поля, но об этом мы поговорим чуть позже.*

Как вы уже знаете, `button1` — это экземпляр класса `Button`. Код же редактирует **поле** этого экземпляра. На диаграмме классов список полей находится сверху, а список методов — снизу.



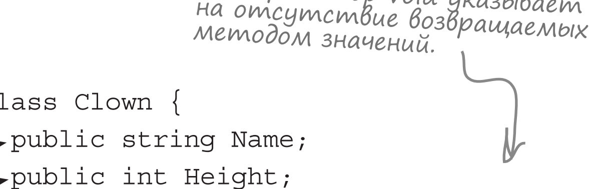
Метод — это то, что объект делает. Поле — это то, что объект знает.

В результате создания Майком трех экземпляров класса `Navigator` его программа создала три объекта, каждый из которых отслеживает свой маршрут. При вызове метода `SetDestination()` для экземпляра `navigator2` пункт назначения указывается только для этого экземпляра. Он никак не влияет на экземпляры `navigator1` и `navigator3`.



Создаем экземпляры!

Для добавления полей достаточно объявить переменные вне методов. Так, все экземпляры будут иметь свои копии этих переменных.



Clown
Name Height
TalkAboutYourself()

```
class Clown {
  public string Name;
  public int Height;

  public void TalkAboutYourself() {
    MessageBox.Show("My name is "
       + Name + " and I'm "
       + Height + " inches tall.");
  }
}
```

При создании экземпляра **не используйте** ключевое слово static ни в объявлении класса, ни в объявлении метода.

Оператор *= означает, что параметр, стоящий справа от оператора, нужно умножить на параметр, стоящий слева.

Возьми в руку карандаш

Клоуны рассказывают о себе при помощи метода TalkAboutYourself. Они называют имя и рост. Напишите, какие сообщения будут содержать всплывающие окна.

```
Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;
oneClown.TalkAboutYourself();
```

«Меня зовут _______ , мой рост ______ дюймов»

```
Clown anotherClown = new Clown();
anotherClown.Name = "Biff";
anotherClown.Height = 16;
anotherClown.TalkAboutYourself();
```

«Меня зовут _______ , мой рост ______ дюймов»

```
Clown clown3 = new Clown();
clown3.Name = anotherClown.Name;
clown3.Height = oneClown.Height - 3;
clown3.TalkAboutYourself();
```

«Меня зовут _______ , мой рост ______ дюймов»

```
anotherClown.Height *= 2;
anotherClown.TalkAboutYourself();
```

«Меня зовут _______ , мой рост ______ дюймов»

Спасибо за память

Создаваемые объекты находятся в так называемой **куче (heap)** — области динамической памяти, выделяемой на стадии исполнения программы. Применение оператора new автоматически резервирует место в памяти под хранение данных.

Вот так выглядит куча до начала работы программы. Да, она действительно пуста.

Возьми в руку карандаш

Решение

Вот как нужно было заполнить пробелы в тексте.

Операторы **new** создают экземпляры класса Clown, резервируя участки памяти и заполняя их данными об объекте.

```
Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;
oneClown.TalkAboutYourself();
```

«Меня зовут Boffo, мой рост 14 дюймов».

```
Clown anotherClown = new Clown();
anotherClown.Name = "Biff";
anotherClown.Height = 16;
anotherClown.TalkAboutYourself();
```

«Меня зовут Biff, мой рост 16 дюймов».

```
Clown clown3 = new Clown();
clown3.Name = anotherClown.Name;
clown3.Height = oneClown.Height - 3;
clown3.TalkAboutYourself();
```

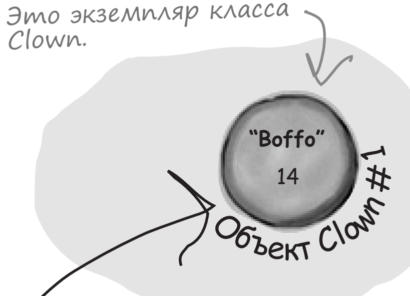
«Меня зовут Biff, мой рост 11 дюймов».

```
anotherClown.Height *= 2;
anotherClown.TalkAboutYourself();
```

«Меня зовут Biff, мой рост 32 дюйма».

Новые объекты добавляются в кучу — динамически распределяемую память.

Что происходит в памяти программы



Программа создает новый экземпляр класса Clown:

```
Clown myInstance = new Clown();
```

В выражении использованы два оператора. Первый объявляет переменную типа Clown (`Clown myInstance;`). Второй создает новый объект и присваивает его только что созданной переменной (`myInstance = new Clown();`). Вот как выглядит куча после выполнения каждого из операторов:

1

```
Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;
oneClown.TalkAboutYourself();
```

Создан первый объект и его поля.

2

```
Clown anotherClown = new Clown();
anotherClown.Name = "Biff";
anotherClown.Height = 16;
anotherClown.TalkAboutYourself();
```

Операторы создают второй объект и присваивают ему данные.

3

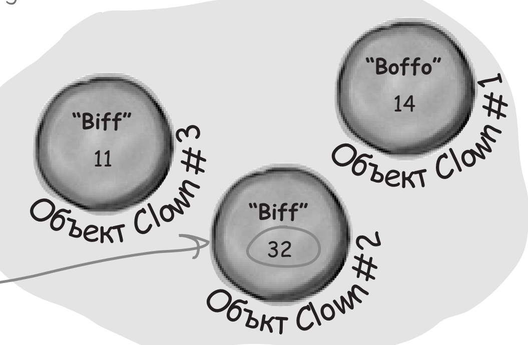
```
Clown clown3 = new Clown();
clown3.Name = anotherClown.Name;
clown3.Height = oneClown.Height - 3;
clown3.TalkAboutYourself();
```

Третий объект Clown создан и помещен в кучу.

4

```
anotherClown.Height *= 2;
anotherClown.TalkAboutYourself();
```

Так как команда new не используется, новый объект не создается. Редактируется только информация, которая уже имеется в памяти.



Значимые имена

При написании кода методов выбирается структура программы. Вы ищете ответы на вопросы: воспользоваться одним методом или, может быть, разбить его на несколько? А может быть, метод тут вообще не нужен? В результате можно получить как интуитивно понятный код, так и нечто запутанное и нечитаемое.

Квалифицированные разработчики пишут читаемый код. В этом помогают комментарии, но ничто не сравнится со значимыми именами методов, классов, переменных и полей.

1. Перед вами пример компактного кода. Это программа, управляющая автоматом по производству шоколадных батончиков.

```
int t = m.chkTemp();
if (t > 160) {
    T tb = new T();
    tb.clsTrpV(2);
    ics.Fill();
    ics.Vent();
    m.airsyschk();
}
```

Метод chkTemp() возвращает целое число... но для чего?

tb, ics, m — ужасные имена! Вы никогда не догадаетесь, зачем нужны эти переменные и какова функция класса T.

Метод clsTrpV() имеет один параметр, но вы в жизни не угадаете его предназначение.

Вы можете, взглянув на этот код, понять, какую функцию он выполняет?

2. Операторы не дают даже намеков на предназначение кода. Зато нужный результат получен при помощи всего одного метода. Но всегда ли максимальная компактность является конечной целью? Давайте разобьем код на несколько методов и сделаем его более простым. Вы на практике убедитесь в необходимости классов и осмысленных имен. Для начала нужно понять, какую задачу решает данный код.

Чтобы определить назначение кода, нужно понять, зачем он был создан. В качестве отправной точки используем страницу из инструкции, в соответствии с которой писалась программа.

Управление автоматом, производящим батончики

Автоматизированная система проверяет температуру нуги каждые 3 минуты. Если она **выше 160 °C**, необходимо **выполнить процедуру охлаждения (CICS)**.

- Закройте клапан турбины #2
- Заполните водой систему охлаждения
- Слейте воду
- Убедитесь, что в системе отсутствует воздух

3 Инструкция не только помогла определить назначение кода, но и показала, как сделать его более понятным. Мы узнали, что проверка условия определяет, не превышает ли параметр **t** значение 160, ведь в инструкции написано, что при 160 °C нуга становится слишком горячей. А буква **m** оказалась классом, который контролирует поведение автомата. В класс входит статический метод проверки температуры нуги и работы воздушной системы. Выделим проверку температуры в отдельный метод и дадим классу и методу смысловые имена.

```
public boolean IsNougatTooHot() {
   int temp = Maker.CheckNougatTemperature();
   if (temp > 160) {
     return true;
   } else {
     return false;
   }
}
```

Тип значений, возвращаемых методом IsNougatTooHot() (Не слишком ли нагрелась нуга).

Назовем класс Maker (Изготовитель), а метод CheckNougatTemperature, (Проверь температуру нуги).

Возвращаемые значения типа Boolean — это true или false.

4 При превышении рекомендуемой температуры инструкция требует выполнения процедуры CICS. Создадим еще один метод и выберем для класса `T` (он управляет турбиной) более значимое имя. Переименуем также класс `ics` (управляющий изолированной системой охлаждения). Этот класс содержит два статических метода: один — для заполнения системы водой (fill), второй — для слива воды (vent):

```
public void DoCICSVentProcedure() {
   Turbine turbineController = new Turbine();
   turbineController.CloseTripValve(2);
   IsolationCoolingSystem.Fill();
   IsolationCoolingSystem.Vent();
   Maker.CheckAirSystem();
}
```

Модификатор void означает, что метод не возвращает никакого значения.

5 Теперь, даже если вы не читали инструкцию и не знаете, что процедура CICS запускается при слишком высокой температуре нуги, **вы все равно можете понять, что делает код**:

```
if (IsNougatTooHot() == true) {
   DoCICSVentProcedure();
}
```

Помните о том, зачем вы пишете код. Это позволит сделать конечный результат простым и легко редактируемым. Выбор смысловых имен упрощает последующую расшифровку программы другими людьми и дает возможность улучшить код!

Структура классов

Почему же методы следует делать интуитивно понятными? **Потому что каждая программа решает проблему или имеет конкретную цель.** Целью далеко не всегда являются серьезные задачи, иногда программы могут писаться просто для забавы! Но каково бы ни было предназначение программы, чем больше код напоминает о том, какая именно проблема с его помощью решается, тем проще такую программу писать, читать и редактировать.

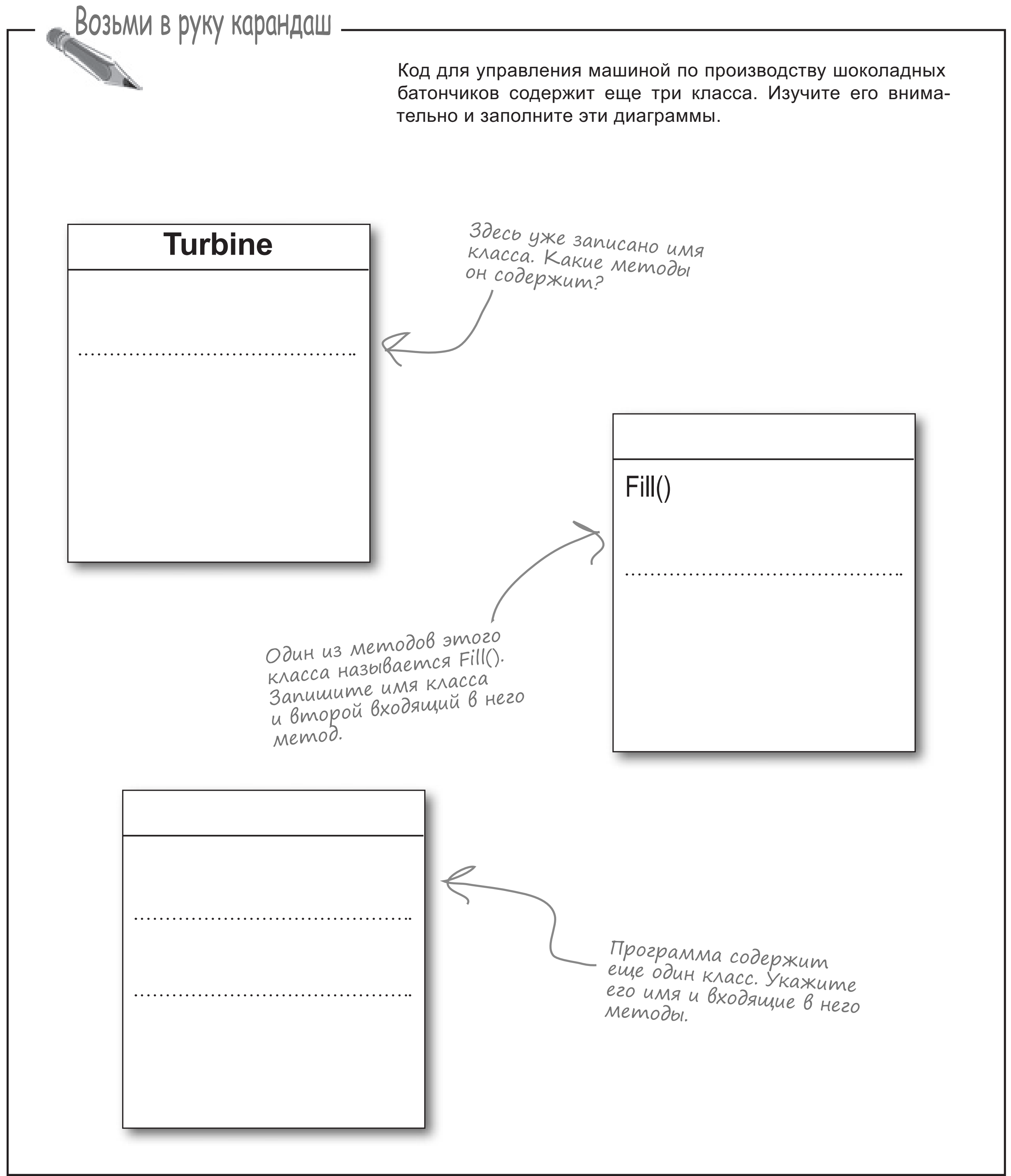
Пример построения диаграммы

Посмотрите на оператор `if` из пункта #5 на предыдущей странице. Вы уже знаете, что операторы находятся внутри методов, а методы в свою очередь — внутри классов? В данном случае оператор `if` принадлежит методу `DoMaintenanceTests()`, который является частью класса `CandyController`. Теперь разберемся, как соотносятся друг с другом код и диаграмма классов.

```
class CandyController {

  public void DoMaintenanceTests() {
    ...
    if (IsNougatTooHot() == true) {
       DoCICSVentProcedure();
    }
    ...
  }

  public void DoCICSVentProcedure() ...

  public boolean IsNougatTooHot() ...

}
```

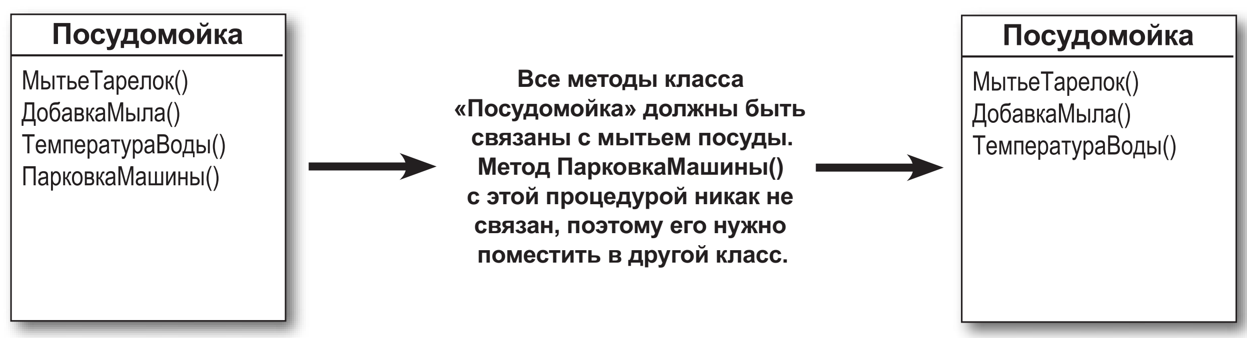
CandyController

DoMaintenanceTests()
DoCICSVentProcedure()
IsNougatTooHot()


Возьми в руку карандаш
Код для управления машиной по производству шоколадных батончиков содержит еще три класса. Изучите его внимательно и заполните эти диаграммы.
Turbine
Здесь уже записано имя класса. Какие методы он содержит?
Fill()
Один из методов этого класса называется Fill(). Запишите имя класса и второй входящий в него метод.
Программа содержит еще один класс. Укажите его имя и входящие в него методы.

Выбор структуры класса при помощи диаграммы

Диаграммы классов позволяют увидеть потенциальные проблемы еще **до того**, как вы начнете писать код. Предварительное обдумывание структуры классов гарантирует связь кода с поставленной перед вами проблемой. Оно позволит избежать работы над ненужным кодом и получить интуитивно понятную и легкую в применении программу.



Возьми в руку карандаш
Решение

Вот так должны выглядеть диаграммы классов, которые вам было предложено заполнить на предыдущей странице.

Понять, что Maker — это название класса, легко по его положению в имени Maker.CheckAirSystem().

Turbine	IsolationCoolingSystem	Maker
CloseTripValve()	Fill()	CheckNougatTemperature()
	Vent()	CheckAirSystem()

Возьми в руку карандаш

Все эти классы содержат ошибки. Напишите, в чем, по вашему мнению, они состоят и как их можно исправить.

Класс23

ВесБатончика()
ПечатьОбертки()
ПечатьОтчета()
Делаем()

Этот класс является частью знакомой вам системы производства шоколадных батончиков.

....

....

....

....

РазносчикПиццы

ДобавитьПиццу()
ПиццаДоставлена()
ПодсчетСуммы()
ВремяВозвращения()

РазносчицаПиццы

ДобавитьПиццу()
ПиццаДоставлена()
ПодсчетСуммы()
ВремяВозвращения()

Эти классы являются частью системы учета доставки пиццы.

....

....

....

....

....

КассовыйАппарат

Продажи()
НетПродаж()
ЗакачатьГаз()
ВозвратДенег()
ВсегоВКассе()
СписокТранзакций()
ДобавитьСумму()
ВычестьСумму()

Класс `КассовыйАппарат` является частью программы автоматизированной системы контроля в круглосуточном магазине.

....

....

....

....

Возьми в руку карандаш
Решение

Вот как мы скорректировали классы. Но это лишь один способ решения проблемы. Можно использовать и другие способы, в зависимости от того, как предполагается использовать класс.

Этот класс является частью знакомой вам системы производства шоколадных батончиков.

Из имени класса Class23.Go() непонятно его назначение. Кроме того, мы выбрали более осмысленное имя для последнего метода.

ДелаемБатончики
ВесБатончика() ПечатьОбертки() ПечатьОтчета() СоздатьБатончик()

Эти классы являются частью системы учета доставки пиццы.

Классы DeliveryGuy и DeliveryGirl выполняли одну задачу. Имеет смысл объединить их друг с другом, добавив поле для ввода половой принадлежности работника.

РазносчикПиццы
Пол
ДобавитьПиццу() ПиццаДоставлена() ПодсчетСуммы() ВремяВозвращения()

Поле Пол было добавлено, чтобы отслеживать результаты работы юношей и девушек по отдельности.

Класс `КассовыйАппарат` является частью программы автоматизированной системы контроля круглосуточного магазина.

Был удален метод ЗакачатьГаз() как единственный не имеющий отношения к работе кассовых аппаратов.

КассовыйАппарат
Продажи() НетПродаж() ВозвратДенег() ВсегоВКассе() СписокТранзакций() ДобавитьСумму() ВычестьСумму()

```
public partial class Form1 : Form {
  public Form1() {
    InitializeComponent();
  }
   private void button1_Click(object sender, EventArgs  e) {
    string result = "";
    Echo e1 = new Echo();
    _________________________
    int x = 0;
    while ( ___________ ) {
      result = result + e1.Hello() + "\n";
      __________________________
      if ( ____________ ) {
        e2.count = e2.count + 1;
      }
      if ( ____________ ) {
        e2.count = e2.count + e1.count;
      }
      x = x + 1;
    }
    MessageBox.Show(result + "Count: " + e2.count);
  }
}
class ____________ {
  public int _________ = 0;
    public string ___________ {
      return "helloooo...";
  }
}
```

Ребус в бассейне

Возьмите фрагменты кода из бассейна и поместите их на пустые строчки. Фрагменты можно использовать несколько раз. В бассейне есть и лишние фрагменты. Полученная в итоге программа должна выводить окно с показанным ниже текстом.

Результат:

```
helloooo...
helloooo...
helloooo...
helloooo...
Count: 10
OK
```

Дополнительный вопрос!

Как решить задачу, чтобы вместо **10** в последней строке оказалось **24**? Для этого достаточно заменить всего один оператор.

Каждый фрагмент можно использовать несколько раз!

```
x
y
e2
count

x < 4
x < 5
x > 0
x > 1

Echo
Tester
Echo( )
Count( )
Hello( )

e1 = e1 + 1;
e1 = count + 1;
e1.count = count + 1;
e1.count = e1.count + 1;

e2 = e1;
Echo e2;
Echo e2 = e1;
Echo e2 = new Echo( );

x == 3
x == 4
```

Ответ на с. 173.

У этой головоломки два решения. Можете найти оба?

Помогите парням

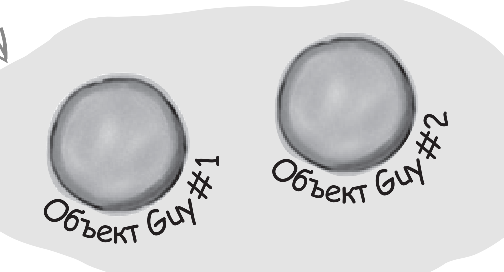
Джо и Боб все время одалживают друг другу деньги. Напишем программу отслеживания истории займов. Для начала нужно понять структуру создаваемого класса.

Guy
Name Cash
GiveCash() ReceiveCash()

Итак, наш класс называется Guy (Парень). Метод GiveCash() отвечает за передачу денег в долг, а метод ReceiveCash() — за их получение. Поля Name и Cash содержат информацию об имени и сумме соответственно.

1 Создадим класс Guy и добавим в форму два его экземпляра
Первое поле формы будет называться `joe` (для слежения за первым объектом), а второе `bob` (для слежения за вторым объектом).

Оператор new, создающий экземпляры, запускается в момент загрузки формы. На рисунке представлен вид кучи после этой операции.



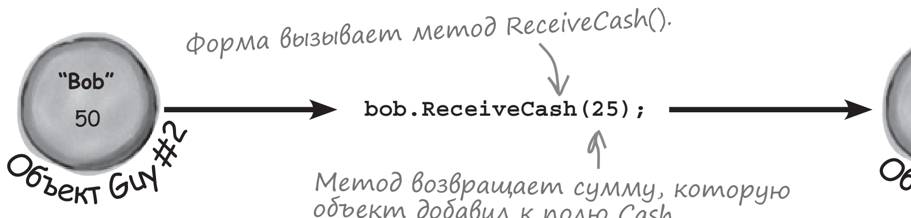
2 Сопоставим каждому объекту Guy поля Name и Cash
Два объекта соответствуют двум парням, у каждого из которых есть свое имя и некоторое количество денег в кошельке.

Поле Name содержит информацию об имени парня, а поле Cash — о сумме его наличности.

"Joe" 100 — Объект Guy #1
"Bob" 50 — Объект Guy #2

3 Парни дают деньги и получают их назад
Метод `ReceiveCash()` увеличивает количество денег у объекта, а метод `GiveCash()` уменьшает этот параметр.

Параметром метода ReceiveCash() является количество получаемых денег. Поэтому запись bob.ReceiveCash(25) означает, что Боб получит 25 долларов.



Проект «Парни»

Создайте проект Windows Forms Application (ведь нам понадобится форма). В окне Solution Explorer создайте класс с именем `Guy`. Добавьте в верхнюю часть файла этого класса строчку `using System.Windows.Forms;`, затем введите следующий код:

```
class Guy {
    public string Name;
    public int Cash;

    public int GiveCash(int amount) {
        if (amount <= Cash && amount > 0) {
            Cash -= amount;
            return amount;
        } else {
            MessageBox.Show(
             "У меня не хватает денег " + amount,
             Name + " говорит...");
            return 0;
        }
    }

    public int ReceiveCash(int amount) {
        if (amount > 0) {
            Cash += amount;
            return amount;
        } else {
            MessageBox.Show(amount + " мне не нужно",
                  Name + " говорит...");
            return 0;
        }
    }
}
```

Поле Name — это строка, с именем парня (Joe), а поле Cash — целое число, указывающее на количество наличных денег.

Метод GiveCash() имеет единственный параметр amount, указывающий, сколько денег следует отдать.

Оператор if проверяет, хватает ли денег на возврат долга. Если да, запрошенная сумма указывается в качестве возвращаемого значения.

Требуемая сумма должна быть больше нуля, иначе деньги будут добавлены в кошелек, а не взяты оттуда.

В случае нехватки денег появляется окно с сообщением, а метод GiveCash() возвращает значение 0 (ноль).

Метод ReceiveCash() также использует в качестве параметра переменную amount, проверяет ее знак, и если она больше нуля, добавляет к переменной cash.

Если переменная amount больше нуля, метод ReceiveCash () добавляет ее к переменной Cash. В противном случае появляется окно с сообщением и возвращается значение 0 (ноль).

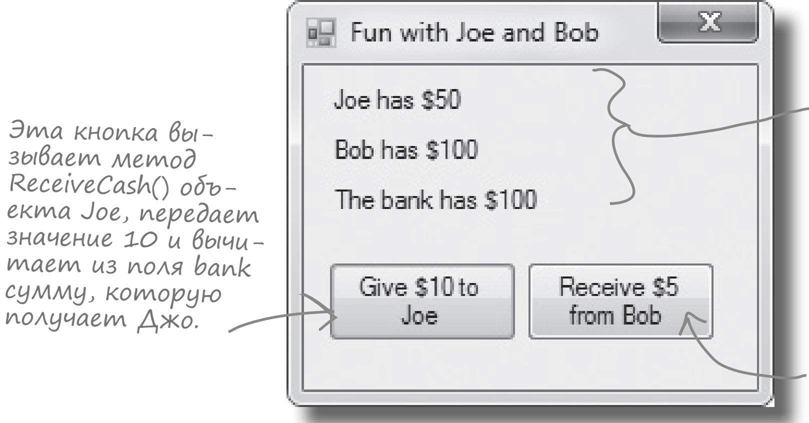
} — Следите за тем, чтобы количество открывающих скобок совпадало с количеством закрывающих. IDE поможет вам в этой задаче.

Что произойдет, если передать методу ReceiveCash() или GiveCash() объекта Guy отрицательное значение?

Форма для взаимодействия с кодом

Теперь нам нужна форма, которая будет работать с экземплярами класса Guy. Она должна содержать метки с именами парней и количеством денег у каждого их них, а также кнопки, управляющие процессом взятия и возврата денег. Еще парни должны откуда-то брать деньги, которые они одалживают друг другу, значит, нам следует добавить банк.

Нам понадобятся две кнопки и три метки.
Две верхние метки должны показывать сумму наличности у каждого из парней. К форме также нужно добавить поле bank — это еще одна метка. **По очереди выделяйте все метки** и **редактируйте их свойство «(Name)»** в окне Properties. Присвоив меткам имена joesCashLabel и bobsCashLabel вместо имен label1 и label2, вы сделаете код более читабельным.



Верхней метке присвойте имя joesCashLabel, средней — bobsCashLabel, а нижней — bankCashLabel. Свойство Text пока можно не редактировать.

Эта кнопка вызывает метод GiveCash() объекта Bob, передает значение 5 и прибавляет его к полю bank.

2 **Поля формы.**
Для отслеживания финансового состояния наших героев потребуются два поля. Назовите их joe и bob. Затем добавьте поле с именем bank для расчета, сколько форма должна взять у объектов, а сколько отдать им. Дважды щелкните на третьей метке и добавьте в появившийся код строки:

```
namespace Your_Project_Name {
    public partial class Form1 : Form {

        Guy joe;
        Guy bob;
        int bank = 100;

        public Form1() {
            InitializeComponent();
        }
```

Поля Joe и Bob объявлены в классе Guy.

Значение поля bank то возрастает, то уменьшается в зависимости от того, сколько денег форма отдала объектам Guy и сколько взяла от них.

Метод, обновляющий метки

Добавим к форме метод UpdateForm(), чтобы метки всегда показывали актуальное количество денег. **Убедитесь в наличии модификатора void**, так как данный метод не должен возвращать значение. Добавьте эти строчки под предыдущий код:

```
public void UpdateForm() {
    joesCashLabel.Text = joe.Name + " имеет $" + joe.Cash;
    bobsCashLabel.Text = bob.Name + " имеет $" + bob.Cash;
    bankCashLabel.Text = "В банке сейчас $" + bank;
}
```

Метки обновляются при помощи полей Name и Cash метода Guy.

Этот метод обновляет метки, изменяя их свойство Text.

Код взаимодействия кнопок с объектами

Убедитесь, что левая кнопка называется button1, а кнопка справа — button2. Дважды щелкните на каждой из кнопок, чтобы добавить методы button1_Click() и button2_Click() соответственно. И для каждой кнопки введите код:

Вы уже знаете, что можно изменить названия элементов управления. Вряд ли button1 или button2 — наилучший вариант. Какие названия выбрали вы?

```
private void button1_Click(object sender, EventArgs e) {
    if (bank >= 10) {
        bank -= joe.ReceiveCash(10);
        UpdateForm();
    } else {
        MessageBox.Show("В банке нет денег.");
    }
}

private void button2_Click(object sender, EventArgs e) {
    bank += bob.GiveCash(5);
    UpdateForm();
}
```

По щелчку на кнопке Give $10 to Joe, форма вызывает метод ReceiveCash() объекта Joe, но только если в банке достаточно денег.

Чтобы Джо мог получить деньги, в банке должно быть не меньше $10. Если их нет, появляется окно с сообщением.

Кнопка Receive $5 from Bob добавляет в банк деньги, полученные от Боба.

При отсутствии денег у Боба метод GiveCash() возвращает 0.

Начальный капитал Джо $50, а Боба — $100

А теперь самостоятельно **укажите начальное состояние полей Cash и Name для обоих экземпляров**. Код расположите под строкой InitializeComponent(), ведь процедура должна выполняться при инициализации формы. Завершив работу, убедитесь, что щелчок на левой кнопке забирает $10 из банка и отдает эту сумму Джо, а вторая кнопка забирает у Боба $5 и возвращает их в банк.

```
public Form1() {
    InitializeComponent();
    // Задайте начальные значения!
}
```

Добавьте код, создающий два экземпляра и задающий начальные значения их полей Name и Cash.

Упражнение

Упражнение Решение

Эти строки следует добавить в код формы под строкой InitializeComponent(), чтобы задать начальные значения полей **Cash** и **Name:**

```
public Form1() {
    InitializeComponent();

    bob = new Guy();
    bob.Name = "Bob";
    bob.Cash = 100;

    joe = new Guy();
    joe.Name = "Joe";
    joe.Cash = 50;

    UpdateForm();
}
```

Мы задаем свойства первого экземпляра Guy. Первая строчка создает объект, а две другие определяют значения его полей.

Аналогичная процедура проводится для второго экземпляра класса Guy.

Метод UpdateForm() обеспечивает актуальный вид меток после вызова формы.

Сохраните проект, мы к нему скоро вернемся.

Часто Задаваемые Вопросы

В: Почему мы не написали `Guy bob = new Guy()`?

О: Потому что поле `bob` уже было объявлено в верхней части формы. Помните, что оператор `int i = 5;` — это по сути два оператора `int i` и `i = 5;`? Сейчас происходит то же самое. Разумеется, можно объявить поле `bob`, написав: `Guy bob = new Guy();`. Но первая часть этого оператора уже имеется в коде формы. Так что вам нужна только вторая часть кода, помещающая в поле `bob` новый образец `Guy()`.

В: А почему бы нам не избавиться от строчки `Guy bob;` в верхней части кода формы?

О: После этого переменная `bob` будет существовать только внутри метода `public Form1()`, поскольку к переменным, объявленным внутри метода, невозможен доступ из других методов. Объявив же переменную вне метода, но при этом внутри добавленной вами формы или класса, вы даете к ней доступ из любого другого метода, принадлежащего этой же форме.

В: Что произойдет если оставить слово `Guy`? Если все-таки написать `Guy bob = new Guy()` вместо `bob = new Guy()`?

О: Форма не будет работать, так как переменная `bob` не задана. Если в начале кода формы было объявлено:

```
public partial class Form1 : Form {
    Guy bob;
```

а затем внутри метода объявлено еще раз:

```
Guy bob = new Guy();
```

получается, что вы объявили две переменные с одинаковыми именами. Одна из них действительна для всей формы, а вторая — только внутри метода. Следующая строка (`bob.Name = "Bob";`) обновляет только локальную переменную. В итоге при запуске программы появляется сообщение об ошибке («NullReferenceException не обработано»), означающее, что ссылка на объект не указывает на экземпляр объекта. То есть вы пытаетесь сослаться на объект, который еще не был создан.

Более простые способы присвоения начальных значений

Практически всегда создание объектов сопровождается присвоением им начальных значений. Объект Guy не исключение – он бесполезен, если не заданы значения полей Name и Cash. Для решения этой задачи в C# существует **инициализатор объектов**. В его основе лежит технология IntelliSense.

Инициализаторы объектов экономят ваше время и увеличивают компактность и читабельность кода.

1. Рассмотрим исходный код, присваивающий полям объекта Joe начальные значения.

```
joe = new Guy();
joe.Name = "Joe";
joe.Cash = 50;
```

2. Удалите две последние строки и точку с запятой после Guy() и добавьте справа фигурную скобку.

```
joe = new Guy() {
```

3. Нажмите пробел. Появится окно со списком полей, для которых могут быть заданы начальные параметры.

```
joe = new Guy() {
```

4. Нажмите Tab, чтобы добавить поле Cash. Затем присвойте ему значение 50.

```
joe = new Guy() { Cash = 50
```

5. Введите запятую, и сразу после нажатия пробела появится еще одно поле.

```
joe = new Guy() { Cash = 50,
```

6. Завершите работу инициализатора. Итак, вы уменьшили свой код на две строчки!

```
joe = new Guy() { Cash = 50, Name = "Joe" };
```

Новое объявление выполняет ту же функцию, что и приведенные вначале три строчки кода. Но оно короче и проще для чтения.

Вы уже использовали инициализатор в своей игре. Просто вернитесь назад.

Создание интуитивно понятных классов

★ **Программа должна решать какую-то задачу.**
Обдумайте проблему. Ответьте на вопросы: легко ли ее поделить на несколько частей? и как бы вы объяснили ее другому человеку?

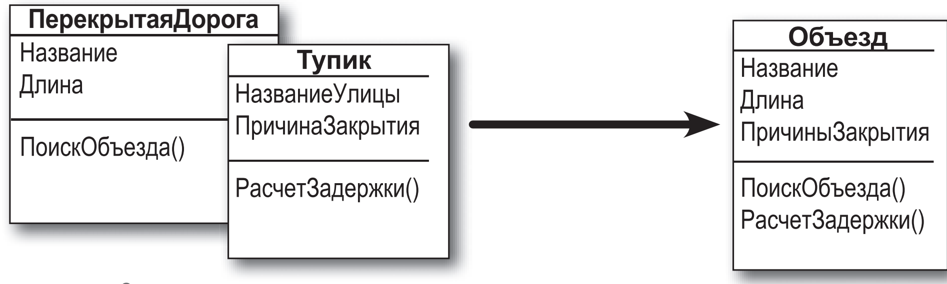
Мне нужно сравнить несколько маршрутов и выбрать самый короткий....

★ **С какими реальными объектами работает программа?**
Программе кормления животных в зоопарке наверняка потребуются классы для различных видов корма и различных видов животных.

★ **Присваивайте классам и методам значимые имена.**
Другие пользователи должны понимать назначение классов и методов по виду их имен.

myInst
Объект obj

★ **Ищите сходство между классами.**
Классы, выполняющие одинаковые функции, имеет смысл объединить. В системе, производящей батончики, может быть несколько турбин, но для закрытия клапана достаточно одного метода, в котором номер турбины будет использован в качестве параметра.

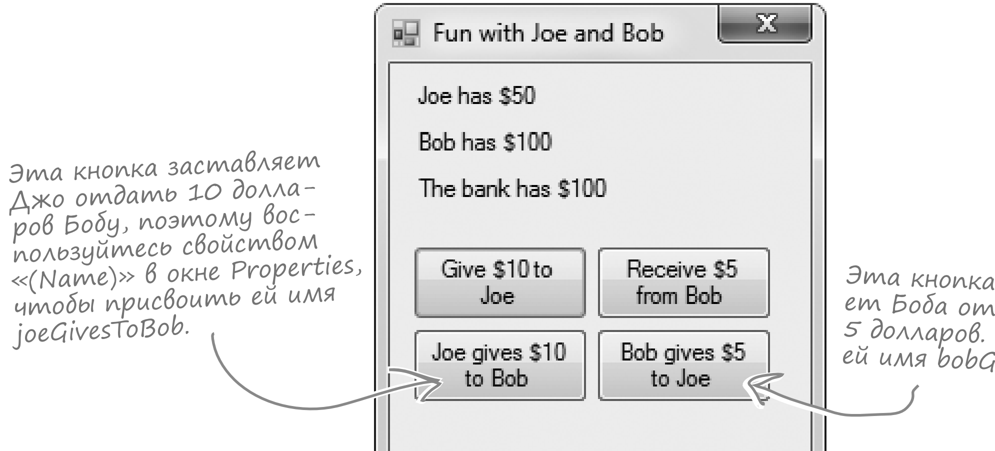


Добавьте кнопки к программе «Веселимся с Джо и Бобом», чтобы заставить парней передавать друг другу деньги.

1. **Присвойте экземпляру Bob начальные значения при помощи инициализатора.**
Вы уже проделывали эту операцию с экземпляром Joe. Потренируйтесь в работе с инициализатором объектов еще раз.

Если вы поспешили и уже щелкнули на кнопке, удалите ее, добавьте заново и присвойте новое имя. Затем удалите старый метод button3_Click() и добавьте новый.

2. **Еще две кнопки для формы**
Пусть при щелчке на первой кнопке Джо отдает Бобу 10 долларов, а при щелчке на второй Боб дает Джо 5 долларов. **Перед двойным щелчком на кнопке** поменяйте ее имя в окне Properties, используя свойство «(Name)». Первой кнопке присвойте имя **`joeGivesToBob`**, а второй — имя **`bobGivesToJoe`**.



3. **Заставим кнопки работать**
Дважды щелкните на кнопке `joeGivesToBob` в конструкторе. Форме будет добавлен метод `joeGivesToBob_Click()`, который запускается при любом щелчке на кнопке. Пусть этот метод заставляет Джо отдавать 10 долларов Бобу. Дважды щелкните на второй кнопке и заставьте новый метод `bobGivesToJoe_Click()` передать 5 долларов от Боба к Джо. Убедитесь, что после передачи денег форма обновляется.

Совет по конструированию форм. Эти кнопки на панели инструментов в IDE позволяют выравнивать элементы в конструкторе, задавать им одинаковый размер, равномерного распределять их в пространстве и перемещать на задний и передний план.

Упражнение Решение

Вот как стала выглядеть программа после того, как к ней добавили кнопки передачи денег от Боба к Джо и обратно.

```
public partial class Form1 : Form {
    Guy joe;
    Guy bob;
    int bank = 100;

    public Form1() {
        InitializeComponent();
        bob = new Guy() { Cash = 100, Name = "Bob" };
        joe = new Guy() { Cash = 50, Name = "Joe" };
        UpdateForm();
    }

    public void UpdateForm() {
        joesCashLabel.Text = joe.Name + " имеет $" + joe.Cash;
        bobsCashLabel.Text = bob.Name + " имеет $" + bob.Cash;
        bankCashLabel.Text = "В банке $" + bank;
    }

    private void button1_Click(object sender, EventArgs e) {
        if (bank >= 10) {
            bank -= joe.ReceiveCash(10);
            UpdateForm();
        } else {
            MessageBox.Show("В банке нет денег.");
        }
    }

    private void button2_Click(object sender, EventArgs e) {
        bank += bob.GiveCash(5);
        UpdateForm();
    }

    private void joeGivesToBob_Click(object sender, EventArgs e) {
        bob.ReceiveCash(joe.GiveCash(10));
        UpdateForm();
    }

    private void bobGivesToJoe_Click(object sender, EventArgs e) {
        joe.ReceiveCash(bob.GiveCash(5));
        UpdateForm();
    }

}
```

Это инициализаторы объектов для двух экземпляров класса Guy.

Чтобы заставить Джо дать деньги Бобу, мы вызываем метод GiveCash() и передаем возвращаемое им значение методу ReceiveCash().

Результаты работы метода GiveCash() используются в качестве параметра метода ReceiveCash().

Важно помнить, кто дает деньги, а кто их получает.

Решение ребуса в бассейне

Требовалось расположить представленные в бассейне фрагменты кода на пустых строчках таким образом, чтобы в итоге получилась работающая программа. Вот как это нужно было сделать:

```
public partial class Form1 : Form {
  public Form1() {
    InitializeComponent();
  }
  private void button1_Click(object sender, EventArgs e) {
    string result = "";
    Echo e1 = new Echo();
    Echo e2 = new Echo();
    int x = 0;
    while ( x < 4 ) {
      result = result + e1.Hello() + "\n";
      e1.count = e1.count + 1;
      if ( x ==3 ) {
        e2.count = e2.count + 1;
      }
      if ( x > 0 ) {
        e2.count = e2.count + e1.count;
      }
      x = x + 1;
    }
    MessageBox.Show(result + "Count: " + e2.count);
  }
}
class Echo {
  public int count = 0;
    public string Hello() {
      return "helloooo...";
  }
}
```

Вот правильный ответ.

А это ответ на дополнительный вопрос!
Echo e2 = e1;

Альтернативное решение:
x == 4
и
x < 4

4 типы и ссылки

10:00 утра.

Куда подевались наши данные?

Без данных программы бесполезны.

Взяв информацию от пользователей, вы производите новую ***информацию***, чтобы вернуть ее им же. Практически все в программировании связано ***с обработкой данных*** тем или иным способом. В этой главе вы познакомитесь с используемыми в C# ***типами данных***, узнаете методы работы с ними и даже ужасный секрет ***объектов*** (только т-с-с-с… объекты — это тоже данные).

Тип переменной определяет, какие данные она может сохранять

Все проекты этой главы являются приложениями Windows Forms. Если вы видите указание создать новый проект, не содержащее дополнительных уточнений, значит, речь идет о приложении Windows Forms, созданном в Visual Studio для рабочего стола Windows.

Количество **встроенных типов** C# велико, и каждый из них хранит свой собственный вид данных. С некоторыми из них вы уже познакомились и даже поработали. Пришла пора узнать о неизвестных доселе типах, которые крайне пригодятся вам в будущем.

Наиболее часто используемые типы

Вряд ли вас удивит тот факт, что типы int, string, bool и double являются самыми распространенными.

- int хранит **целые** числа от –2 147 483 648 до 2 147 483 647;
- string хранит текст произвольной длины (в том числе и пустую строку "");
- bool хранит логические значения — true или false;
- double хранит **вещественные** числа от ±5.0 · 10^{324} до ±1.7 · 10^{308} до 16 значащих цифр. Подобный диапазон выглядит странным и сложным, но на самом деле все очень просто. Словосочетание «значащие цифры» указывает на *точность* числа: и 35 048 410 000 000, и 1 743 059, и 14.43857, и 0.00004374155 имеют по семь значащих цифр. Запись 10^{308} означает, что вы можете хранить любое число не больше 10^{308}, при условии, что количество значащих цифр не превышает 16. С другой стороны диапазона — 10^{-324}, позволяет хранить числа не меньше 10^{-324}... но, как несложно догадаться, опять же при условии, что количество значащих цифр не превышает 16.

Форма представления, при которой число хранится в виде мантиссы и показателя степени, называется числом с плавающей точкой (запятой).

Целочисленные типы

Часто вы меняете тип переменной, а проблема может быть решена и при помощи «циклического присваивания», о котором мы поговорим через пару страниц.

Когда оперативная память компьютера стоила дорого, а процессоры работали медленно, использование неверного типа данных могло серьезно замедлить работу программы. К счастью, времена изменились и теперь для хранения целых чисел в большинстве случаев достаточно типа int. Но иногда требуются дополнительные возможности, поэтому в C# присутствуют такие типы, как:

- byte хранит **целые** числа от 0 до 255;
- sbyte хранит **целые** числа от –128 до 127;
- short хранит **целые** числа от –32 768 до 32 767;
- ushort хранит **целые** числа от 0 до 65,535;
- uint хранит **целые** числа от 0 до 4 294 967 295;
- long хранит **целые** числа в диапазоне от минус до плюс 9 триллионов;
- ulong хранит **целые** числа от 0 до примерно 18 триллионов.

Буква «s» означает «со знаком». То есть число может быть отрицательным.

Буква «u» означает «без знака».

Типы для хранения *очень* БОЛЬШИХ и *очень* маленьких чисел

Иногда семи значащих цифр недостаточно. Бывает так, что 10^{38} — недостаточно большое, а 10^{-45} — недостаточно малое. С такими проблемами сталкиваются в финансовом учете и научных исследованиях, и для них в C# предназначены дополнительные типы:

- `float` хранит любое число в диапазоне от $\pm 1.5 \cdot 10^{-45}$ до $\pm 3.4 \cdot 10^{38}$ с 7 значащими цифрами;
- double хранит любое число в диапазоне от $\pm 5.0 \cdot 10^{-324}$ до $\pm 1.7 \cdot 10^{308}$ с 15–16 значащими цифрами.
- `decimal` хранит любое число в диапазоне от $\pm 1.0 \cdot 10^{-28}$ до $\pm 7.9 \cdot 10^{28}$ с 28–29 значащими цифрами.

Тип double более популярен, чем float. Значения этого типа используют многие свойства XAML.

Тип данных decimal часто встречается в программах финансового учета.

Свойство Value элемента управления numericUpDown относится к типу данных decimal.

«Константа» — это число, которое вы вводите в код. В выражении «int i = 5;» 5 — это константа.

Константы тоже имеют тип

Числа, используемые в программе на C#, называются константами... и все они принадлежат какому-то типу. Попробуйте написать код, присваивающий значение `14.7` переменной типа `int`:

```
int myInt = 14.7;
```

При компиляции вы увидите:

	Description
1	Cannot implicitly convert type 'decimal' to 'int'. An explicit conversion exists (are you missing a cast?)

IDE сообщает, что константа `14.7` принадлежит типу `double`. Вставив в конец букву `F` (`14.7F`), вы поменяете тип константы на `float`. А в форме `14.7M` оно будет принадлежать уже типу `decimal`.

«M» означает «money» (деньги). Я не шучу!

Попытавшись присвоить константу типа float переменной типа double или decimal, вы увидите окно с подсказкой.

Еще несколько встроенных типов

Для хранения единичных символов, например Q, 7 или $ используется тип `char`. Константы этого типа всегда заключаются в одиночные кавычки (`'x'`, `'3'`). В кавычки можно заключить и **esc-последовательность** (`'\n'` — это перенос строки, `'\t'` — знак табуляции). Хотя в коде эти последовательности фигурируют в виде пары символов, программа хранит их в памяти в виде одного.

И наконец, тип таких данных, как **`object`**. Вы уже получали объекты, создавая экземпляры классов. Любой из них мог быть переменной типа `object`. О том, как работают объекты и переменные, которые на них ссылаются, мы еще поговорим в этой главе.

О связи между типами char и byte вы узнаете в главе 9.

У Калькулятора в Windows 7 существует режим «Программист», позволяющий использовать бинарные и десятичные числа одновременно!

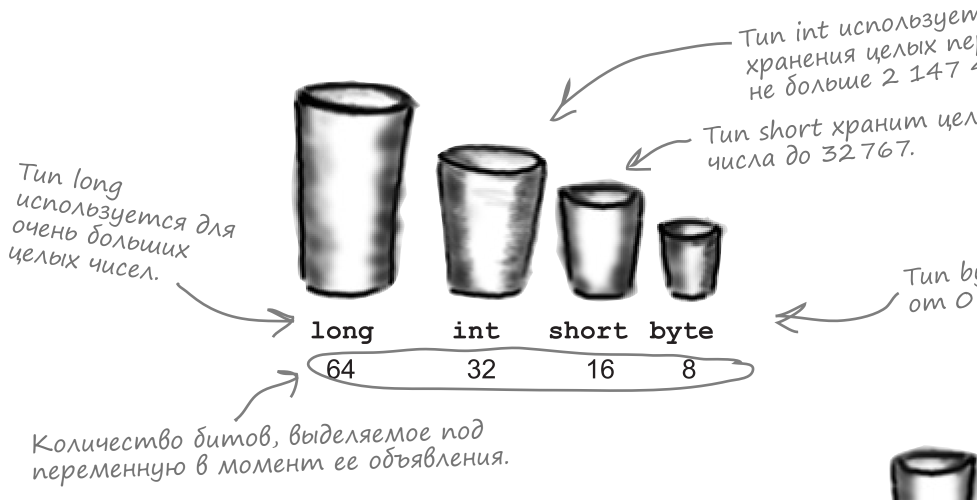
Встроенный калькулятор Windows можно использовать для перевода десятичных чисел в двоичные: перейдите в Инженерный режим, введите число и установите переключатель в положение **Bin**. Для обратного преобразования достаточно вернуть переключатель в положение **Dec**. Проделайте эту операцию с **пограничными значениями целых типов** (скажем, с –32 768 и 255). Вы можете объяснить, ***почему*** в C# используются именно такие значения?

Наглядное представление переменных

Не все переменные попадают в кучу. Значимые типы хранят данные в другой части памяти, которая называется стеком. Более подробно мы поговорим об этом в главе 14.

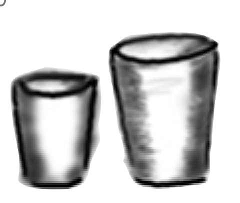
Данные занимают место в памяти. (Помните кучу из предыдущей главы?) Значит, следует учитывать, *сколько* пространства требуется под строку или число в программе. Именно поэтому мы пользуемся переменными. Они позволяют выделить достаточно места для хранения данных.

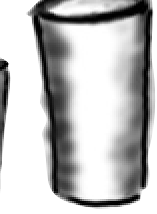
Представим переменную в виде стакана. В C# используются разные стаканы для хранения данных разных типов. Чем больше переменная, тем больший стакан ей требуется.



Числа с десятичной точкой хранятся по-другому. Для большинства таких чисел подойдет тип `float`, занимающий меньше всего места в памяти. Если вам требуется большая точность, используйте тип `double`. А для финансовых приложений, в которых хранится информация о курсах валют, используется тип `decimal`.

Впрочем, речь не только о цифрах. (Вы же не наливаете горячий кофе в пластиковый стаканчик, а холодный — в бумажный.) Компилятор C# умеет обрабатывать и нечисленные типы данных. Тип `char` хранит один символ, а тип `string` позволяет хранить целый набор символов. При этом под объект `string` не выделяется предустановленное место в памяти. Он расширяется в зависимости от количества помещаемой в него информации. А тип данных `bool` хранит значения true и false, с которыми вы уже сталкивались при работе с оператором `if`.





10 литров в 5-литровой банке

Объявив тип переменной, вы фактически объясняете компилятору, как ее следует воспринимать. Компилятор видит стаканы, а не то, что в них налито. Поэтому такой код работать не будет:

```
int leaguesUnderTheSea = 20000;

short smallerLeagues = leaguesUnderTheSea;
```

Хотя число 20 000 попадает в диапазон, заданный для типа данных `short`, переменная `leaguesUnderTheSea` была объявлена как `int`, и компилятор не может положить ее в контейнер `short`. Следовательно, вам всегда нужно следить за совпадением типов данных.

Компилятор «видит», что вы пытаетесь положить стакан типа int в стакан типа short. Содержимое стакана int при этом не имеет значения.

Такое поведение имеет смысл, так как позволяет избежать ситуации, когда вы попытаетесь поместить содержимое большого стакана в стакан маленького объема.

Возьми в руку карандаш

Обведите три оператора, которые не будут компилироваться из-за несовпадения типов данных или из-за того, что им пытаются присвоить слишком большое или слишком маленькое значение.

```
int hours = 24;                      string taunt = "your mother";

short y = 78000;                     byte days = 365;

bool isDone = yes;                   long radius = 3;

short RPM = 33;                      char initial = 'S';

int balance = 345667 - 567;          string months = "12";
```

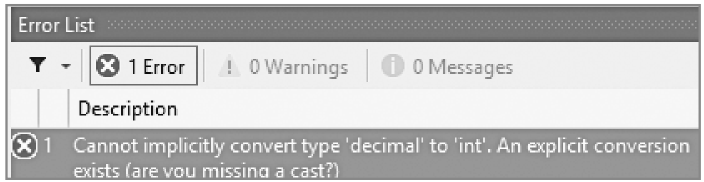
Приведение типов

Посмотрим, что произойдет при попытке назначить значение типа decimal переменной типа int.

1. Создайте новый проект и добавьте кнопку. Для метода **Click()** этой кнопки напишите:

```
decimal myDecimalValue = 10;
int myIntValue = myDecimalValue;

MessageBox.Show("The myIntValue is " + myIntValue);
```

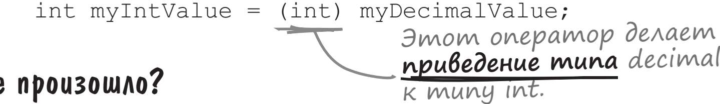
2. Попытавшись запустить программу, вы получите сообщение об ошибке:



IDE предполагает, что вы забыли осуществить приведение типов (casting).

3. Устраните ошибку, **преобразовав** тип decimal в int. Внесите во вторую строку следующие изменения:

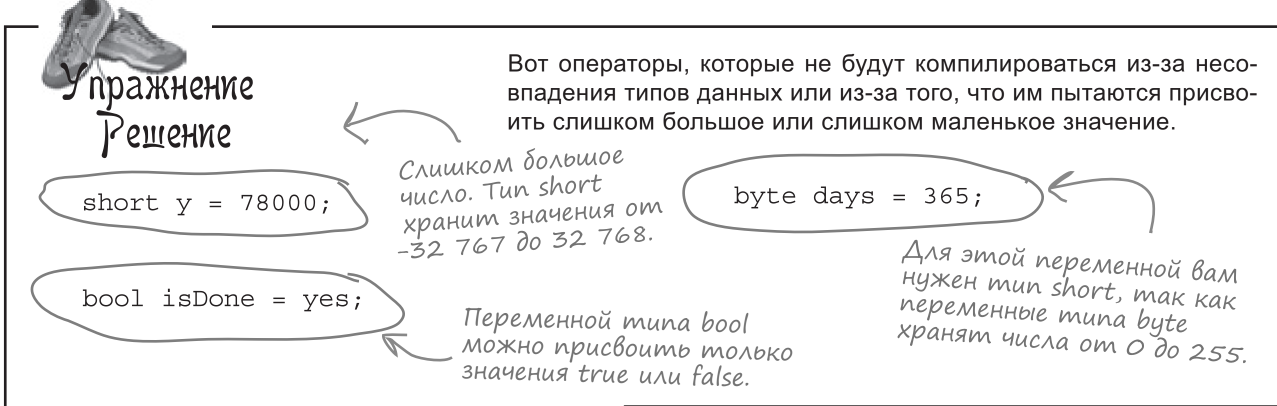
```
int myIntValue = (int) myDecimalValue;
```



Что же произошло?

Компилятор не позволяет присваивать значения несовпадающих типов, даже если переменная попадает в правильный диапазон. Приведение типа — это объяснение компилятору, что вы знаете о несовпадении типов, но в данном конкретном случае осознанно осуществляете операцию присваивания.

Вернитесь в начало предыдущей главы и посмотрите, как приведение типов использовалось для передачи значения NumericUpDown форме Talker Tester.



О типах в C# можно прочитать тут:
http://msdn.microsoft.com/ru-ru/library/s1ax56ch.aspx

Автоматическая коррекция слишком больших значений

Вы уже видели, что тип `decimal` может быть приведен к типу `int`. Получается, *любое* число может быть приведено к *любому типу*. Но это не означает сохранения ***значения***. Приведем переменную типа `int` со значением 365 к типу `byte`. Число 365 выходит за границы диапазона этого типа. Но вместо сообщения об ошибке произойдет **циклическое присваивание**: например, 256 после приведения к типу `byte` превратится в 0, 257 — в 1, а 365 — в **109**. Как только вы дойдете до 255, произойдет переход к нулевому значению.

Фактически «зацикливание» — это операция деления по модулю. Переключите калькулятор в Инженерный режим, наберите 365 Mod 256, и вы получите 109.

Да! Это делал за вас оператор + .

Вы пользовались оператором +, который **автоматически преобразовывал данные**. Когда вы добавляли число и логическое значение к строке, указанное вами значение автоматически преобразовывалось к типу string. Операторы + (или *, / или -), примененные к значениям различных типов, **автоматически преобразовывают меньший тип в больший**. Вот пример:

```
int myInt = 36;
float myFloat = 16.4F;;
myFloat = myInt + myFloat;
```

После числа, которое присваивается переменной типа float, нужно добавлять F, чтобы указать компилятору на его принадлежность к типу float.

Диапазон значений типа `int` больше диапазона значений типа `float`, оператор + преобразует переменную `myInt` к типу `float` и только потом прибавляет ее к переменной `myFloat`.

Возьми в руку карандаш

Операция приведения работает не со всеми типами. Создайте новый проект, добавьте кнопку, дважды щелкните на ней и введите следующие строки. Попытайтесь построить программу. Зачеркните строки с ошибками. Это поможет понять, для каких типов приведение допустимо, а для каких 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;
```

Иногда приведение типов происходит автоматически

Есть два случая, когда вам не требуется делать приведение типов. Автоматически эта процедура выполняется, во-первых, при проведении арифметических операций:

```
long l = 139401930;

short s = 516;

double d = l - s;

d = d / 123.456;

MessageBox.Show("Ответ " + d);
```

Параметр типа short вычитается из параметра типа long, а оператор = преобразует результат к типу double.

Благодаря оператору + происходит преобразование типа decimal к типу string.

Во-вторых, автоматическое преобразование происходит при **объединении** строк оператором +. Все числа при этом преобразуются к типу string. Рассмотрим пример, в котором первые две строки кода написаны правильно, третья не может быть скомпилирована.

```
long x = 139401930;

MessageBox.Show("Ответ " + x);

MessageBox.Show(x);
```

Компилятор выдает сообщение об ошибке в связи с неправильным аргументом (аргументом в C# называется значение, передаваемое методу в качестве параметра). Параметр метода `MessageBox.Show()` должен принадлежать типу `string`, код же передает переменную типа `long`. Впрочем, вы легко можете осуществить это преобразование при помощи метода `ToString()`. Этим методом обладают все объекты. (И все созданные вами классы имеют метод `ToString()`, возвращающий имя класса.) Именно с его помощью можно преобразовать x в параметр метода `MessageBox.Show()`:

```
MessageBox.Show(x.ToString());
```

Упражнение Решение

Итак, вы убедились, что приведение возможно далеко не для всех типов. Зачеркнутые строки не позволяют осуществить построение программы.

```
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;
```

Аргументы метода должны быть совместимы с типами параметров

Попытайтесь вызвать метод `MessageBox.Show(123)`, то есть передать методу `MessageBox.Show()` константу `(123)` вместо строки. IDE не позволит построить программу. Появится сообщение об ошибке: «Аргумент '1': преобразование из типа `int` в тип `string` невозможно». Иногда преобразование происходит автоматически: например, если ожидалось значение типа `int`, а вы передали методу значение типа short, — но в случае с переменными типа `int` и `string` это невозможно.

Параметр — это то, что вы определяете внутри метода, а аргумент — что в него передается. Метод с параметром типа int может принять аргумент типа byte.

Не только метод `MessageBox.Show()`, *все* методы, даже написанные вами, будут показывать ошибку компиляции, если передать им параметр неправильного типа. Попробуйте использовать на практике этот совершенно правильный метод:

Ошибка «неверный аргумент» означает, что вы попытались вызвать метод с переменными, тип которых не совпадает с его параметрами.

```
public int MyMethod(bool yesNo) {

   if (yesNo) {
      return 45;
   } else {
      return 61;
   }

}
```

Код, вызывающий этот параметр, не должен передавать ему переменную yesNo. Передать он должен только логическое значение. Переменная yesNo будет вызываться исключительно внутри метода.

Все работает, пока вы передаете методу то, что он ожидает получить (логическую переменную), вызов `MyMethod(true)` или `MyMethod(false)` позволяет легко скомпилировать код.

Переменной, параметру или полю можно назначить произвольное значение при помощи типа **object**.

Но что получится, если передать методу число или строку? Вы получите уже знакомое сообщение об ошибке. Попытайтесь передать параметр типа `Boolean`, но в качестве возвращаемого значения укажите строку или передайте результат методу `MessageBox.Show()`. Код перестанет работать, ведь метод возвращает значение типа `int`, а не `long` или `string`, которое требуется методу `MessageBox.Show()`.

Вы делали это в проекте "Save the Humans"— вернитесь и убедитесь сами; надеемся, вы сможете найти этот фрагмент.

Не нужно писать в явном виде if (yesNo == true), так как оператор if всегда проверяет истинность условия. Для проверки невыполнения условия используйте оператор ! (это оператор отрицания). if (!yesNo) означает то же самое, что и if (yesNo == false). Поэтому не удивляйтесь, когда видите в образцах кода сокращенную запись if (yesNo) или if (!yesNo) вместо развернутой проверки соблюдения условия.

Вы можете использовать для имен переменных даже зарезервированные слова. Достаточно поставить в начале знак @.

Упражнение

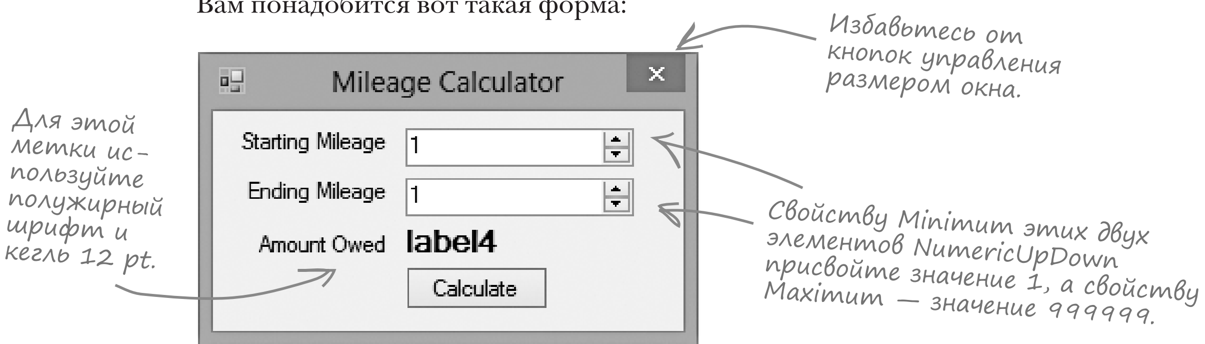
В C# существует около 77 **зарезервированных слов**. Их нельзя использовать в качестве имен переменных. К концу данной книги вы должны познакомиться с ними всеми, а пока посмотрите на список слов, с которыми вы уже сталкивались. Напишите, зачем они нужны.

`namespace`	
`for`	
`class`	
`public`	
`else`	
`while`	
`using`	
`if`	
`new`	

Ответы на с. 215.

Создадим калькулятор расходов для деловой поездки. Вы будете вводить данные с одометра в начале и конце путешествия, а счетчик будет вычислять, какое расстояние было пройдено и какую сумму денег вам вернут в бухгалтерии, при условии, что за каждую пройденную милю полагается $.39.

1 Новый проект Windows.
Вам понадобится вот такая форма:



Завершив создание формы, дважды щелкните на кнопке и добавьте код.

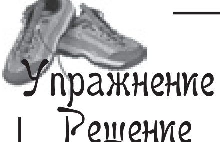
2 Переменные, которые потребуются для калькулятора.
Поместите переменные в строчки, объявляющие класс, в верхней части кода `Form1`. Вам потребуются две целочисленные переменные для начальных и конечных показаний одометра. Присвойте им имена `startingMileage` и `endingMileage`. Затем вам потребуются три нецелых числа. Выберите для них тип `double` и имена `milesTraveled` (Пройдено миль), `reimburseRate` (Коэффициент возмещения) и `amountOwed` (Должны денег). Переменной `reimburseRate` присвойте значение .39.

3 Чтобы заставить калькулятор работать.
Добавьте код для метода `button1_Click()`:

- ★ Убедитесь, что значение поля *StartingMileage* меньше значения поля *EndingMileage*. В противном случае должно выводится окно с текстом «Начальный пробег не может превышать конечный». Присвойте этому окну имя «Cannot Calculate» (Невозможно рассчитать).
- ★ Вычтите начальный пробег из конечного и умножьте полученный результат на тариф:

```
milesTraveled = endingMileage -= startingMileage;
amountOwed = milesTraveled *= reimburseRate;
label4.Text = "$" + amountOwed;
```

4 Запуск программы.
Убедитесь в вводе правильных значений. Укажите начальный пробег больше конечного и убедитесь в появлении окна с сообщением.



Вот как выглядит код для первой части упражнения по расчету компенсации.

```
public partial class Form1 : Form
{
    int startingMileage;
    int endingMileage;
    double milesTraveled;
    double reimburseRate = .39;
    double amountOwed;
    public Form1() {
        InitializeComponent();
    }
    private void button1_Click(object sender, EventArgs e){
      startingMileage = (int) numericUpDown1.Value;
      endingMileage = (int)numericUpDown2.Value;
        if (startingMileage <= endingMileage){
            milesTraveled = endingMileage -= startingMileage;
            amountOwed = milesTraveled *= reimburseRate;
            label4.Text = "$" + amountOwed;
        } else {
            MessageBox.Show(
      "Начальный пробег не может превышать конечный",
                        "Cannot Calculate Mileage");
        }
    }
}
```

Так как в данном случае число может превысить 999999, типы short или byte не подходят.

Помните, что для элемента управления umericUpDown вам нужно поменять тип decimal на тип int?

Здесь рассчитываете количество пройденных миль, которое затем умножается на тариф возмещения расходов.

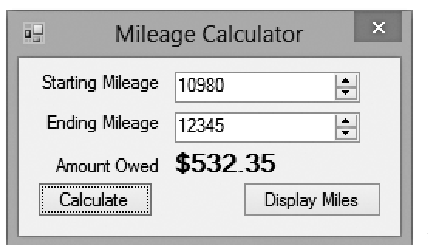
Здесь используется альтернативный способ вызова методаMessageBox.Show(). Первый параметр — это отображаемое сообщение, а второй — заголовок.

Кнопка, кажется, работает, но есть одна проблема. Вы можете указать ее?

Отладка калькулятора расстояний

В калькуляторе расстояний присутствует ошибка. Если код работает не так, как вы ожидали, для этого есть причина, и вам нужно ее найти. Попробуем понять, куда в данном случае вкралась ошибка и как ее можно исправить.

Еще одна кнопка.
Попытаемся понять, почему не работает форма, добавив еще одну кнопку, которая будет показывать значение поля `milesTraveled`. (Это можно сделать при помощи отладчика!)



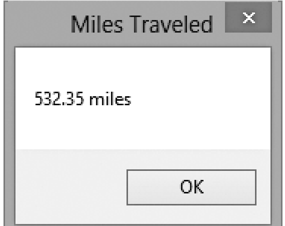
Если сначала щелкнуть на кнопке Calculate, а потом на этой кнопке, должно появиться окно диалога с указанием пробега.

Закончив конструирование формы, дважды щелкните на кнопке Display Miles, чтобы добавить код.

Понадобится всего одна строка.
Нам нужно окно, в котором будет отображаться значение переменной `milesTraveled`, не так ли? Для этого достаточно одной строки:

```
private void button2_Click(object sender, EventArgs e) {
       MessageBox.Show(milesTraveled + " miles", "Miles Traveled");
}
```

Запуск программы.
Введите какие-нибудь значения и посмотрите, что получится. После ввода начального и конечного пробега щелкните на кнопке Calculate. После этого щелчок на кнопке Display Miles покажет значение поля `milesTraveled`.



Вот только странно...
При любых введенных значениях пробег совпадает с суммой возмещения. Почему?

Комбинация с оператором =

Обратите внимание на оператор, который использовался для вычитания начального пробега из конечного (-=). Проблема в том, что он не только вычитает, но и присваивает значение переменной. То же самое происходит в строке, где мы умножаем пройденные мили на тариф возмещения. Поэтому нужно заменить операторы -= и *= на - и *:

```
private void button1_Click(object sender, EventArgs e)
{
  startingMileage = (int) numericUpDown1.Value;
  endingMileage = (int)numericUpDown2.Value;
    if (startingMileage <= endingMileage){
        milesTraveled = endingMileage -= startingMileage;
        amountOwed = milesTraveled *= reimburseRate;
        label4.Text = "$" + amountOwed;
    } else {
        MessageBox.Show("Начальный пробег не может превышать конечный",
                        "Cannot Calculate Mileage");
    }
```

Это так называемые ***составные операторы присваивания (compound operators)****.*

При такой записи код не будет менять значение переменных `endingMileage` *и* `milesTraveled`*.*

```
milesTraveled = endingMileage - startingMileage;
amountOwed = milesTraveled * reimburseRate;
```

Помогли ли нам значимые имена переменных? Конечно! Посмотрим на функцию каждой переменной. По названию `milesTraveled` (Пройденные мили)вы понимаете, что эту переменную форма отображает неправильно, и догадываетесь, как можно исправить ситуацию. Вам было бы намного сложнее локализовать проблему, если бы код выглядел вот так:

```
mT = eM -= sM;
aO = mT *= rR;
```

По таким именам переменных невозможно понять, какую функцию они выполняют.

Объекты тоже используют переменные

До этого момента мы рассматривали объекты отдельно. На самом же деле это еще один тип данных. Объекты обрабатываются аналогично цифрам, строкам и логическим значениям. И точно так же используют переменные:

Работа с типом int

① Пишем оператор объявления типа.

```
int myInt;
```

② Присваиваем новой переменной значение.

```
myInt = 3761;
```

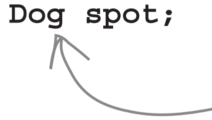
③ Используем переменную в коде.

```
while (i < myInt) {
```

Работа с объектом

❶ Пишем оператор объявления типа.

```
Dog spot;
```



При объявлении переменной название класса используется в качестве типа.

❷ Присваиваем новой переменной значение:

```
spot = new Dog();
```

❸ Проверяем одно из полей объекта.

```
while (spot.IsHappy) {
```

Получается, что нет разницы с чем работать, с объектом или с числом. В любом случае я работаю с переменной.

Да, объекты — это всего лишь еще один тип переменных.

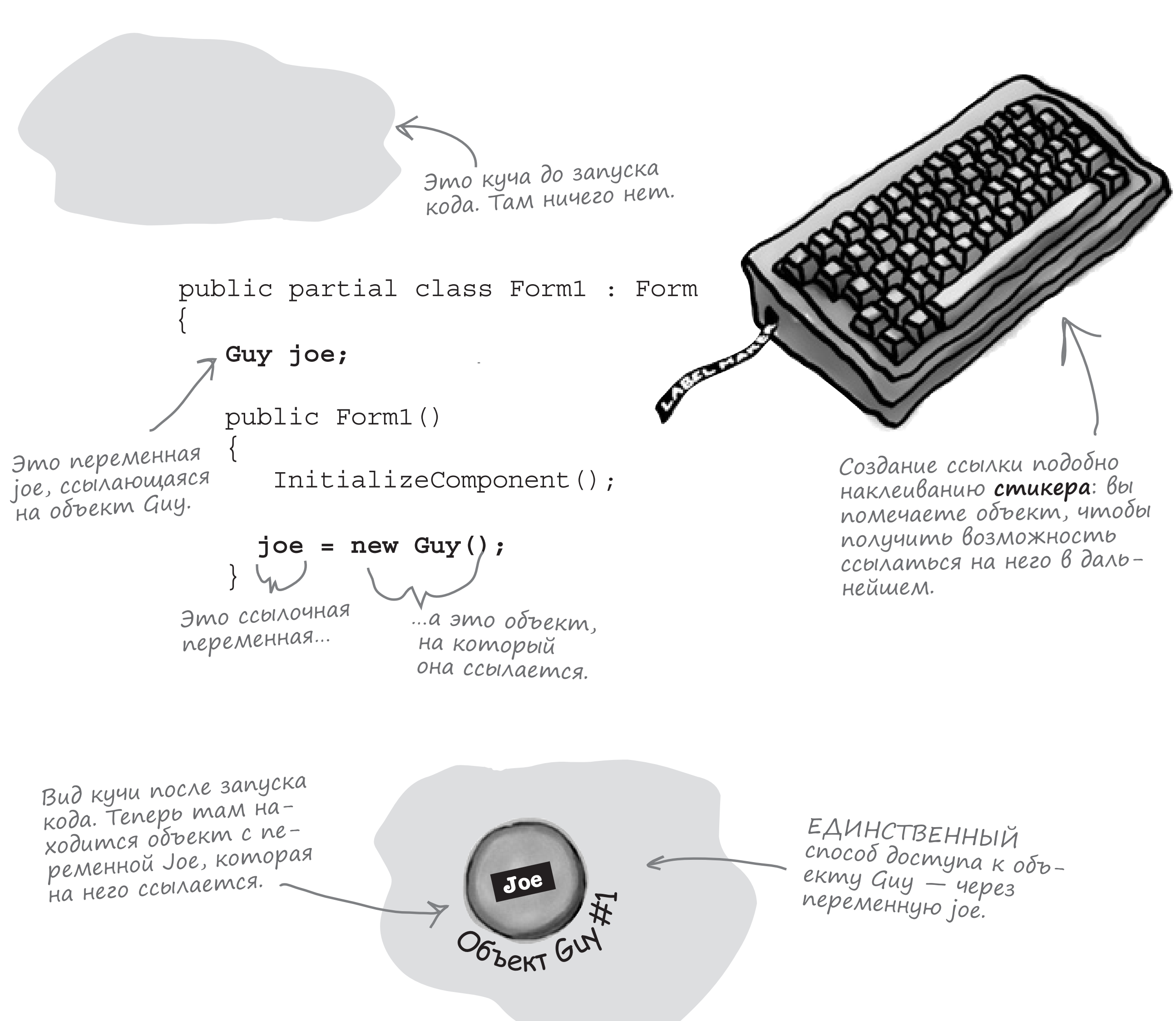
Если программе нужны целые большие числа, используйте тип `long`. Для целых малых чисел используйте тип `short`. Если вам нужны значения Да/Нет, используйте тип `boolean`. А если вам нужно нечто лающее, используйте тип `Dog`. Во всех случаях вы работаете с переменными.

Переменные ссылочного типа

Для создания нового объекта вы пишете `new Guy()`. Но этого недостаточно; в куче появится объект **Guy**, но у вас не будет к нему *доступа*. ***Вам требуется ссылка на объект.*** Здесь на помощь приходят **переменные ссылочного типа** — это переменные типа **Guy**, имеющие имя, например **joe**. В итоге получается, что **joe** — это ссылка на только что созданный объект **Guy**. Именно она используется для доступа к объекту.

Это называется **созданием экземпляра** объекта.

Все переменные типа object являются ссылочными. Давайте рассмотрим пример:




```
public partial class Form1 : Form
{
    Guy joe;

    public Form1()
    {
        InitializeComponent();

        joe = new Guy();
    }
```

Ссылки подобны маркерам

Подозреваем, что на вашей кухне есть отдельные емкости для сахара и соли. Если поменять местами наклейки на них, еда будет несъедобной, ведь содержимое емкостей при этом местами не поменялось. ***Аналогично действуют ссылки.*** Ярлыки можно поменять местами и заставить указывать на разные вещи, но какие именно данные и методы будут доступны, определяет только сам **объект**, а не ссылка на него.

Для работы с хранящимися в памяти объектами используются ссылки, которые являются переменными. Тип этих переменных определяется классом объекта, на который они ссылаются.

Метод button1_Click формы Form1 содержит переменную «Джо», указывающую на этот объект.

Это объект типа Guy. Он ОДИН, но на него МНОГО ссылок.

Джо

программист

клиент

брат

`Guy dad = joe;`

эйты!

папа

дядяДжо

Переменная «папа» является ссылкой на экземпляр класса Guy.

Это ссылочные переменные, указывающие на ОДИН И ТОТ ЖЕ объект.

Различные методы могут использовать экземпляр Guy в различных целях. Поэтому ссылочным переменным разумно присвоить разные имена в зависимости от контекста.

Обращение к объекту никогда не проходит напрямую. К примеру, невозможно записать `Guy.GiveCash()`, если `Guy` принадлежит типу object. Компилятор не понимает, куда именно вы обращаетесь, так как в куче может храниться несколько экемпляров `Guy`. Вам нужна ссылочная переменная, например `joe`, которой будет присвоен конкретный экземпляр `Guy joe = new Guy()`.

Теперь можно вызывать не статические методы `joe.GiveCash()`, ведь компилятор «знает», к какому экземпляру обратиться. Как показано на рисунке, возможен ***набор ссылок на один экземпляр***. Можно написать `Guy dad = joe` и вызвать метод `dad.GiveCash()` (папа.ДайДенег()). Именно этим сын Джо занимается каждый день!

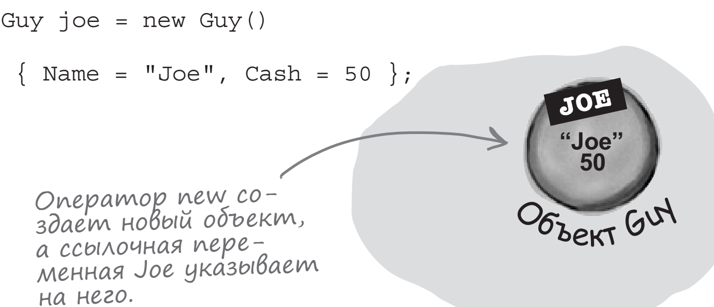
При отсутствии ссылок объект превращается в мусор

Если все ссылки на объект исчезают, программа теряет к нему доступ. Теперь объект предназначен для **сборки мусора (garbage collection)**. C# избавляется от всех объектов, на которые нет ссылок, освобождая место в памяти.

Объект остается в куче, пока на него есть хотя бы одна ссылка. Как только последняя ссылка исчезает, объект удаляется.

1 Вот код создания объекта.

```
Guy joe = new Guy()
 { Name = "Joe", Cash = 50 };
```

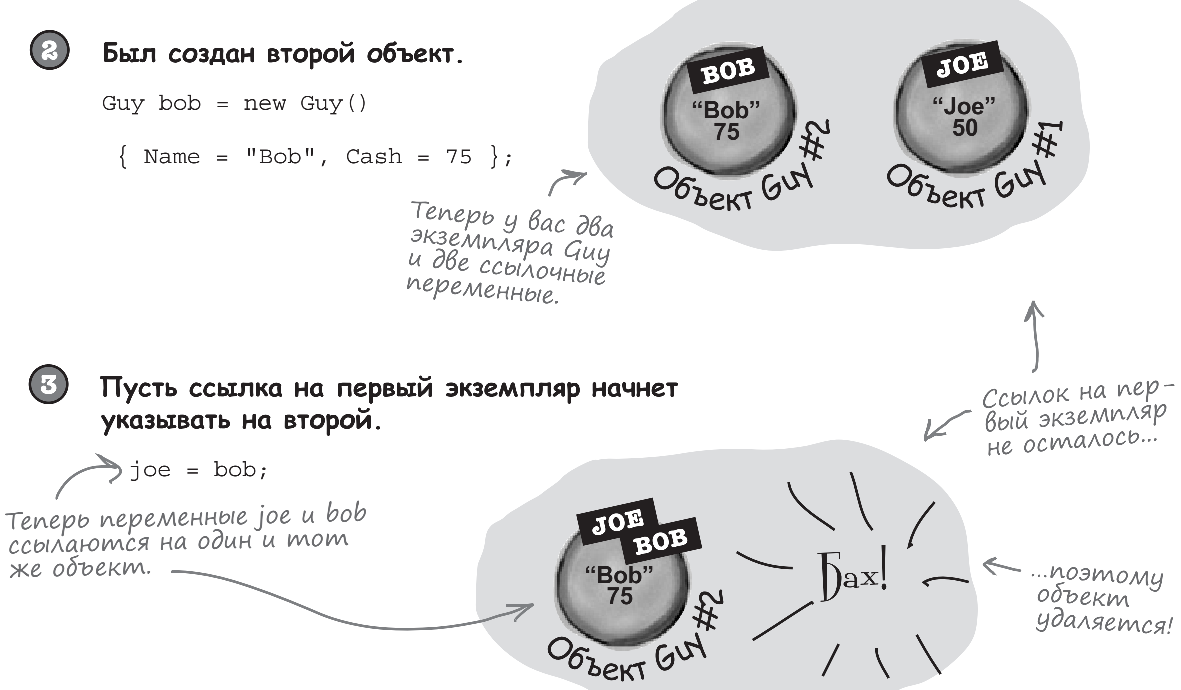


2 Был создан второй объект.

```
Guy bob = new Guy()
 { Name = "Bob", Cash = 75 };
```

3 Пусть ссылка на первый экземпляр начнет указывать на второй.

```
joe = bob;
```

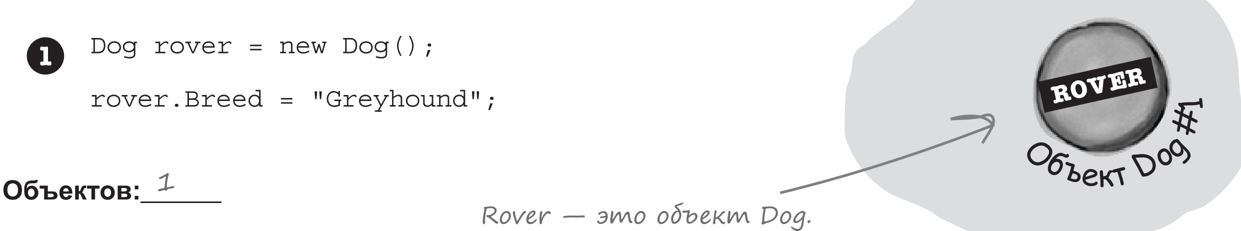


Побочные эффекты множественных ссылок

Обращаться с ссылочными переменными нужно аккуратно. В большинстве случаев кажется, что вы просто перенаправляете ссылку на другой объект. Но при этом можно нечаянно удалить все ссылки на другой объект. И далеко не всегда это будет тот результат, который вам нужен. Рассмотрим пример:

1

```
Dog rover = new Dog();
rover.Breed = "Greyhound";
```



Объектов: 1

Ссылок: 1

2

```
Dog fido = new Dog();
fido.Breed = "Beagle";
Dog spot = rover;
```

ROVER
SPOT
Объект Dog #1
FIDO
Объект Dog #2

Fido — это еще один объект Dog. При этом Spot — это всего лишь дополнительная ссылка на первый объект.

Объектов: 2

Ссылок: 3

3

```
Dog lucky = new Dog();
lucky.Breed = "Dachshund";
fido = rover;
```

Объектов: 2

Ссылок: 4

Теперь ваша очередь. Вот длинный код, разбитый на блоки. Укажите, сколько объектов и сколько ссылок на них имеется на каждой стадии. Справа нарисуйте объекты в куче и их метки.

1.

```
Dog rover = new Dog();
rover.Breed = "Greyhound";
Dog rinTinTin = new Dog();
Dog fido = new Dog();
Dog quentin = fido;
```

Объектов:______

Ссылок:_____

2.

```
Dog spot = new Dog();
spot.Breed = "Dachshund";
spot = rover;
```

Объектов:______

Ссылок:_____

3.

```
Dog lucky = new Dog();
lucky.Breed = "Beagle";
Dog charlie = fido;
fido = rover;
```

Объектов:______

Ссылок:_____

4.

```
rinTinTin = lucky;
Dog laverne = new Dog();
laverne.Breed = "pug";
```

Объектов:______

Ссылок:_____

5.

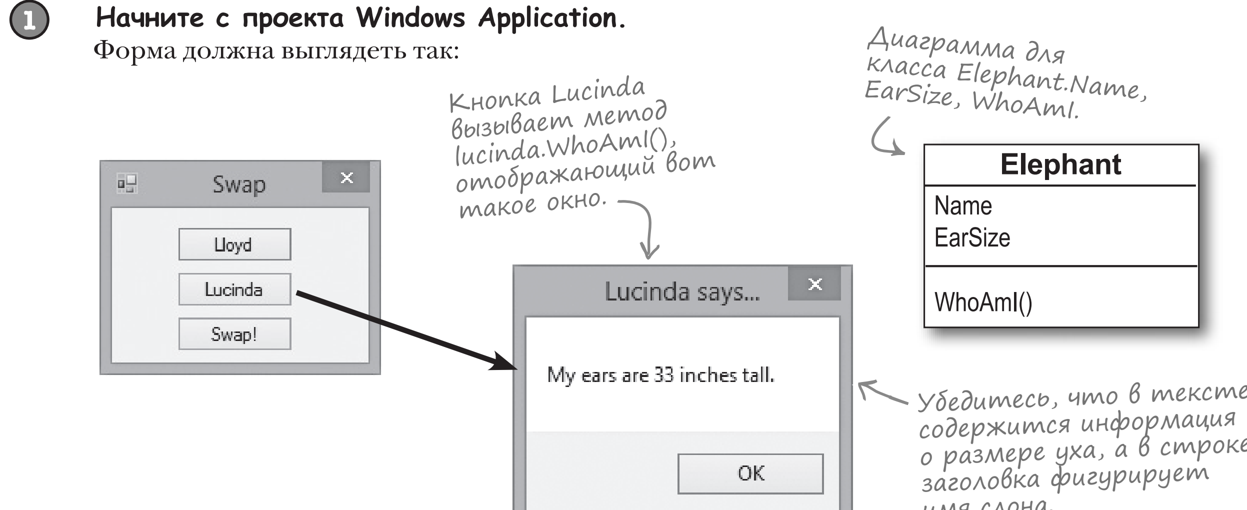
```
charlie = laverne;
lucky = rinTinTin;
```

Объектов:______

Ссылок:_____

Напишите программу для класса `elephant` (слон). Получите два экземпляра `elephant` и поменяйте указывающие на них ссылки таким образом, чтобы ***ни один из экземпляров*** не был уничтожен.

1 **Начните с проекта Windows Application.**
Форма должна выглядеть так:



2 **Класс Elephant**
Согласно диаграмме классов `Elephant` вам потребуется поле типа `int` с названием `EarSize` и поле типа `String` с названием `Name`. (Убедитесь в наличии модификатора public.) Добавьте метод `WhoAmI()`, вызывающий окно с информацией о размере уха и имени слона.

3 **Еще два экземпляра Elephant и ссылки на них**
Добавьте к классу `Form1` два поля `Elephant` (сразу под объявлением класса) с именами `Lloyd` и `Lucinda`. Присвойте им следующие начальные значения:

```
lucinda = new Elephant() { Name = "Lucinda", EarSize = 33 };
lloyd = new Elephant() { Name = "Lloyd", EarSize = 40 };
```

4 **Кнопки Lloyd и Lucinda**
Кнопка Lloyd должна вызывать метод `lloyd.WhoAmI()`, а кнопка Lucinda – метод `lucinda.WhoAmI()`.

5 **Кнопка переключения**
Это самая сложная часть. Кнопка Swap должна ***поменять местами*** две ссылки. То есть щелчок на ней заставляет переменные `Lloyd` и `Lucinda` ссылаться на другие объекты и вызвает окно с сообщением «Objects swapped» (Замена объектов). После щелчка на кнопке Swap щелчок на кнопке Lloyd должен вызывать окно Lucinda, и наоборот. Повторный щелчок на кнопке Swap должен возвращать все обратно.

Как только на объект не остается ни одной ссылки, он удаляется. Что же делать? Представьте, что вы решили перелить пиво в стакан, который в данный момент наполнен водой. Чтобы сделать это, вам потребуется еще один пустой стакан...

Возьми в руку карандаш
Решение

Вот как выглядит куча с объектами и ссылками на них.

1

```
Dog rover = new Dog();
rover.Breed = "Greyhound";
Dog rinTinTin = new Dog();
Dog fido = new Dog();
Dog quentin = fido;
```

Объектов: 3

Ссылок: 4

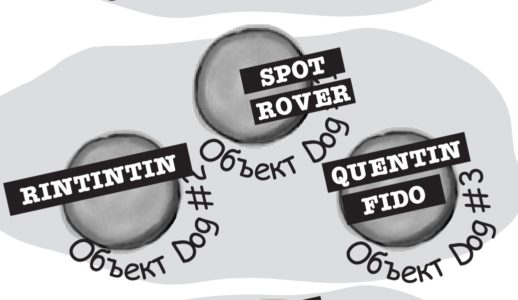
Создан новый объект Dog со ссылкой Spot. В результате операции Spot = Rover объект исчезает.

2

```
Dog spot = new Dog();
spot.Breed = "Dachshund";
spot = rover;
```

Объектов: 3

Ссылок: 5



Создан новый объект Dog, но после Fido=Rover предыдущий объект, на который ссылался Fido, исчезает.

3

```
Dog lucky = new Dog();
lucky.Breed = "Beagle";
Dog charlie = fido;
fido = rover;
```

Charlie, как и Fido, стал указывать на object #3. Затем Fido начал ссылаться на object #1.

Объектов: 4

Ссылок: 7

Исчезла последняя ссылка на объект Dog #2.

4

```
rinTinTin = lucky;
Dog laverne = new Dog();
laverne.Breed = "pug";
```

Объектов: 4

Ссылок: 8

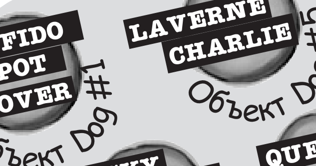
Ссылка Rin Tin Tin теперь указывает на объект Lucky, поэтому предыдущий объект был удален.

5

```
charlie = laverne;
lucky = rinTinTin;
```

Объектов: 4

Ссылок: 8



Происходит перенаправление ссылок, но новые объекты не создаются. Последний оператор не делает ничего, так как обе ссылки и так указывали на один и тот же объект.

Посмотрим, как же поменять между собой ссылки на два экземпляра, чтобы ни один из них при этом не оказался удаленным.

Это определение класса Elephant. Не забудьте строчку «using System.Windows.Forms;», иначе оператор MessageBox не будет работать.

```
using System.Windows.Forms;

class Elephant {

  public int EarSize;
  public string Name;

  public void WhoAmI() {
     MessageBox.Show("My ears are " + EarSize + " inches tall.",
        Name + " says…");
  }
}
```

Операторы using **можно поместить в фигурные скобки внутри пространства имен.**

Это код класса Form1 из файла Form1.cs.

```
public partial class Form1 : Form {

    Elephant lucinda;
    Elephant lloyd;

    public Form1()
    {
        InitializeComponent();
        lucinda = new Elephant()
             { Name = "Lucinda", EarSize = 33 };
        lloyd = new Elephant()
             { Name = "Lloyd", EarSize = 40 };
    }

    private void button1_Click(object sender, EventArgs e) {
        lloyd.WhoAmI();
    }

    private void button2_Click(object sender, EventArgs e) {
        lucinda.WhoAmI();
    }

    private void button3_Click(object sender, EventArgs e) {
        Elephant holder;
        holder = lloyd;
        lloyd = lucinda;
        lucinda = holder;
        MessageBox.Show("Objects swapped");
    }
}
```

Ссылка Holder нужна для того, чтобы объект Lloyd не исчез, после того как ссылка будет перенаправлена на объект Lucinda.

Оператор new не используется, так как нам не нужен еще один экземпляр Elephant.

Типы данных string и array отличаются от всех остальных отсутствием фиксированной длины.

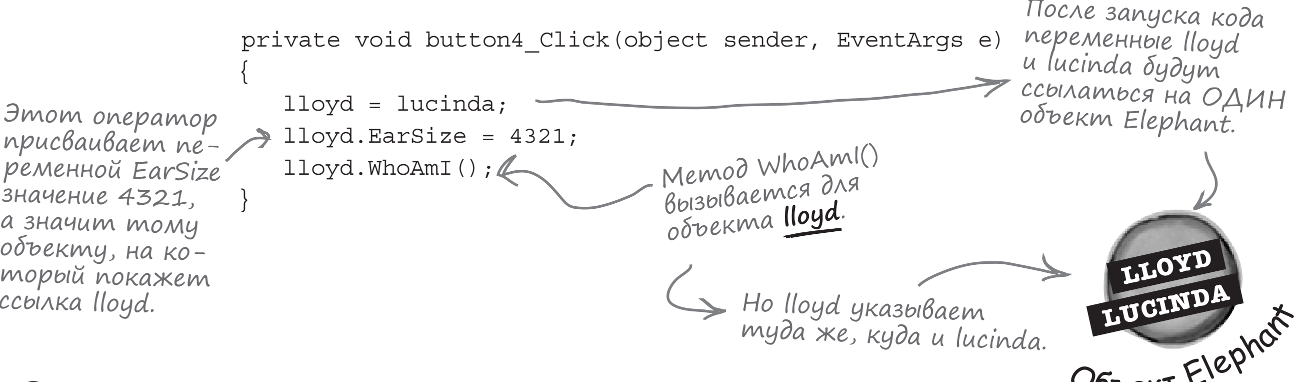
Как вы считаете, почему к классу Elephant не был добавлен метод Swap()?

Две ссылки — это ДВА способа редактировать данные объекта

Наличие множественных ссылок на объект создает опасность непреднамеренного редактирования. Другими словами, одна ссылка может ***внести изменения*** в объект, в то время как другая рассчитана на использование ***старой версии*** этого объекта. Посмотрите сами:

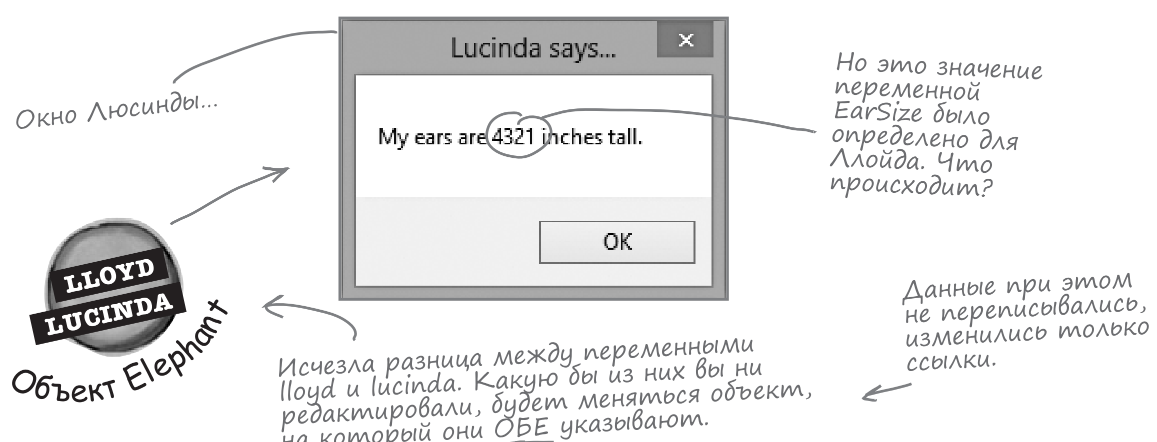
1. Добавим форме еще одну кнопку.

2. Добавим к кнопке код. А теперь подумайте, что произойдет после щелчка на этой кнопке?




```
private void button4_Click(object sender, EventArgs e)
{
   lloyd = lucinda;
   lloyd.EarSize = 4321;
   lloyd.WhoAmI();
}
```

3. Щелкните на кнопке, чтобы проверить свои догадки. Смотрите, это окно Lucinda. При том что вы вызывали метод `WhoAmI()` для Ллойда.



Особый случай: массивы

Строки и массивы отличаются от знакомых вам типов данных тем, что не имеют заранее заданного размера (поразмыслите над этим).

Для слежения за набором данных одного типа, например списка высот или группы собак, используются **массивы (array)**. Этот термин обозначает **группу переменных**, которая обрабатывается как единый объект. Вы получаете возможность хранить и редактировать большие наборы данных. Массивы объявляются аналогично другим переменным. Нужно указать имя массива и его тип:

Объявление переменной myArray можно совместить с присвоением начальных значений:

bool[] myArray = new bool[15];

Для объявления массива нужно указать его тип, поставив следом квадратные скобки.

```
bool[] myArray;
myArray = new bool[15];
myArray[4] = true;
```

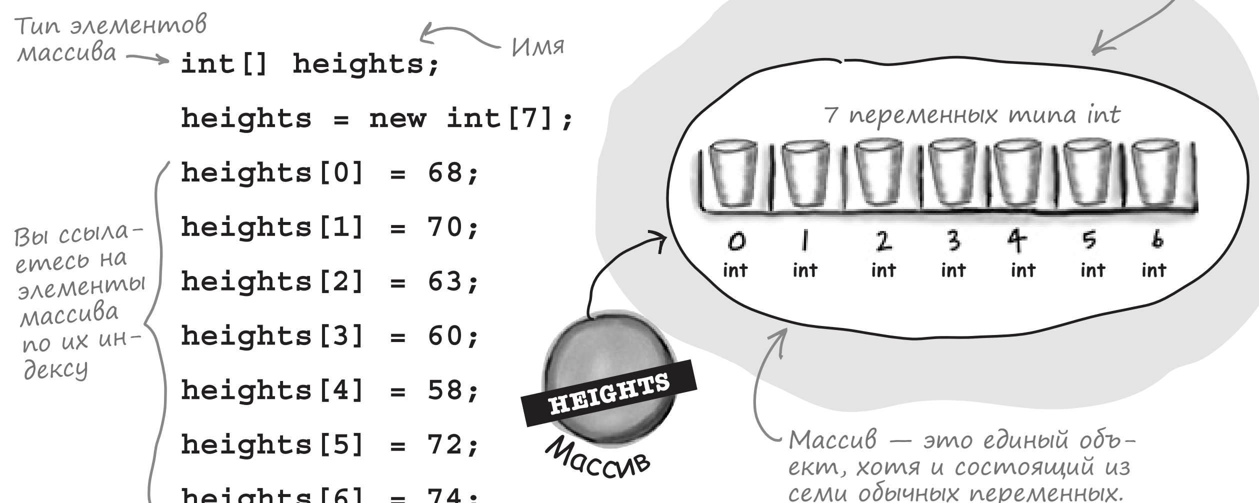
Этот массив состоит из 15 элементов.

Новый массив создается командой new, как и любой другой объект.

Нумерация элементов массива начинается с 0. Эта строка присваивает пятому элементу массива значение true.

Каждый элемент массива — это всего лишь переменная

Прежде всего вы должны **объявить ссылочную переменную** для массива. Затем при помощи оператора new **создается объект** с указанием размера. Все готово для **помещения элементов** в массив. Вот пример кода создания массива и вид кучи. Первый элемент массива всегда имеет нулевой **индекс**.



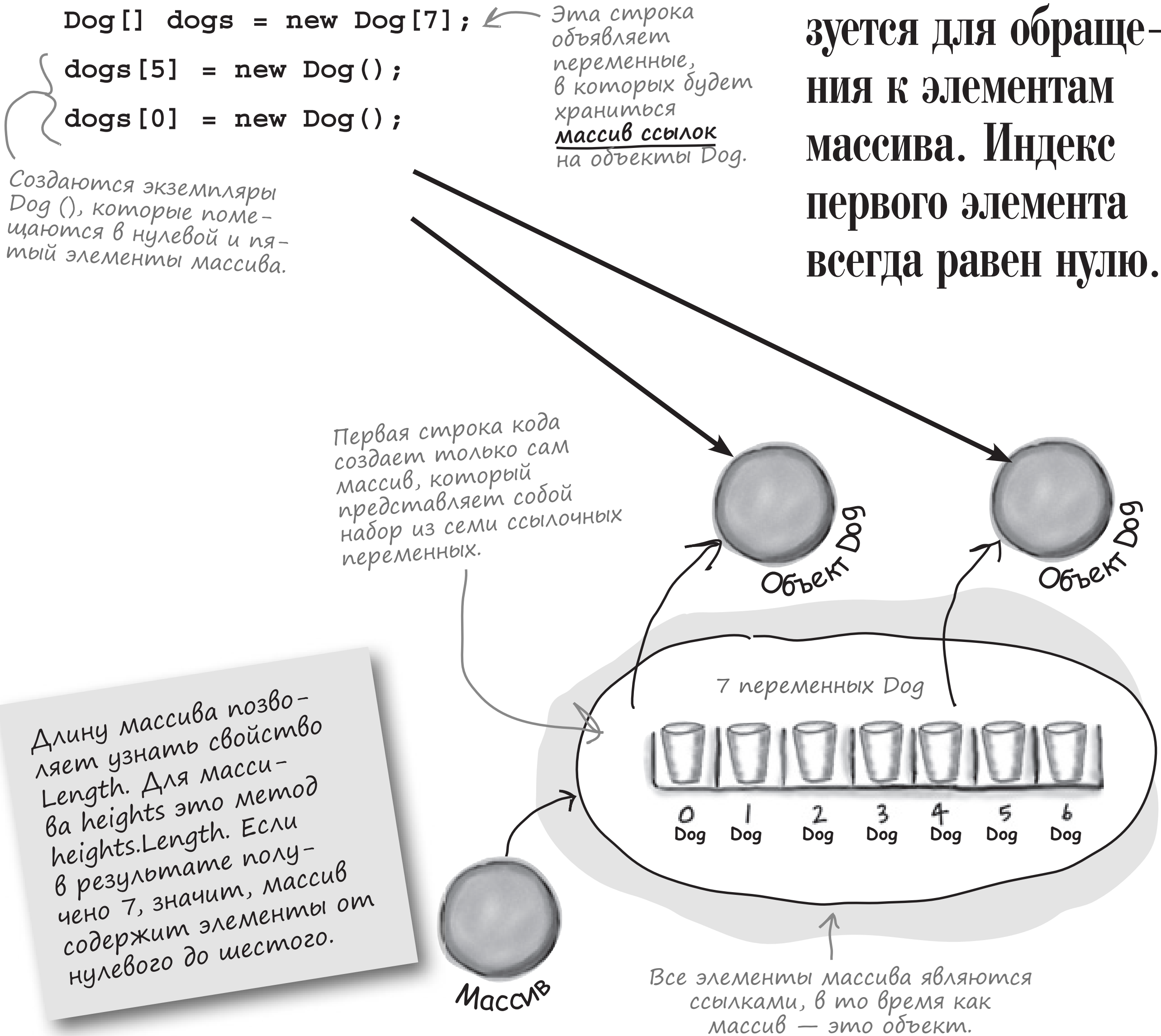
Массив может состоять из ссылочных переменных

Массив может содержать не только числа или строки, но и ссылки на объекты. Словом, только от вас зависит, переменные какого типа вы хотите таким образом организовать. Можно создать как массив целых чисел, так и массив объектов типа Duck.

Рассмотрим код создания массива из 7 переменных типа Dog. Строка инициализации создает только ссылочные переменные. За ней следуют две строки new Dog(), то есть создаются два экземпляра класса Dog.

Число в квадратных скобках называется индексом (index) и используется для обращения к элементам массива. Индекс первого элемента всегда равен нулю.

```
Dog[] dogs = new Dog[7];
dogs[5] = new Dog();
dogs[0] = new Dog();
```



Добро пожаловать на распродажу сэндвичей от Джо!

В заведении Джо множество видов мяса, хлеб на любой вкус и больше приправ, чем вы можете себе представить. Нет только меню! Сможете создать для него программу, которая будет генерировать меню случайным образом?

MenuMaker

Randomizer
Meats
Condiments
Breads

GetMenuItem()

Новый проект, содержащий класс MenuMaker.
Для меню вам потребуются ингредиенты. Информацию о них лучше всего хранить в виде массивов. Ингредиенты должны смешиваться случайным образом, образуя сэндвич. В этом вам поможет встроенный класс `Random`, генерирующий случайные числа. Итак, наш класс будет иметь четыре поля: `Randomizer` для хранения ссылок на объект `Random` и три массива типа `string` для информации о мясе, приправах и хлебе.

```
class MenuMaker {
    public Random Randomizer;

    string[] Meats = { "Roast beef", "Salami", "Turkey", "Ham", "Pastrami" };
    string[] Condiments = { "yellow mustard", "brown mustard",
                  "honey mustard", "mayo", "relish", "french dressing" };
    string[] Breads = { "rye", "white", "wheat", "pumpernickel",
                  "italian bread", "a roll" };
}
```

*Поле Randomizer содержит **ссылку** на объект Random. Метод Next() генерирует случайные числа.*

Из элементов этих трех массивов будет случайным образом создаваться сэндвич.

Для доступа к элементу укажите его номер в скобках. Например, элемент Breads[2] имеет значение white.

Это инициализатор коллекций, с которым вы подробно познакомитесь в главе 8.

Метод GetMenuItem().
Наш класс должен создавать сэндвичи, поэтому добавим к нему соответствующий метод. Он будет использовать метод `Next()` объекта `Random` для выбора элемента из каждого массива. При передаче методу `Next()` целочисленного параметра он возвращает случайное число, меньшее, чем значение этого параметра. То есть результатом работы метода `Randomizer.Next(7)` будет случайное число от 0 до 6.

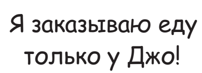
Но как узнать, какой параметр нужен методу `Next()`? Он легко вычисляется при помощи свойства массивов `Length`. Именно так вы получите случайный номер элемента массива.

```
public string GetMenuItem() {
    string randomMeat = Meats[Randomizer.Next(Meats.Length)];
    string randomCondiment = Condiments[Randomizer.Next(Condiments.Length)];
    string randomBread = Breads[Randomizer.Next(Breads.Length)];
    return randomMeat + " with " + randomCondiment + " on " + randomBread;
}
```

Метод GetMenuItem() возвращает строку с названием сэндвича.

Случайный элемент массива Meats попадает в метод randomMeat путем передачи параметра Meats.Length методу Next() объекта Random. Массив Meats состоит из 5 элементов, значит, Meats.Length равно 5, и метод Next(5) возвратит случайное число от 0 до 4.

Как это работает...



Meats.Length возвращает число элементов массива Meats. Так что **randomizer.Next(Meats.Length)** *дает случайное число, большее или равное нулю, но меньшее, чем количество элементов массива Meats.*

```
Meats[Randomizer.Next(Meats.Length)]
```

Meats — это массив строк, состоящий из пяти элементов. Meats[0] = "Roast Beef" (Ростбиф), а Meats[3] = "Ham" (Ветчина).

3 **Построение формы**

Добавьте к форме шесть меток и задайте свойство Text. Для этого воспользуйтесь объектом MenuMaker. Объекту потребуется присвоить начальные значения, используя новый экземпляр класса Random.

```
public Form1() {
    InitializeComponent();

    MenuMaker menu = new MenuMaker() { Randomizer = new Random() };

    label1.Text = menu.GetMenuItem();
    label2.Text = menu.GetMenuItem();
    label3.Text = menu.GetMenuItem();
    label4.Text = menu.GetMenuItem();
    label5.Text = menu.GetMenuItem();
    label6.Text = menu.GetMenuItem();
}
```

Присвойте полю Randomizer объекта MenuMaker новый экземпляр класса Random.

А что будет, если не указать начальные значения поля Randomizer? И как этого можно избежать?

Все готово для запуска метода GetMenuItem() и получения случайного меню из шести пунктов.

После запуска программы шесть меток покажут шесть случайных рецептов сэндвичей.

Sloppy Joe's Menu

Salami with honey mustard on white
Pastrami with french dressing on pumpernickel
Turkey with honey mustard on italian bread
Ham with yellow mustard on pumpernickel
Turkey with honey mustard on pumpernickel
Turkey with yellow mustard on wheat

Ссылки позволяют объектам обращаться друг к другу

Elephant
Name EarSize
WhoAmI() TellMe() SpeakTo()

Вы уже видели, как формы обращаются к объектам при помощи ссылочных переменных, которые вызывают методы и проверяют значения полей. Для объектов определены те же операции, что и для форм, потому что **форма — это объект**. При обращении объектов друг к другу используется ключевое слово `this`. С его помощью делается ссылка на текущий экземпляр класса.

Метод, заставляющий слона говорить.
Добавим к классу `Elephant` метод, первым параметром которого будет сообщение от объекта `elephant`. Вторым параметром будет имя слона:

```
public void TellMe(string message, Elephant whoSaidIt) {
    MessageBox.Show(whoSaidIt.Name + " says: " + message);
}
```

Можно добавить метод `button4_Click()`, но сделать это нужно **до оператора, перезагружающего ссылки** (`lloyd = lucinda;`)!

```
lloyd.TellMe("Hi", lucinda);
```

Мы вызвали метод `TellMe()` (Скажи мне) для Ллойда и передали ему два параметра: строку Hi и ссылку на объект Lucinda. Параметр `whoSaidIt` (Кто это сказал) используется для доступа к параметру `Name` любого слона, имя которого будет передано методу `TellMe()` вторым параметром.

Вызов одного метода другим.
Добавим метод `SpeakTo()` к классу `Elephant`. Именно здесь мы используем ключевое слово **`this`**. Оно является ссылкой, **позволяющей объекту рассказать о себе**.

```
public void SpeakTo(Elephant whoToTalkTo, string message) {
    whoToTalkTo.TellMe(message, this);
}
```

Этот метод класса Elephant вызывает метод TalkTo() (Поговори с), позволяющий одному слону разговаривать с другим.

Посмотрим, как это работает.

```
lloyd.SpeakTo(lucinda, "Hello");
```

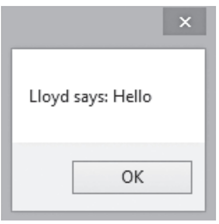
При вызове метода `SpeakTo()` для Ллойда вы используете параметр `talkTo` (являющийся ссылкой на Люсинду), чтобы вызвать метод `TellMe()` для Люсинды.

```
whoToTalkTo.TellMe(message, this);
```

Ллойд использует whoToTalkTo (со ссылкой на Люсинду) для вызова TellMe()

ключевое слово this заменяется ссылкой на объект Ллойд

```
lucinda.TellMe(message, [ссылка на Ллойда]);
```



В результате появляется окно диалога с обращением к Люсинде:

Сюда объекты еще не отправлялись

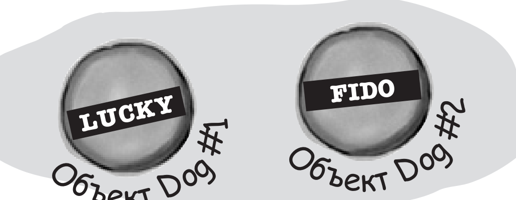
С объектами связано еще одно важное ключевое слово. Только что созданная ссылка, которая пока никуда не указывает, имеет по умолчанию значение `null`.

```
Dog fido;

Dog lucky = new Dog();
```

Пока существует один объект, ссылка fido имеет значение null.

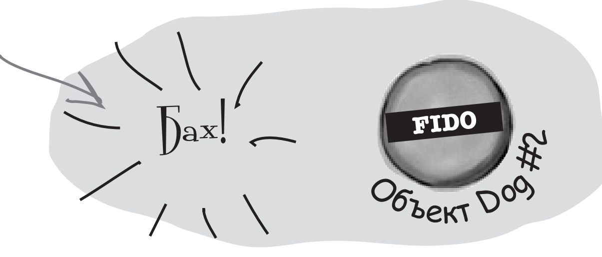
Когда ссылка fido указывает на объект, она больше не равна null.

```
fido = new Dog();
```



Если присвоить lucky значение null, объект будет уничтожен, так как на него не останется ссылок.

```
lucky = null;
```



Часто задаваемые вопросы

В: Форма — это только объект?

О: Да! Именно поэтому код класса начинается с объявления этого класса. Откройте код формы и убедитесь сами. Затем откройте файл `Program.cs` для любой из написанных программ и посмотрите на метод `Main()`, вы найдете там строку `new Form1()`.

В: Зачем мне может потребоваться ключевое слово `null`?

О: Вот для такой проверки условия:

```
if (lloyd == null) {
```

Оно имеет значение `true`, если ссылка `lloyd` указывает на `null`.

Кроме того, ключевое слово `null` ***позволяет*** быстро избавиться от ставшего ненужным объекта. Если ссылки на объект остаются, а вам он уже не нужен, достаточно воспользоваться ключевым словом `null`, и объект будет уничтожен.

В: Что на самом деле происходит с объектами после попадания в мусор?

О: Помните, в начале второй главы мы говорили об **Общеязыковой исполняющей среде (CLR)**? Это виртуальная машина, управляющая всеми программами .NET. *Виртуальной машиной* называется способ изоляции работающих программ от остальной оперативной системы. Виртуальная машина управляет используемой памятью. Она следит за всеми объектами, и как только последняя ссылка на какой-нибудь из объектов исчезает, она освобождает оперативную память, которая была выделена под этот объект.

Часто Задаваемые Вопросы

В: Я до сих пор не очень понимаю, как работают ссылки.

О: Ссылки — это способ доступа к методам и полям. Создав ссылку на объект `Dog`, вы получаете возможность пользоваться любыми методами, определенными для этого объекта. Для нестатических методов `Dog.Bark()` и `Dog.Beg()` можно определить ссылку `spot`. После чего для доступа к ним достаточно написать `spot.Bark()` и `spot.Beg()`. Ссылки позволяют редактировать поля объекта. Например, для внесения изменений в поле `Breed` достаточно написать `spot.Breed`.

В: Получается, что, редактируя объект посредством одной ссылки, я меняю его значение для остальных ссылок?

О: Да. Если `rover` ссылается на тот же объект, что и spot, заменив `rover.Breed` на beagle, вы получите значение beagle и для `spot.Breed`.

В: Я не понимаю, почему переменные разных типов имеют разный размер. Зачем это нужно?

О: Переменные определяют размер присваиваемого значения. Если вы объявите переменную типа `long` и присвоите ей значение (например, 5), CLR все равно выделит памяти на большое значение. В конце концов, на то они и переменные, чтобы все время меняться.

CLR предполагает, что вы выбираете тип осознанно и не будете объявлять переменную ненужного типа. Сейчас переменная хранит небольшое значение, но потом оно может стать большим. Поэтому для нее заранее выделен нужный объем памяти.

В: Напомните мне еще раз, какую функцию выполняет ключевое слово `this`?

О: `this` — это специальная переменная, используемая только внутри объекта. Она ссылается на поля и методы выбранного экземпляра. Это полезно при работе с классом, методы которого обращаются к другим классам. Объект может передать **ссылку на себя** другому объекту. Если `Spot` вызывает один из методов `Rover` при помощи параметра `this`, объект `Rover` получает ссылку на объект `Spot`.

Экземпляры объектов используют ключевое слово this для ссылки на самих себя.

КЛЮЧЕВЫЕ МОМЕНТЫ

При рассмотрении модификатора var в главе 14 вы узнаете, в каком случае тип переменной не объявляется заранее.

- Объявляя переменную, вы ВСЕГДА указываете ее тип. Иногда при этом задаются и начальные значения.
- Существует множество **значимых типов** для хранения значений разного размера. Для огромных чисел используйте тип `long`, а для самых маленьких (до 255) — `bytes`.
- Невозможно присвоить значение большего типа переменной, принадлежащей к меньшему типу.
- При работе с константами используйте суффикс F для обозначения типа float (15.6F), а суффикс M — для обозначения типа decimal (36.12M).
- Некоторые типы могут преобразовываться друг в друга автоматически (например, `short` в `int`). Для других случаев используйте операцию приведения типов.
- Зарезервированные слова (например, `for`, `while`, `using`, `new`) нельзя использовать в качестве имен переменных.
- Ссылки подобны наклейкам: вы можете иметь множество ссылок на один и тот же объект.
- Если на объект нет ни одной ссылки, он отправляется в мусор, и место, занимаемое им в памяти, освобождается.

Возьми в руку карандаш

Перед вами массив объектов `Elephant` и цикл для поиска слона с самыми большими ушами. Определите значение `biggestEars.Ears` **после** каждой итерации цикла `for`.

```
private void button1_Click(object sender, EventArgs e)
{
  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 };

  Elephant biggestEars = elephants[0];
  for (int i = 1; i < elephants.Length; i++)
  {
    if (elephants[i].EarSize > biggestEars.EarSize)
    {
      biggestEars = elephants[i];
    }
  }
  MessageBox.Show(biggestEars.EarSize.ToString());
}
```

Вы создаете массив из 7 ссылок Elephant().

Первый индекс любого массива равен 0, поэтому первым элементом будет Elephants[0].

Точка останова в этой строке поможет отследить значения elephants[i] всех элементов массива.

Будьте внимательны, этот цикл начинается со **второго элемента** массива (с индексом 1) и выполняется шесть раз, пока i не становится равной длине массива.

Итерация #1 biggestEars.EarSize = ________

Итерация #2 biggestEars.EarSize = ________

Итерация #3 biggestEars.EarSize = ________

Итерация #4 biggestEars.EarSize = ________

Итерация #5 biggestEars.EarSize = ________

Итерация #6 biggestEars.EarSize = ________

Ответ на с. 216.

Магниты с кодом

Магниты с фрагментами кода упали с холодильника. Верните их на место, чтобы получить код, приводящий к появлению показанной ниже формы.

```
int y = 0;
```

```
refNum = index[y];
```

```
islands[0] = "Bermuda";
islands[1] = "Fiji";
islands[2] = "Azores";
islands[3] = "Cozumel";
```

```
int refNum;
while (y < 4) {
```

```
result += islands[refNum];
```

```
MessageBox.Show(result);
```

```
index[0] = 1;
index[1] = 3;
index[2] = 0;
index[3] = 2;
```

```
}
```

```
}
```

```
string[] islands = new string[4];
```

```
result += "\nisland = ";
```

```
int[] index = new int[4];
```

```
y = y + 1;
```

```
private void button1_Click (object sender, EventArgs e)
{
```

```
string result = "";
```

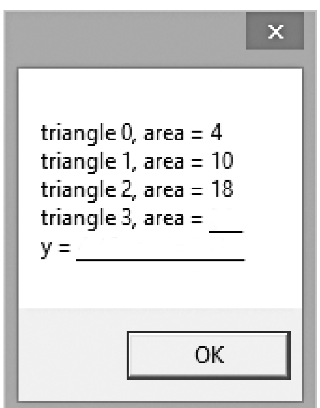
```
island = Fiji
island = Cozumel
island = Bermuda
island = Azores
OK
```

Ответ на с. 217.

Ребус в бассейне

Возьмите фрагменты кода из бассейна и поместите их на пустые строчки. Каждый фрагмент можно использовать несколько раз. В бассейне есть и лишние фрагменты. Нужно получить окно, в котором для каждого треугольника (triangle) будет показана его площадь (area).

Результат:



Дополнительный вопрос!

Воспользуйтесь фрагментами из бассейна, чтобы составить код, заполняющий даже пустые поля в нижней части окна диалога.

```
class Triangle
{
    double area;
    int height;
    int length;
    public static void Main(string[] args)
    {
        string results = "";
        __________
        __________________________
        while ( ________ )
        {
            ________________________
            _____.height = (x + 1) * 2;
            _____.length = x + 4;
            __________________
            results += "triangle " + x + ", area";
            results += " = " + _____.area + "\n";
            ___________
        }
        ___________
        x = 27;
        Triangle t5 = ta[2];
        ta[2].area = 343;
        results += "y = " + y;
        MessageBox.Show(results +
            ", t5 area = " + t5.area);
    }
    void setArea()
    {
        ____________ = (height * length) / 2;
    }
}
```

Это точка входа. Предположим, что она находится в файле с правильными строками using в верхней части.

Подсказка: Метод SetArea() НЕ является статическим.

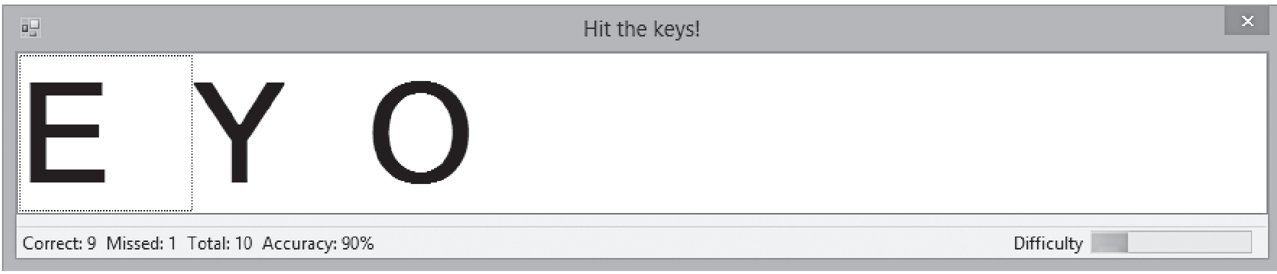
Каждый фрагмент кода можно использовать несколько раз!

area
ta.area
ta.x.area
ta[x].area
x
y
4, t5 area = 18
4, t5 area = 343
27, t5 area = 18
27, t5 area = 343
Triangle [ ] ta = new Triangle(4);
Triangle ta = new [ ] Triangle[4];
Triangle [ ] ta = new Triangle[4];
ta[x] = setArea();
ta.x = setArea();
ta[x].setArea();
int x;
int y;
int x = 0;
int x = 1;
int y = x;
28
30.0
x = x + 1;
x = x + 2;
x = x - 1;
ta.x
ta(x)
ta[x]
x < 4
x < 5
ta = new Triangle();
ta[x] = new Triangle();
ta.x = new Triangle();

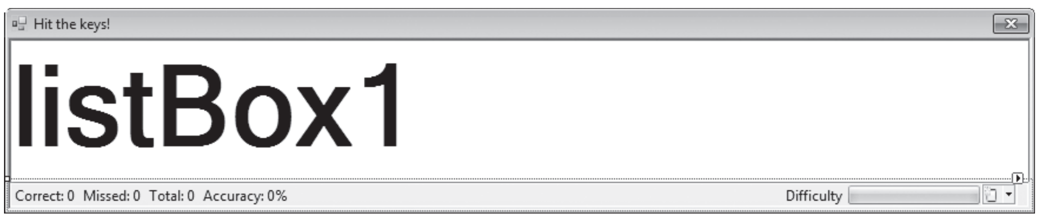
Ответ на с. 218.

Играем в печатную машинку

Вы достигли первых результатов… и знаете достаточно, чтобы создать собственную игру! Пусть в форме случайным образом появляются буквы. Если игрок вводит букву правильно, она исчезает, а уровень повышается. Если же буква введена неправильно, уровень понижается. По мере ввода букв игра усложняется. Как только форма оказывается заполнена буквами, игра заканчивается!



1. **Форма.**
Вот как она должна выглядеть в конструкторе форм.



★ Уберите кнопки управления размерами окна. Для свойства **`FormBorderStyle`** выберите вариант **`Fixed3D`**. Теперь игрок не сможет случайно перетащить форму и изменить ее размер. Установите размер 876 x 174.

★ Перетащите из окна Toolbox на форму элемент управления **`ListBox`**. Присвойте свойству **`Dock`** значение Fill. Свойству **`MultiColumn`** присвойте значение True. Для свойства **`Font`** выберите кегль 72 пункта и полужирное начертание.

★ В окне Toolbox раскройте группу All Windows Forms. Найдите элемент управления **`Timer`** и дважды щелкните на нем, чтобы добавить его к форме.

★ Затем найдите элемент управления **`StatusStrip`** и двойным щелчком добавьте к форме строку состояния. В нижней части конструктора формы должны появиться значки элементов управления **`StatusStrip`** и **`Timer`**:

Вы познакомились с тремя новыми элементами управления!

Задать свойства для ListBox, StatusStrip и Timer вы сможете знакомым способом.

2 Параметры элемента управления StatusStrip

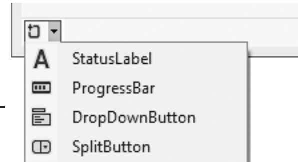
Обратите внимание на первый рисунок данного раздела. В нижней части строки состояния вы увидите метки:

Correct: 18 Missed: 3 Total: 21 Accuracy: 85%

А с другой стороны — метку и индикатор:

Difficulty

Щелкните на стрелке справа от элемента управления `StatusStrip` и выберите в появившемся меню вариант `StatusLabel`:



Добавьте StatusLabel к элементу StatusStrip, выбрав этот вариант в раскрывающемся списке. Затем выполните следующие действия:

★ В окне Properties введите в поле (Name) имя correctLabel, а в поле Text — "Correct: 0". Добавьте еще три таких элемента: missedLabel, totalLabel и accuracyLabel, присвоив их свойству Text значения "Missed: 0", "Total: 0" и "Accuracy: 0%" соответственно.

★ Добавьте еще один элемент StatusLabel. Его свойству Spring присвойте значение True, свойству TextAlign — значение MiddleRight, а свойству Text — значение Difficulty. Наконец, добавьте элемент ProgressBar и назовите его difficultyProgressBar.

★ Присвойте свойству SizingGrip элемента StatusStrip значение False (если вы выделили дочерний элемент StatusLabel или ProgressBar, для возвращения внимания IDE к родительскому элементу StatusStrip нажмите клавишу Escape).

3 Параметры элемента управления Timer.

Вы заметили, что элемент управления Timer на форме отсутствует? Дело в том, что это *невизуальный элемент управления*. Он не меняет вид формы. Его функция — **снова и снова вызывать некий метод**. Присвойте свойству **`Interval`** значение 800, чтобы вызов метода происходил каждые 800 миллисекунд. Затем **дважды щелкните на значке *timer1*** в конструкторе, чтобы добавить к форме метод **`timer1_Tick`**. Вот код для этого метода:

```
private void timer1_Tick(object sender, EventArgs e)
{
    // Добавим случайную клавишу к элементу ListBox
    listBox1.Items.Add((Keys)random.Next(65, 90));
    if (listBox1.Items.Count > 7)
    {
        listBox1.Items.Clear();
        listBox1.Items.Add("Игра окончена!");
        timer1.Stop();
    }
}
```

Скоро вы добавите поле random. Можете ли вы сейчас назвать его тип?

Класс Timer обладает методом Start() , но в этом проекте мы его вызывать не будем. Вместо этого присвоим его свойству Enabled значение True, что приведет к автоматическому пуску.

Класс, отслеживающий статистику игры.
Так как в форме должно отображаться общее количество нажатых, пропущенных и правильно нажатых клавиш, а также точность, значит, мы должны отслеживать все эти данные. Кажется, нам потребуется новый класс! Назовите его `Stats`. Он будет содержать четыре поля типа `int` с именами `Total` (Всего), `Missed` (Пропущено), `Correct` (Правильно) и `Accuracy` (Точность), а также метод `Update` с одним параметром типа `bool`, который получает значение `true`, если нажатая игроком клавиша совпала с буквой, появившейся в окне ListBox, и значение `false`, если игрок ошибся.

Stats
Total Missed Correct Accuracy
Update()

```
class Stats
{
    public int Total = 0;
    public int Missed = 0;
    public int Correct = 0;
    public int Accuracy = 0;

    public void Update(bool correctKey)
    {
        Total++;

        if (!correctKey)
        {
            Missed++;
        }
        else
        {
            Correct++;
        }

        Accuracy = 100 * Correct / (Missed + Correct);
    }
}
```

При каждом вызове метода Update() вычисляется процент правильных попаданий, а результат этих вычислений помещается в поле Accuracy.

Поля для отслеживания объектов Stats и Random.
Вам потребуется экземпляр класса **`Stats`** для хранения информации, поэтому добавьте поле **`stats`**. Как вы уже видели, поле **`random`**, содержащее объект **`Random`**, пока отсутствует.

Напишите в верхней части кода формы:

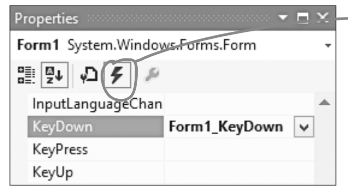
```
public partial class Form1 : Form
{
    Random random = new Random();
    Stats stats = new Stats();
    ...
```

Прежде чем продолжить, присвойте свойству Enabled таймера значение True. Для элемента ProgressBar сделайте свойство Maximum равным 701, а для свойства KeyPreview формы выберите значение True. Попробуйте ответить на вопрос, что будет, если этого не сделать?

Контроль за нажатием клавиш.

Осталось сделать так, чтобы правильно нажатая игроком буква удалялась из окна ListBox), а статистика `StatusStrip` обновлялась.

Вернитесь к конструктору форм и выделите форму. Щелкните на кнопке с изображением молнии в верхней части окна Properties, а затем дважды щелкните на строчке **KeyDown**, чтобы добавить метод `Form1_KeyDown()`, вызываемый при каждом нажатии клавиши. Вот код для этого метода:



Эта кнопка меняет вид окна Properties. Кнопка слева от нее возвращает окно к привычному виду.

Вы получаете список событий (events). О том, что это такое, мы поговорим позднее.

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    // Если пользователь правильно нажимает клавишу, удалите букву из ListBox
    // и увеличьте скорость появления букв
    if (listBox1.Items.Contains(e.KeyCode))
    {
        listBox1.Items.Remove(e.KeyCode);
        listBox1.Refresh();
        if (timer1.Interval > 400)
            timer1.Interval -= 10;
        if (timer1.Interval > 250)
            timer1.Interval -= 7;
        if (timer1.Interval > 100)
            timer1.Interval -= 2;
        difficultyProgressBar.Value = 800 - timer1.Interval;

        // При правильном нажатии клавиши обновляем объект Stats,
        // вызывая метод Update() с аргументом true
        stats.Update(true);
    }
    else
    {
        // При неправильном нажатии клавиши обновляем объект Stats,
        // вызывая метод Update() с аргументом false
        stats.Update(false);
    }

    // Обновление меток элемента StatusStrip
    correctLabel.Text = "Correct: " + stats.Correct;
    missedLabel.Text = "Missed: " + stats.Missed;
    totalLabel.Text = "Total: " + stats.Total;
    accuracyLabel.Text = "Accuracy: " + stats.Accuracy + "%";
}
```

Оператор if проверяет наличие в ListBox нажатой игроком буквы.

Уменьшая числа, вычитаемые из параметра timer1.Interval, вы облегчаете игру, а увеличивая — усложняете ее.

При нажатии клавиши метод Form1_KeyDown() вызывает метод Update() объекта Stats, обновляющий статистику, а затем результат отображается в StatusStrip.

Игра запускается только один раз. Подумайте, как сделать так, чтобы после появления надписи «Game Over» игрок смог начать новую партию?

Запуск игры.

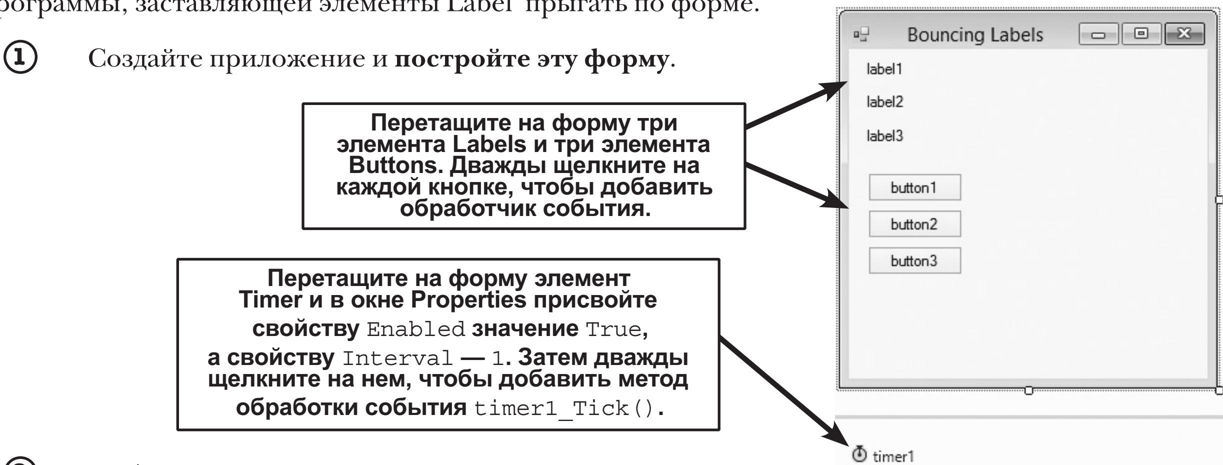
В окно ListBox должно помещаться ровно 7 букв, поэтому может потребоваться поменять размер шрифта. Для изменения уровня сложности отредактируйте значения, вычитаемые из параметра `timer1.Interval` в методе `Form1_KeyDown()`.

Элементы управления — это тоже объекты

Упражнение!

Вы построили множество форм, перетаскивая элементы управления с панели. И вот выясняется, что они являются нашими старыми знакомыми — объектами. А следовательно, на них можно ссылаться и работать с ними так же, как и с собственноручно написанными экземплярами классов. Убедимся в этом на примере программы, заставляющей элементы Label прыгать по форме.

① Создайте приложение и **постройте эту форму**.



② **Добавьте класс** `LabelBouncer`. Вот его код:

```
using System.Windows.Forms;

class LabelBouncer {
    public Label MyLabel;

    public bool GoingForward = true;

    public void Move() {
        if (MyLabel != null) {
            if (GoingForward == true) {
                MyLabel.Left += 5;
                if (MyLabel.Left >= MyLabel.Parent.Width - MyLabel.Width) {
                    GoingForward = false;
                }
            }
            else
            {
                MyLabel.Left -= 5;
                if (MyLabel.Left <= 0) {
                    GoingForward = true;
                }
            }
        }
    }
}
```

Строка "using" нужна потому, что Label находится в этом пространстве имен.

Этому классу принадлежит поле MyLabel с типом Label, значит, в поле находится ссылка на объект Label. Как и любая ссылка, она начинается со значения null. Затем она будет указывать на одну из меток формы.

Значение этой переменной типа Boolean в процессе перемещения метки по форме меняется с true на false и обратно.

Метод Move() определяет, коснулась ли метка правого края формы, используя оператор >= для сравнения Left с шириной формы.

Как вы думаете, зачем вычитать ширину метки из ширины формы?

В процессе перетаскивания элемента управления по форме IDE задает свойства Top и Left. Ваша программа может воспользоваться этими свойствами.

Для перемещения метки по форме нужно создать новый экземпляр класса `LabelBouncer`, сделать ссылку поля `MyLabel` на элемент Label формы и раз за разом вызывать метод `Move()`.

При этом `LabelBouncer` будет двигать метку, меняя свойство `Left`. Если свойство `GoingForward` — `true`, добавляем 5 и сдвигаемся вправо; в противном случае 5 вычитается, и смещение идет влево.

У каждого элемента управления есть свойство `Parent`, содержащее ссылку на форму, **так как форма — это тоже объект**!

③ **Вот код формы**. Попробуйте понять, как он функционирует. Массив объектов `LabelBouncer` перемещает метки взад и вперед, а обработчик события `Tick` элемента Timer снова и снова вызывает методы `Move()`.

Форма хранит массив ссылок `LabelBouncer` в поле `bouncers`. При вызове метода `ToggleBouncing()` она проверяет элемент массива при помощи парамера index. Если элемент имеет значение null, создается новый объект `LabelBouncer`, и его ссылка сохраняется в массив; в противном случае элементу присваивается значение `null`.

```
public partial class Form1 : Form {

    public Form1() {
        InitializeComponent();
    }

    LabelBouncer[] bouncers = new LabelBouncer[3];

    private void ToggleBouncing(int index, Label labelToBounce) {
        if (bouncers[index] == null) {
            bouncers[index] = new LabelBouncer();
            bouncers[index].MyLabel = labelToBounce;
        }
        else {
            bouncers[index] = null;
        }
    }

    private void button1_Click(object sender, EventArgs e) {
        ToggleBouncing(0, label1);
    }

    private void button2_Click(object sender, EventArgs e) {
        ToggleBouncing(1, label2);
    }

    private void button3_Click(object sender, EventArgs e) {
        ToggleBouncing(2, label3);
    }

    private void timer1_Tick(object sender, EventArgs e) {
        for (int i = 0; i < 3; i++) {
            if (bouncers[i] != null) {
                bouncers[i].Move();
            }
        }
    }
}
```

Каждая кнопка вызывает метод ToggleBouncing(), передавая ему индекс массива и ссылку на один из элементов Labels формы.

Вы понимаете, что происходит с обработчиком событий кнопки? __Ваша задача__ понять, каким образом включается перемещение меток взад и вперед.

Так как элементы управления являются всего лишь объектами, ссылки на них можно передавать методам в качестве параметров и сохранять в массивах, полях и переменных.

Элемент Timer использует цикл for для вызова каждого метода Move() объекта LabelBouncer, при условии, что он не равен null. Равенство null останавливает перемещение по форме.

Bouncing Labels

label1 →

← label2

label3 →

button1

button2

button3

Щелчком на button1 начните перемещение метки label1. Повторным щелчком остановите его. Остальные кнопки управляют другими метками.

Метки будут отскакивать от краев формы, даже если растянуть ее вширь или сделать уже.

Еще раз напомним, что в C# существует примерно 77 **зарезервированных слов**. Вам было предложено объяснить назначение некоторых из них. Вот правильные ответы.

namespace	Пространство имен — это логическое понятие, объединяющее связанные друг с другом классы и типы.
for	Это цикл, выполняющий оператор или блок операторов, пока определенное выражение не примет значение false.
class	Класс — это определение объекта. Классы имеют свойства и методы.
public	Ключевое слово, дающее уровень доступа с максимальными правами. Public class может использоваться любым другим классом.
else	Оператор, который выполняется, если проверка условия if вернула значение false.
while	Цикл while выполняется до тех пор, пока условие имеет значение true.
using	Определяет пространства имен, предустановленные и созданные вами классы, которые используются в программе.
if	Оператор, выбирающий другие операторы по результатам проверки условия.
new	Оператор, создающий новый экземпляр объекта.

Возьми в руку карандаш

Решение

Вот как выглядят результаты определения самого большого слоновьего уха на каждой итерации цикла for.

```
private void button1_Click(object sender, EventArgs e)
{
  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 };

  Elephant biggestEars = elephants[0];
  for (int i = 1; i < elephants.Length; i++)
  {
    if (elephants[i].EarSize > biggestEars.EarSize)
    {
      biggestEars = elephants[i];
    }
  }
  MessageBox.Show(biggestEars.EarSize.ToString());
}
```

Помните, что цикл начинается со второго элемента массива? Как вы думаете, почему?

Ссылка biggestEars отслеживает самый большой элемент в цикле.

Поместите сюда точку останова для проверки результата.

Цикл for начинается со второго слона и сравнивает размеры ушей. Если уши следующего слона больше, ссылка biggestEars начинает указывать на него. Таким образом определяется слон с самыми большими ушами.

Итерация #1 biggestEars.EarSize = 40

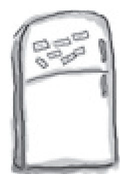
Итерация #2 biggestEars.EarSize = 42

Итерация #3 biggestEars.EarSize = 42

Итерация #4 biggestEars.EarSize = 44

Итерация #5 biggestEars.EarSize = 44

Итерация #6 biggestEars.EarSize = 45



Решение задачи с магнитами

Вот каким образом нужно было расположить на холодильнике магниты с фрагментами кода.

```
private void button1_Click (object sender, EventArgs e)
{
    string result = "";

    int[] index = new int[4];

    index[0] = 1;
    index[1] = 3;
    index[2] = 0;
    index[3] = 2;

    string[] islands = new string[4];

    islands[0] = "Bermuda";
    islands[1] = "Fiji";
    islands[2] = "Azores";
    islands[3] = "Cozumel";

    int y = 0;

    int refNum;
    while (y < 4) {

        refNum = index[y];
        result += "\nisland = ";
        result += islands[refNum];
        y = y + 1;

    }

    MessageBox.Show(result);
}
```

Здесь задаются начальные значения массива index[].

Здесь задаются начальные значения массива islands[].

Итоговая строка составляется при помощи оператора +=.

В цикле while элементы массива index[] используются как индексы элементов массива islands[].

```
island = Fiji
island = Cozumel
island = Bermuda
island = Azores
```

OK

Решение ребуса в бассейне

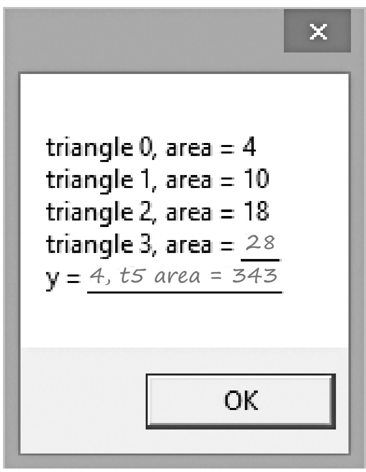
Вы заметили, что класс имеет точку входа и при этом создает экземпляр себя? В C# такое вполне допустимо.

```
class Triangle
{
    double area;
    int height;
    int length;
    public static void Main(string[] args)
    {
        string results = "";
        int x = 0;
        Triangle[] ta = new Triangle[4];
        while ( x < 4 )
        {
            ta[x] = new Triangle();
            ta[x].height = (x + 1) * 2;
            ta[x].length = x + 4;
            ta[x].setArea();
            results += "triangle " + x + ", area";
            results += " = " + ta[x].area + "\n";
            x = x + 1;
        }
        int y = x;
        x = 27;
        Triangle t5 = ta[2];
        ta[2].area = 343;
        results += "y = " + y;
        MessageBox.Show(results +
            ", t5 area = " + t5.area);
    }
    void setArea()
    {
        area = (height * length) / 2;
    }
}
```

После этой строки вы получаете массив из четырех ссылок Triangle, но пока нет ни одного объекта Triangle!

Цикл while создает четыре экземпляра Triangle, четыре раза вызывая оператор new.

Ответ на дополнительный вопрос:



Метод setArea() использует поля height и length для вычисления значения поля area. Так как метод не является статическим, он может вызываться только для экземпляра Triangle.

5 инкапсуляция

Пусть личное остается... личным

Вы когда-нибудь мечтали о том, чтобы вашу личную жизнь оставили в покое? Иногда объекты чувствуют то же самое. Вы же не хотите, чтобы посторонние люди читали ваши записки или рассматривали банковские выписки. Вот и объекты не хотят, чтобы ***другие*** объекты имели доступ к их полям. В этой главе мы поговорим об **инкапсуляции**. Вы научитесь **закрывать объекты** и добавлять методы, **защищающие данные от доступа**.

Кэтлин — профессиональный массовик-затейник

Она организует великолепные званые ужины. Но для нее настали тяжелые времена, так как, по мнению клиентов, Кэтлин недостаточно быстро называет стоимость услуг.

После получения от нового клиента информации о количестве участников, желаемых напитках и оформлении она с помощью специальной диаграммы выполняет довольно сложные вычисления, чтобы определить конечную стоимость. К сожалению, это медленный процесс, и пока Кэтлин считает, клиенты звонят ее конкурентам.

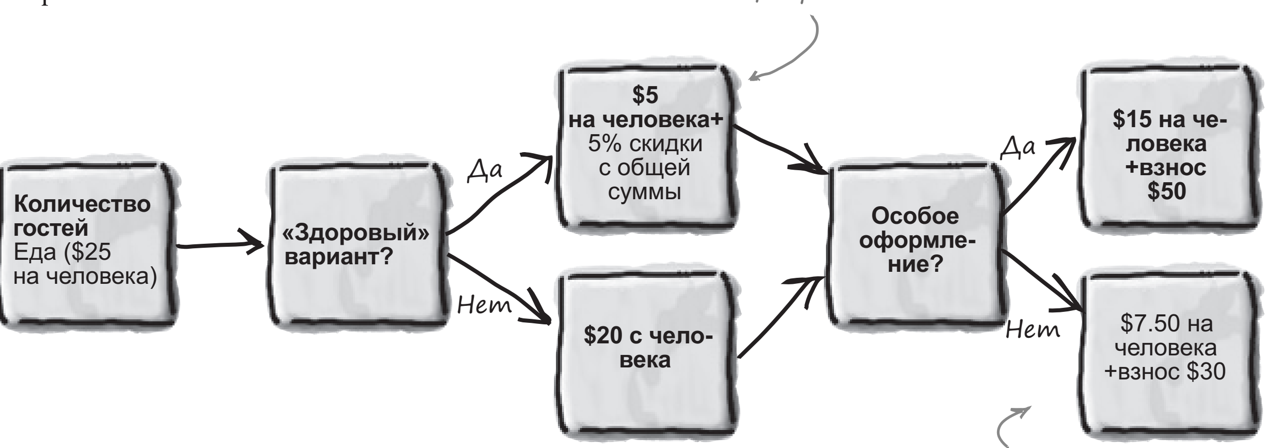
Вы можете написать программу, автоматизирующую расчеты, и спасти бизнес Кэтлин. Только представьте, каким обедом она вас отблагодарит!

Как происходит оценка

Вот фрагмент записей Кэтлин, касающийся расчетов стоимости мероприятия:

> **Оценка стоимости обеда**
>
> - За каждого гостя — $25.
> - Большинство обедов сервируется с алкогольными напитками — это дополнительные $20 на человека. Можно выбрать «здоровый» вариант — это стоит всего $5 на человека, вместо алкоголя подаются соки и газировка. «Здоровый» вариант намного проще в организации, поэтому скидка составит 5% на всю стоимость мероприятия.
> - Обычное оформление стоит $7.50 на человека плюс первоначальный взнос $30. Стоимость особого оформления увеличивается до $15 на человека, а первоначальный взнос составит $50.

Для наглядности представим эти правила в виде диаграммы:



Вы создадите для Кэтлин программу

Перевернув страницу, вы обнаружите упражнение, в процессе выполнения которого вам предстоит создать для Кэтлин планировщик мероприятий. Вот предварительный план работ.

Вы построите форму, в которой Кэтлин будет указывать параметры вечеринки. Достаточно указать количество человек и поставить или снять флажки, касающиеся особого оформления и здорового питания, и в нижней части формы появится стоимость будущего праздника.

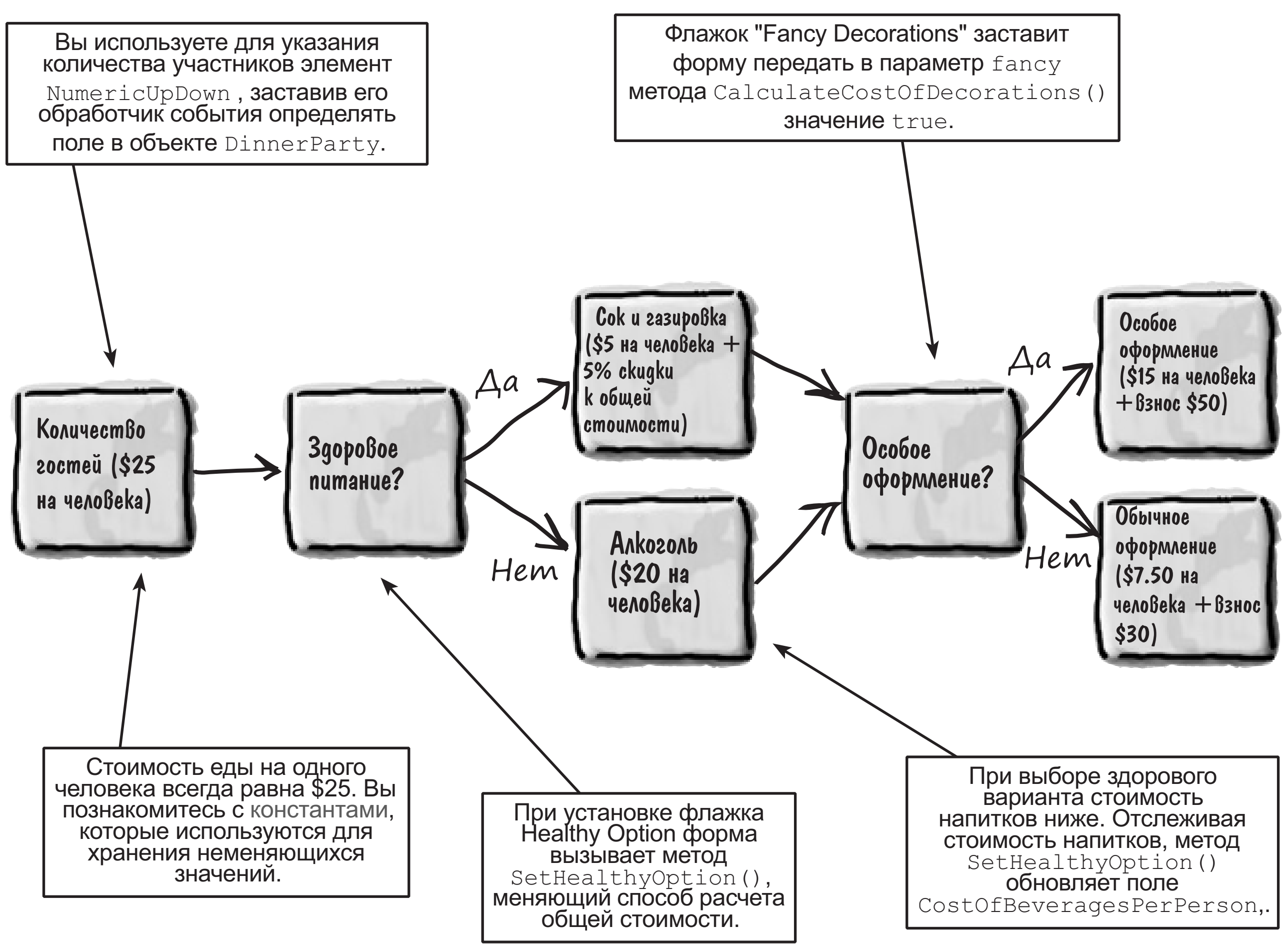
DinnerParty
NumberOfPeople CostOfBeveragesPerPerson CostOfDecorations
SetHealthyOption() CalculateCostOfDecorations() CalculateCost()

Логика программы будет встроена в класс DinnerParty. Форма создаст объект DinnerParty, сохранит ссылки на него в поле и использует поля и методы для вычислений стоимости.

Вот как должна выглядеть верхняя часть формы. Для вычисления стоимости применяется поле `dinnerParty`. Первым делом форма инициализируется значениями по умолчанию, а затем вычисляет стоимость `DisplayDinnerPartyCost()`, который вызывается при каждом изменении параметров пользователем.

```
public partial class Form1 : Form
{
    DinnerParty dinnerParty;

    public Form1()
    {
        InitializeComponent();
        dinnerParty = new DinnerParty() { NumberOfPeople = 5 };
        dinnerParty.SetHealthyOption(false);
        dinnerParty.CalculateCostOfDecorations(true);
        DisplayDinnerPartyCost();
    }
    ...
```

Вот как будет работать класс `DinnerParty`. Текущее состояние объекта `DinnerParty` — хранящееся в полях значения — определяет способ вычисления стоимости. Выбор здорового питания и особого оформления, а также изменение количества участников влияет на состояние объекта, что заставляет метод `CalculateCost()` пересчитать результат.



Каждый раз при установке флажка и изменении количества участников метод обработчика событий использует поля и методы объекта `DinnerParty` для обновления его состояния. Затем вызывается метод `CalculateCost()`, который считает итоговую сумму вечеринки и отображает ее в метке.

Разобрались? Тогда начнем работу!

Создадим программу, оценивающую стоимость обедов.

DinnerParty

NumberOfPeople
CostOfBeveragesPerPerson
CostOfDecorations

SetHealthyOption()
CalculateCostOfDecorations()
CalculateCost()

Диаграмма класса DinnerParty, который вам нужно создать.

1. Создайте проект Windows Application, добавьте класс `DinnerParty.cs` и оформите его в соответствии с приведенной слева диаграммой. Методы затрагивают расчеты стоимости «здорового» варианта, стоимости оформления, а также общей стоимости мероприятия. Два последних поля должны относиться к типу `decimal`, а первое — к типу `int`. Убедитесь, что в конце каждой константы типа `decimal` **стоит буква M** (например, `10.0M`).

2. Так как стоимость еды на одного человека не будет меняться, ее можно объявить как ***константу***. Вот как это делается:

```
public const int CostOfFoodPerPerson = 25;
```

3. Вернитесь на предыдущую страницу, чтобы убедиться, что вы работаете с нужными методами. Один из методов возвращает значение типа `decimal`, два других — нет. Метод `CalculateCostOfDecorations()` вычисляет стоимость оформления в зависимости от количества приглашенных. Метод `CalculateCost()` вычисляет общую стоимость, складывая цену оформления, еды и напитков в зависимости от количества приглашенных. При выборе «здорового варианта» делается скидка с общей стоимости.

4. Добавьте к форме код:

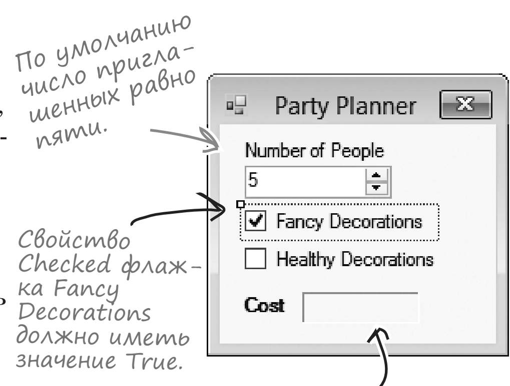
```
DinnerParty dinnerParty;
public Form1() {
  InitializeComponent();
  dinnerParty = new DinnerParty() { NumberOfPeople = 5 };
  dinnerParty.SetHealthyOption(false);
  dinnerParty.CalculateCostOfDecorations(true);
  DisplayDinnerPartyCost();
}
```

Вы объявляете поле dinnerParty и добавляете четыре строчки.

Метод SetHealthyOption() использует логический параметр (healthyOption) для обновления значений поля CostOfBeveragesPerPerson.

5. Форма должна выглядеть так. Используйте элемент `NumericUpDown`, чтобы задать максимальное количество людей равным 20, а минимальное — 1. По умолчанию количество приглашенных равно 5. Избавьтесь от кнопок управления размерами окна.

Флажки названы fancyBox и healthyBox. Для элемента управления NumericUpDown можно сохранить имя по умолчанию.



Имя этой метки costLabel. Свойство Text оставлено пустым, свойство BorderStyle имеет значение Fixed3D, а свойство AutoSize — значение false.

6 Сделаем так, чтобы итоговая сумма автоматически менялась при изменении флажков и показаний счетчика `NumericUpDown`. Для этого вам потребуется метод, отображающий сумму.

Добавим его к классу Form1. Он будет вызываться при щелчке на элементе `NumericUpDown`:

Этот метод вызывается всеми методами, связанными с формой. Именно так обновляется значение метки Cost.

```
private void DisplayDinnerPartyCost()
{
  decimal Cost = dinnerParty.CalculateCost(checkBox2.Checked);
  costLabel.Text = Cost.ToString("c");
```

Метод рассчитывает стоимость обеда и передает результат метке Cost.

Присвойте метке, отображающей цену, имя costLabel.

Аргумент "c" метода ToString() отображает валюту с использованием принятого по соглашению символа.

Значение true появляется при установке флажка Healthy Option.

7 Соединим поле `NumericUpDown` с переменной `NumberOfPeople`, принадлежащей классу `DinnerParty`, чтобы в форме начала отображаться сумма расходов. Дважды щелкните на элементе `NumericUpDown`, в код будет добавлен ***обработчик событий (event handler)***. Так называется метод, запускаемый при каждом изменении элемента управления. Он сбросит количество гостей. Впишите следующий код:

При двойном щелчке на кнопке IDE добавляет обработчик события Click.

```
private void numericUpDown1_ValueChanged(
                                    object sender, EventArgs e)
{
   dinnerParty.NumberOfPeople = (int) numericUpDown1.Value;
   DisplayDinnerPartyCost();
}
```

Ой, ошибка в коде. Можете указать, где именно? Не волнуйтесь, если вы этого пока не видите.

Значение numericUpDown.Value принадлежит типу Decimal, поэтому требуется операция приведения типов.

Ой... код содержит ошибку. Вы ее видите? Если нет, не волнуйтесь. Скоро мы все объясним!

Из формы методу передается логическое значение fancyBox.Checked.

Первый метод рассчитывает стоимость мероприятия, а второй отображает конечную сумму в форме.

8 Дважды щелкните на флажке Fancy Decorations и убедитесь, что сначала он вызывает метод `CalculateCostOfDecorations()`, а потом метод `DisplayDinnerPartyCost()`. Затем дважды щелкните на флажке Healthy Option и убедитесь, что он сначала вызывает метод `SetHealthyOption()` класса `DinnerParty`, а затем метод `DisplayDinnerPartyCost()`.

Упражнение

Вот такой код должен содержаться в файле DinnerParty.cs.

Использование констант гарантирует, что данные параметры останутся неизменными на протяжении всей программы.

```
class DinnerParty {
    const int CostOfFoodPerPerson = 25;
    public int NumberOfPeople;
    public decimal CostOfBeveragesPerPerson;
    public decimal CostOfDecorations = 0;

    public void SetHealthyOption(bool healthyOption) {
        if (healthyOption) {
            CostOfBeveragesPerPerson = 5.00M;
        } else {
            CostOfBeveragesPerPerson = 20.00M;
        }
    }

    public void CalculateCostOfDecorations(bool fancy) {
        if (fancy)
        {
            CostOfDecorations = (NumberOfPeople * 15.00M) + 50M;
        } else {
            CostOfDecorations = (NumberOfPeople * 7.50M) + 30M;
        }
    }
    public decimal CalculateCost(bool healthyOption) {
        decimal totalCost = CostOfDecorations +
                ((CostOfBeveragesPerPerson + CostOfFoodPerPerson)
                    * NumberOfPeople);

        if (healthyOption) {
            return totalCost * .95M;
        } else {
            return totalCost;
        }
    }
}
```

Создав объект, форма использует инициализатор для указания параметра NumberOfPeople. Затем методы SetHealthyOption() и CalculateCostOfDecorations() задают значения других полей.

Оператор if всегда проверяет, соблюдается ли условие, поэтому достаточно написать «if (Fancy)» вместо «if (Fancy == true)».

Мы использовали скобки, чтобы гарантировать правильность математических вычислений.

5-процентная скидка при условии, что обед подается без алкоголя.

Не нужно добавлять в класс DinnerParty строку «using System.Windows.Forms;», так как в нем не используется метод MessageBox.Show() или другие элементы из пространства имен .NET Framework.

Для переменных, содержащих цены, выбирается тип `decimal`. Всегда помещайте букву M после цифры: чтобы присвоить переменной значение $35.26, нужно написать `35.26M`. Это легко запомнить, потому что M — первая буква в слове money (деньги)**!**

```
public partial class Form1 : Form {
    DinnerParty dinnerParty;
    public Form1() {
        InitializeComponent();
        dinnerParty = new DinnerParty() { NumberOfPeople = 5 };
        dinnerParty.CalculateCostOfDecorations(fancyBox.Checked);
        dinnerParty.SetHealthyOption(healthyBox.Checked);
        DisplayDinnerPartyCost();
    }

    private void fancyBox_CheckedChanged(object sender, EventArgs e) {
        dinnerParty.CalculateCostOfDecorations(fancyBox.Checked);
        DisplayDinnerPartyCost();
    }

    private void healthyBox_CheckedChanged(object sender, EventArgs e) {
        dinnerParty.SetHealthyOption(healthyBox.Checked);
        DisplayDinnerPartyCost();
    }

    private void numericUpDown1_ValueChanged(object sender, EventArgs e) {
        dinnerParty.NumberOfPeople = (int)numericUpDown1.Value;
        DisplayDinnerPartyCost();
    }

    private void DisplayDinnerPartyCost() {
        decimal Cost = dinnerParty.CalculateCost(healthyBox.Checked);
        costLabel.Text = Cost.ToString("c");
    }
}
```

Метод DisplayDinnerPartyCost передает данные метке, которая отображает сумму расходов сразу после загрузки формы.

Флажки меняют значение переменных healthyOption и Fancy с true на false и обратно.

*Мы **присвоили флажкам имена** healthyBox и fancyBox, чтобы вы могли следить, что происходит в их методах обработчиков событий.*

Сумма должна пересчитываться и отображаться при каждом изменении количества гостей и каждой установке флажка.

Метод ToString() преобразует переменные в строки. Аргумент «c» при этом преобразуется в локальную денежную единицу. Аргумент «f3» форматирует результат в виде типа decimal с тремя знаками после запятой, «0» (ноль) превращает результат в целое число, «0%» — в целое с процентами, а «n» дает число с запятой в качестве разделителя групп разрядов. Вы сами можете посмотреть, как это работает!

Кэтлин тестирует программу

Потрясающе! Как быстро я теперь могу назвать сумму!

Роб — один из любимых клиентов Кэтлин. Она организовала его свадьбу, и теперь он хочет поручить ей планирование званого обеда.

Роб (по телефону): Привет, Кэтлин! Как там подготовка моей вечеринки?

Кэтлин: Великолепно. Мы уже заказали оформление. Тебе должно понравиться.

Роб: Фантастика! Послушай, мне только что позвонила тетя жены. Она с мужем собирается погостить у нас пару недель. Сколько будет стоит включение в список гостей еще двух человек?

Кэтлин: Подожди минутку...

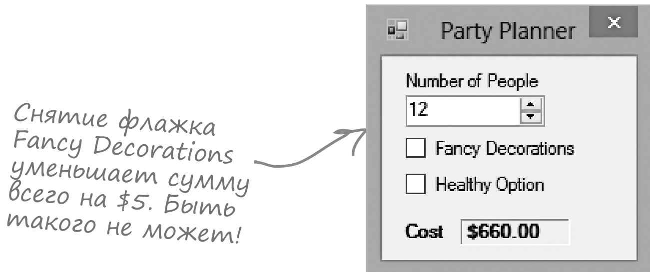
Флажок Fancy Decorations установлен по умолчанию, так как его свойство Checked имеет значение true. Если количество гостей равно 10, стоимость обеда составит $575.

Этот скриншот был снят в США, поэтому на нем фигурирует знак доллара. Жители Великобритании, Франции и Японии увидят значок фунта, евро и йены соответственно, так как для преобразования десятичной цифры в строку применяется метод ToString("c").

Изменив параметр Number of People с 10 на 12, в итоге получаем $665. Сумма кажется ей слишком маленькой...

Кэтлин: Кажется, общая стоимость обеда возрастет с $575 до $665.

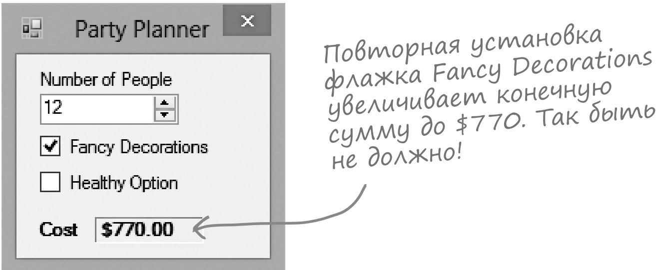
Роб: Всего на $90? Соблазнительно! А какая цена получится, если убрать особое оформление?



Кэтлин: Ээээ… получается сумма в $660.

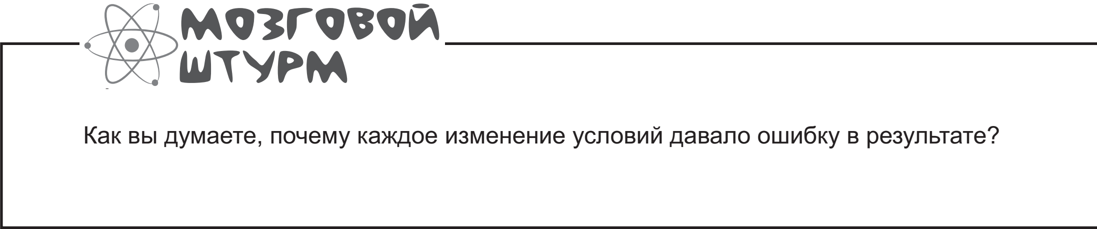
Роб: $660? Я думал, оформление стоит $15 с человека. Вы что, поменяли цену? Если разница составит всего $5, оставим особый вариант. Хотя, должен заметить, что в твоих ценах я уже запутался.

Кэтлин: Я только что получила новую программу для оценки расходов. Но кажется, она где-то ошибается. Подожди секунду, я попробую снова добавить к счету особое оформление.



Кэтлин: Роб, произошла ошибка. С особым оформлением цена возрастает до $770. Что-то я не доверяю этому приложению. Верну-ка я его на доработку и посчитаю вручную, как и раньше. Я могу позвонить тебе завтра?

Роб: Я не готов платить $770 за двух дополнительных гостей. Сначала ты озвучила мне намного более разумную цену. Я дам $665 и ни цента больше!



Как вы думаете, почему каждое изменение условий давало ошибку в результате?

Каждый вариант нужно было считать отдельно

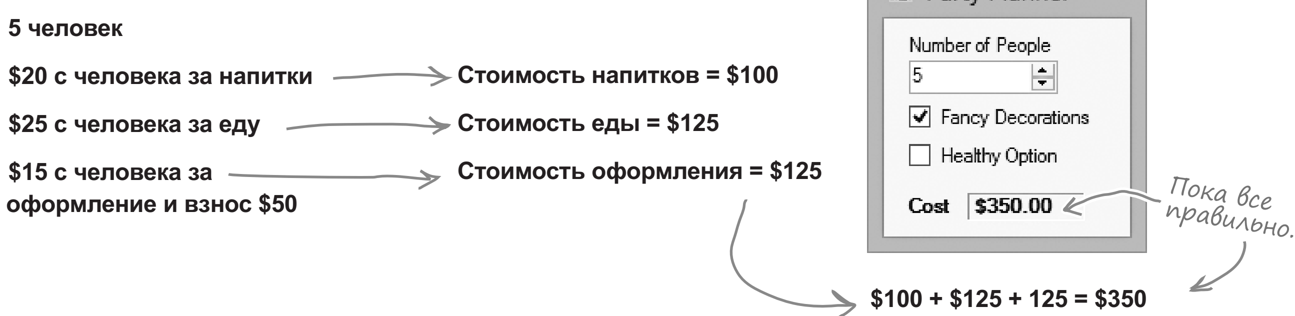
Расслабьтесь

Это не ваша вина!

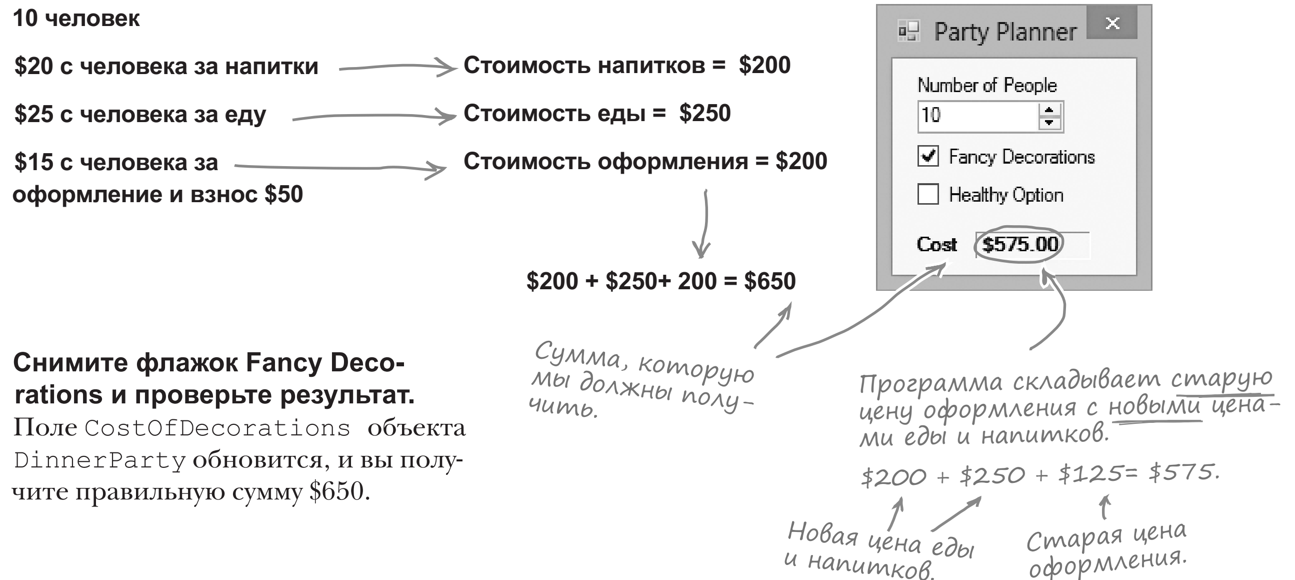
В код специально была помещена ошибка, чтобы показать, какие проблемы возникают, когда объекты используют поля друг друга, и то, как сложно эти проблемы отследить.

Подсчет сумм велся строго по диаграмме Кэтлин, но мы не учли того, каким образом на общий счет влияет изменение каждого параметра формы.

При запуске формы количество гостей по умолчанию равно 5, кроме того, установлен флажок Fancy Decorations. Флажок Healthy Option не установлен. При этих условиях стоимость мероприятия равна $350. Вот как получается эта сумма:



При изменении количества гостей приложение должно пересчитывать сумму аналогичным образом. Но этого не происходит:



Снимите флажок Fancy Decorations и проверьте результат. Поле `CostOfDecorations` объекта `DinnerParty` обновится, и вы получите правильную сумму $650.

Проблема под увеличительным стеклом

Рассмотрим метод, обрабатывающий изменение состояния элемента numericUpDown. Он берет значение переменной NumberofPeople и вызывает метод DisplayDinnerPartyCost(). Именно здесь осуществляется пересчет конечной суммы.

```
private void numericUpDown1_ValueChanged(
                              object sender, EventArgs e) {

   dinnerParty.NumberOfPeople = (int)numericUpDown1.Value;

   DisplayDinnerPartyCost();

}
```

Эта строка задает значение параметра NumberofPeople для экземпляра DinnerParty на основе введенных в форму данных.

Данный метод вызывает метод CalculateCost(), но забывает вызвать метод CalculateCostofDecorations().

То есть при изменении значения в поле NumberofPeople показанный ниже метод никогда не вызывается:

```
public void CalculateCostOfDecorations(bool Fancy) {

    if (Fancy) {

        CostOfDecorations = (NumberOfPeople * 15.00M) + 50M;

    } else {

        CostOfDecorations = (NumberOfPeople * 7.50M) + 30M;

    }

}
```

Эта переменная сохраняет значение $125, полученное при первом вызове формы.

Повторная установка флажка Fancy Decorations снова запускает метод CalculateCostOf Decorations(), что приводит к коррекции данных.

Это не единственная часть программы, которая работает некорректно. Флажки работают **несогласовано**: один вызывает метод, задающий состояние объекта, а второй передается в метод в виде аргумента. Любой программист скажет, что это ***противоречит здравому смыслу***!

Сложно? Не судите себя слишком строго. Задача была поставлена таким образом, что в результате появилась программа с концептуальными проблемами! К концу этой главы вы создадите более правильную и простую версию.

Предполагалось, что все три варианта будут выбираться одновременно!

К сожалению, пользователи не всегда используют классы так, как предполагал разработчик. К счастью, в C# есть функция, позволяющая гарантировать корректную работу программы, даже когда пользователь делает вещи, о которых вы и предположить не могли. Она называется **инкапсуляцией (encapsulation)**.

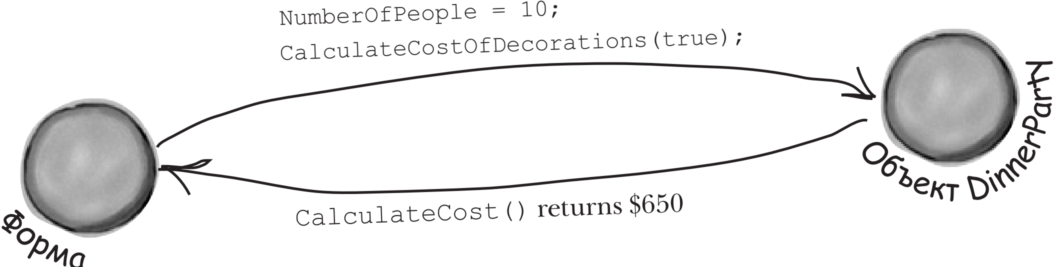
...иногда этим «пользователем» являетесь вы сами!

Неправильное использование объектов

Программа не принесла Кэтлин пользы, так как форма, проигнорировав метод `CalculateCostOfDecorations()`, сразу перешла к полям класса `DinnerParty`. Этот класс написан без ошибок, но программа все равно работает некорректно.

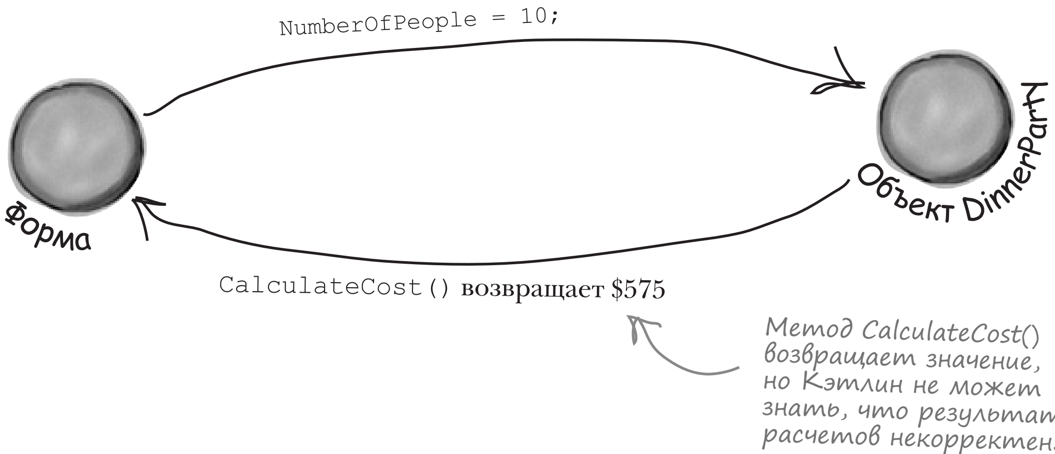
1 Как нужно было бы вызывать класс DinnerParty

Класс `DinnerParty` предоставляет форме прекрасный метод расчета конечной стоимости оформления. Достаточно указать количество гостей и вызвать метод `CalculateCostOfDecorations()`, и метод `CalculateCost()` покажет правильную сумму.



2 Как этот класс на самом деле вызывается

После указания количества гостей форма сразу вызывает метод `CalculateCost()`, игнорируя расчет стоимости оформления. То есть пропускается целый этап, и в итоге получается неверная сумма.



Инкапсуляция как управление доступом к данным

Подобных проблем можно избежать: достаточно оставить один способ работы с классом. Для этого в C# в объявлении переменных используется ключевое слово **private**. До этого момента вы встречали только модификатор public. Поля объектов, помеченные этим модификатором, были доступны для чтения и редактирования любым другим объектом. Модификатор private **делает поле доступным только изнутри объекта** (или из другого объекта *этого же класса*).

В C# поля по умолчанию считаются закрытыми, поэтому модификатор private зачастую опускается.

Статические методы имеют доступ к закрытым полям всех экземпляров класса.

```
class DinnerParty {
   private int numberOfPeople;

   ...

   public void SetPartyOptions(int people, bool fancy) {
      numberOfPeople = people;
      CalculateCostOfDecorations(fancy);
   }

   public int GetNumberOfPeople() {
      return numberOfPeople;
   }
```

Для ограничения доступа к полю достаточно воспользоваться ключевым словом private при его объявлении. В итоге доступ к полю numberOfPeople экземпляра DinnerParty будет только у экземпляров этого класса. Другие объекты не смогут его «увидеть».

Но другим объектам также требуется информация о количестве гостей. Значит, нужно добавить задающие этот параметр методы. Это позволит гарантировать вызов метода CalculateCostOfDecorations() при каждом изменении количества гостей, избавив нас от надоевшей ошибки.

Закрыв поле, содержащее информацию о количестве гостей, мы оставим форме всего один способ передать эти данные классу DinnerParty, гарантировав правильный расчет стоимости оформления. Закрытие доступа к данным с последующим написанием кода для их использования называется *инкапсуляцией (encapsulation)*.

Инкапсулированный — заключенный в защитный слой или мембрану. *Подводники полностью* инкапсулированы *в подводной лодке.*

Доступ к методам и полям класса

Доступ к полям и методам, помеченным словом public, имеет любой класс. Вся содержащаяся в них информация подобна открытой книге... вы уже видели, как подобные вещи могут стать причиной непредсказуемых результатов. Инкапсуляция меняет уровень доступа к данным. Рассмотрим на примере, как это работает:

1 Супершпион Херб Джонс стоит на страже интересов, свободы и счастья, работая секретным агентом в СССР. Его объект `ciaAgent` (Агент ЦРУ) является экземпляром класса `SecretAgent` (Секретный агент).

SecretAgent
Alias RealName Password
AgentGreeting()

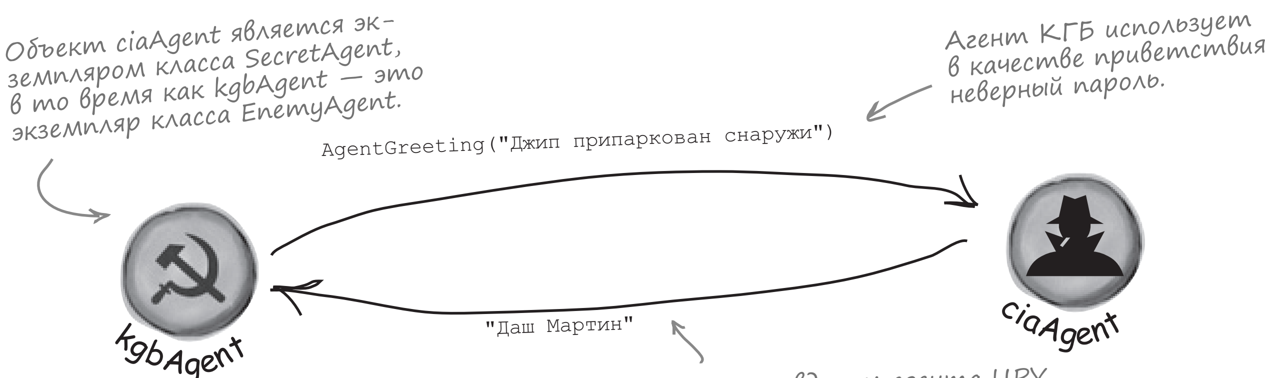
ciaAgent

```
Реальное имя (RealName): "Херб Джонс"
Псевдоним (Alias): "Даш Мартин"
Пароль (Password): "Ворона летает в полночь"
```

2 У агента Джонса есть план, как избежать агентов КГБ. Он добавил метод `AgentGreeting()`, параметром которого является пароль. Не получив правильного пароля, он называет только свой псевдоним — Даш Мартин.

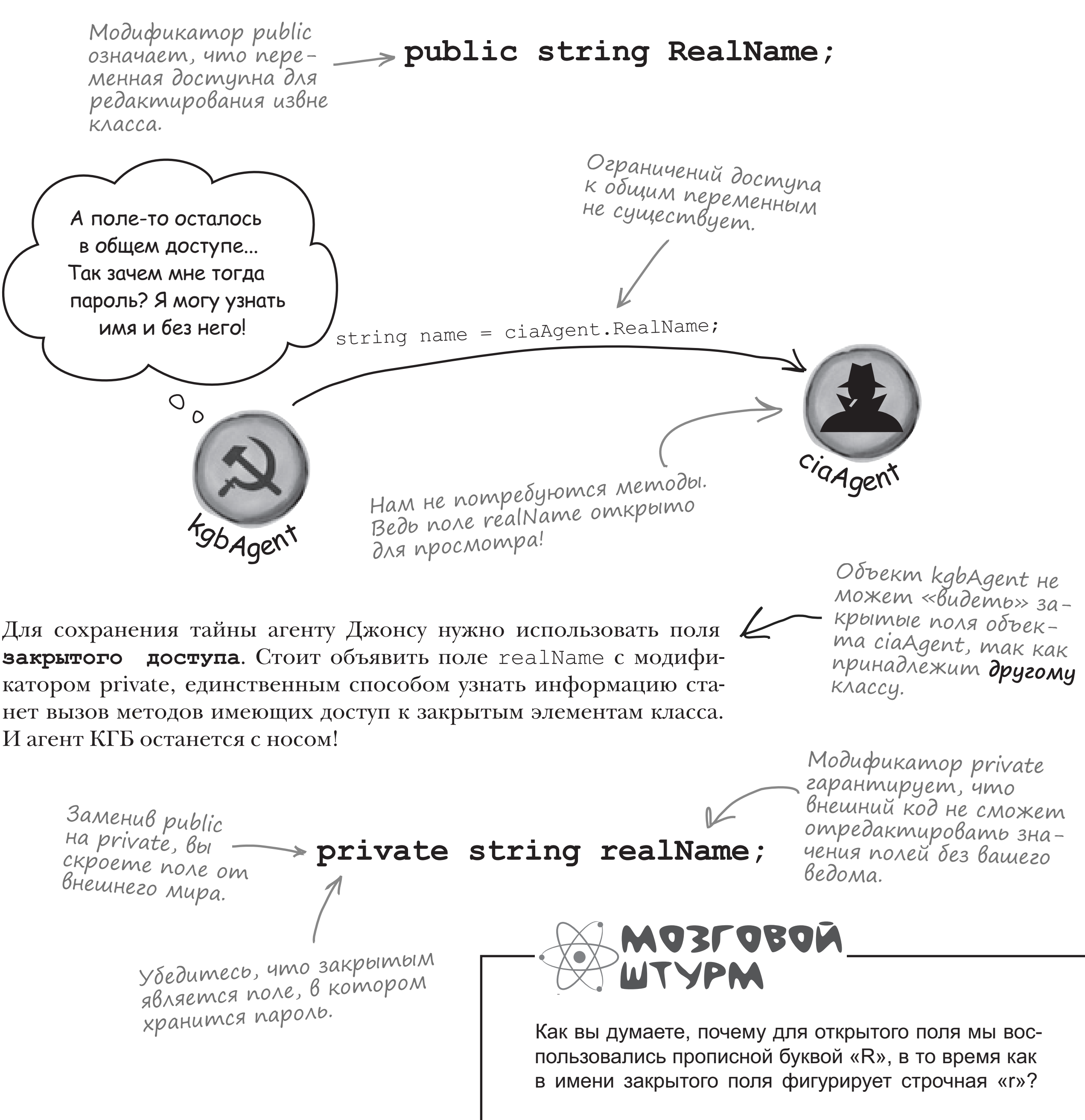
EnemyAgent
Borscht Vodka
ContactComrades() OverthrowCapitalists()

3 Кажется, что это вполне надежная защита, не так ли? Если объект, вызывающий метод, не «знает» правильного пароля, он не «узнает» настоящее имя агента Джонса.



НА САМОМ ЛИ ДЕЛЕ защищено поле realName?

Итак, мы закончили на том, что, не зная пароля, агент КГБ не сможет узнать настоящее имя агента ЦРУ. Теперь посмотрим на объявление поля `realName`:



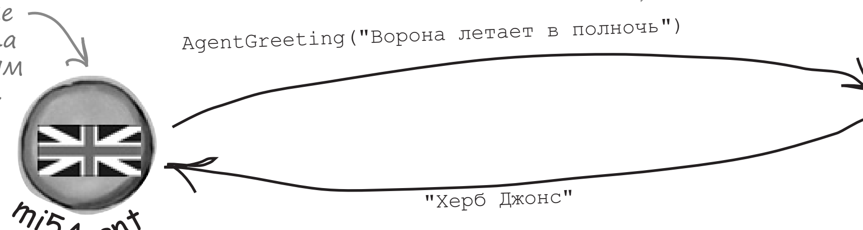
Для сохранения тайны агенту Джонсу нужно использовать поля **закрытого доступа**. Стоит объявить поле `realName` с модификатором private, единственным способом узнать информацию станет вызов методов имеющих доступ к закрытым элементам класса. И агент КГБ останется с носом!

МОЗГОВОЙ ШТУРМ

Как вы думаете, почему для открытого поля мы воспользовались прописной буквой «R», в то время как в имени закрытого поля фигурирует строчная «r»?

Закрытые поля и методы доступны только изнутри класса

Существует всего один способ доступа к информации, хранящейся в закрытых полях: использование полей и методов общего доступа, возвращающих значение. Но если агентам КГБ и МИ5 требуется метод `AgentGreeting()`, коллеги из ЦРУ могут видеть все что угодно: любой класс **имеет доступ к закрытым полям любого своего экземпляра**.



Единственным способом получения информации из закрытых полей являются методы общего доступа, которые возвращают данные.

Часто задаваемые вопросы

В: Что произойдет, если класс с закрытыми полями не даст мне доступа к данным, в то время как мне это нужно?

О: В этом случае вы не сможете получить доступ извне. При конструировании классов нужно гарантировать доступ для других объектов. Закрытые поля являются важной частью инкапсуляции, но нужно оставлять удобный способ доступа к данным на случай, если вдруг возникнет такая необходимость.

В: А зачем запрещать доступ к полям объектам из другого класса?

О: Иногда классу приходится отслеживать информацию, необходимую для выполнения каких-то операций, но при этом другим объектам эта информация не нужна. Скажем, при генерации случайных чисел применяются так называемые начальные числа (*seeds)*. О том, как именно происходит генерация, мы говорить не будем, достаточно знать, что каждый экземпляр `Random` содержит массив из нескольких дюжин чисел, которые гарантируют, что метод `Next()` всегда даст на выходе случайное число. Создав новый экземпляр `Random`, вы не сможете увидеть этот массив. Да это и не нужно. Имея доступ, вы могли бы добавлять значения, что привело бы к генерации неслучайных чисел. Поэтому начальные числа должны быть полностью инкапсулированы.

В: А почему все обработчики событий объявляются со словом `private`?

О: Формы в C# сконструированы таким образом, что запуск обработчиков событий может осуществляться только при помощи элементов формы. Модификатор `private` означает, что метод может использоваться только внутри класса. По умолчанию обработчики событий не могут управляться посторонними формами или объектами. Но нет правила, это предписывающего. Вы можете щелкнуть два раза на кнопке и указать в объявлении обработчика событий модификатор `public`. И код все равно будет компилироваться и запускаться.

Возьми в руку карандаш

Перед вами класс с закрытыми полями. Обведите операторы, которые **не будут компилироваться,** если их запустить извне класса, используя **экземпляр объекта mySuperChef**.

```
class SuperChef
{
       public string cookieRecipe;
       private string secretIngredient;
       private const int loyalCustomerOrderAmount = 60;
       public int Temperature;
       private string ingredientSupplier;

       public string GetRecipe (int orderAmount)
       {
              if (orderAmount >= loyalCustomerOrderAmount)
              {
                     return cookieRecipe + " " + secretIngredient;
              }
              else
              {
                     return cookieRecipe;
              }
      }
}
```

1. `string ovenTemp = mySuperChef.Temperature;`

2. `string supplier = mySuperChef.ingredientSupplier;`

3. `int loyalCustomerOrderAmount = 54;`

4. `mySuperChef.secretIngredient = "кардамон";`

5. `mySuperChef.cookieRecipe = "3 яйца, 2.5 чашки муки, 1 ст. л. соли, 1 ст. л. ванили и 1.5 чашки сахара смешать. Выпекать 10 минут при температуре 190 °C. Приятного аппетита!";`

6. `string recipe = mySuperChef.GetRecipe(56);`

7. Какое значение будет иметь переменная `recipe` после запуска всех этих строк кода?

....

....

Возьми в руку карандаш
Решение

Вот какие операторы **не будут компилироваться,** если запустить их извне класса, используя **экземпляр объекта mySuperChef**.

```
class SuperChef
{
       public string cookieRecipe;
       private string secretIngredient;
       private const int loyalCustomerOrderAmount = 60;
       public int Temperature;
       private string ingredientSupplier;

       public string GetRecipe (int orderAmount)
       {
              if (orderAmount >= loyalCustomerOrderAmount)
              {
                     return cookieRecipe + " " + secretIngredient;
              }
              else
              {
                     return cookieRecipe;
              }
       }
}
```

Единственным способом получения секретного компонента является заказ всех ингредиентов. Внешний код не имеет непосредственного доступа к этому полю.

1. `string ovenTemp = mySuperChef.Temperature;`

Попытка присвоить переменную типа int переменной типа string.

2. `string supplier = mySuperChef.ingredientSupplier;`

3. `int loyalCustomerOrderAmount = 54;`

4. `mySuperChef.secretIngredient = "кардамон";`

Строки 2 и 4 не будут компилироваться из-за закрытых полей ingredientSupplier и secretIngredient.

5. `mySuperChef.cookieRecipe = "3 яйца, 2.5 чашки муки, 1 ст. л. соли, 1 ст. л. ванили и 1.5 чашки сахара смешать. Выпекать 10 минут при температуре 190 °C. Приятного аппетита!";`

6. `string recipe = mySuperChef.GetRecipe(56);`

Созданная вами локальная переменная loyalCustomerAmount, которой было присвоено значение 54, не меняет значение переменной объекта loyalCustomerAmount, поэтому секретный компонент не выдается.

7. Какое значение будет иметь переменная `recipe` после запуска всех этих строк кода?

3 яйца, 2.5 чашки муки, 1 ст. л. соли, 1 ст. л. ванили и 1.5 чашки сахара смешать.

Выпекать 10 минут при температуре 190 °C. Приятного аппетита!

Потому что иногда возникает необходимость скрыть некую информацию от остальной части программы.

Большинство пользователей поначалу не могут принять идею инкапсуляции, так как не понимают, зачем нужно скрывать поля, методы или свойства одного класса от другого. Но постепенно вы поймете причины, по которым отдельные классы имеет смысл делать невидимыми для остальной части программы.

Инкапсуляция делает классы...

★ **Легкими в применении**
Вы уже знаете, что поля нужны для отслеживания данных. И большинство из них применяют методы для обновления информации — эти методы не нужны никакому другому классу. Часто поля, методы и свойства одного класса совершенно не нужны в других частях программы. Пометив их словом private, вы уберете их из окна IntelliSense.

★ **Легкими в управлении**
Помните программу Кэтлин? Проблема возникала потому, что форма имела непосредственный доступ к полю. Если бы поле было закрытым, программа работала бы правильно.

★ **Гибкими**
Иногда по прошествии времени возникает необходимость внести в программу изменения. С хорошо инкапсулированными классами не возникает вопросов по их дальнейшему использованию.

Инкапсуляция означает скрытие информации одного класса от другого. Она помогает предотвратить появление ошибок.

Как плохо инкапсулированные классы могут помешать редактированию программы в будущем?

Программе Майка не помешала бы инкапсуляция

Геокэшингом называется игра, в которой игроки ищут тайники, с помощью GPS определяют их координаты и сообщают об этом в Интернете.

Помните программу Майка из главы 3? Майк увлекся геокэшингом и надеется на помощь навигатора. Он давно не обновлял программу и сейчас испытывает проблемы. Класс Route в его программе хранит маршрут между двумя точками. Но Майк не помнит, как им пользоваться! Вот что получается при попытках редактировать код:

★ Свойству StartPoint были присвоены координаты дома Майка, а свойству EndPoint — координаты офиса. Свойство Length показало, что расстояние равно 15.3. Но метод GetRouteLength() вернул 0 в качестве результата.

★ Свойству SetStartPoint() были присвоены координаты дома, а свойству SetEndPoint() — координаты офиса. Метод GetRouteLength() вернул значение 9.51, в то время как свойство Length показало расстояние 5.91.

★ При попытках использовать свойства StartPoint и SetEndPoint() метод GetRouteLength() всегда возвращал 0, такое же значение показывало свойство Length.

★ Попытавшись взять для задания начальной и конечной точек метод SetStartPoint() и свойство EndPoint соответственно, Майк увидел, что свойство Length имеет значение 0, а метод GetRouteLength() приводит к сообщению об ошибке... что-то про невозможность деления на ноль.

Не могу вспомнить, мне требовалось поле StartPoint или метод SetStartPoint(). Но раньше все работало!

Возьми в руку карандаш

Вот объект Route из программы Майка. Какие свойства и методы **вы** бы пометили как **private**, чтобы облегчить их использование?

Route
StartPoint EndPoint Length
GetRouteLength() GetStartPoint() GetEndPoint() SetStartPoint() SetEndPoint() ChangeStartPoint() ChangeEndPoint()

....

....

....

....

....

Есть много потенциально правильных способов решения этой задачи! Запишите лучшие.

Представим объект в виде «черного ящика»

Иногда можно услышать, как программисты называют объекты «черным ящиком». Примерно так все и выглядит. При вызове методов объекта вы вряд ли заботитесь о том, как именно они работают, по крайней мере, не на текущем этапе обучения. Вам пока нужно, чтобы метод взял указанные вами значения и выдал нужный результат

Открыв через некоторое время старую программу, вы вряд ли вспомните, как предполагалось использовать те или иные переменные. Вот тут инкапсуляция может сильно облегчить вам жизнь!

Мой объект Route работает! Но вот как его **сейчас** приспособить его для геокэшинга?

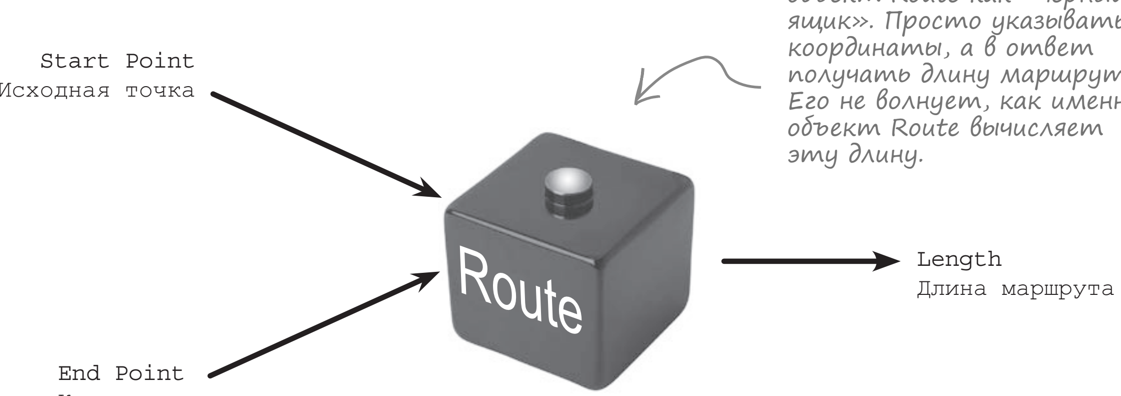
Когда в главе 3 Майк создавал навигатор, он хорошо знал, как работает объект Route. Но прошло время...

Майк успешно использовал навигатор, и теперь он хочет **повторно использовать** объект Route.

Если бы Майк вспомнил об инкапсуляции при создании объекта Route, сегодня у него не болела бы голова!

Правильно инкапсулировав классы сегодня, вы облегчите себе работу с ними завтра.

Майк хочет воспринимать объект Route как «черный ящик». Просто указывать координаты, а в ответ получать длину маршрута. Его не волнует, как именно объект Route вычисляет эту длину.



Получается, что инкапсулированный класс делает **ровно то же**, что и неинкапсулированный.

Естественно! Основное отличие инкапсулированного класса в том, что он предотвращает появление ошибок и более прост в использовании.

Испортить инкапсулированный класс легко: воспользуйтесь функцией поиска с последующей заменой, чтобы заменить все модификаторы `private` на `public`.

Как ни странно, после этого программа все равно будет благополучно скомпилирована. И даже будет работать так же, как и раньше. Именно поэтому многим пользователям так сложно понять, зачем вообще нужна инкапсуляция.

До этого момента вы учились тому, как заставить программу **выполнять действия**. Инкапсуляция же не меняет способ достижения поставленной цели. Она похожа на игру в шахматы: скрывая определенную информацию на стадии создания классов, вы задаете стратегию взаимодействия этих классов в будущем. Чем лучше будет эта стратегия, тем более гибкой получится программа и тем большего количества ошибок вы избежите.

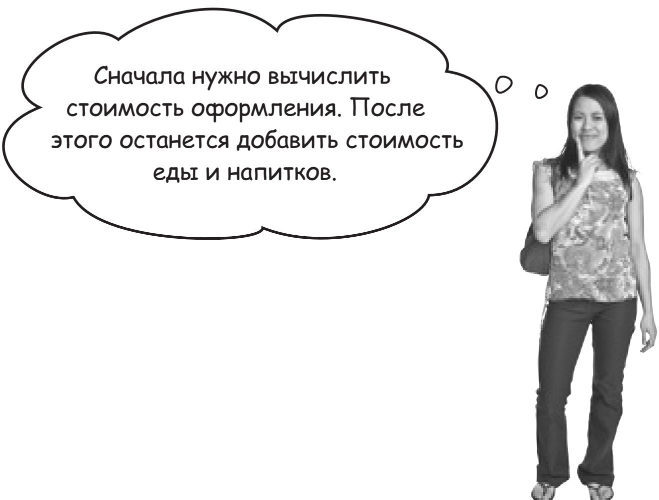
Как в шахматах существует множество вариантов ходов, так и количество стратегий инкапсуляции является почти бесконечным.

Как правильно инкапсулировать классы

★ **Думайте о способах неправильного использования полей.**
Что может пойти не так, если выбрать неправильный уровень доступа?

★ **Вся ли информация в классе может быть открытой?**
Если все поля и методы в классе являются открытыми, следует более детально подумать об инкапсуляции.

★ **Какие поля будут участвовать в вычислениях?**
Они — первые кандидаты на инкапсуляцию. Если позже к программе будет добавлен метод, использующий значение одного из таких полей, это может привести к ошибке.



★ **Открывайте доступ к полям и методам только в случае необходимости.**
Если у вас нет веской причины оставлять общий доступ к полю или методу, не делайте этого. Оставив доступ ко всем полям, вы только запутаете программу. Впрочем, не стоит впадать и в другую крайность — делать все поля закрытыми. Правильно выбрав уровни доступа, вы сэкономите время в будущем.

Инкапсуляция сохраняет данные нетронутыми

Иногда значение поля меняется по ходу выполнения программы. Если программе в явном виде не сказано возвращать поля в исходное состояние, вычисления делаются с новыми значениями. В некоторых случаях именно это и требуется, то есть программа должна выполнять некие вычисления при каждом изменении значения поля. Помните программу Кэтлин, в которой стоимость мероприятия пересчитывалась при каждом изменении количества гостей? Мы можем избежать такого поведения, просто инкапсулировав поля. После этого потребуется метод, получающий их значение, и другой метод, который присваивает значения полям и выполняет все необходимые расчеты.

Для закрытых полей мы использовали стильВерблюда, а для открытых СтильВерблюда. Во втором случае все слова в пишутся с прописной буквы, в то время как в первом случае первое слово целиком пишется строчными буквами. Прописные буквы при таком написании выглядят как «горбы» верблюда.

Код читается проще, если для записи имен полей, свойств, переменных и методов используется определенный регистр. Описанного способа придерживаются многие программисты.

Пример инкапсуляции

Класс `Farmer` (Фермер) использует поле для хранения информации о количество коров (numberOfCows), которое затем умножается на количество мешков с кормом, необходимое для одной коровы:

```
class Farmer
{
       private int numberOfCows;
}
```

Сделаем это поле закрытым, чтобы никто не мог изменить его, не отредактировав при этом поле bagsOfFeed (мешки с кормом). Рассинхронизация этих полей приводит к ошибкам в программе!

Ввод пользователем количества коров в форму должен менять значение поля `numberOfCows`. Значит, потребуется метод, возвращающий значение этого поля форме:

```
public const int FeedMultiplier = 30;
public int GetNumberOfCows()
{
       return numberOfCows;
}

public void SetNumberOfCows(int newNumberOfCows)
{
       numberOfCows = newNumberOfCows;
       BagsOfFeed = numberOfCows * FeedMultiplier;
}
```

Одной корове требуется 30 мешков корма.

Этот метод сообщает другим классам количество коров.

Они выполняют одну и ту же функцию!

numberOfCows — закрытое поле, поэтому для написания его имени использован стильВерблюда.

Метод, задающий число коров, который гарантирует одновременное изменение переменной BagsOfFeed.

Инкапсуляция при помощи свойств

Свойства (properties) объединяют функции полей и методов. Они используются для чтения и записи **вспомогательного поля (backing field)**. Именно так называется поле, заданное свойством.

```
private int numberOfCows;

public int NumberOfCows
{
    get
    {
        return numberOfCows;
    }

    set
    {
        numberOfCows = value;
        BagsOfFeed = numberOfCows * FeedMultiplier;
    }

}
```

Закрытое поле numberOfCows станет **вспомогательным полем** свойства NumberOfCows.

Свойства часто объединяются с обычным объявлением полей. Это объявление для NumberOfCows.

Метод чтения вызывается каждый раз, когда свойство NumberOfCows нужно **прочитать**. В данном случае он возвращает значение закрытого свойства numberOfCows.

Метод записи вызывается при каждой **записи** в свойство NumberOfCows. Он имеет параметр **value**, содержащий значение, записываемое в поле.

Методы чтения и записи используют так же, как поля. Вот код для кнопки, которая задает количество коров, а в ответ получает количество мешков с кормом:

```
private void button1_Click(object sender, EventArgs e) {
    Farmer myFarmer = new Farmer();
    myFarmer.NumberOfCows = 10;

    int howManyBags = myFarmer.BagsOfFeed;

    myFarmer.NumberOfCows = 20;
    howManyBags = myFarmer.BagsOfFeed;
}
```

В этой строчке метод записи задает значение закрытого поля numberOfCows и тем самым обновляет открытое поле BagsOfFeed.

Так как метод записи NumberOfCows обновил поле BagsOfFeed, вы можете получить его значение.

Поле NumberOfCows запускает метод записи, передавая значение 20. Запрос к полю BagsOfFeed запускает метод чтения, возвращающий значение 20*30=600.

Приложение для проверки класса Farmer

Создайте новое приложение Windows Forms для проверки класса **Farmer** и его свойств. Для вывода результатов будет использован метод **Console.WriteLine()**.

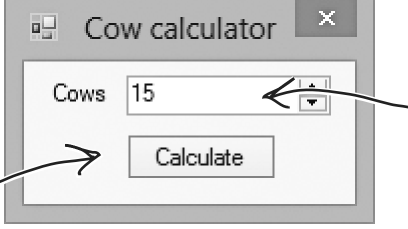
Консольный вывод в окне Output.

Когда в приложении Windows Forms результат выводится при помощи метода Console.WriteLine(), появляется окно Output. Обычно в приложениях WinForms консольный вывод не применяется, но мы воспользовались им как обучающим инструментом.

1 Добавьте к проекту класс Farmer:

```
class Farmer {
    public int BagsOfFeed;
    public const int FeedMultiplier = 30;

    private int numberOfCows;
    public int NumberOfCows {
        // (добавьте методы чтения и записи
        // с предыдущей страницы)
    }
}
```

2 Создайте форму:



Кнопка называется «вычислить» и использует открытые данные класса Farmer для вывода результата.

Присвойте параметрам Value, Minimum и Maximum элемента NumericUpDown значения 15, 5 и 300 соответственно.

3 Код формы использует метод Console.WriteLine() для отправки итоговых данных в **окно Output** (это окно вызывается также командой Output из меню Debug >> Windows). Методу WriteLine() можно передать несколько параметров, и первый — это выводимая строка. Включив в эту строку «{0}», вы выведете первый параметр, «{1}» — второй параметр, «{2}» — третий параметр и т. д.

```
public partial class Form1 : Form {
    Farmer farmer;
    public Form1() {
        InitializeComponent();
        farmer = new Farmer() { NumberOfCows = 15 };
    }
    private void numericUpDown1_ValueChanged(object sender, EventArgs e) {
        farmer.NumberOfCows = (int)numericUpDown1.Value;
    }
    private void calculate_Click(object sender, EventArgs e) {
        Console.WriteLine("I need {0} bags of feed for {1} cows",
            farmer.BagsOfFeed, farmer.NumberOfCows);
    }
}
```

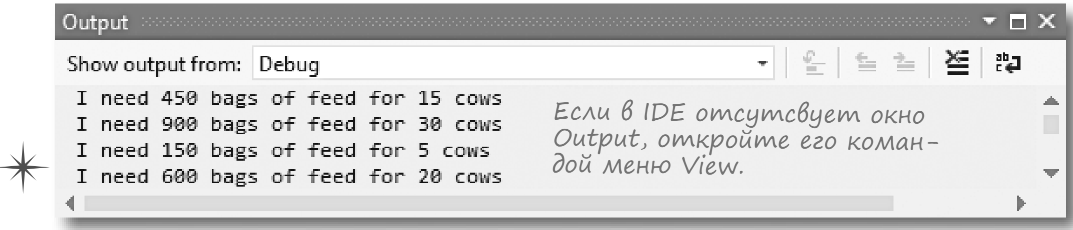
Метод Console.WriteLine() отправляет строчку с текстом в окно Output.

Метод WriteLine () замещает "{0}" значением первого параметра, а "{1}" — значением второго параметра.

Не забудьте, что элементы управления следует «привязать» к обработчикам событий! Дважды щелкните на Button и NumericUpDown в конструкторе, чтобы IDE создала заглушки их методов-обработчиков события.

Автоматические свойства

Кажется, наш счетчик коров работает корректно. Запустите программу и щелкните на кнопке для проверки. Сделайте количество коров равным 30 и снова щелкните на кнопке. Повторите эту операцию для 5 коров, а потом для 20 коров. Вот что должно появиться в окне Output:



Вы понимаете, почему это стало причиной ошибки?

Но есть небольшая проблема. Добавьте к форме кнопку, которая выполняет оператор:

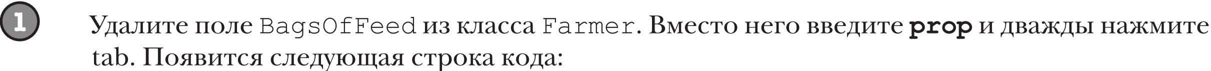
```
farmer.BagsOfFeed = 5;
```

Запустите программу. Все работает до нажатия новой кнопки. Попробуйте после этого нажать кнопку Calculate. Окажется, что 5 мешков корма требуется <u>для любого количества коров</u>! После редактирования параметра `NumericUpDown` кнопка Calculate снова начнет работать корректно.

Полностью инкапсулируем класс Farmer

Напечатав prop и дважды нажав tab, вы добавите к коду автоматическое свойство.

Проблема в том, что класс **не полностью инкапсулирован**. С помощью свойств мы инкапсулировали переменную `NumberOfCows`, но переменная `BagsOfFeed` до сих пор общедоступна. Это крайне распространенная проблема. Настолько распространенная, что в C# существует автоматическая процедура ее решения. Просто замените поле общего доступа `BagsOfFeed` автоматическим свойством:



1. Удалите поле `BagsOfFeed` из класса `Farmer`. Вместо него введите **prop** и дважды нажмите tab. Появится следующая строка кода:

```
public int MyProperty { get; set; }
```

2. Снова нажмите Tab, чтобы выделить поле `MyProperty`. Введите имя `BagsOfFeed`:

```
public int BagsOfFeed { get; set; }
```

Теперь у вас свойство вместо поля. Компилятор обрабатывает эту информацию как вспомогательное поле.

3. Впрочем, проблема еще не решена. Для ее решения сделайте свойство **доступным только для чтения**:

```
public int BagsOfFeed { get; private set; }
```

При попытке построить код вы получите сообщение об ошибке в строчке, задающей свойство `BagsOfFeed`: **метод записи недоступен**, — ведь вы не можете редактировать свойство `BagsOfFeed` вне класса `Farmer`. Удалите строчку кода, соответствующую второй кнопке. Теперь класс `Farmer` хорошо инкапсулирован!

Редактируем множитель feed

При построении счетчика коров множитель, указывающий количество корма на одну особь, мы определили как константу. Но представим, что нам требуется его изменить. Вы уже видели, как доступ к полям одного класса со стороны других классов может стать причиной ошибки. Именно поэтому **общий доступ к полям и методам имеет смысл оставлять только там, где это необходимо**. Так как программа никогда не обновляет `FeedMultiplier`, нам не требуется запись в это поле из других классов. Поэтому сделаем его свойством доступным только для чтения, которое использует вспомогательное поле.

Свойство возвращает вспомогательное поле feedMultiplier. Метод записи отсутствует, то есть оно доступно только для чтения. Метод чтения при этом открыт, то есть значение поля FeedMultiplier можно прочитать из любого другого класса.

1 Удалите строчку

```
public const int FeedMultiplier = 30;
```

Воспользуйтесь комбинацией prop-tab-tab, чтобы добавить свойство, доступное только для чтения. Но вместо автоматического свойства добавьте вспомогательное поле:

```
private int feedMultiplier;
public int FeedMultiplier { get { return feedMultiplier; } }
```

Так как вместо константы общего доступа у нас закрытое поле типа int, его имя теперь начинается со строчной буквы f.

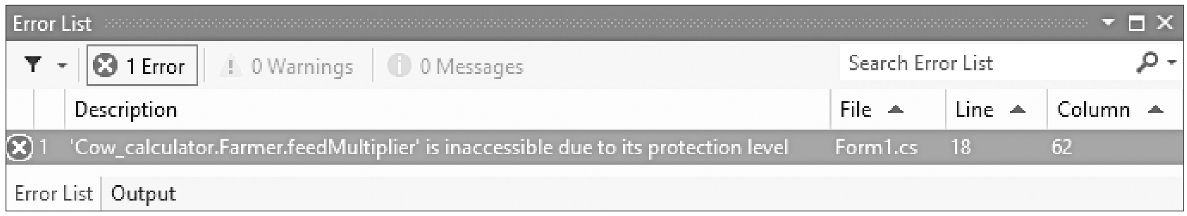
2 Запуск кода после внесения в него изменений покажет абсурдный результат. Свойство `BagsOfFeed` **всегда возвращает 0 мешков**.

Дело в том, что переменная `FeedMultiplier` не была инициализирована. Поэтому она по умолчанию имеет значение ноль. Добавим инициализатор объекта:

```
public Form1() {
    InitializeComponent();
    farmer = new Farmer() { NumberOfCows = 15, feedMultiplier = 30 };
```

Теперь **программа не компилируется**! Вот как выглядит сообщение об ошибке:

Проверьте окно Error List. В нем можно увидеть предупреждения, например, о том, что вы пытаетесь использовать переменную, которую забыли инициализировать.



Дело в том, что инициализатор объекта работает только с открытыми полями и свойствами. Что же делать, если требуется инициализировать закрытые поля?

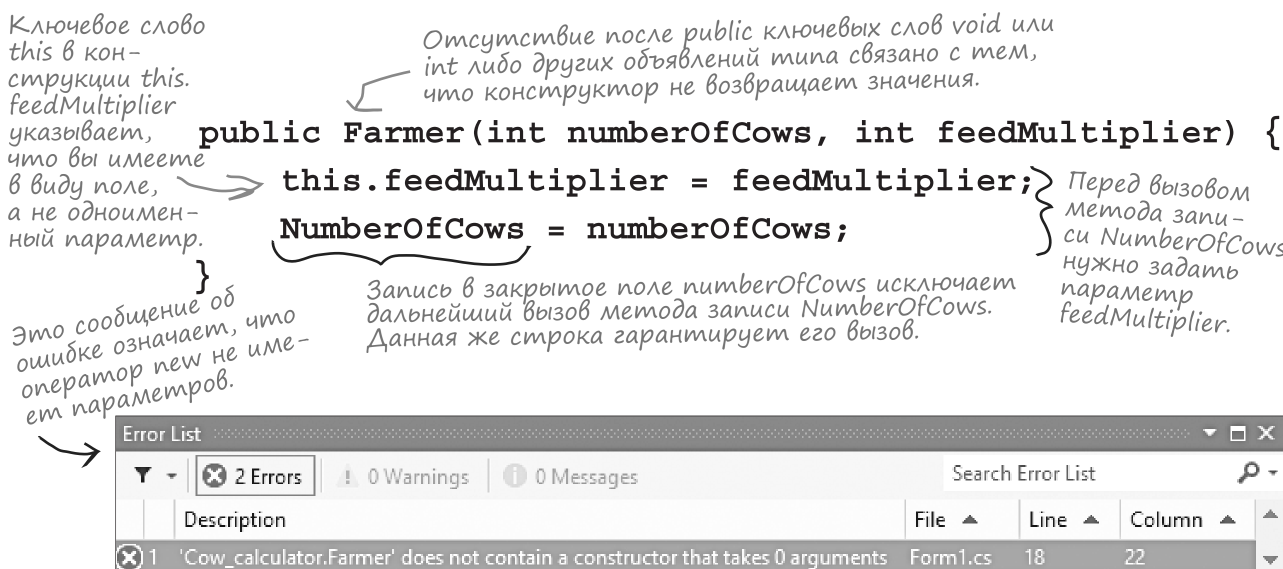
Конструктор

Итак, вы уже убедились, что с закрытыми полями инициализатор объектов не работает. К счастью, существует особый метод, называемый **конструктором (constructor)**. Это **самый первый метод, который выполняется** при создании класса оператором new. Передавая конструктору параметры, вы указываете значения, которые требуется инициализировать. Но этот метод **не имеет возвращаемого значения**, так как напрямую не вызывается. Параметр передается оператору new. А как вы уже знаете, этот оператор возвращает объект, поэтому конструктору возвращать уже ничего не нужно.

Чтобы снабдить класс конструктором, добавьте метод, имеющий имя класса и не имеющий возвращаемого значения.

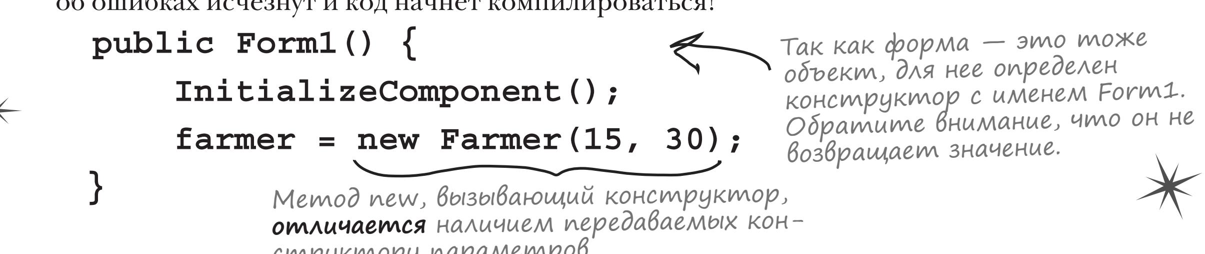
Добавление конструктора к классу Farmer

Требуется добавить всего две строчки кода, но как много они значат. Как вы помните, в классе должны присутствовать данные о количестве коров и мешков корма на одну корову. Добавим эту информацию к конструктору в качестве параметров. Для переменной feedMultiplier требуется начальное значение, так как она более не является константой.



Настройки формы, необходимые для работы с конструктором

Теперь нужно сделать так, чтобы оператор new, создающий объект Farmer, использовал конструктор вместо инициализатора объекта. После редактирования оператора new сообщения об ошибках исчезнут и код начнет компилироваться!



Конструкторы под микроскопом

Внимательно рассмотрим конструктор `Farmer`, чтобы составить представление о том, как он работает.

Конструкторы не возвращают значения.

Данный конструктор имеет два параметра: количество коров и количество корма на одну корову.

```
public Farmer(int numberOfCows, int feedMultiplier) {
    this.feedMultiplier = feedMultiplier;
    NumberOfCows = numberOfCows;
}
```

Так как второй оператор вызывает метод записи NumberOfCows, для вычисления параметра BagsOfFeed вам нужно значение feedMultiplier.

Чтобы отличить поле feedMultiplier от одноименного параметра, мы воспользовались словом this.

Так как this означает ссылку на текущий объект, запись this.feedMultiplier является ссылкой на поле. Так что сначала мы присваиваем закрытому полю feedMultiplier второй параметр конструктора.

Часто задаваемые вопросы

В: Бывают ли конструкторы без параметров?

О: Да. Классы часто снабжены конструктором, не имеющим параметров. И вы уже видели пример — **конструктор вашей формы**. Посмотрите на его объявление:

```
public Form1() {
  InitializeComponent();
}
```

Как видите, конструктор формы действительно не имеет параметров. Откройте файл `Form1.Designer.cs` и найдите метод `InitializeComponent()`, щелкнув на значке + рядом со строчкой «Windows Form Designer generated code».

Этот метод задает не только начальные значения всех элементов управления формы, но и их свойства. Перетащите на форму новый элемент управления, отредактируйте его свойства в окне Properties и обратите внимание на изменения, которые произойдут с методом `InitializeComponent()`.

Метод `InitializeComponent()` вызывается внутри конструктора формы, поэтому инициализация всех элементов управления происходит в момент создания формы. (Помните, что все формы не более чем объекты, использующие методы .NET Framework из пространства имен `System.Windows.Forms` для отображения окон, кнопок и других элементов управления.)

Будьте осторожны!

Как различать одноименные параметры и поля?

Вы заметили, что параметр конструктора `feedMultiplier` *называется так же, как вспомогательное поле свойства* `FeedMultiplier`*? Чтобы использовать в конструкторе последнее, не забудьте про ключевое слово* `this`*: имя* `feedMultiplier` *указывает на параметр, а запись* `this.feedMultiplier` *на доступ к закрытому полю.*

Часто Задаваемые Вопросы

В: Зачем нужна процедура создания методов чтения и записи? Почему нельзя просто создать поле?

О: Поля нужны для вычислений или иных действий. Вспомните о проблеме Кэтлин — после указания количества гостей в классе `DinnerParty` форма не запускала метод, пересчитывающий стоимость оформления. Заменив поле на метод записи, мы гарантируем, что этот пересчет будет сделан. (Через пару страниц вы убедитесь в этом!)

В: Чем же отличаются просто методы от методов чтения и записи?

О: Ничем! Это особые виды методов — для других объектов они выглядят как поля и вызываются при записи в поле. Метод чтения в качестве значения возвращает тип поля. А метод записи имеет только один параметр с именем `value`, тип которого совпадает с типом поля. Вместо громоздких «метод чтения» и «метод записи» можно говорить просто — «свойства».

В: Получается, что в свойство можно превратить ЛЮБОЙ оператор?

О: Вы абсолютно правы. Все, что можно реализовать при помощи метода, можно превратить в свойство. Свойства могут вызывать методы, получать доступ к полям и даже создавать объекты и экземпляры. Но все эти функции реализуются только в момент доступа к свойству, поэтому не стоит превращать в свойства операторы, не имеющие отношения к процедурам чтения или записи.

В: Если метод записи имеет параметр `value`, почему этот параметр не указан в скобках как `int value`, как это происходит с другими методами?

О: C# позволяет не писать информацию, которая не потребуется компилятору. Параметр объявляется без ваших явных указаний. Это не имеет особого значения, когда вы вводите один или два оператора, но если требуется ввести сотню, подобный подход реально экономит время и уменьшает количество возможных ошибок.

Метод записи ***всегда*** имеет единственный параметр `value`, тип которого ***всегда*** совпадает с типом свойства. C# получает всю необходимую информацию о типе и параметре в момент, когда вы набрали "`set {`". Больше ничего набирать не нужно.

В: То есть я могу не добавлять в конструктор возвращаемое значение?

О: Именно так! Конструктор не имеет возвращаемого значения, так как принадлежит к типу `void`. Было бы излишне заставлять вас набирать `void` в начале каждого конструктора.

В: Могу ли я использовать только метод чтения или только метод записи?

О: Конечно! Воспользовавшись свойством get без свойства set, вы получите свойство только для чтения. Например, класс `SecretAgent` может иметь поле `ReadOnly` (только для чтения) для имени (name):

```
string name = "Dash Martin";
public string Name {
   get { return name; }
}
```

А если воспользоваться свойством set без свойства get, вспомогательное поле будет доступно только для записи. Класс `SecretAgent` создает свойство `Password`, в которое другие шпионы могут только записывать информацию, но не читать ее:

```
public string Password {
 set {
  if (value == secretCode) {
    name = "Herb Jones";
  }
}.
```

В: А как же с объектами, которые мы создавали, не создав для них конструктор? Получается, что он нужен далеко не всегда?

О: Нет, это означает, что C# автоматически создает конструктор с нулевым параметром, если вы не делаете этого. Вы можете заставить пользователя создать экземпляр вашего класса, чтобы воспользоваться конструктором.

Свойства (методы чтения и записи) — это особый вид методов, запускаемых при попытке прочитать свойство или сделать запись в него.

Полезная информация: первая строка метода, содержащая модификатор доступа, возвращаемое значение, имя и параметры, называется сигнатурой метода. Свойства также обладают сигнатурами.

Возьми в руку карандаш

Форма использует экземпляр класса `CableBill` (Счет за телевидение) с именем `thisMonth` (Этот месяц) и при нажатии кнопки вызывает метод `GetThisMonthsBill()` (Получить счет за этот месяц). Укажите значение переменной `amountOwed` (Сумма, которую я должен) после выполнения кода.

```
class CableBill {
    private int rentalFee;
    public CableBill(int rentalFee) {
        this.rentalFee = rentalFee;
        discount = false;
    }

    private int payPerViewDiscount;
    private bool discount;
    public bool Discount {
        set {
            discount = value;
            if (discount)
            payPerViewDiscount = 2;
            else
            payPerViewDiscount = 0;
        }
    }

    public int CalculateAmount(int payPerViewMoviesOrdered) {
        return (rentalFee - payPerViewDiscount) * payPerViewMoviesOrdered;
    }
}
```

```
1. CableBill january = new CableBill(4);
   MessageBox.Show(january.CalculateAmount(7).ToString());
```

Значение переменной amountOwed?

```
2. CableBill february = new CableBill(7);
   february.payPerViewDiscount = 1;
   MessageBox.Show(february.CalculateAmount(3).ToString());
```

Значение переменной amountOwed?

```
3. CableBill march = new CableBill(9);
   march.Discount = true;
   MessageBox.Show(march.CalculateAmount(6).ToString());
```

Значение переменной amountOwed?

Часто Задаваемые Вопросы

В: Почему имена одних полей начинаются с прописной буквы, а других — со строчной? Это что-то означает?

О: Да, означает. Для вас. Но не для компилятора. C# все равно, какие имена вы выбираете для переменных. Выбор странных имен затруднит чтение кода в будущем. Вы можете запутаться в одноименных переменных, все отличие имен которых заключается в регистре первой буквы. В C# регистр имеет значение. Внутри одного метода можно иметь две переменные с именами `Party` и `party`. Это не помешает компиляции кода. Вот несколько советов по выбору имен для переменных, которые упростят чтение программы в будущем.

1. Имена полей закрытого доступа должны начинаться со строчной буквы.

2. Имена свойств и полей общего доступа должны начинаться с прописной буквы.

3. Имена параметров методов должны начинаться со строчной буквы.

4. В некоторых методах, особенно это касается конструкторов, имена параметров совпадают с именами полей. В итоге параметр маскирует поле, то есть операторы методов, которые используют это имя, ссылаются на параметр, а не на поле. Эта проблема решается при помощи ключевого слова this: достаточно добавить его к имени, и компилятор поймет, что вы имеете в виду поле, а не параметр.

Возьми в руку карандаш

Код содержит ошибки. Напишите, в чем, по вашему мнению, они заключаются и как их исправить.

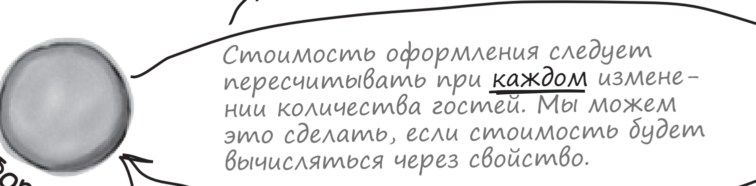
```
class GumballMachine {
       private int gumballs;

       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)
       {
              if (this.coinsInserted >= price) { // проверка поля
                     gumballs -= 1;
                     return "Вот ваша жевательная резинка";
              } else {
                     return "Сумма недостаточна";
              }
       }
}
```

Упражнение

Используем полученную информацию, чтобы заставить программу Кэтлин работать корректно.

1 Заставим счетчик Dinner Party считать правильно

Исправить ошибку можно при условии, что метод `CalculateCostOfDecorations()` вызывается при каждом изменении параметра `NumberOfPeople`. Для этого мы добавим свойство Cost.



Свойство `Cost` при следующем доступе к нему возвращает сумму $650.

Если гарантировать пересчет стоимости при каждом доступе к свойству Cost, нам останется задать параметры объекта DinnerParty и получить его свойство Cost.

2 Зададим параметры вечеринки через свойства.

Мы собираемся пересмотреть класс `DinnerParty`. Возможно, вы захотите создать для этого новой проект. Начните с создания трех автоматических свойств:

```
public int NumberOfPeople { get; set; }

public bool FancyDecorations { get; set; }

public bool HealthyOption { get; set; }
```

Диаграмма для нового класса DinnerParty.

DinnerParty
NumberOfPeople: int FancyDecorations: bool HealthyOption: bool Cost: decimal
private methods: *CalculateCostOfDecorations()* *CalculateCostOfBeverages PerPerson();*

Еще вам нужен **конструктор** с такой сигнатурой:

```
public DinnerParty(int numberOfPeople, bool healthyOption,
                   bool fancyDecorations)
```

3 Создадим закрытые методы для вычисления промежуточной стоимости.

Вот сигнатуры методов, помогающих в расчете стоимости. Добавьте к ним вычисления:

```
private decimal CalculateCostOfDecorations() { ... }

private decimal CalculateCostOfBeveragesPerPerson() { ... }
```

Они очень похожи на методы, которые вы писали в начале этой главы.

4 Добавим предназначенное только для чтения свойство `Cost`.

Напишите код свойства `Cost` для обсчета стоимости вечеринки:

```
public decimal Cost {
    get {
        // Напишите код, вычисляющий стоимость
    }
}
```

Подсказка. Начните с переменной типа decimal с именем totalCost, а затем воспользуйтесь составными операторами += и *=, чтобы отредактировать ее значение.

5 Обновим форму, чтобы воспользоваться свойствами.

Вот полный код формы. Для передачи информации в объект используется конструктор и три свойства (`NumberOfPeople`, `FancyDecoration` и `HealthyOption`), а свойство `Cost` дает конечную стоимость.

```
public partial class Form1 : Form
{
    DinnerParty dinnerParty;
    public Form1()
    {
        InitializeComponent();
        dinnerParty = new DinnerParty((int)numericUpDown1.Value,
                            healthyBox.Checked, fancyBox.Checked);
        DisplayDinnerPartyCost();
    }

    private void fancyBox_CheckedChanged(object sender, EventArgs e)
    {
        dinnerParty.FancyDecorations = fancyBox.Checked;
        DisplayDinnerPartyCost();
    }

    private void healthyBox_CheckedChanged(object sender, EventArgs e)
    {
        dinnerParty.HealthyOption = healthyBox.Checked;
        DisplayDinnerPartyCost();
    }

    private void numericUpDown1_ValueChanged(object sender, EventArgs e)
    {
        dinnerParty.NumberOfPeople = (int)numericUpDown1.Value;
        DisplayDinnerPartyCost();
    }

    private void DisplayDinnerPartyCost()
    {
        decimal Cost = dinnerParty.Cost;
        costLabel.Text = Cost.ToString("c");
    }
}
```

Форма хранит экземпляр объекта DinnerParty и обновляет его свойства при каждом изменении количества участников или других параметров вечеринки.

Форма использует конструктор объекта `DinnerParty` для присвоения корректных начальных значений. Вы должны гарантировать наличие данного конструктора в классе `DinnerParty`.

Метод обновляет отображаемую в форме стоимость вечеринки, обращаясь к свойству Cost при каждом обновлении формы.

Форма стала проще, так как теперь нам не нужно обращаться к методам, которые выполняют вычисления. Вычисления инкапсулированы в свойстве `Cost`.

Этот принцип называется “разделением ответственности” и часто используется в программах. Форма имеет отношение к пользовательскому интерфейсу, в то время как объект DinnerParty связан с вычислением стоимости.

Обратили внимание, как мало функций у новой формы? Она всего лишь задает свойства объектов на основе введенных пользователем данных и в зависимости от этих свойств корректирует конечный результат. Подумайте, как код пользовательского ввода и вывода отделен от кода, выполняющего вычисления.

```
class DinnerParty {
    public const int CostOfFoodPerPerson = 25;

    public int NumberOfPeople { get; set; }

    public bool FancyDecorations { get; set; }

    public bool HealthyOption { get; set; }

    public DinnerParty(int numberOfPeople, bool healthyOption, bool fancyDecorations) {
        NumberOfPeople = numberOfPeople;
        FancyDecorations = fancyDecorations;
        HealthyOption = healthyOption;
    }

    private decimal CalculateCostOfDecorations() {
        decimal costOfDecorations;
        if (FancyDecorations)
        {
            costOfDecorations = (NumberOfPeople * 15.00M) + 50M;
        }
        else
        {
            costOfDecorations = (NumberOfPeople * 7.50M) + 30M;
        }
        return costOfDecorations;
    }

    private decimal CalculateCostOfBeveragesPerPerson() {
        decimal costOfBeveragesPerPerson;
        if (HealthyOption)
        {
            costOfBeveragesPerPerson = 5.00M;
        }
        else
        {
            costOfBeveragesPerPerson = 20.00M;
        }
        return costOfBeveragesPerPerson;
    }

    public decimal Cost {
        get {
            decimal totalCost = CalculateCostOfDecorations();
            totalCost += ((CalculateCostOfBeveragesPerPerson()
                          + CostOfFoodPerPerson) * NumberOfPeople);
            if (HealthyOption)
            {
                totalCost *= .95M;
            }
            return totalCost;
        }
    }
}
```

Эти свойства задаются в конструкторе, обновляются формой и применяются при расчете конечной стоимости.

Это конструктор DinnerParty. Он задает три свойства, базируясь на переданных ему значениях.

Сделав этот метод закрытым, вы гарантировали отсутствие доступа к нему извне, тем самым исключив вероятность его некорректного применения.

Закрытые методы, участвующие в расчете стоимости, обращаются к свойствам, получая при этом самую свежую информацию из формы.

Присутствовавший в первой версии программы метод SetHealthyOption() превратился в свойство HealthyOption.

Если у вас есть метод, начинающийся с «Set», который задает поле, а затем обновляет состояние объекта, превращение его в свойство сделает его предназначение более очевидным.

Это один из способов, которыми инкапсуляция упрощает понимание и повторное применение классов.

Теперь, когда вычисления стали закрытыми и инкапсулировались в свойстве Cost, форма потеряла возможность пересчитывать стоимость оформления без учета текущих параметров. Мы исправили ошибку, из-за которой Кэтлин чуть не лишилась одного из клиентов!

Возьми в руку карандаш. Решение

Вот какие значения должна иметь переменная `amountOwed` после выполнения кода:

```
1. CableBill january = new CableBill(4);
   MessageBox.Show(january.CalculateAmount(7).ToString());
```

Значение переменной amountOwed? **28**

```
2. CableBill february = new CableBill(7);
   february.payPerViewDiscount = 1;
   MessageBox.Show(february.CalculateAmount(3).ToString());
```

Значение переменной amountOwed? **не компилируется**

```
3. CableBill march = new CableBill(9);
   march.Discount = true;
   MessageBox.Show(march.CalculateAmount(6).ToString());
```

Значение переменной amountOwed? **42**

Возьми в руку карандаш. Решение

Вот какие ошибки содержит код:

Имя price относится не к полю, а к параметру конструктора. Эта строчка присваивает ПАРАМЕТРУ значение, возвращаемое методом чтения Price, который еще даже не задан! Строчка станет корректной, если заменить имя параметра конструктора на Price (с прописной буквы).

Ключевое слово this у неверной переменной gumballs. this.gumballs — это свойство, в то время как gumballs — это параметр.

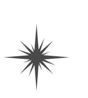
Этот параметр маскирует закрытое поле Price, в то время как метод должен проверять значение вспомогательного поля price.

```
public GumballMachine(int gumballs, int price)
{
        gumballs = this.gumballs;
        price = Price;
}
public string DispenseOneGumball(int price, int coinsInserted)
{
        if (this.coinsInserted >= price) { // проверка поля
             gumballs -= 1;
             return "Вот ваша жевательная резинка";
        } else {
             return "Сумма недостаточна";
        }
}
```

Ключевое слово this не имеет отношения к данному параметру. Оно должно стоять рядом с параметром price.

Потратьте пару минут, чтобы изучить этот код. Он демонстрирует наиболее распространенные ошибки, которые новички допускают при работе с объектами.

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, какое совершенство можно было бы получить, ***изменив всего несколько элементов***? Именно по этой причине **наследование** является одним из самых мощных инструментов C#. В этой главе вы узнаете, как **производный класс** повторяет поведение родительского, сохраняя при этом **гибкость** редактирования. Вы научитесь **избегать дублирования кода** и **облегчите последующее редактирование своих программ**.

Организация дней рождения — это тоже работа Кэтлин

Созданная нами программа работает, и Кэтлин с ней не расстается. Теперь она организует не только званые обеды, но и дни рождения, а их стоимость рассчитывается по другой схеме. Эту схему хотелось бы добавить в программу.

Ваша программа сможет рассчитать стоимость дня рождения?

Эти пункты остались без изменений.

- За каждого гостя оплата $25.
- Обычное оформление стоит $7.50 на человека плюс первоначальный взнос $30. Стоимость особого оформления увеличивается до $15 на человека, а первоначальный взнос — $50.
- Если участников меньше четырех, готовим (8-дюймовый) торт ($40), более четырех — (16-дюймовый) торт ($75).
- Надпись на торте по $0.25 за букву. На первом торте помещается до 16 букв, на втором — до 40.

Приложение должно рассчитывать стоимость вечеринок обоего типа. Каждую форму расчета нужно поместить на свою вкладку.

Теперь нам нужно учитывать стоимость тортов и надписей.

Для дней рождения отсутствует «здоровый» вариант. Подумайте, к каким ошибкам это приведет, если в новый проект вставить код, скопированный из написанного в прошлой главе класса DinnerParty?

Нам нужен класс BirthdayParty

Чтобы программа получила возможность рассчитывать стоимость вечеринок другого типа, нам нужно поменять форму.

BirthdayParty
NumberOfPeople CostOfDecorations CakeSize CakeWriting Cost

Вы займетесь этим через минуту, а пока представим общую картину.

Вот как мы это будем делать:

1. **Создание класса BirthdayParty**
   Новый класс будет подсчитывать стоимость, исходя из выбранного варианта оформления, а также количества букв на торте.

2. **Добавление к форме элемента TabControl**
   Работать с вкладками просто. Выберите нужную и перетащите на нее элементы управления.

3. **Перетаскивание элементов управления Dinner Party на первую вкладку**
   После этого они будут работать точно так же, как и раньше, просто чтобы их увидеть, потребуется перейти на нужную вкладку.

4. **Добавление элементов управления Birthday Party на вторую вкладку**
   Вы выберете интерфейс для расчета стоимости дней рождения так же, как создавали в свое время интерфейс для расчета стоимости званых обедов.

5. **Связывание нового класса с элементами управления**
   Вам потребуется добавить ссылку `BirthdayParty` на поля формы и код для новых элементов управления.

Часто задаваемые вопросы

В: Почему нельзя просто создать экземпляр `DinnerParty`, как это делал Майк, когда ему потребовалось сравнить три маршрута?

О: Потому что новый экземпляр класса `DinnerParty` годится только для расчета стоимости званых обедов. Два экземпляра одного класса полезны, когда требуется работать с данными одного типа. Но для хранения **других данных** вам потребуется **другой класс**.

В: И что же мне поместить в этот новый класс?

О: Перед тем как приступить к созданию класса, нужно понять, какую проблему он будет решать. В данном случае вы должны поговорить с Кэтлин, ведь это она будет пользоваться программой. Впрочем, у вас есть ее заметки! Определить поля, методы и свойства класса можно, продумав его поведение (что он ***должен делать***) и его состояние (что он ***должен знать***).

Планировщик мероприятий, версия 2.0

Начнем новый проект — сделаем для Кэтлин версию программы, которая сможет рассчитать стоимость дней рождения *и* званых обедов. Начнем с инкапсулированного класса `BirthdayParty`, который и будет выполнять все расчеты.

Поля и свойства, содержащие информацию о денежных суммах, должны принадлежать типу decimal.

BirthdayParty
NumberOfPeople: int FancyDecorations: bool Cost: decimal CakeWriting: string CakeWritingTooLong: bool *private ActualLength: int*
private methods: *CalculateCostOfDecorations()* *CakeSize()* *MaxWritingLength()*

1 **Добавим к программе класс BirthdayParty.**
Вы уже знаете, что делать со свойствами `NumberOfPeople` и `FancyDecorations`. Они аналогичны свойствам класса `DinnerParty`. Вставим их в наш новый класс, а затем добавим остальное поведение.

★ Добавьте константу `CostOfFoodPerPerson` и свойства `NumberOfPeople` и `FancyDecorations`. Также нам потребуется **закрытое свойство** типа **int** с именем `actualLength`. (Да, свойства тоже могут быть закрытыми!)

```
class BirthdayParty
{
    public const int CostOfFoodPerPerson = 25;

    public int NumberOfPeople { get; set; }

    public bool FancyDecorations { get; set; }

    public string CakeWriting { get; set; }

    public BirthdayParty(int numberOfPeople,
                         bool fancyDecorations, string cakeWriting)
    {
        NumberOfPeople = numberOfPeople;
        FancyDecorations = fancyDecorations;
        CakeWriting = cakeWriting;
    }
```

Для инициализации объекта BirthdayParty необходимы данные о количестве гостей, виде оформления и надписи на торте. В этом случае стоимость будет правильно рассчитана.

Конструктор задает состояние объектов, определяя свойства таким образом, чтобы впоследствии можно было рассчитать стоимость.

Свойство CakeWriting добавим на следующей странице.

- Для хранения сведений о надписи на торте потребуется свойство CakeWriting типа string. Его метод чтения возвращает содержимое вспомогательного поля cakeWriting.
- Метод записи CakeWriting сначала задает поле cakeWriting, а затем проверяет, не слишком ли велика длина строки, и делает поле actualLength соотетствующим количеству добавленных букв.
- Метод записи CakeWriting должен знать размер торта (зависит от количества приглашенных) и максимально допустимое количество букв (зависит от размера торта). Добавим методы вычисления этих параметров.

Если надпись слишком велика, закрытое свойство ActualLength вычислит, сколько букв помещается на торте.

Свойства тоже могут быть закрытыми. Это свойство обладает только методом чтения, который определяет реальную длину надписи, чтобы использовать ее при вычислении.

```
private int ActualLength
{
    get
    {
        if (CakeWriting.Length > MaxWritingLength())
            return MaxWritingLength();
        else
            return CakeWriting.Length;

    }
}
```

Этот блок if/else проверяет длину надписи и обновляет поле actualLength, присваивая ему количество букв, которые могут поместиться на торте.

```
private int CakeSize() {
    if (NumberOfPeople <= 4)
        return 8;
    else
        return 16;
}

private int MaxWritingLength()
{
    if (CakeSize() == 8)
        return 16;
    else
        return 40;
}
```

Обратили внимание, что часть скобок отсутствует? При наличии в блоке кода всего одного оператора скобки можно опустить.

Для блоков, состоящих из одной строки, фигурные скобки необязательны

После оператора if или внутри цикла while часто находится всего один оператор. Только представьте себе, сколько скобок появилось бы при наличии многочисленных проверок условия и циклов в программе, если этот единственный оператор обязательно требовалось бы заключать в скобки. Поэтому в C# вполне допустима запись:

```
for (int i = 0; i < 10; i++)
    DoTheJob(i);
```

```
if (myValue == 36)
    myValue *= 5;
```

Продолжим работу над классом BirthdayParty...

★ Завершим создание класса BirthdayParty, добавив к нему свойство Cost. В классе DinnerParty к стоимости оформления прибавлялась стоимость напитков, а теперь нужно добавить еще и стоимость торта.

Свойство возвращает true, если надпись не помещается на торте. Мы его используем, чтобы сказать Кэтлин "TOO LONG".

```
    public bool CakeWritingTooLong
    {
        get
        {
            if (CakeWriting.Length > MaxWritingLength())
                return true;
            else
                return false;
        }
    }

    private decimal CalculateCostOfDecorations()
    {
        decimal costOfDecorations;
        if (FancyDecorations)
            costOfDecorations = (NumberOfPeople * 15.00M) + 50M;
        else
            costOfDecorations = (NumberOfPeople * 7.50M) + 30M;
        return costOfDecorations;
    }

    public decimal Cost
    {
        get
        {
            decimal totalCost = CalculateCostOfDecorations();
            totalCost += CostOfFoodPerPerson * NumberOfPeople;
            decimal cakeCost;
            if (CakeSize() == 8)
                cakeCost = 40M + ActualLength * .25M;
            else
                cakeCost = 75M + ActualLength * .25M;
            return totalCost + cakeCost;
        }
    }

}
```

Это свойство обладает только методом чтения, но не меняет состояние объекта. Оно всего лишь использует поля и методы для вычисления логического значения.

Этот метод аналогичен методу, которым мы пользовались в классе DinnerParty.

Класс BirthdayParty обладает свойством Cost типа decimal, как и класс DinnerParty. Но вычисления ведутся при помощи метода CakeSize() и поля actualLength (которое задано свойством CakeWriting).

Вернитесь на предыдущую страницу и посмотрите, каким образом свойство CakeWriting задает поле actualLength. Если надпись не помещается, переменной actualLength присваивается максимально допустимое для данного торта значение. Как только надпись достигает предела, рост стоимости прекращается.

2 **Элемент управления TabControl**

Перетащите на форму элемент управления `TabControl` и измените его размер. Измените название каждой вкладки с помощью свойства `TabPages`. Редактирование свойств каждой вкладки осуществляется в отдельном окне диалога. При помощи свойства `Text` присвойте вкладкам имена Dinner Party и Birthday Party соответственно.

Перемещайтесь между вкладками щелчками. С помощью свойства TabPages поменяйте текст каждой вкладки. Щелкните на расположенной рядом со вкладкой кнопке "..." и выберите свойство Text.

3 **Вставка элементов управления на вкладку Dinner Party**

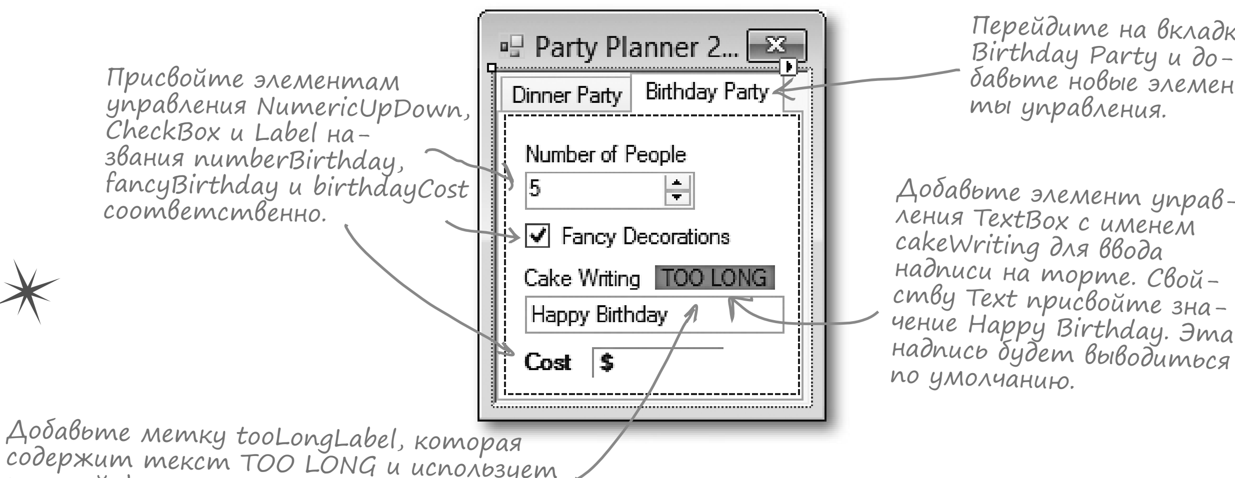
Откройте программу Party Planner (из главы 5) в отдельном окне IDE. Выделите элементы управления, скопируйте и **вставьте на вкладку Dinner Party**. Щелкните **внутри** вкладки, чтобы гарантировать правильное размещение элементов (в противном случае вы увидите сообщение о невозможности добавить компоненты в контейнер типа `TabControl`).

Эти элементы управления доступны только на вкладке Dinner Party.

Таким способом вы добавите только элементы управления, **а не связанные с ними обработчики событий**. Нужно также проверить свойство (Name) в окне Properties для каждого элемента. Убедитесь, что все элементы управления сохранили имена, а также произведите двойной щелчок на каждом из них, чтобы добавить пустой обработчик событий.

4 **Пользовательский интерфейс для вкладки Birthday Party**

На вкладке Birthday Party должен присутствовать элемент `NumericUpDown` для количества гостей, элемент `CheckBox` для особого оформления и элемент `Label` с трехмерной рамкой для итоговой суммы. Кроме того, потребуется элемент управления `TextBox` для ввода надписи на торте.



Продолжим работу над формой...

5 **Соединяем все вместе.**

Все отдельные части уже готовы, осталось только написать код, который заставит форму работать.

- ★ Вам потребуются поля со ссылками на объекты `BirthdayParty` и `DinnerParty`, которым будут присвоены начальные значения при помощи конструктора.
- ★ Код для обработчиков событий, связанных с различными элементами управления со вкладки Dinner Party, возьмите из главы 5. Вот как должен выглядеть код для формы:

```
public partial class Form1 : Form {
    DinnerParty dinnerParty;
    BirthdayParty birthdayParty;
    public Form1() {
        InitializeComponent();
        dinnerParty = new DinnerParty((int)numericUpDown1.Value,
                          healthyBox.Checked, fancyBox.Checked);
        DisplayDinnerPartyCost();

        birthdayParty = new BirthdayParty((int)numberBirthday.Value,
                          fancyBirthday.Checked, cakeWriting.Text);
        DisplayBirthdayPartyCost();
    }

    // Обработчики событий для fancyBox, healthyBox и numericUpDown1
    // а также метод DisplayDinnerCost() аналогичны показанным
    // в упражнении Dinner Party в конце главы 5.
```

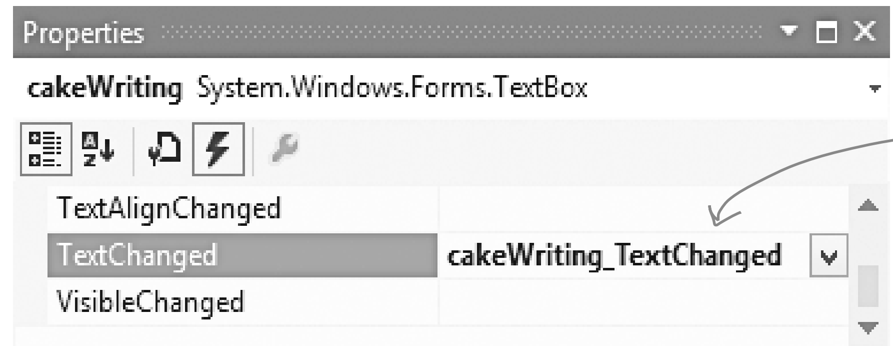
Связанный с формой конструктор присваивает экземпляру BirthdayParty начальные значения подобно тому, как это было сделано для экземпляра DinnerParty.

- ★ Добавьте код к обработчику событий элемента `NumericUpDown`, чтобы задать свойство `NumberOfPeople` и заставить функционировать флажок Fancy Decorations.

```
    private void numberBirthday_ValueChanged(object sender, EventArgs e) {
        birthdayParty.NumberOfPeople = (int)numberBirthday.Value;
        DisplayBirthdayPartyCost();
    }

    private void fancyBirthday_CheckedChanged(object sender, EventArgs e) {
        birthdayParty.FancyDecorations = fancyBirthday.Checked;
        DisplayBirthdayPartyCost();
    }
```

Обработчики событий элементов CheckBox и NumericUpDown такие же, как и для вкладки dinner party.

★ Откройте страницу Events в окне Properties и добавьте к текстовому полю `cakeWriting` новый обработчик событий `TextChanged`. Щелкните на кнопке со значком молнии для перехода к списку событий. Выделите элемент TextBox и найдите в списке событие TextChanged. Двойным щелчком на нем добавьте новый обработчик события.



Когда вы выделите поле cakeWriting и дважды щелкнете на строчке TextChanged в списке Events окна Properties, IDE добавит обработчик событий, который будет запускаться при каждом изменении текста в поле.

```
private void cakeWriting_TextChanged(object sender, EventArgs e) {
    birthdayParty.CakeWriting = cakeWriting.Text;
    DisplayBirthdayPartyCost();
}
```

★ Добавьте метод `DisplayBirthdayPartyCost()` ко всем обработчикам событий, чтобы итоговая сумма автоматически обновлялась при любых изменениях.

Свойство Visible позволяет отображать и прятать элементы управления.

Класс BirthdayParty обладает свойством, которое позволяет форме отобразить предупреждение.

Способ, которым форма обрабатывает надпись на торте, очень прост, потому что класс BirthdayParty **инкапсулирован**. Форма должна всего лишь задать свойства объекта. Все остальное объект сделает самостоятельно.

```
    private void DisplayBirthdayPartyCost() {
        tooLongLabel.Visible = birthdayParty.CakeWritingTooLong;
        decimal cost = birthdayParty.Cost;
        birthdayCost.Text = cost.ToString("c");
    }
}
```

Способ обработки надписи, количества гостей и размера торта встроен в методы записи NumberOfPeople и CakeWriting, поэтому форма должна всего лишь передать им нужные значения.

...вот вы и закончили создание формы!

6 Программа готова!

Убедитесь, что программа работает корректно. Если надпись на торте слишком длинная, должно появляться окно с сообщением. Проверьте правильность расчета конечной суммы. Если все функционирует, значит, работа сделана!

Party Planner 2...
Dinner Party | Birthday Party
Number of People: 5
[x] Fancy Decorations
[ ] Healthy Option
Cost $350.00

Запустите программу и откройте вкладку Dinner Party. Убедитесь, что она работает точно так же, как и старая программа Party Planner.

Проверим правильность расчетов для 10 гостей. Еда $25 x 10 = $250, 16-дюймовый торт стоит $75, обычное оформление: $7.50 x 10 = $75, единовременный взнос: $30, 21 буква на торте по цене $.25 за букву = $5.25.

Перейдите на вкладку Birthday Party. Убедитесь, что значение поля Cost меняется при изменении количества гостей и установке флажка Fancy Decorations.

Party Planner 2...
Dinner Party | Birthday Party
Number of People: 5
[x] Fancy Decorations
Cake Writing: Happy Birthday
Cost $328.50

Итого $250 + $75 + $75 + $30 + $5.25 = $435.25. Все правильно!

Party Planner 2...
Dinner Party | Birthday Party
Number of People: 4
[x] Fancy Decorations
Cake Writing TOO LONG
Happy Birthday Myrtle
Cost $254.00

При вводе информации в поле Cake Writing обработчик событий TextChanged должен обновлять значение поля Cost.

Party Planner 2...
Dinner Party | Birthday Party
Number of People: 10
[ ] Fancy Decorations
Cake Writing: Happy Birthday Myrtle
Cost $435.25

Если надпись слишком велика, класс BirthdayParty присваивает свойству CakeWritingTooLong значение true и вычисляет стоимость для максимально допустимой длины. Форме не нужно производить никаких вычислений.

Дополнительный взнос за мероприятия с большим количеством гостей

Благодаря программе дела Кэтлин пошли в гору, и теперь она может позволить себе брать дополнительную плату за мероприятия с очень большим количеством гостей (более 12 человек). Как же добавить к программе еще один платеж?

- ★ Метод `DinnerParty.Cost` должен проверять значение переменной `NumberOfPeople`, и если возвращаемое значение превышает 12, добавлять $100.
- ★ Аналогично для `BirthdayParty.Cost`.

Подумайте, как добавить еще один платеж к классам `DinnerParty` и `BirthdayParty`. Какой код следует написать? С какими элементами этот код должен быть связан?

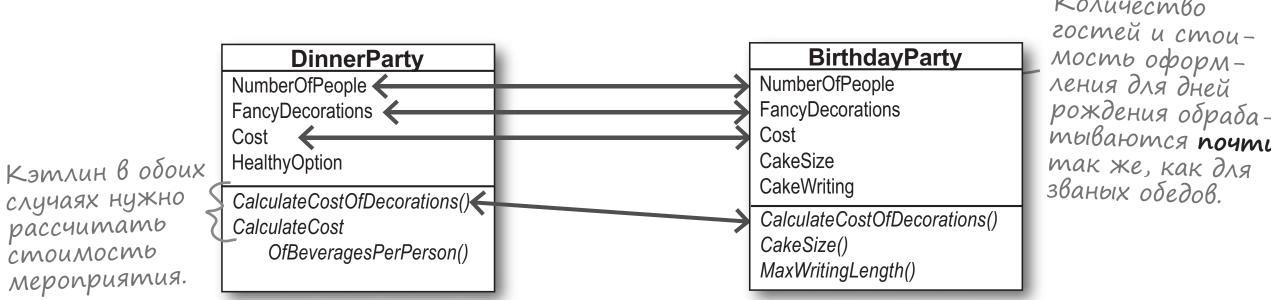
Кажется, что это просто... но что может случиться при сосуществовании трех одинаковых классов? А четырех? А двенадцати? А что, если в будущем вам потребуется отредактировать код? Вы представляете себе, как трудно ***менять одинаковым образом*** множество ***родственных*** классов?

Вы правы! Повторение одного и того же кода в разных классах неэффективно. И к тому же увеличивается вероятность ошибок.

К счастью, в C# существует более продуктивный способ создания связанных друг с другом классов: ***наследование (inheritance)***.

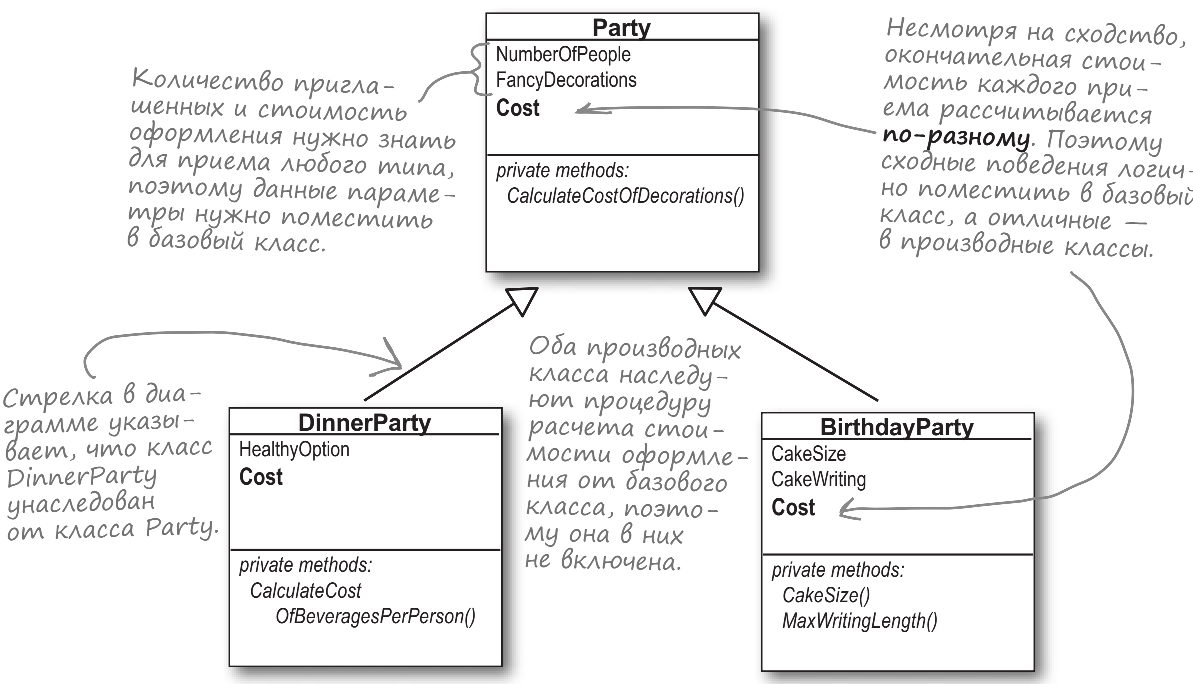
Наследование

Классы `DinnerParty` и `BirthdayParty` не случайно имеют одинаковый код. При написании программ C# часто создаются классы, соответствующие процессам из реального мира, и эти процессы, как правило, связаны друг с другом. Ваши классы имеют **одинаковый код**, так как процессы, взятые за их основу (дни рождения и званые обеды), имеют **одинаковые признаки**.



Званые обеды и дни рождения относятся к приемам

Если существует несколько классов, являющихся частными случаями другого, более общего класса, можно заставить их **наследовать** от этого класса. При этом они становятся классами, **производными (subclass)** от **базового (base class)**.



Модель классов: от общего к частному

Наследование в C# является копией процессов, происходящих в реальном мире. **Иерархия** присутствует в виде перехода от общих вещей к частностям. В модели классов классы, расположенные ниже в иерархии, **наследуют** от вышестоящих.

> **На-сле-до-вать, гл.**
> иметь признаки родителя или предка. *Она хочет, чтобы ребенок* ***унаследовал*** *ее большие карие глаза.*

Симулятор зоопарка

Львы, тигры и медведи... о боже! А еще бегемоты, волки и случайно затесавшаяся кошка. Вам нужно написать симулятор зоопарка. (Не бойтесь, сам код писать не потребуется, на данном этапе достаточно создать диаграмму классов, представляющую всех животных).

Мы получили небольшой список животных, которые должны попасть в программу. Каждому животному будет соответствовать объект со специфическими свойствами.

Программа должна легко читаться и редактироваться другими программистами, если позднее потребуется добавить другие классы или других животных.

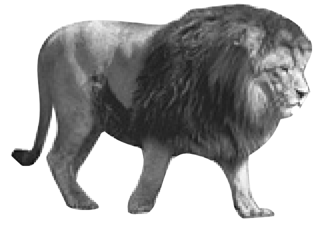
С чего же начать? Перед обсуждением **отдельных** животных нужно понять, что они имеют **общего**: выделить характеристики, подходящие ***всем*** видам. Именно на их основе будет построен класс, от которого будут наследовать другие классы.

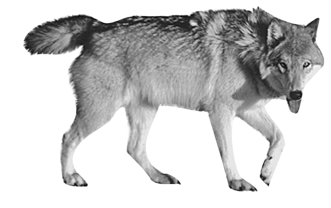
Предок, суперкласс и базовый класс обозначают одно и то же. Аналогичная ситуация с терминами «потомок» и «подкласс».

Некоторые называют класс, расположеный наверху дерева наследования, «базовым»... при этом он не является САМЫМ верхним, так как все классы наследуют от класса Object или его подкласса.

1 **Что у животных общего**

Посмотрите на шесть животных. Как связаны лев, бегемот, тигр, кошка, волк и собака? Именно эти общие для всех свойства лягут в основу базового класса.





Наследование позволяет избежать дублирования кода в производных классах

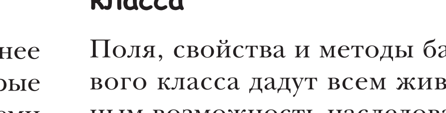
Так как дублирующийся код сложно редактировать и еще сложнее читать, выберем методы и поля для базового класса Animal, которые будут написаны **только один раз** и которые будут унаследованы всеми производными классами. Начнем с полей общего доступа:

- ★ `Picture`: картинка, которую можно поместить в PictureBox.
- ★ `Food`: тип пищи. Пока у этого поля только два значения: meat (мясо) и grass (трава).
- ★ `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.

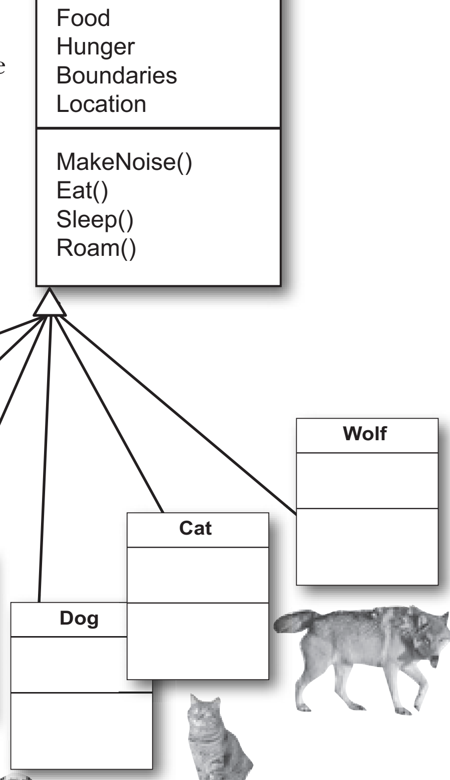
Кроме того, в классе Animal присутствуют четыре метода, которые могут быть унаследованы:

- ★ `MakeNoise()`: метод, позволяющий издавать звуки.
- ★ `Eat()`: поведение при получении предпочитаемого корма.
- ★ `Sleep()`: метод, заставляющий животное спать.
- ★ `Roam()`: метод, учитывающий перемещения по вольеру.

❷ Построение базового класса



Поля, свойства и методы базового класса дадут всем животным возможность наследовать общее состояние и поведение. Логично, что этот класс должен называться Animal (Животное).



Можно было сделать и другой выбор. Например, написать класс ZooOccupant, учитывающий расходы на содержание животных, или класс Attraction, показывающий привлекательность для посетителей. Но мы остановились на классе Animal. Вы согласны?

Животные издают звуки

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)**.

Свойства и методы из базового класса Animal необязательно использовать в производных классах в неизменном виде. Вы можете вообще их не использовать!

Что вам нужно перекрыть?

Все животные едят. Но если собака любит мясо, то бегемоту подавай воз травы. Как может выглядеть код подобного поведения? Как собака, так и бегемот перекроют метод `Eat()`. Бегемота этот метод заставить потреблять, скажем, 10 кг травы за раз. В то время как вызванный для собаки метод `Eat()` уменьшит запасы пищи в зоопарке на одну банку собачьего корма весом 300 г.

*Производный класс наследует от базового все его поведения, но вы можете их **отредактировать**.*

3 Что каждое животное из класса Animal делает по-своему или не делает вообще?

Что животное из каждого производного класса делает такого, чего остальные животные не делают? Если собака ест собачью еду, то ее метод `Eat()` должен перекрыть метод `Animal.Eat()`. Бегемоты умеют плавать, поэтому для них определен метод `Swim()`, который в классе Animal вообще отсутствует.

Сено — это вкусно! Я бы съел стог-другой прямо сейчас.

Сено?! Нет, мне хочется мяса!

Animal
Picture Food Hunger Boundaries Location
MakeNoise() Eat() Sleep() Roam()

Для некоторых животных требуется перекрыть методы **`MakeNoise()`** и **`Eat()`**. А для кого нужно перекрыть метод **`Sleep()`** или **`Roam()`**? И нужно ли это делать вообще? Подумайте также, для каких животных будут перекрываться свойства.

Разбиваем животных на группы

«Выдержанный вермонтский чеддер» — это вид сыра, который относится к ежедневно потребляемым продуктам и в свою очередь входит в категорию «еда». Эта последовательность представлена наглядной моделью классов. К счастью для нас, в C# такие вещи легко сделать. Можно создать цепочку классов, наследующих друг от друга. И вы получите базовый класс `Food` с производным классом `DairyProduct`, который в свою очередь является базовым для класса `Cheese`, содержащего в себе производный класс `Cheddar`, передающий свои признаки классу `AgedVermontCheddar`.

4 **Поиск классов, имеющих много общего**

Вам не кажется, что волки и собаки во многом похожи? Они относятся к семейству псовых и имеют сходное поведение. Скорее всего, им придется по вкусу одна и та же еда. И спят они одинаковым способом. Теперь рассмотрим домашних кошек, тигров и львов. Они одинаково относятся к местам своего обитания. Наверное, имеет смысл создать для них базовый класс `Feline` (Кошачьи), который будет производным от класса `Animal`. Это позволит избежать дублирования кода.

Иерархия классов

Конструкция, в которой под базовым классом располагаются производные классы, в свою очередь являющиеся базовыми для других классов, называется **иерархией (class hierarchy)**. Подобный подход не только позволяет избежать многократного дублирования кода, но и делает код намного более читабельным. Например, при просмотре кода симулятора зоопарка, наткнувшись на метод или свойство из класса Feline, вы *сразу поймете*, что они имеют отношение к кошкам. Иерархия становится картой, позволяющей отследить происходящее в программе.

5 **Завершение построения иерархии**

Вам осталось добавить классы Feline и Canine, и иерархия будет готова.

Animal
Picture
Food
Hunger
Boundaries
Location
MakeNoise()
Eat()
Sleep()
Roam()

Класс Feline перекрывает метод Roam(), поэтому все унаследованные свойства будут иметь дело с новым методом, а не с тем, который был определен в классе Animal.

Feline
Roam()

Hippo
MakeNoise()
Eat()
Swim()

Волки и собаки одинаково едят и спят, но издают разные звуки.

Canine
Eat()
Sleep()

Lion
MakeNoise()
Eat()

Все кошки двигаются одинаково, поэтому метод Roam() для них общий. Но они по-разному питаются и издают разные звуки, поэтому методы Eat() и MakeNoise(), унаследованные от класса Animal, перекрываются.

Tiger
MakeNoise()
Eat()

Cat
MakeNoise()
Eat()

Волки и собаки питаются одинаково, поэтому их общий метод Eat() помещен в класс Canine.

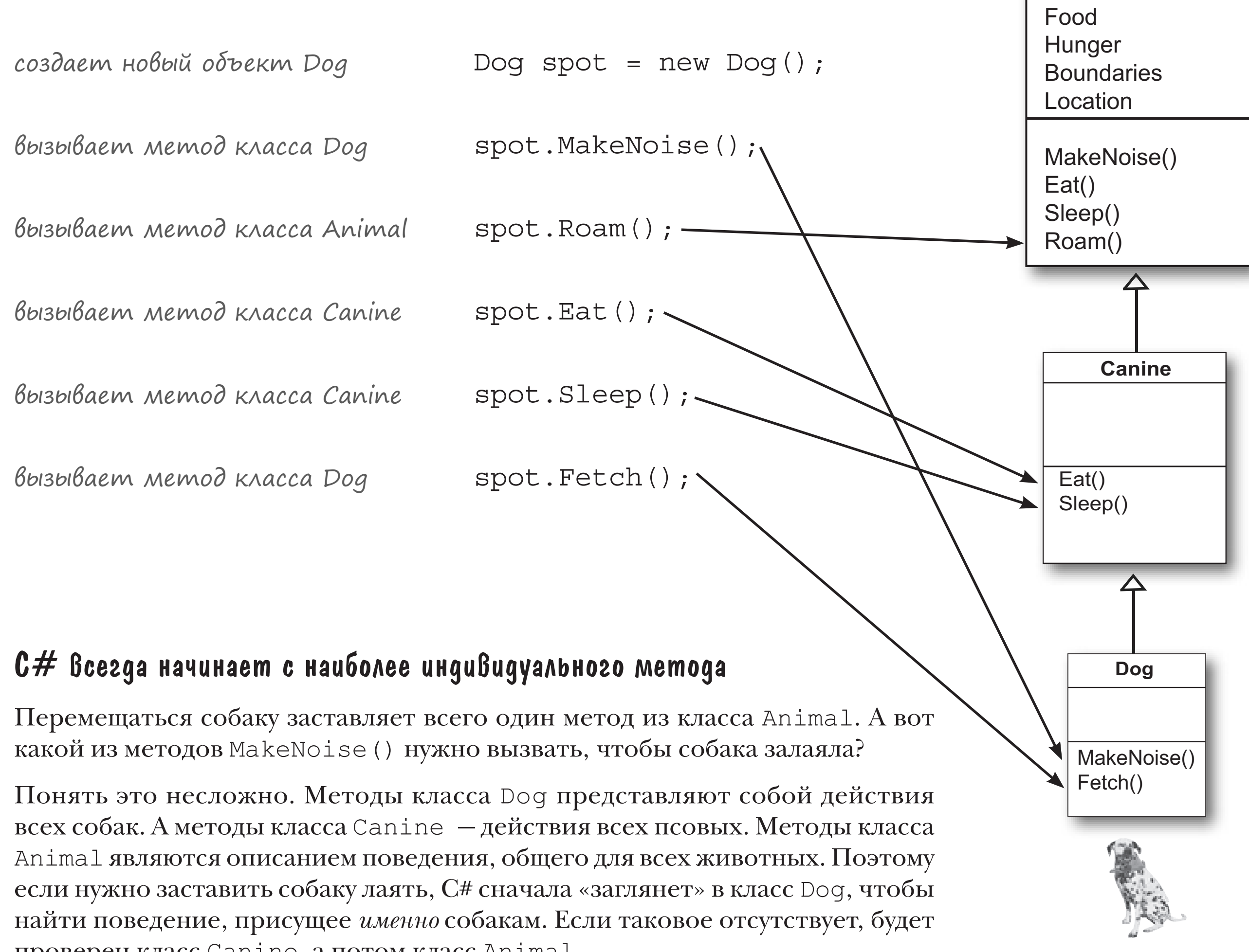
Wolf
MakeNoise()

Dog
MakeNoise()

Производные классы *расширяют* базовый

Вы не ограничены методами, которые производный класс наследует от базового... впрочем, вы это уже знаете! В конце концов, вы же уже создавали классы. А при наследовании класс просто расширяется за счет добавления к базовому классу полей, свойств и методов. Можно легко добавить собакам метод `Fetch()` (Принести дичь). Новый метод не будет ничего наследовать и ничего перекрывать, ведь он определен только для собак и никак не повлияет на классы Wolf, Canine, Animal, Hippo и любые другие.

> и-е-рар-хи-я, сущ.
> расположение групп одна под другой в соответствии с их рангом. *Президент компании прошел весь путь от курьера до верхов корпоративной **иерархии**.*



C# всегда начинает с наиболее индивидуального метода

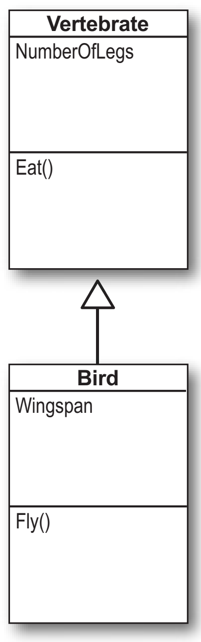
Перемещаться собаку заставляет всего один метод из класса `Animal`. А вот какой из методов `MakeNoise()` нужно вызвать, чтобы собака залаяла?

Понять это несложно. Методы класса `Dog` представляют собой действия всех собак. А методы класса `Canine` — действия всех псовых. Методы класса `Animal` являются описанием поведения, общего для всех животных. Поэтому если нужно заставить собаку лаять, C# сначала «заглянет» в класс `Dog`, чтобы найти поведение, присущее *именно* собакам. Если таковое отсутствует, будет проверен класс `Canine`, а потом класс `Animal`.

Синтаксис наследования

При наследовании имена производного и базового классов разделяются двоеточием (:). Производный класс получает **все поля, свойства и методы** базового класса.

При наследовании все поля, свойства и методы базового класса автоматически добавляются в производный класс.




```
class Vertebrate
{
    public int NumberOfLegs;
    public void Eat() {
       // код, заставляющий есть
    }
}
```

Класс Bird (Птица) наследует от класса Vertebrate (Позвоночное).

```
class Bird : Vertebrate
{
    public double Wingspan;
    public void Fly() {
       // код, заставляющий летать
    }
}
```

Вы расширяете класс, добавив в конец его объявления двоеточие и имя базового класса.

```
public button1_Click(object sender, EventArgs e) {
    Bird tweety = new Bird();
    tweety.Wingspan = 7.5;
    tweety.Fly();
    tweety.NumberOfLegs = 2;
    tweety.Eat();
}
```

Так как tweety — это экземпляр объекта Bird, он имеет все методы и поля этого объекта.

Так как класс Bird — производный по отношению к классу Vertebrate, все экземпляры Bird имеют поля и методы, определенные в классе Vertebrate.

Часто задаваемые вопросы

В: Почему стрелка указывает от производного класса к базовому? Не логичнее было бы рисовать ее наоборот?

О: Это было бы не совсем точно. Заставляя один класс наследовать от другого, вы встраиваете это отношение в производный класс, а базовый остается без изменений. Это имеет смысл, если подумать о происходящем с точки зрения базового класса. Поведение базового класса не только никак не меняется, он даже не знает о появлении производных классов. Методы класса, поля и свойства никак не затрагиваются. А вот производный класс свое поведение меняет. Все его вновь создаваемые экземпляры получают свойства, поля и методы базового класса. И все это происходит благодаря единственному двоеточию! Стрелка на диаграмме является частью производного класса, и поэтому она нацелена на базовый класс.

Возьми в руку карандаш

Посмотрите на эти модели и объявления классов и обведите некорректные операторы.

Aircraft
AirSpeed Altitude
TakeOff() Land()

FirePlane
BucketCapacity
FillBucket()

```
class Aircraft {
    public double AirSpeed;
    public double Altitude;
    public void TakeOff() { ... };
    public void Land() { ... };
}

class FirePlane : Aircraft {
    public double BucketCapacity;
    public void FillBucket() { ... };
}

public void FireFightingMission() {
    FirePlane myFirePlane = new FirePlane();
    new FirePlane.BucketCapacity = 500;
    Aircraft.Altitude = 0;
    myFirePlane.TakeOff();
    myFirePlane.AirSpeed = 192.5;
    myFirePlane.FillBucket();
    Aircraft.Land();
}
```

Sandwich
Toasted SlicesOfBread
CountCalories()

BLT
SlicesOfBacon AmountOfLettuce
AddSideOfFries()

```
class Sandwich {
    public boolean Toasted;
    public int SlicesOfBread;
    public int CountCalories() { ... }
}

class BLT : Sandwich {
    public int SlicesOfBacon;
    public int AmountOfLettuce;
    public int AddSideOfFries() { ... }
}

public BLT OrderMyBLT() {
    BLT mySandwich = new BLT();
    BLT.Toasted = true;
    Sandwich.SlicesOfBread = 3;
    mySandwich.AddSideOfFries();
    mySandwich.SlicesOfBacon += 5;
    MessageBox.Show("В моем сэндвиче"
        + mySandwich.CountCalories + "калорий".);
    return mySandwich;
}
```

Возьми в руку карандаш

Решение

Вот какие операторы следовало обвести как неработающие.

Aircraft
AirSpeed Altitude
TakeOff() Land()

FirePlane
BucketCapacity
FillBucket()

```
class Aircraft {
    public double AirSpeed;
    public double Altitude;
    public void TakeOff() { ... };
    public void Land() { ... };
}

class FirePlane : Aircraft {
    public double BucketCapacity;
    public void FillBucket() { ... };
}

public void FireFightingMission() {
    FirePlane myFirePlane = new FirePlane();
    new FirePlane.BucketCapacity = 500;
    Aircraft.Altitude = 0;
    myFirePlane.TakeOff();
    myFirePlane.AirSpeed = 192.5;
    myFirePlane.FillBucket();
    Aircraft.Land();
}
```

Ключевое слово **new** так использовать нельзя.

Эти операторы используют имя класса вместо имени экземпляра myFirePlane.

Sandwich
Toasted SlicesOfBread
CountCalories()

BLT
SlicesOfBacon AmountOfLettuce
AddSideOfFries()

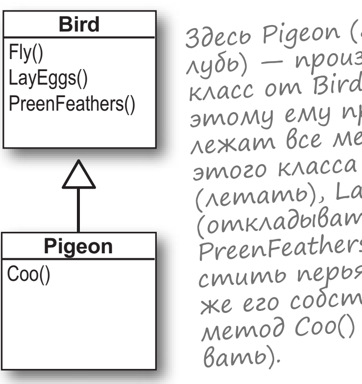
```
class Sandwich {
    public boolean Toasted;
    public int SlicesOfBread;
    public int CountCalories() { ... }
}

class BLT : Sandwich {
    public int SlicesOfBacon;
    public int AmountOfLettuce;
    public int AddSideOfFries() { ... }
}

public BLT OrderMyBLT() {
    BLT mySandwich = new BLT();
    BLT.Toasted = true;
    Sandwich.SlicesOfBread = 3;
    mySandwich.AddSideOfFries();
    mySandwich.SlicesOfBacon += 5;
    MessageBox.Show("В моем сэндвиче "
       + mySandwich.CountCalories + "калорий".);
    return mySandwich;
}
```

Эти свойства принадлежат экземпляру, в то время как операторы пытаются вызывать их по имени классов.

После имени метода CountCalories отсутствуют скобки ().

При наследовании поля свойства и методы базового класса добавляются к производному...

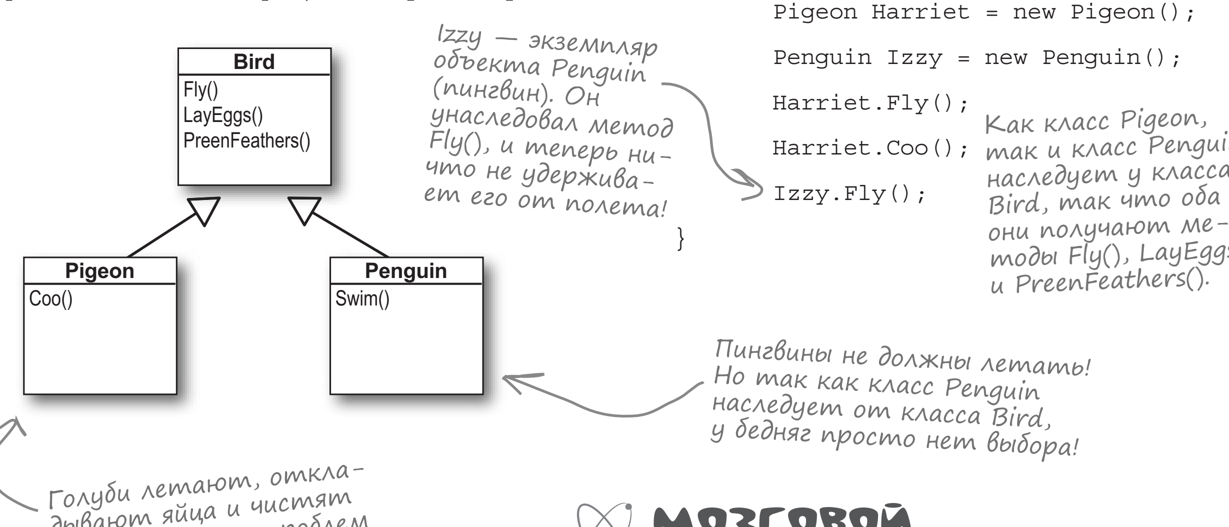
Наследование является простым, если производному классу нужны ***все*** методы, свойства и поля базового класса.




```
class Bird {
   public void Fly() {
    // код, заставляющий птицу летать
   }

   public void LayEggs() { ... };

   public void PreenFeathers() { ... };
}


class Pigeon : Bird {
   public void Coo() { ... }
}


class Penguin : Bird {
   public void Swim() { ... }
}


public void BirdSimulator() {

      Pigeon Harriet = new Pigeon();

      Penguin Izzy = new Penguin();

      Harriet.Fly();

      Harriet.Coo();

      Izzy.Fly();

}
```

...не все птицы летают!

Что делать, если базовый класс имеет метод, который в производном классе требуется ***отредактировать***?



МОЗГОВОЙ ШТУРМ

Если бы этот код писали вы, как бы вы избавили пингвина от необходимости летать?

Перекрытие методов

Иногда нужно, чтобы производный класс унаследовал *не все* поведения базового, а только *часть их*. Чтобы изменить унаследованное ненужное поведение, достаточно **перекрыть (override)** метод.

Ключевое слово virtual

Чтобы производный класс получил возможность перекрывать методы, используйте ключевое слово **virtual**.

```
class Bird {
   public virtual void Fly() {
      // код, заставляющий птицу летать
   }
}
```

Это ключевое слово показывает, что производный класс может перекрыть метод Fly().

2 **Добавление одноименного метода в производный класс**

Переопределенный метод должен иметь такую же сигнатуру, то есть то же самое возвращаемое значение и параметры. В его объявлении используется ключевое слово `override`.

```
class Penguin : Bird {
   public override void Fly() {
     MessageBox.Show("Пингвины не летают!")
   }
}
```

Чтобы перекрыть метод Fly(), добавьте в производный класс идентичный метод и воспользуйтесь ключевым словом override.

При перекрытии сигнатура нового метода должна совпадать с сигнатурой исходного метода из базового класса. В случае с методом Fly это означает отсутствие возвращаемого значения и параметров.

Используйте ключевое слово override для добавления в производный класс методов, замещающих методы унаследованные. Перекрывать можно методы, помеченные в базовом классе словом virtual.

Sandwich
Toasted
SlicesOfBread
CountCalories()

BLT
SlicesOfBacon
AmountOfLettuce
AddSideOfFries()

Вместо базового класса можно взять один из производных

Наследование позволяет использовать производный класс вместо базового. К примеру, если метод `Recipe()` берет объект `Cheese`, в то время как класс `AgedVermontCheddar` наследует от класса `Cheese`, методу `Recipe()` можно передать экземпляр `AgedVermontCheddar`. В результате метод `Recipe()` будет иметь доступ только к полям, методам и свойствам класса `Cheese`, но не «увидит» элементы класса `AgedVermontCheddar`.

1. Допустим, у нас имеется метод, анализирующий объекты `Sandwich`:

```
public void SandwichAnalyzer(Sandwich specimen) {
    int calories = specimen.CountCalories();
    UpdateDietPlan(calories);
    PerformBreadCalculations(specimen.SlicesOfBread, specimen.Toasted);
}
```

2. Методу можно передать объект сэндвич, а можно сэндвич с беконом, салатом и помидорами BLT (сэндвич). Эти свойства мы наследуем от класса `Sandwich`:

```
public button1_Click(object sender, EventArgs e) {
    BLT myBLT = new BLT();
    SandwichAnalyzer(myBLT);
}
```

Подробно об этом мы поговорим в следующей главе.

3. По диаграмме классов всегда можно ***спуститься вниз*** — ссылочной переменной можно присвоить экземпляр одного из производных классов. Но движение ***вверх*** по диаграмме классов запрещено.

```
public button2_Click(object sender, EventArgs e) {
    Sandwich mySandwich = new Sandwich();
    BLT myBLT = new BLT();
    Sandwich someRandomSandwich = myBLT;
    BLT anotherBLT = mySandwich;    // <--- ЭТО НЕ КОМПИЛИРУЕТСЯ!!!
}
```

Значение myBLT можно присвоить любой переменной класса Sandwich, так как BLT — это подвид сэндвича.

Но нельзя присвоить значение mySandwich переменной BLT, так как далеко не каждый сэндвич содержит бекон, лук, помидоры. Поэтому последняя строка компилироваться не будет.

Ниже показана короткая программа, у которой отсутствует фрагмент кода. Вам нужно сопоставить фрагменты кода (слева) с возможными результатами. Не все строчки, приведенные под заголовком «Результат», должны использоваться, хотя некоторые могут использоваться больше одного раза.

Инструкции

1. **Заполните четыре пробела в коде.**
2. **Совместите фрагменты и результаты.**

```
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);
  }
}

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 = "";
    [                    ]
    System.Windows.Forms.MessageBox.Show(q);
  }
}
```

Вот точка входа в программу, она не показывает форму, а только вызывает окно с сообщением.

Подсказка: как следует подумайте, что именно это означает.

Вставьте сюда фрагмент (три строчки)

Фрагменты 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();
```

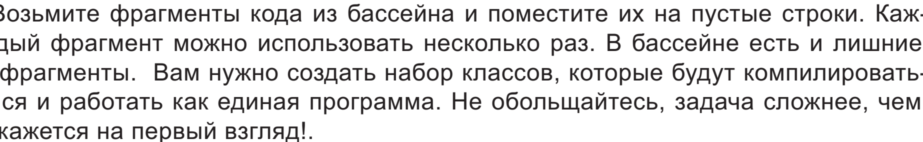
```
q += a2.m1();
q += a2.m2();
q += a2.m3();
```

Результат:

```
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 пользуйтесь средствами IDE, намного полезнее будет решить задачу на бумаге!)

Ребус в бассейне



Возьмите фрагменты кода из бассейна и поместите их на пустые строки. Каждый фрагмент можно использовать несколько раз. В бассейне есть и лишние фрагменты. Вам нужно создать набор классов, которые будут компилироваться и работать как единая программа. Не обольщайтесь, задача сложнее, чем кажется на первый взгляд!.

```
class Rowboat ...................... {
   public .............. rowTheBoat() {
      return "stroke natasha";
   }
}
```

```
class ............... {
   private int ............... ;
   .............. void ........................ (.......) {
      length = len;
   }
   public int getLength() {
      .................... ;
   }
   public ..................... move() {
      return " ............... ";
   }
}
```

```
class TestBoats {
   ................................ Main(){
      ........... xyz = "";
      ........... b1 = new Boat();
      Sailboat b2 = new                    ();
      Rowboat .......... = new Rowboat();
      b2.setLength(32);
      xyz = b1. ..........  ();
      xyz += b3. ..........  ();
      xyz += ......... .move();
      System.Windows.Forms.MessageBox.Show(xyz);
   }
}
class .................... : Boat {
   public ............................        () {
      return " ..................... ";
   }
}
```

Подсказка: это точка входа в программу.

Результат:

drift drift hoist sail

OK

Rowboat
Sailboat
subclasses
;
Boat
Testboats
override
return
continue
virtual
int len
drift
hoist sail
stroke natasha
rowTheBoat
b1
break
:
int b1
int length
string
move
int b2
int b3
void
setLength
b2
b3
length
len
int b2
int
static
public
private
getLength

Смесь сообщений

```
a = 6;   → 56
b = 5;   → 11
a = 5;   → 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() {
```

Вы всегда можете использовать конкретное вместо общего. При наличии строчки кода, в которой требуется класс Canine, можно создать ссылку на класс Dog. Поэтому строчка:

```
A a2 = new C();
```

означает, что вы создаете экземпляр C и ссылку из класса A с именем a2, указывающую на него. Впрочем, имена A, a2 и C не очень наглядны, поэтому приведем несколько примеров со значимыми именами:

```
Sandwich mySandwich = new BLT();

Cheese ingredient= new AgedVermontCheddar();

Songbird tweety = new NorthernMockingbird();
```

```
q += b.m1();
q += c.m2();
q += a.m3();
```
→ B's m1, A's m2, A's m3,

```
q += c.m1();
q += c.m2();
q += c.m3();
```
→ B's m1, A's m2, C's m3, 13

```
q += a.m1();
q += b.m2();
q += c.m3();
```
→ A's m1, A's m2, C's m3, 13

```
q += a2.m1();
q += a2.m2();
q += a2.m3();
```
→ B's m1, A's m2, C's m3, 13

Варианты:

- 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

Решение ребуса в бассейне

```
class Rowboat : Boat {
   public string rowTheBoat() {
      return "stroke natasha";
   }
}
class Boat {
   private int length ;
   public void setLength ( int len )
{
      length = len;
   }
   public int getLength() {
      return length ;
   }
   public virtual string move() {
      return "drift";
   }
}
```

```
class TestBoats {
  public static void Main(){
    string xyz = "";
    Boat b1 = new Boat();
    Sailboat b2 = new Sailboat();
    Rowboat b3 = new Rowboat();
    b2.setLength(32);
    xyz = b1.move();
    xyz += b3.move();
    xyz += b2.move();
    System.Windows.Forms.MessageBox.Show(xyz);
  }
}
class Sailboat : Boat {
  public override string move () {
    return "hoist sail";
  }
}
```

Часто Задаваемые Вопросы

В: В ребусе в бассейне точка входа указывала наружу, означает ли это, что в программе отсутствует форма Form1?

О: Для нового проекта Windows Application IDE создает все необходимые файлы, включая файл Program.cs (содержащий статический класс с точкой входа) и файл Form1.cs (содержащий пустую форму Form1).

Попробуйте при создании нового проекта выбрать вариант Empty Project вместо Windows Application. Добавьте файл с классами через окно Solution Explorer и введите код из решения ребуса в бассейне. Так как в программе должно появляться окно диалога, необходимо добавить ссылку на форму. Щелкните правой кнопкой мыши на строчке References в окне Solution Explorer, выберите команду Add Reference, в открывшемся окне перейдите на вкладку .NET и выберите строчку System.Windows. Forms. (Это IDE делает автоматически при создании проекта Windows Application.) Затем выберите команду Properties в меню Project и укажите в списке output type вариант Windows Application.

Запустите программу и посмотрите на результат! Поздравляем, вы только что создали программу с нуля!

Если вам нужно вспомнить, что такое метод Main() и точка входа, перечитайте начало главы 2.

Окно Class View открывается командой меню View. Обычно окно пристыковано к окну Solution Explorer и показывает классы из вашего решения; в некоторых случаях это очень удобно.

В: Можно ли наследовать от класса, содержащего точку входа?

О: Да. Точка входа ***должна быть*** статическим методом, но этот метод ***может и не принадлежать*** к статическому классу. (Ключевое слово `static` означает невозможность создания экземпляров класса, но его методы доступны с момента запуска программы. В ребусе в бассейне метод `TestBoats.Main()` можно вызвать из любого другого метода без объявления ссылочной переменной или создания экземпляров при помощи оператора `new`.)

В: Я не понимаю, почему эти методы называют «виртуальными», они вполне реальные!

О: Ключевое слово virtual относится к способам обработки методов в .NET. Используется так называемая таблица виртуальных методов, которая отслеживает, какие методы были унаследованы, а какие перекрыты.

Class View
<Search>
SandwichAnalyzer
Project References
SandwichAnalyzer
BLT
Base Types
Sandwich
Form1
Program
Sandwich
CountCalories()
SlicesOfBread
Toasted
Solution Explorer | Class View

Для просмотра членов класса раскройте его в окне Class View.

Воспользуйтесь папкой Base Types из Class View для знакомства с иерархией наследования классов.

В: Почему я могу двигаться по диаграмме классов только вверх?

О: Классы, расположенные в диаграмме сверху, являются более *общими*. Именно от них наследуют более детализированные классы (скажем, `Рубашка` или `Автомобиль` могут наследовать от классов `Одежда` или `Транспорт`). Если вам нужен транспорт, вам подойдет как автомобиль, так и мотоцикл или даже поезд. Но если вам требуется именно автомобиль, вы не сможете выбирать все транспортные средства.

Именно так работает наследование. При наличии метода с параметром `Транспорт` и при условии, что класс Мотоцикл наследует от класса `Транспорт`, вы можете передать методу экземпляр `Мотоцикл`. Если же параметром метода является `Мотоцикл`, вы не сможете передать объект `Транспорт`, так как это может оказаться `Поезд`, и C# не будет знать, что делать, при попытках метода получить доступ к свойству `Руль`.

Методам, параметры которых работают с базовым классом, можно передавать экземпляры производного класса.

Есть важная причина!

Ключевые слова `virtual` и `override` не являются формальностью. Они меняют способ работы вашей программы. Впрочем, мы не заставляем верить нам на слово, давайте рассмотрим пример.

На этот раз вместо приложения Windows Forms будет создано консольное приложение. Оно не имеет формы.

1 **Создайте консольное приложение, выбрав вариант Console application.**
Добавьте пять классов через окно Solution Explorer: `Jewels` (Драгоценности), `Safe` (Сейф), `Owner` (Владелец), `Locksmith` (Слесарь) и `JewelThief` (Вор).

2 **Код для новых классов.**
Добавьте следующий код:

Если при создании нового проекта вместо варианта Windows Forms application выбрать вариант Console Application, IDE создаст только новый класс с именем Program, содержащий пустой метод Main() в качестве точки входа. Вывод информации осуществляется в консольное окно. Данный вид приложений будет подробно разбираться в следующих главах.

```
class Jewels {
    public string Sparkle() {
        return "Sparkle, sparkle!";
    }
}
```

блеск

Объект Safe хранит ссылку Jewels в поле contents (содержимое). Он возвращает эту ссылку только при вызове метода Open() (Открыть) с правильным значением переменной combination (комбинация).

```
class Safe {
    private Jewels contents = new Jewels();
    private string safeCombination = "12345";
    public Jewels Open(string combination)
    {
        if (combination == safeCombination)
            return contents;
        else
            return null;
    }
    public void PickLock(Locksmith lockpicker) {
        lockpicker.WriteDownCombination(safeCombination);
    }
}
```

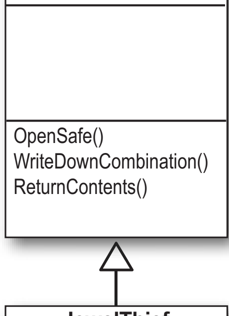
Обратите внимание, что слово private скрывает переменные contents и combination.

Слесарь (Locksmith) может подобрать цифровую комбинацию, вызвав метод PickLock() и передав ему ссылку на себя. При помощи этой комбинации сейф вызывает свой метод WriteDownCombination().

```
class Owner {
    private Jewels returnedContents;
    public void ReceiveContents(Jewels safeContents) {
        returnedContents = safeContents;
        Console.WriteLine("Thank you for returning my jewels! " + safeContents.Sparkle());
    }
}
```

3 **Класс JewelThief наследует от класса Locksmith.**

Похитители драгоценностей – это бывшие слесари! Они могут подобрать код и открыть сейф, но вместо того чтобы вернуть содержимое сейфа владельцу, они крадут его!




```
class Locksmith {
    public void OpenSafe(Safe safe, Owner owner) {
        safe.PickLock(this);
        Jewels safeContents = safe.Open(writtenDownCombination);
        ReturnContents(safeContents, owner);
    }

    private string writtenDownCombination = null;
    public void WriteDownCombination(string combination) {
        writtenDownCombination = combination;
    }

    public void ReturnContents(Jewels safeContents, Owner owner) {
        owner.ReceiveContents(safeContents);
    }
}
```

Метод OpenSafe() из класса Locksmith подбирает ключ, открывает сейф и возвращает его содержимое владельцу.

```
class JewelThief : Locksmith {
    private Jewels stolenJewels = null;
    public void ReturnContents(Jewels safeContents, Owner owner) {
        stolenJewels = safeContents;
        Console.WriteLine("I'm stealing the contents! " + stolenJewels.Sparkle());
    }
}
```

Объект JewelThief наследует методы OpenSafe() и WriteDownCombination(). Но когда метод OpenSafe() вызывает метод ReturnContents(), чтобы вернуть драгоценности владельцу, вор забирает их себе!

4 **Метод Main() для класса Program.**

Пока не запускайте программу! Попробуйте угадать, что будет написано в консоли.

```
class Program {
    static void Main(string[] args) {
        Owner owner = new Owner();
        Safe safe = new Safe();
        JewelThief jewelThief = new JewelThief();
        jewelThief.OpenSafe(safe, owner);
        Console.ReadKey();
    }
}
```

Метод ReadKey() не допускает завершения программы, пока пользователь не нажмет клавишу.

Возьми в руку карандаш

Внимательно посмотрите код программы и запишите сообщение, которое появится в консоли. (Подсказка: определите, какие свойства класс `JewelThief` наследует от класса `Locksmith`!)

Производный класс умеет скрывать методы

Запустите программу JewelThief. Так как это консольное приложение, вывод результатов осуществляется через командное окно. Вот как это выглядит:

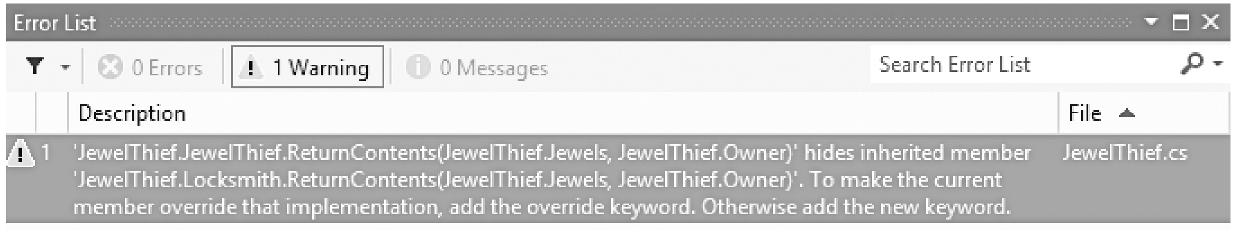
Вы ожидали, что программа напишет кое-что другое? Например:

`I'm stealing the contents! Sparkle, sparkle! (Я краду драгоценности! Круто!)`

Но кажется, наш вор JewelThief повторяет действия слесаря Locksmith! Как это могло произойти?

Сравнение скрытия и перекрытия методов

Причиной, по которой объект `JewelThief` при вызове метода `ReturnContents()` ведет себя как объект `Locksmith`, является способ, которым класс `JewelThief` объявил этот метод. Подсказка находится в предупреждениях, которые посылает приложение:



Так как класс `JewelThief` наследует от класса `Locksmith` и по идее замещает метод `ReturnContents()` своим собственным, кажется, что этот метод перекрывается. Но на самом деле это не так. Объект `JewelThief` скрывает метод `ReturnContents()`.

При скрытии производный класс замещает (технически он «переобъявляет») одноименный метод базового класса. В итоге в производном классе оказываются два метода с одинаковым именем: один унаследованный и один определенный самостоятельно.

Если имя и сигнатура создаваемого метода совпадают с именем и сигнатурой наследуемых методов, новый метод производного класса скрывает метод базового класса.

Для вызова скрытых методов используйте различные ссылки

Объект `JewelThief` скрывает метод `ReturnContents()`, что заставляет его действовать как объект `Locksmith`. Одну версию данного метода объект `JewelThief` наследует от объекта `Locksmith`, затем он определяет вторую собственную версию, и в итоге мы имеем два разных метода с одинаковыми именами. Это означает, что нужны различные способы их вызова.

На самом деле при наличии экземпляра `JewelThief` для вызова нового метода `ReturnContents()` можно использовать ссылочную переменную `JewelThief`. А если для вызова используется ссылочная переменная `Locksmith`, будет вызван скрытый метод `ReturnContents()`.

```
// Производный класс JewelThief скрывает метод базового класса Locksmith,
// поэтому вы можете увидеть поведение одного объекта
// в зависимости от того, какой ссылкой он вызывается!

// При вызове объекта JewelThief ссылочной переменной Locksmith вызывается
// метод ReturnContents() из базового класса
Locksmith calledAsLocksmith = new JewelThief();
calledAsLocksmith.ReturnContents(safeContents, owner);

// При вызове объекта JewelThief ссылочной переменной JewelThief вызывается
// его собственный метод ReturnContents(), а одноименный метод
// из базового класса скрывается.
JewelThief calledAsJewelThief = new JewelThief();
calledAsJewelThief.ReturnContents(safeContents, owner);
```

Скрывая методы, пользуйтесь ключевым словом new

Внимательно прочитайте это предупреждение. Мы знаем, что предупреждения никто не читает, но на этот раз сделайте исключение. **Чтобы осуществить текущую реализацию, добавьте ключевое слово override. Если предполагается сокрытие, используйте ключевое слово new.**

Вернемся в программу и добавим ключевое слово **new**.

```
new public void ReturnContents(Jewels safeContents, Owner owner) {
```

Сразу же после этого сообщение об ошибке исчезнет. Хотя программа все равно не станет работать так, как нужно! По-прежнему вызывается метод `ReturnContents()`, определенный для объекта `Locksmith`. Почему так происходит? Дело в том, что вызов этого метода осуществляется ***через метод, определенный в классе Locksmith***, а именно через `Locksmith.OpenSafe()`, несмотря на то что его начальные значения были заданы в классе `JewelThief`. Если бы `JewelThief` просто **скрывал** метод `ReturnContents()` из базового класса, его собственный метод `ReturnContents()` никогда бы не был вызван.

Подумайте, как заставить объект `JewelThief` перекрыть, а не скрыть метод `ReturnContents()`? Сможете догадаться до того, как перевернете страницу?

Ключевые слова override и virtual

Нам требуется, чтобы класс `JewelThief` всегда использовал свой собственный метод `ReturnContents()`, вне зависимости от способа вызова. Именно так должен работать механизм наследования, и именно это называется **перекрытием**. Вы можете легко заставить класс это сделать. Для начала воспользуйтесь ключевым словом **`override`** при объявлении метода `ReturnContents()`:

```
class JewelThief {
    ...
    override public void ReturnContents
               (Jewels safeContents, Owner owner)
```

Но этого недостаточно. Попытавшись скомпилировать программу, вы получите сообщение об ошибке:

Сообщение предупреждает, что объект `JewelThief` не может перекрыть унаследованный метод `ReturnContents()` из-за отсутствия ключевых слов `virtual`, `abstract` или `override` в объявлении класса `Locksmith`. Эту ошибку легко исправить! Используйте в объявлении метода `ReturnContents()` в классе `Locksmith` ключевое слово `virtual`.

```
class Locksmith {
    ...
    virtual public void ReturnContents
          (Jewels safeContents, Owner owner)
```

Теперь, запустив программу, вы увидите следующее:

```
file:///C:/Users/Public/Documents/Visual Studio 2012...
I'm stealing the contents! Sparkle, sparkle!
```

Э*того* мы и добивались!

Обычно мне требуется перекрывать, а не скрывать методы. Но если я все-таки скрываю их, я всегда должен использовать ключевое слово `new`, не так ли?

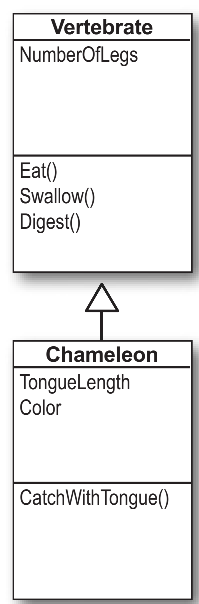
Именно так. В большинстве случаев методы требуется перекрывать, но имеется и возможность скрыть их.

Работая с производным классом, который является расширением базового, вы, скорее всего, будете использовать перекрытие методов. Поэтому если вы заметили, что компилятор предупреждает о скрытии методов, не оставляйте это без внимания! Подумайте, действительно ли вы хотите скрыть метод, или, может быть, вы просто забыли написать ключевые слова `virtual` и `override`. Корректное использование ключевых слов `virtual`, `override` и `new` позволяет избежать проблем, с которыми вы столкнулись в последней программе!

Чтобы перекрыть метод базового класса, всегда помечайте его ключевым словом virtual. И всегда используйте ключевое слово override, когда хотите перекрыть метод в производном классе. Если этого не сделать, некоторые методы внезапно могут оказаться скрытыми.

Ключевое слово base

Иногда возникает необходимость доступа к перекрытым методам или свойствам базового класса. К счастью, существует ключевое слово **base**, дающее доступ к любым методам базового класса.



1. Все животные едят, поэтом класс `Vertebrate` (Позвоночные) должен иметь метод `Eat()`, в качестве параметра которого используется объект `Food` (Еда).

```
class Vertebrate {
   public virtual void Eat(Food morsel) {
      Swallow(morsel);
      Digest();
   }
}
```

2. Хамелеоны ловят пищу языком. Поэтому класс `Chameleon` наследует от класса `Vertebrate`, переписывая при этом метод `Eat()`.

```
class Chameleon : Vertebrate {
   public override void Eat(Food morsel) {
      CatchWithTongue(morsel);
      Swallow(morsel);
      Digest();
   }
}
```

Хамелеоны глотают и переваривают еду, как и любое другое животное. Нужно ли нам в этом случае дублировать данный код?

3. Воспользуйтесь ключевым словом **base** для вызова перекрытого метода, и вы получите доступ как к новой, так и к старой версии метода `Eat()`.

```
class Chameleon : Vertebrate {
   public override void Eat(Food morsel) {
      CatchWithTongue(morsel);
      base.Eat(morsel);
   }
}
```

Эта строчка вызывает метод Eat () из базового класса, от которого наследует объект Chameleon.

Вы получили некоторое представление о процедуре наследования, но можете ли вы рассказать, каким образом наследование облегчает редактирование кода в будущем?

Если в базовом классе присутствует конструктор, он должен остаться и в производном классе

Если в классе присутствуют конструкторы, то все классы, которые от него наследуют, **должны вызывать хотя бы один из этих конструкторов**. При этом конструктор производного класса может иметь свои собственные параметры.

Добавьте эту строку в конец объявления конструктора производного класса, и при любой его инициализации будет **вызываться конструктор из базового класса**.

```
class Subclass : BaseClass {
    public Subclass(список параметров)
        : base(список параметров базового класса) {
      // сначала выполняется конструктор базового класса,
      // а потом все остальные операторы
    }
}
```

Это конструктор производного класса.

Оператор new можно вызвать и без присваивания результата его работы переменной. Оператор `new MySubclass();` создает экземпляр класса MySubclass: Из-за отсутствия ссылок он будет быстро удален сборщиком мусора.

Конструктор базового класса вызывается первым

Убедитесь в этом сами!

1 Создайте базовый класс с конструктором, вызывающим окно диалога

Добавьте к форме кнопку, которая инициализирует ***базовый класс*** и вызывает окно диалога:

```
class MyBaseClass {
    public MyBaseClass(string baseClassNeedsThis) {
        MessageBox.Show("This is the base class: " + baseClassNeedsThis);
    }
}
```

Этот параметр нужен базовому конструктору.

2 Добавьте производный класс, но не вызывайте конструктор

Добавьте к форме кнопку, которая делает то же самое для ***производного класса***:

```
class MySubclass : MyBaseClass{
    public MySubclass(string baseClassNeedsThis, int anotherValue) {
        MessageBox.Show("This is the subclass: " + baseClassNeedsThis
            + " and " + anotherValue);
    }
}
```

Выберите команду Build >> Build Solution, и вы получите сообщение об ошибке.

Эта ошибка означает, что производный класс не вызвал конструктор из базового класса.

3 Заставьте сначала вызывать конструктор из базового класса

Затем инициализируйте производный класс и посмотрите, ***в каком порядке*** появятся два окна диалога.

```
class MySubclass : MyBaseClass{
    public MySubclass(string baseClassNeedsThis, int anotherValue)
        : base(baseClassNeedsThis)
    {
      // остаток кода не изменился
```

Так мы послали базовому классу параметр, который требуется его конструктору.

Эта строка вызовет конструктор базового класса. После этого сообщение об ошибке исчезнет, и программа начнет работать.

Теперь мы готовы завершить программу для Кэтлин!

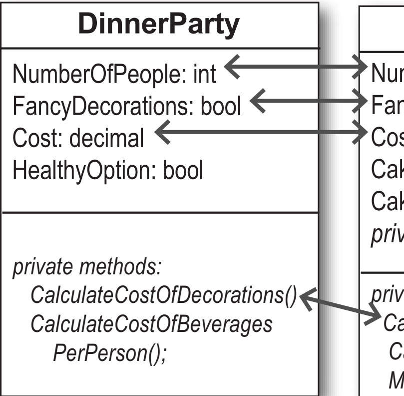
После последнего обновления программа Кэтлин получила возможность рассчитывать стоимость дней рождения. Теперь Кэтлин хочет брать еще по $100 за вечеринки с количеством гостей больше 12. Сначала казалось, что вам придется набирать весь код заново, но теперь, познакомившись с процедурой наследования, вы знаете, как этого избежать.

Если все было сделано правильно, можно отредактировать оба класса, не затрагивая форму!

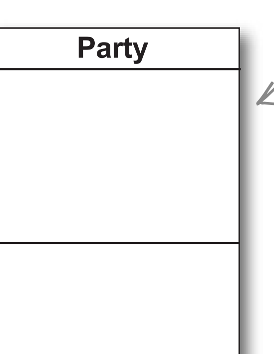
Завершим программу для Кэтлин, создав базовый класс `Party`, обладающий поведением классов `DinnerParty` и `BirthdayParty`.

Расположим эти классы рядом. Какие свойства и методы являются общими?

Новая модель классов.
Первым шагом к созданию хорошей программы является проектирование. У нас по-прежнему есть все те же классы `DinnerParty` и `BirthdayParty`, но теперь они наследуют от класса `Party`. При этом они должны обладать тем же, что и раньше, набором свойств, чтобы не пришлось вносить изменения в форму.



Добавим базовый класс Party.
Создайте новое приложение **Windows Forms** и добавьте к нему класс `Party`. Затем добавьте классы `DinnerParty` и `BirthdayParty` (из проекта в начале этой главы), обновив их таким образом, чтобы они стали потомками класса `Party`.



Первым делом нужно добавить пустой класс Party и превратить классы DinnerParty и BirthdayParty в его потомков. Расширением пустого класса может быть что угодно, поэтому программа готова к построению.

Поместим общие поведения в суперкласс Party.

Скопируйте константу `CostOfFoodPerPerson`, свойства `NumberOfPeople` и `FancyDecorations` и метод `CalculateCostOfDecorations()` из класса `DinnerParty` или `BirthdayParty` (они идентичны) и вставьте их в класс `Party`. Затем удалите перечисленное из обоих подклассов.

Создайте в классе `Party` свойство `Cost`, пометьте его ключевым словом `virtual` и укажите, что в подклассах оно перекрывается.

Party
NumberOfPeople: int FancyDecorations: bool virtual Cost: decimal
private methods: *CalculateCostOfDecorations()*

DinnerParty
HealthyOption: bool override Cost: decimal
private methods: *CalculateCostOfBeverages PerPerson();*

BirthdayParty
CakeWriting: string CakeWritingTooLong: bool override Cost: decimal *private ActualLength: int*
private methods: *CakeSize()* *MaxWritingLength()*

Оба класса используют свойства NumberOfPeople и FancyDecorations одним и тем же способом. Имеет смысл унаследовать их от суперкласса Party.

Хотя оба класса вычисляют стоимость по-разному, они оба пользуются методом CalculateCostOfDecorations(). Но он является закрытым, и подклассы не имеют к нему доступа.

К счастью, можно воспользоваться наследованием. Объявим свойство Cost в суперклассе как виртуальное и расширим его в подклассы.

Вот расширенное свойство Cost. Оно объявляется с ключевым словом override и вызывает base.Cost.

Сложнее всего понять, какую часть двух свойств `Cost` в подклассах следует скопировать в базовый класс `Party`. Можно создать автоматическое свойство `Cost`, ничего не поменяв в одноименных свойствах подклассов. Но в данном случае вам нужно исследовать свойства `Cost` исходных классов `DinnerParty` и `BirthdayParty`, определить, что у них общего, и перенести как можно больше строк в базовый класс.

Подсказка. Свойства `Cost` в обоих классах должны начинаться со следующих строк:

```
override public decimal Cost {
    get {
        decimal totalCost = base.Cost;
```

Не забудьте добавить **$100 за вечеринки с количеством приглашенных более 12** к базовому свойству `Cost` в классе `Party`.

Упражнение Решение

Вот как следовало отредактировать классы `DinnerParty` и `BirthdayParty`, чтобы они унаследовали свойства базового класса `Party`. Именно это позволило вам добавить новый платеж ($100), не редактируя при этом форму!

```
class Party
{
    public const int CostOfFoodPerPerson = 25;

    public int NumberOfPeople { get; set; }

    public bool FancyDecorations { get; set; }

    private decimal CalculateCostOfDecorations()
    {
        decimal costOfDecorations;
        if (FancyDecorations)
            costOfDecorations = (NumberOfPeople * 15.00M) + 50M;
        else
            costOfDecorations = (NumberOfPeople * 7.50M) + 30M;
        return costOfDecorations;
    }

    virtual public decimal Cost
    {
        get {
            decimal totalCost = CalculateCostOfDecorations();
            totalCost += CostOfFoodPerPerson * NumberOfPeople;

            if (NumberOfPeople > 12)
                totalCost += 100;

            return totalCost;
        }
    }
}

class BirthdayParty : Party
{
    public BirthdayParty(int numberOfPeople,
                         bool fancyDecorations, string cakeWriting)
    {
        NumberOfPeople = numberOfPeople;
        FancyDecorations = fancyDecorations;
        CakeWriting = cakeWriting;
    }
```

Эти свойства и константа в классах DinnerParty и BirthdayParty идентичны, поэтому мы их оттуда удалили и перенесли в суперкласс.

Этот метод также присутствовал в обоих подклассах, поэтому он был перенесен в базовый класс Party.

Не забудьте пометить Cost как virtual!

Эти две строчки совпадали в исходных классах DinnerParty и BirthdayParty, поэтому мы перенесли их в базовое свойство Cost. В класс Party переместилось максимально возможное число поведений

Теперь для дней рождения и званых обедов появились собственные классы, расширяющие базовый класс Party. Мы легко добавим платеж $100, когда гостей больше 12. Укажите это в базовом классе, и подклассы унаследуют данное поведение.

Класс BirtdayParty расширяет класс Party.

Конструктор BirthdayParty остается без изменений, хотя он задает свойства, принадлежащие базовому классу.

```
    public string CakeWriting { get; set; }

    private int ActualLength
    {
        get
        {
            if (CakeWriting.Length > MaxWritingLength())
                return MaxWritingLength();
            else
                return CakeWriting.Length;

        }
    }

    private int CakeSize() {
        if (NumberOfPeople <= 4)
            return 8;
        else
            return 16;
    }

    private int MaxWritingLength() {
        if (CakeSize() == 8)
            return 16;
        else
            return 40;
    }


    public bool CakeWritingTooLong {
        get {
            if (CakeWriting.Length > MaxWritingLength())
                return true;
            else
                return false;
        }
    }

    override public decimal Cost {
        get {
            decimal totalCost = base.Cost;
            decimal cakeCost;
            if (CakeSize() == 8)
                cakeCost = 40M + ActualLength * .25M;
            else
                cakeCost = 75M + ActualLength * .25M;
            return totalCost + cakeCost;
        }
    }
}
```

Свойства CakeWriting и ActualLength используются только классом BirthdayParty, поэтому они никуда не переносятся.

Свойство CakeWriting, свойство ActualLength и используемые ими методы остаются в классе BirthdayParty. Как и свойство CakeWritingTooLong.

Мы перенесли в базовый класс первые два оператора свойства Cost, так как они одинаковы в классах DinnerParty и BirthdayParty. Для выполнения этих двух операторов свойство Cost класса BirthdayParty вызывает base.Cost.

Продолжение на с. 300.

Упражнение Решение

Последний класс в программе для Кэтлин. Код формы не изменился!

Свойство HealthyOption используется только для званых обедов, и не учитывается при праздновании дней рождения, поэтому мы оставим его в исходном классе.

```
class DinnerParty : Party {
    public bool HealthyOption { get; set; }

    public DinnerParty(int numberOfPeople, bool healthyOption,
                       bool fancyDecorations) {
        NumberOfPeople = numberOfPeople;
        FancyDecorations = fancyDecorations;
        HealthyOption = healthyOption;
    }

    private decimal CalculateCostOfBeveragesPerPerson() {
        decimal costOfBeveragesPerPerson;
        if (HealthyOption)
            costOfBeveragesPerPerson = 5.00M;
        else
            costOfBeveragesPerPerson = 20.00M;
        return costOfBeveragesPerPerson;
    }

    override public decimal Cost {
        get {
            decimal totalCost = base.Cost;
            totalCost += CalculateCostOfBeveragesPerPerson() * NumberOfPeople;
            if (HealthyOption)
                totalCost *= .95M;
            return totalCost;
        }
    }
}
```

Метод CalculatecostOfBeveragesPerPerson() и конструктор остаются в DinnerParty, так как класс BirthdayParty их не использует.

Свойство Cost работает так же, как в классе BirthdayParty. При помощи base.Cost оно выполняет операторы в Party.Cost и использует полученный результат как отправную точку для итоговых расчетов.

Программа работает! Теперь мой бизнес пошел в гору, спасибо огромное!

Минимальное пересечение классов является важным принципом проектирования, который называется «разделение ответственности».

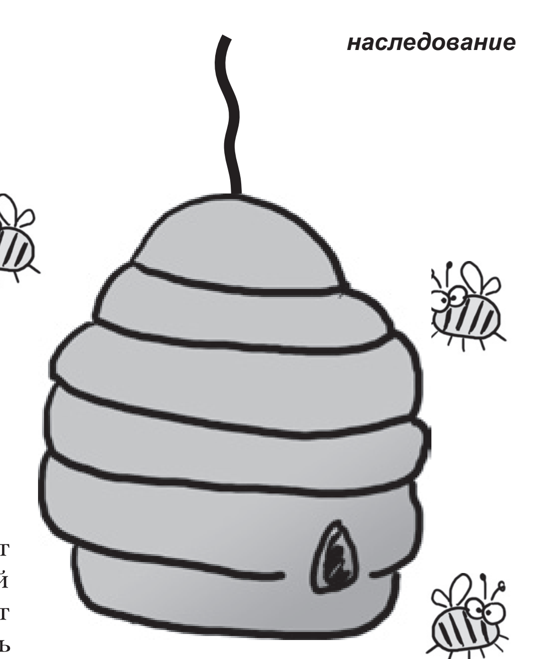
Правильно спроектированные классы легко редактируются. Нам потребовалась бы масса усилий, чтобы добавить $100, когда участников больше 12, в классы `DinnerParty` и `BirthdayParty`. Но наследование позволило ограничится парой строк кода благодаря перемещению в базовый класс поведений, которые являются общими для свойств `Cost` в подклассах.

Это хороший пример разделения ответственности, ведь каждый класс содержит только код, касающийся определенной части решаемой задачи. Код, связанный со зваными обедами, попал в класс `DinnerParty`, код для дней рождения оказался в классе `BirthdayParty`, а общий для них код попал в класс `Party`.

Но подумайте вот о чем. Ответственность за пользовательский интерфейс лежит на объекте `Form`. Все расчеты стоимости инкапсулированы в `Cost` классов `DinnerParty` и `BirthdayParty`. Мы решили, что объект `Form` должен преобразовывать переменную `cost` типа `decimal` в «денежную» строку. Правильно ли это?

Помните, что программу можно написать разными способами и универсального «правильного» ответа не существует. Даже если этот ответ дан в книге!

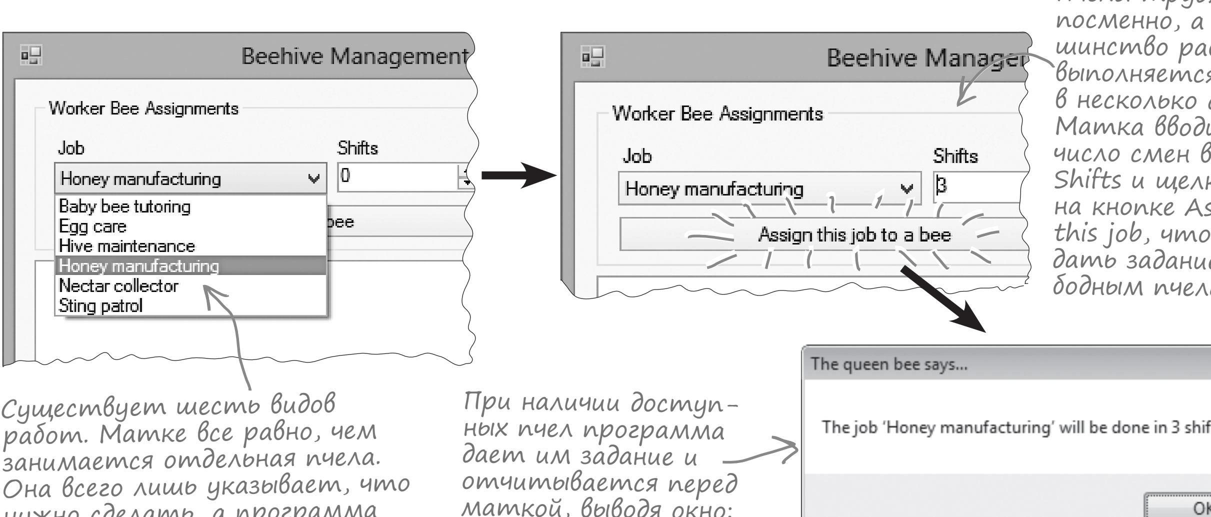
Система управления ульем



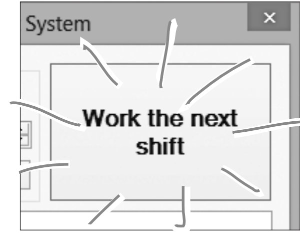
Теперь ваша помощь нужна пчелиной матке! Улей вышел из-под контроля, и ей нужна программа, которая поможет им управлять. Улей полон рабочих пчел, имеется и список заданий. Нужно распределить задания между пчелами с учетом их специализации.

Постройте систему, управляющую поведением рабочих пчел. Вот как она должна функционировать:

Матка раздает задания рабочим
Существует шесть видов работ. Некоторые пчелы умеют собирать нектар и делать мед, другие могут строить улей и защищать его от врагов. Есть пчелы, которые могут выполнять вообще любую работу. Вам нужно написать программу, дающую пчеле задание, которое она в состоянии выполнить.



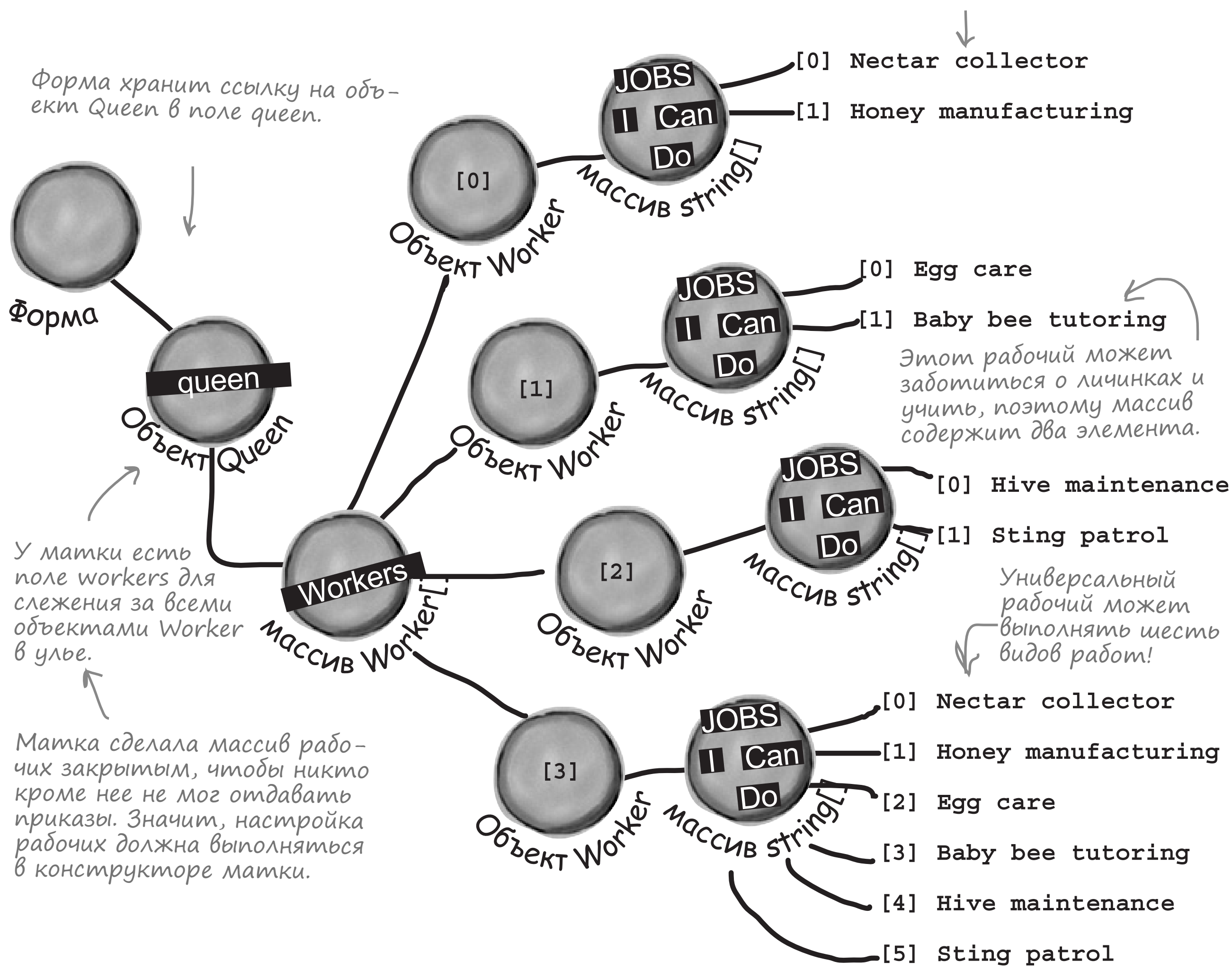
Время работать
Раздав задания, матка заставляет пчел отрабатывать очередную смену щелчком на кнопке «Работать! Следующая смена». Программа отчитывается, какие пчелы работали в эту смену, какую работу они выполняли и сколько смен им еще осталось трудиться именно над этим заданием.




```
Report for shift #1
Worker #1 is doing 'Honey manufacturing' for 2 more shifts
```

Как нам обустроить управление ульем

Этот проект делится на две части. Начнем с обзора системы управления ульем. Вам потребуются два класса, `Queen` (Матка) и `Worker` (Рабочий). С ними мы свяжем построенную для системы форму. И следует **гарантировать инкапсуляцию классов**, чтобы при переходе ко второй части решения они вам не помешали.



Сначала форма создает массив, затем — каждого рабочего, а затем добавляет его к массиву.

Конструктор каждого объекта Worker принимает один параметр — массив строк с доступными рабочему видами деятельности.

```
Worker[] workers = new Worker[4];
workers[0] = new Worker(new string[] { "Nectar collector", "Honey manufacturing" });
workers[1] = new Worker(new string[] { "Egg care", "Baby bee tutoring" });
workers[2] = new Worker(new string[] { "Hive maintenance", "Sting patrol" });
workers[3] = new Worker(new string[] { "Nectar collector", "Honey manufacturing",
   "Egg care", "Baby bee tutoring", "Hive maintenance", "Sting patrol" });
queen = new Queen(workers);
```

Форма имеет поле, указывающее на объект Queen. Она инициализирует поле, передавая созданный массив объектов Worker конструктору объекта Queen.

При щелчке на кнопке «назначить» вызывается метод AssignWork матки, давая ей возможность проверить наличие доступных рабочих.

Матка проверяет каждого рабочего, способен ли он к определенной деятельности.

Метод AssignWork() объекта Queen просматривает массив рабочих, вызывая метод DoThisJob(), пока не найдет подходящего кандидата.

Матка спрашивает Рабочего, может ли он 4 смены обслуживать улей.

Матка может назначать рабочим задания и отправлять их на следующую смену.

Матка заставляет каждого рабочего отработать смену, составляя после этого отчет о проделанной работе.

Метод DidYouFinish() объекта Worker заставляет отработать еще одну смену и возвращает true при завершении работы.

Пчелиной матке нужна помощь! Используйте все знания о классах и объектах и создайте систему управления ульем, следящую за рабочими пчелами. Сначала вы спроектируете форму, добавите классы Queen и Worker и получите базовую рабочую систему.

Иногда в диаграмме классов могут появиться закрытые поля и типы.

Queen
private workers: Worker[] *private shiftNumber: int*
AssignWork() WorkTheNextShift()

Объект `Queen` указывает, какую работу нужно сделать.

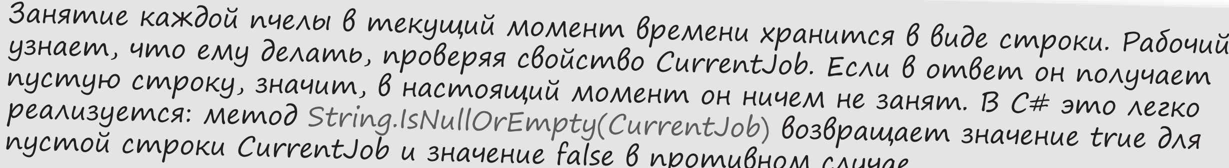
- ★ Массив объектов `Worker` отслеживает текущую занятость каждой рабочей пчелы. Эта информация хранится в закрытых полях `Worker[]`.
- ★ Форма вызывает метод `AssignWork()` (Назначить задание), передавая строку с названием работы и переменную типа int с количеством смен. При наличии свободной пчелы, которая в состоянии выполнять эту работу, возвращается значение true.
- ★ Кнопка Work the next shift вызывает метод `WorkTheNextShift()`, заставляющий каждый объект Worker отработать одну смену и затем проверяющий его состояние, чтобы сформировать отчет.
- ★ Скриншот на странице справа показывает, что именно возвращает метод `WorkTheNextShift()`. Сначала он создает строку («Report for shift #13»). Затем внутри цикла `for` выполняет два оператора `if` для каждого объекта `Worker` в массиве `workers[]`. Первый оператор проверяет, завершил ли рабочий свое задание («Worker #2 finished the job»), а второй — занят ли рабочий. В случае положительного результата выводится количество оставшихся смен.

Свойства CurrentJob (ТекущееЗадание) и ShiftsLeft (ОставшиесяСмены) предназначены только для чтения.

Worker
CurrentJob: string ShiftsLeft: int *private jobsICanDo: string[]* *private shiftsToWork: int* *private shiftsWorked: int*
DoThisJob() DidYouFinish()

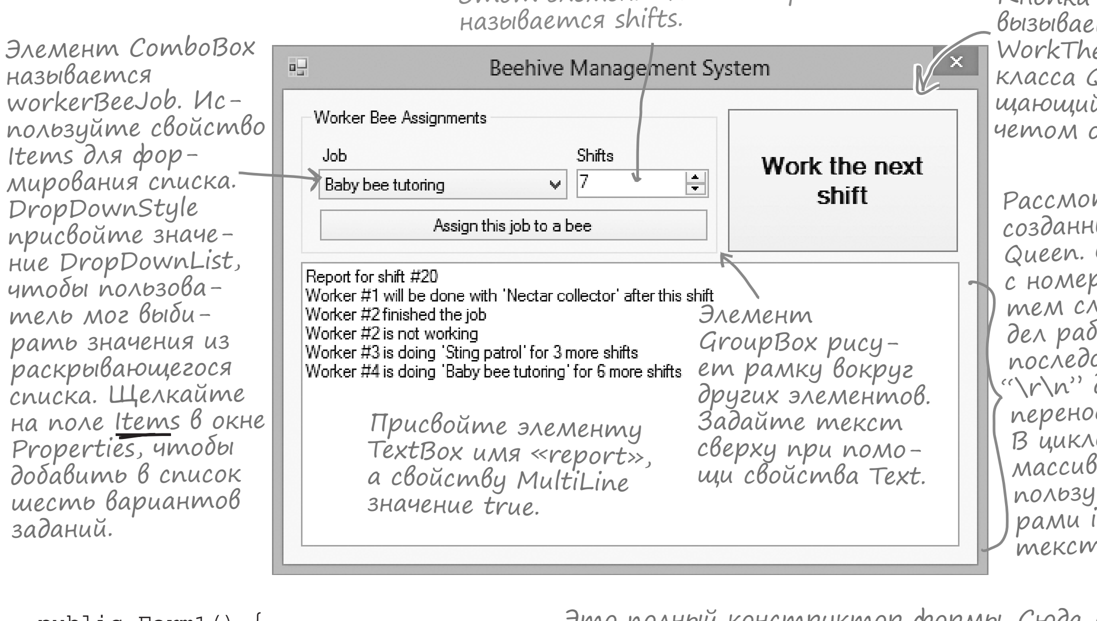
Посмотрим на функции массива Worker.

- ★ Свойство `CurrentJob` дает понять объекту `Queen`, какую работу выполняет каждый рабочий. Если рабочий на момент проверки не занят, возвращается пустая строка.
- ★ Раздача заданий рабочим происходит при помощи метода `DoThisJob()`. Если рабочий не занят и в состоянии выполнять указанную работу, метод возвращает значение true.
- ★ Когда вызывается метод `DidYouFinish()`, рабочий отрабатывает смену. Метод проверяет, сколько смен осталось отработать. Если работа завершена, метод возвращает `true` и присваивает вместо текущего задания пустую строку, давая возможность получить следующее задание. В противном случае возвращается `false`.


Занятие каждой пчелы в текущий момент времени хранится в виде строки. Рабочий узнает, что ему делать, проверяя свойство CurrentJob. Если в ответ он получает пустую строку, значит, в настоящий момент он ничем не занят. В C# это легко реализуется: метод String.IsNullOrEmpty(CurrentJob) возвращает значение true для пустой строки CurrentJob и значение false в противном случае.

1 **Построим форму.**

Форма крайне проста, все сложное осталось в классах `Queen` и `Worker`. На ней закрытое поле Queen и две кнопки, вызывающие методы `AssignWork()` и `WorkTheNextShift()`. Нужно добавить `ComboBox` для заданий (см. на предыдущих страницах), `NumericUpDown`, две кнопки и текстовое поле для отчета. Еще нам понадобится конструктор формы.



Кнопка nextShift вызывает метод WorkTheNextShift() из класса Queen, возвращающий строку с отчетом о сменах.

Рассмотрим отчет, созданный объектом Queen. Он начинается с номера смены, затем следует список дел работников. esc-последовательности "\r\n" добавляют знак переноса в строку. В цикле просмотрите массив workers и воспользуйтесь операторами if для генерации текста.

```
public Form1() {
    InitializeComponent();
    workerBeeJob.SelectedIndex = 0;
    Worker[] workers = new Worker[4];
    workers[0] = new Worker(new string[] { "Nectar collector", "Honey manufacturing" });
    workers[1] = new Worker(new string[] { "Egg care", "Baby bee tutoring" });
    workers[2] = new Worker(new string[] { "Hive maintenance", "Sting patrol" });
    workers[3] = new Worker(new string[] { "Nectar collector", "Honey manufacturing",
                "Egg care", "Baby bee tutoring", "Hive maintenance", "Sting patrol" });
    queen = new Queen(workers);
}
```

Это полный конструктор формы. Сюда мы поместили код с предыдущей страницы. Дополнительная строка заставляет ComboBox отображать первый элемент (чтобы список не был пустым при загрузке формы).

Форме потребуется поле Queen с именем queen. Массив Worker нужно передать конструктору объекта Queen.

2 **Создадим классы Worker и Queen.**

Вы уже практически все знаете о классах `Worker` и `Queen`. Осталась всего пара моментов. Метод `Queen.AssignWork()` циклически просматривает массив `worker` и пытается дать работу каждому объекту `worker` при помощи метода `DoThisJob()`. Объект `Worker` в массиве строк `jobsICanDo` проверяет свою способность выполнить предложенную работу. Если это возможно, закрытому полю `shiftsToWork` присваивается количество смен, параметру `CurrentJob` – название работы, а переменной `shiftsWorked` – значение «ноль». Отработанная смена увеличивает эту переменную на 1. Предназначенное только для чтения свойство `ShiftsLeft` дает матке возможность оценить, сколько смен еще требуется отработать.

Упражнение Решение

```
class Worker {
    public Worker(string[] jobsICanDo) {
        this.jobsICanDo = jobsICanDo;
    }

    public int ShiftsLeft {
        get {
            return shiftsToWork - shiftsWorked;
        }
    }

    private string currentJob = "";
    public string CurrentJob {
        get {
            return currentJob;
        }
    }

    private string[] jobsICanDo;
    private int shiftsToWork;
    private int shiftsWorked;

    public bool DoThisJob(string job, int numberOfShifts) {
        if (!String.IsNullOrEmpty(currentJob))
            return false;
        for (int i = 0; i < jobsICanDo.Length; i++)
            if (jobsICanDo[i] == job) {
                currentJob = job;
                this.shiftsToWork = numberOfShifts;
                shiftsWorked = 0;
                return true;
            }
        return false;
    }

    public bool DidYouFinish() {
        if (String.IsNullOrEmpty(currentJob))
            return false;
        shiftsWorked++;
        if (shiftsWorked > shiftsToWork) {
            shiftsWorked = 0;
            shiftsToWork = 0;
            currentJob = "";
            return true;
        }
        else
            return false;
    }
}
```

Конструктор задает свойство JobsICanDo, представляющее собой строковый массив. Он закрыт, так как предполагается, что матка только просит рабочего выполнить работу, но не должна при этом проверять, может ли он это сделать.

Свойство ShiftsLeft, предназначенное только для чтения, показывает, сколько смен осталось отработать до завершения задания.

Свойство CurrentJob (только для чтения) показывает матке, какие работы следует провести.

Матка использует метод DoThisJob() объекта worker для назначения заданий рабочим, рабочий проверяет свойство JobsICanDo, чтобы понять, может ли он выполнять данную работу.

Оператор !— означающий НЕТ— проверяет, является ли строка пустой и имеет ли значения null. Именно таким образом осуществляется проверка несоблюдения условия.

Матка использует метод DidYouFinish() объекта worker, чтобы заставить рабочих отработать следующую смену. Метод возвращает значение true, если это **последняя смена** текущего работника. В отчете появляется строчка, что после этой смены пчела свободна.

Сначала проверяется поле currentJob. Если рабочий ничем не занят, возвращается значение false, и метод прекращает работу. В противном случае параметр ShiftsWorked увеличивается на 1, и проверяется, сделана ли работа (путем сравнения с параметром ShiftsToWork). При положительном результате метод возвращает true.

```
class Queen {
    public Queen(Worker[] workers) {
        this.workers = workers;
    }

    private Worker[] workers;
    private int shiftNumber = 0;

    public bool AssignWork(string job, int numberOfShifts) {
        for (int i = 0; i < workers.Length; i++)
            if (workers[i].DoThisJob(job, numberOfShifts))
                return true;
        return false;
    }

    public string WorkTheNextShift() {
        shiftNumber++;
        string report = "Отчет для смены#" + shiftNumber + "\r\n";
        for (int i = 0; i < workers.Length; i++)
        {
            if (workers[i].DidYouFinish())
                report += "Рабочий #" + (i + 1) + " закончил работу\r\n";
            if (String.IsNullOrEmpty(workers[i].CurrentJob))
                report += "Рабочий #" + (i + 1) + " не работает\r\n";
            else
              if (workers[i].ShiftsLeft > 0)
                report += "Рабочий #" + (i + 1) + " выполняет '" + workers[i].CurrentJob
                    + "' еще " + workers[i].ShiftsLeft + " смен\r\n";
              else
                report += "Рабочий #" + (i + 1) + " закончит '"
                    + workers[i].CurrentJob + "' после этой смены\r\n";
        }
        return report;
    }
}
```

Массив рабочих является закрытым, так как после раздачи заданий ни один другой класс не должен иметь возможности их менять... и даже видеть их, ведь приказы рабочим может отдавать только матка. Значение поля задает конструктор.

Матка сначала пытается дать задание первому рабочему. Если он не умеет делать работу указанного типа, она переходит к следующему. Как только находится пчела, которая в состоянии взять задание, метод возвращает значение true, и цикл завершается.

Метод WorkTheNextShift() объекта queen заставляет рабочего трудиться следующую смену и добавляет строчку в отчет.

Вы знаете код для конструктора. Вот код для остальной части формы:

```
private Queen queen;

private void assignJob_Click(object sender, EventArgs e) {
    if (queen.AssignWork(workerBeeJob.Text, (int)shifts.Value) == false)
        MessageBox.Show("Для этого задания рабочих нет '"
            + workerBeeJob.Text + "'", "Матка говорит...");
    else
        MessageBox.Show("Задание '" + workerBeeJob.Text + "' будет закончено через "
            + shifts.Value + " смен", "Матка говорит...");
}

private void nextShift_Click(object sender, EventArgs e) {
    report.Text = queen.WorkTheNextShift();
}
```

Поле queen используется формой для хранения ссылок на объект Queen, который, в свою очередь, содержит массив ссылок на объекты worker.

Кнопка assignJob вызывает метод AssignWork() объекта queen, дающий пчелам задания, и отображает окно с сообщением о том, пригоден ли выбранный рабочий для указанной деятельности.

Кнопка nextShift заставляет матку раздать задания на следующую смену. При этом в текстовом поле формы появляется отчет.

Совершенствуем систему управления ульем при помощи наследования

Основа нашей системы готова, теперь воспользуемся наследованием для учета количества меда, съедаемого пчелами. Каждая пчела потребляет мед, а больше всего его нужно матке. Поэтому создадим базовый класс `Bee`, от которого будут наследовать классы `Queen` и `Worker`.

Вы добавите класс Bee, который расширяют классы Queen и Worker. В классе Bee будут находиться базовые члены, вычисляющие потребление меда.

Иногда на диаграмме классов показываются возвращаемые значения и закрытые члены.

Bee
- HoneyUnitsConsumedPerMg: const double
- HoneyConsumptionRate(): virtual double

Метод HoneyConsumptionRate() вычисляет количество меда, потребляемое пчелой за смену.

В каждую последующую смену объект Worker потребляет больше меда. Класс Worker расширит класс Bee и перекроет метод HoneyConsumptionRate().

Queen
- private workers: Worker[]
- private shiftNumber: int
- AssignWork()
- WorkTheNextShift()

Worker
- CurrentJob: string
- ShiftsLeft: int
- private jobsICanDo: string[]
- private shiftsToWork: int
- private shiftsWorked: int
- DoThisJob()
- DidYouFinish()
- override HoneyConsumptionRate()

Матке нужно расширить класс Bee и вызвать метод HoneyConsumptionRate() для добавления в отчет сведений о съеденном за смену меде.

При выполнении упражнений, состоящих из двух частей, для второй части стоит создать новый проект. Тогда при необходимости вы сможете вернуться к исходному решению. Для создания нового проекта щелкните правой кнопкой мыши на имени уже имеющегося проекта и выберите в появившемся меню команду Add Existing Item. После чего перейдите в папку со старым проектом и выделите файлы, которые нужно добавить в новый. IDE скопирует эти файлы. Но не забудьте, что IDE НЕ меняет пространство имен, поэтому вам придется редактировать эти строки в файле классов вручную. Если проект содержит форму, не забудьте добавить конструктор (.Designer.cs) и файлы с ресурсами (.resx), для них также потребуется изменить пространство имен вручную.

Работа еще не закончена! Матке позвонила пчела-бухгалтер и сказала, что нужно вести учет потребляемого рабочими меда. Прекрасный шанс применить на практике все, что вы узнали о наследовании! Добавьте суперкласс `Bee`, который будет вычислять потребление меда для каждой смены.

1 Создадим класс Bee.

В классе Bee присутствует метод `HoneyConsumptionRate()`, вычисляющий потребление меда конкретной пчелой за одну смену. Превратим классы `Worker` и `Queen` в его расширения.

```
class Bee {
    public const double HoneyUnitsConsumedPerMg = .25;

    public double WeightMg { get; private set; }

    public Bee(double weightMg) {
        WeightMg = weightMg;
    }

    virtual public double HoneyConsumptionRate() {
        return WeightMg * HoneyUnitsConsumedPerMg;
    }
}
```

Конструктор Bee в качестве параметра принимает вес пчелы в миллиграммах и на его основе вычисляет базовое потребление меда.

2 Превратим классы Queen и Worker в расширения класса Bee.

Классы `Queen` и `Worker` будут наследовать базовое потребление меда от их нового предка, суперкласса `Bee`. Нужно, чтобы их конструкторы вызывали конструктор базового класса.

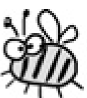
- ★ Превратим класс `Queen` в потомка класса `Bee`. К конструктору добавим параметр `weightMg` типа `double`, который будет передан обратно в базовый конструктор.
- ★ Превратим класс `Worker` в потомка класса `Bee`, с его конструктором следует осуществить такую же манипуляцию.

Подсказка: вы можете использовать в своих интересах сообщение об ошибке «does not contain a constructor», которое уже видели в этой главе! Заставьте класс Worker наследовать от класса Bee и выполните построение проекта. Когда IDE отобразит ошибку, двойной щелчок на ней автоматически перебросит вас на код конструктора Worker. Очень удобно!

3 Заствим форму присваивать вес матке и рабочим.

Вы отредактировали конструкторы `Queen` и `Worker`, теперь осталось **внести изменения в конструктор формы**, чтобы при создании новых экземпляров `Worker` и `Queen` он передавал данные о весе в их конструкторы. Рабочий #1 весит 175 мг, рабочий #2 — 114 мг, рабочий #3 — 149 мг, рабочий #4 — 155 мг, а матка — 275 мг.

(Теперь следует скомпилировать ваш код.)



4 Перекройте метод HoneyConsumptionRate() объекта Worker

Класс `Queen` потребляет мед подобно базовому классу `Bee`. Рабочие потребляют одно и то же количество меда... пока они ничего не делают. В процессе работы потребуется 0.65 дополнительной единицы меда за каждую смену.

Это означает, что объект `Queen` может использовать базовый метод `HoneyConsumptionRate()`, унаследованный от суперкласса `Bee`, а объект `Worker` должен перекрыть этот метод, добавив 0.65 единицы меда на отработанную смену. Для наглядности можно поместить этот параметр в константу `honeyUnitsPerShiftWorked`.

IDE поможет нам начать. Введите для класса `Worker` «public override», и как только вы нажмете пробел, IDE покажет список всех доступных для перекрытия методов:

```
public override
        Equals(object obj)
        GetHashCode()
        HoneyConsumptionRate()      double Bee.HoneyConsumptionRate()
        ToString()
```

Выберите `HoneyConsumptionRate()` в окне IntelliSense. После этого IDE сгенерирует заглушку метода, которая вызывает базовый метод. Введите туда код, который первым делом выводит результат работы метода `base.HoneyConsumptionRate()`, а затем добавьте дополнительные 0.65 единицы, съеденные за отработанную смену.

5 Добавим потребление меда в отчет.

Нужно отредактировать метод `WorkTheNextShift()` объекта `Queen`, чтобы он начал следить за медом, потребляемым объектом `Queen` и всеми объектами `Worker`, вызывая метод `HoneyConsumptionRate()` каждого из объектов и прибавляя результат его работы к общей сумме. В конце отчета добавьте вот эту строчку (заменив `XXX` количеством единиц съеденного меда):

```
Total honey consumed for the shift: XXX units
```

Это делается добавлением трех строк к методу WorkTheNextShift().

Так как все пчелы обладают методом `HoneyConsumptionRate()`, а объекты `Queen` и `Worker` принадлежат классу Bee, не существует ли единого способа вызова этого метода для любого объекта `Bee`, вне зависимости от того, какая именно это пчела?

Упражнение Решение

Конструктор получает новый параметр, который возвращается в базовый конструктор. Это позволяет форме инициализировать объект-пчелу, присвоив ей вес.

```
class Worker : Bee
{
    public Worker(string[] jobsICanDo, double weightMg)
        : base(weightMg)
    {
        this.jobsICanDo = jobsICanDo;
    }

    const double honeyUnitsPerShiftWorked = .65;

    public override double HoneyConsumptionRate()
    {
        double consumption = base.HoneyConsumptionRate();
        consumption += shiftsWorked * honeyUnitsPerShiftWorked;
        return consumption;
    }

    // Остальная часть класса Worker осталась без изменений
    // ...

}
```

Класс Worker перекрывает метод HoneyConsumptionRate(), сверхпотребление меда работающими пчелами.

Благодаря наследованию вы легко смогли добавить в классы Queen и Worker поведение, учитывающее потребление меда. Только представьте, что вместо этого вам пришлось бы вводить повторяющийся код вручную!

В форме меняется только конструктор.

```
public Form1()
{
    InitializeComponent();
    workerBeeJob.SelectedIndex = 0;
    Worker[] workers = new Worker[4];
    workers[0] = new Worker(new string[] { "Nectar collector", "Honey manufacturing" }, 175);
    workers[1] = new Worker(new string[] { "Egg care", "Baby bee tutoring" }, 114);
    workers[2] = new Worker(new string[] { "Hive maintenance", "Sting patrol" }, 149);
    workers[3] = new Worker(new string[] { "Nectar collector", "Honey manufacturing",
        "Egg care", "Baby bee tutoring", "Hive maintenance", "Sting patrol" }, 155);
    queen = new Queen(workers, 275);
}
```

Единственным изменением формы является добавление веса в конструкторы Worker и Queen.

```
class Queen : Bee
{
    public Queen(Worker[] workers, double weightMg)
        : base(weightMg)
    {
        this.workers = workers;
    }

    private Worker[] workers;
    private int shiftNumber = 0;

    public bool AssignWork(string job, int numberOfShifts)
    {
        for (int i = 0; i < workers.Length; i++)
            if (workers[i].DoThisJob(job, numberOfShifts))
                return true;
        return false;
    }

    public string WorkTheNextShift()
    {
        double honeyConsumed = HoneyConsumptionRate();

        shiftNumber++;
        string report = "Report for shift #" + shiftNumber + "\r\n";
        for (int i = 0; i < workers.Length; i++)
        {
            honeyConsumed += workers[i].HoneyConsumptionRate();

            if (workers[i].DidYouFinish())
                report += "Worker #" + (i + 1) + " finished the job\r\n";
            if (String.IsNullOrEmpty(workers[i].CurrentJob))
                report += "Worker #" + (i + 1) + " is not working\r\n";
            else
                if (workers[i].ShiftsLeft > 0)
                    report += "Worker #" + (i + 1) + " is doing '" + workers[i].CurrentJob
                        + "' for " + workers[i].ShiftsLeft + " more shifts\r\n";
                else
                    report += "Worker #" + (i + 1) + " will be done with '"
                        + workers[i].CurrentJob + "' after this shift\r\n";
        }

        report += "Total honey consumed for the shift: " + honeyConsumed + " units\r\n";

        return report;
    }
}
```

Конструктор объекта Queen редактируется так же, как и конструктор объекта Worker.

Этот код не изменился.

Вычисление меда, съеденного за смену, должно начинаться с потребления меда объектом Queen в настоящий момент.

Просматривая в цикле всех рабочих, метод добавляет потребляемый ими мед к общему количеству.

Этот код также остается без изменений.

После того как к отчету добавлены сведения обо всех рабочих, матке останется добавить последнюю строку с общим количеством съеденного за смену меда.

7 интерфейсы и абстрактные классы

Пусть классы держат обещания

Действия значат больше, чем слова.

Иногда возникает необходимость сгруппировать объекты по **выполняемым функциям**, а не по классам, от которых они наследуют. Здесь вам на помощь приходят **интерфейсы** — они позволяют работать с любым классом, отвечающим вашим потребностям. Но **чем больше возможностей, тем выше ответственность**, и если классы, реализующие интерфейс, **не выполнят обязательств**… программа компилироваться не будет.

Вернемся к нашим пчелам

Было принято решение превратить систему учета из предыдущей главы в полноценный симулятор улья. Вот как выглядит спецификация новой версии программы:

Симулятор улья

Для лучшего представления жизни в улье укажем специальные возможности рабочих пчел:

- Все пчелы потребляют мед и обладают весом.
- Матка раздает задания, следит за отчетами и отправляет рабочих на следующую смену.
- Рабочие трудятся посменно.
- Охранники «точат» жало, ищут врагов и жалят их.
- Сборщики меда ищут цветы, собирают нектар и возвращаются в улей.

Классы Bee и Worker практически не изменились. Поэтому для новых функций достаточно расширить имеющиеся классы.

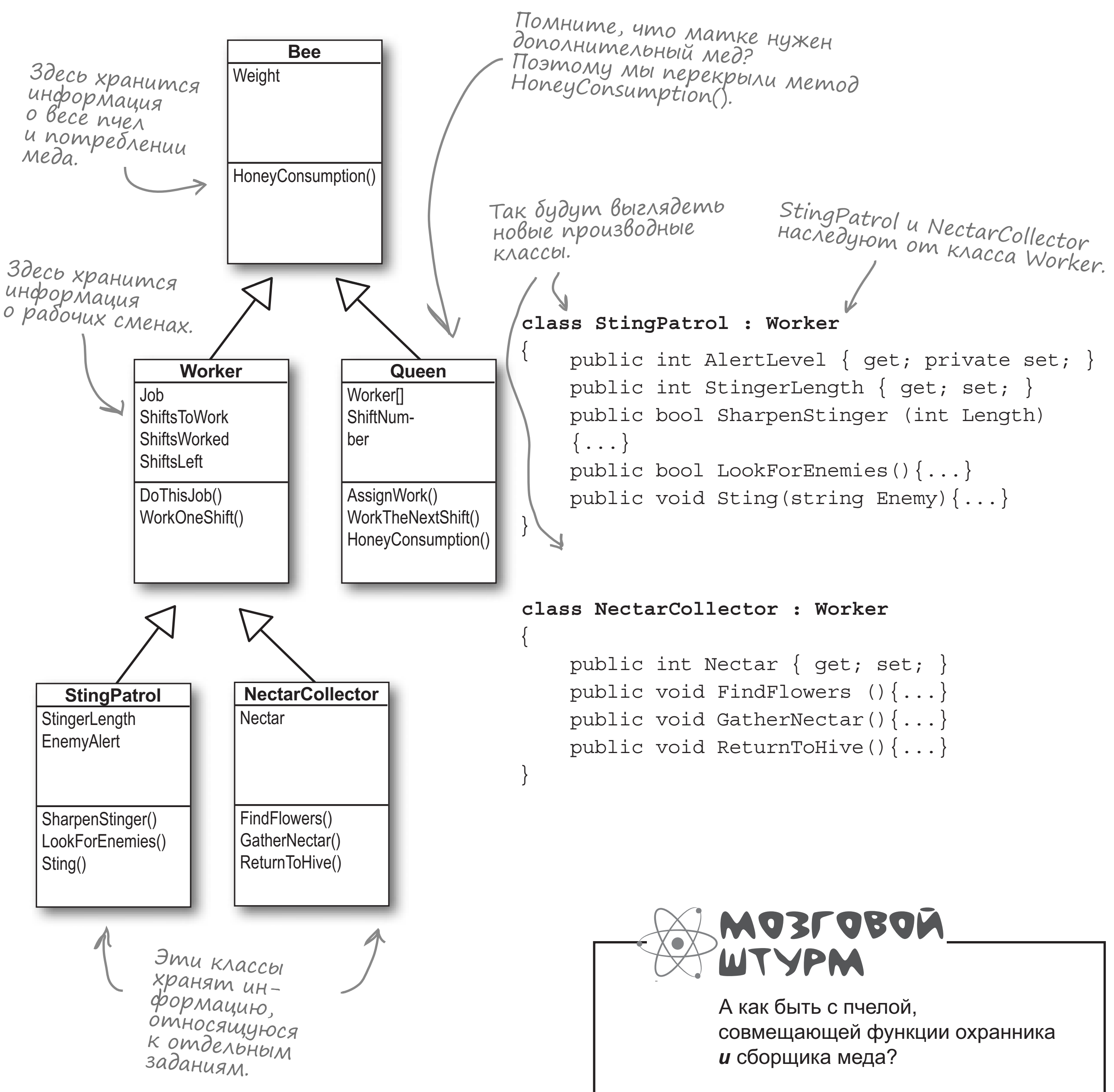
Кажется, нам потребуется сохранять данные о рабочих пчелах в зависимости от выполняемых ими функций.

Многое остается неизменным

В новом симуляторе улья пчелы потребляют мед тем же способом, что и раньше. Матка по-прежнему должна раздавать задания и отслеживать, сколько смен осталось отработать каждой пчеле. Рабочие по-прежнему трудятся посменно. Просто их деятельность претерпела небольшие изменения.

Классы для различных типов пчел

Вот иерархия с классами `Worker` и `Queen`, наследующими от класса `Bee`. При этом класс `Worker` имеет производные классы `NectarCollector` (Сборщик меда) и `StingPatrol` (Охранник).




```
class StingPatrol : Worker
{
    public int AlertLevel { get; private set; }
    public int StingerLength { get; set; }
    public bool SharpenStinger (int Length)
    {...}
    public bool LookForEnemies(){...}
    public void Sting(string Enemy){...}
}
```

```
class NectarCollector : Worker
{
    public int Nectar { get; set; }
    public void FindFlowers (){...}
    public void GatherNectar(){...}
    public void ReturnToHive(){...}
}
```

МОЗГОВОЙ ШТУРМ

А как быть с пчелой, совмещающей функции охранника ***и*** сборщика меда?

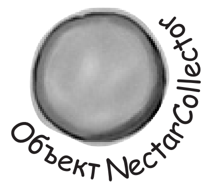
Интерфейсы

Класс, реализующий интерфейс, должен включать в себя все методы и свойства, указанные в определении интерфейса.

Наследовать класс может только от одного класса. Поэтому создание двух производных классов `StingPatrol` и `NectarCollector` не поможет нам описать пчелу, которая в состоянии выполнять задания **разных** типов.

Метод `DefendTheHive()` (Защита улья) из класса Queen может заставить объекты `StingPatrol` защищать улей. Но если матка захочет, чтобы за эту работу принялись другие пчелы, она не сможет дать им команду:

```
class Queen {
  private void DefendTheHive(StingPatrol patroller) { ... }
}
```



Объект `NectarCollector` умеет собирать нектар, а экземпляры `StingPatrol` борются с врагами. Но даже если матка научит сборщиков нектара защищать улей, добавив методы `SharpenStinger()` и `LookForEnemies()` в определение их класса, она все равно не сможет передать их своему методу `DefendTheHive()`. Впрочем, можно воспользоваться двумя методами:

```
private void DefendTheHive(StingPatrol patroller);
private void AlternateDefendTheHive(NectarCollector patroller);
```

Даже если матка добавит методы защиты объекту NectarCollector, она не сможет передать их своему методу DefendTheHive(), так как он ожидает ссылки StingPatrol. Приравнять же ссылку StingPatrol объекту NectarCollector невозможно.

Можно добавить аналогичный метод с названием AlternateDefendTheHive(), который будет ссылаться на объект NectarCollector, но код получится слишком громоздким и неудобным.

Методы DefendTheHive() и AlternateDefendTheHive() будут отличаться только типом параметра. А чтобы заставить защищать улей объекты BabyBeeCare или Maintenance, вам потребуются дополнительные, «альтернативные» методы.

Но это плохое решение. Вы получаете два фрагмента кода, единственным различием которых является то, что один метод имеет параметр `StingPatrol`, а второй — `NectarCollector`.

К счастью, решить подобные проблемы позволяют **интерфейсы (interfaces)**. Они определяют, какие методы ***должны*** присутствовать в классе.

Методы, указанные в определении интерфейса, ***должны быть*** реализованы. В противном случае ***компилятор выдаст сообщение об ошибке***. Код методов может быть написан непосредственно в рассматриваемом классе или же унаследован от базового класса. Интерфейс не интересует происхождение методов и свойств, главное — чтобы при компиляции кода они были на своем месте.

Ключевое слово interface

Чтобы добавить интерфейс к программе, писать методы не нужно. Достаточно указать их параметры и тип возвращаемого ими значения. И поставить в конце строки точку с запятой.

Интерфейсы не хранят данные, поэтому вы **не сможете добавить к ним поля**. Но *можно* добавить определения свойств. Дело в том, что интерфейс определяет список методов класса с определенными именами, типами и параметрами. Если вам кажется, что проблема может быть решена добавлением поля к интерфейсу, попробуйте **вместо этого добавить свойство** — вполне вероятно, это именно то, что вам было нужно.

Интерфейсам стоит давать имена, начинающиеся с прописной буквы I.
Нет правила, обязывающего вас так поступать, но это делает код более простым. Чтобы убедиться, насколько это облегчает жизнь, установите курсор на пустую строчку внутри любого метода и наберите I, — IntelliSense сразу же покажет вам список доступных интерфейсов .NET.

```
interface IStingPatrol
{
    int AlertLevel { get;}
    int StingerLength { get; set;}
    bool LookForEnemies();
    int SharpenStinger(int length);
}
```

Интерфейс объявляется:

Интерфейсы не хранят данные, они не имеют полей... но они могут иметь свойства.

Любой класс, реализующий этот метод, должен иметь эти методы и свойства, иначе программа не будет компилироваться.

Любому классу, реализующему интерфейс, понадобится метод SharpenStinger() с параметром типа int.

```
interface INectarCollector
{
    void FindFlowers();
    void GatherNectar();
    void ReturnToHive();
}
```

В интерфейсе достаточно упомянуть имена методов, их код писать не требуется. Код написан в классе, реализующем данный метод.

Так как же помочь матке? Теперь она может создать метод, берущий в качестве параметра любой объект, который знает, как защитить улей:

```
private void DefendTheHive(IStingPatrol patroller)
```

Ссылку IStingPatrol можно передать ЛЮБОМУ объекту, реализующему данный интерфейс.

Этот метод может использовать объект `StingPatrol`, `NectarStinger` или любую другую пчелу, знающую, как защитить улей. При реализации `IStingPatrol` метод `DefendTheHive()` гарантирует наличие у объекта свойств и методов, необходимых для защиты улья.

Все элементы открытого интерфейса по умолчанию являются открытыми. То есть с помощью интерфейса вы определяете открытые методы и свойства любого реализующего его класса.

Экземпляр NectarStinger

Используйте **двоеточие**, чтобы **реализовать** интерфейс. После двоеточия сначала указывается класс, от которого происходит наследование, затем список интерфейсов. Если наследования не происходит, то интерфейсы перечисляются в произвольном порядке.

*Как и в случае наследования, для **реализации** интерфейса используется двоеточие.*

*Этот класс **наследует** от класса Worker и реализует интерфейсы INectarCollector и IStingPatrol.*

```
class NectarStinger : Worker, INectarCollector,
IStingPatrol {
        public int AlertLevel
            { get; private set; }

        public int StingerLength
            { get; set; }

        public int Nectar { get; set; }

        public bool LookForEnemies() {...}
        public int SharpenStinger(int length)

        {...}
        public void FindFlowers() {...}
        public void GatherNectar() {...}
        public void ReturnToHive() {...}
}
```

Интерфейсы перечисляются через запятую.

Экземпляр NectarStinger реализует оба интерфейса, ему требуются все их методы и свойства.

Каждому методу в интерфейсе соответствует метод в классе. Иначе программа не будет компилироваться.

Созданный вами объект NectarStinger сможет выполнять работу пчел как из класса NectarCollector, так и из класса StingPatrol.

Класс, реализующий интерфейс так же, как и обычный, создает экземпляры при помощи оператора new и использует методы:

```
NectarStinger bobTheBee = new NectarStinger();

bobTheBee.LookForEnemies();

bobTheBee.FindFlowers();
```

Данная концепция усваивается достаточно тяжело. Если вам что-то непонятно, продолжайте чтение. В этой главе вы найдете много наглядных примеров.

Часто задаваемые вопросы

В: Зачем симулятору улья интерфейсы? Ведь мы добавляем еще один класс `NectarStinger` и все равно получаем дублирующийся код?

О: Интерфейсы и не предназначены для борьбы с дублирующимся кодом. Они просто позволяют использовать один и тот же класс в разных ситуациях. Вам требовалось создать класс рабочих пчел, которые могут выполнять два задания. Интерфейсы дают возможность получить класс, выполняющий произвольное количество заданий. Скажем, у вас есть метод `PatrolTheHive()`, работающий с объектом `StingPatrol`, и метод `CollectNectar()` для объекта `NectarCollector`. При этом хочется, чтобы класс `StingPatrol` мог наследовать от класса `NectarCollector` или наоборот, ведь в каждом классе есть открытые методы и свойства, отсутствующие у другого. А теперь подумайте, как можно создать класс, экземпляры которого могут быть переданы обоим методам. Есть какие-нибудь идеи?

Проблему решают интерфейсы. Создав ссылку IStingPatrol, вы можете указать на любой объект, реализующий `IStingPatrol`, какому бы классу этот объект ни принадлежал. Можно указать как на объект `StingPatrol`, так и на объект `NectarStinger` или еще на что-нибудь. При этом вы можете использовать все методы и свойства, которые являются частью интерфейса `IStingPatrol`, независимо от типа объекта.

Разумеется, вам придется создать новый класс, который и будет реализовывать интерфейс. Так что этот инструмент не позволит избежать создания дополнительных классов или сократить количество дублирующегося кода. Он всего лишь дает возможность получить класс для выполнения нескольких работ без привлечения наследования. Ведь наследуются все методы, свойства и поля другого класса.

Как при работе с интерфейсами избежать дублирующегося кода? Можно создать отдельный класс с именем `Stinger` и кодом, относящимся к укусам или сбору нектара. После чего объекты `NectarStinger` и `NectarCollector` смогут создать закрытый экземпляр `Stinger` и для сбора нектара будут использовать его методы и задавать его свойства.

Классы, реализующие интерфейсы, должны включать ВСЕ методы интерфейсов

Реализация интерфейсов означает, что в классе должны присутствовать все объявленные в интерфейсе методы и свойства. Если это не так, программа не компилируется. Если класс реализует несколько интерфейсов, он должен включать в себя все свойства и методы каждого из них. Впрочем, можете не верить на слово...

1. **Создайте новое приложение и добавьте в него класс IStingPatrol.cs**
IDE, как обычно, добавит файл со строкой **`class IStingPatrol`**. Замените ее на interface **`IStingPatrol`** и введите код интерфейса, приведенный пару страниц назад. Вы только что *добавили* к проекту *интерфейс*! Теперь программа будет компилироваться.

2. **Добавьте к проекту класс Bee**
Но пока не добавляйте ни свойства, ни методы. Заставьте этот класс реализовывать интерфейс IStingPatrol:

```
class Bee : IStingPatrol
{
}
```

3. **Попытайтесь скомпилировать программу**
Выберите команду Rebuild в меню Build. Компилятор не запустится:

Error List

4 Errors | 0 Warnings | 0 Messages | Search Error List

	Description	File	Line	Column
1	'ch07_Bee_Interfaces.Bee' does not implement interface member 'ch07_Bee_Interfaces.IStingPatrol.SharpenStinger(int)'	Bee.cs	9	11
2	'ch07_Bee_Interfaces.Bee' does not implement interface member 'ch07_Bee_Interfaces.IStingPatrol.LookForEnemies()'	Bee.cs	9	11
3	'ch07_Bee_Interfaces.Bee' does not implement interface member 'ch07_Bee_Interfaces.IStingPatrol.StingerLength'	Bee.cs	9	11
4	'ch07_Bee_Interfaces.Bee' does not implement interface member 'ch07_Bee_Interfaces.IStingPatrol.AlertLevel'	Bee.cs	9	11

Error List | Output

Пометка does not implement (не реализует) будет выведена для каждого члена класса IStingPatrol. Компилятор **на самом деле** хочет, чтобы вы реализовали каждый метод интерфейса.

4. **Добавьте в класс Bee методы и свойства**
Добавьте методы `LookForEnemies()` и `SharpenStinger()`. Убедитесь, что их сигнатуры совпадают с представленными в интерфейсе. Метод `LookForEnemies()` должен возвращать логическое значение, а метод `SharpenStinger()` — принимать параметр типа `int` и возвращать такое же значение (сейчас они могут возвращать значения-заполнители). Добавьте свойство `AlertLevel` типа `int` с методом чтения (путь возвращает произвольное значение) и автоматическое свойство `StingerLength` типа `int` с методами чтения и записи.

Убедитесь, что все члены класса Bee помечены как `public`. Теперь программа компилируется!

Учимся работать с интерфейсами

Использовать интерфейсы легко. Вы поймете это, попрактиковавшись. Поэтому создайте новый проект Console Application — и начнем!

1 Вот класс TallGuy (Высокий парень) и код метода Main() в файле *Program.cs*, который создает экземпляр класса при помощи инициализатора объекта и вызывает его метод TalkAboutYourself() (Рассказ о себе). Пока ничего нового:

```
class TallGuy {
    public string Name;
    public int Height;

    public void TalkAboutYourself() {
        Console.WriteLine("My name is " + Name + " and I'm "
           + Height + " inches tall.");
    }
}

static void Main(string[] args) {
    TallGuy tallGuy = new TallGuy() { Height = 74, Name = "Jimmy" };
    tallGuy.TalkAboutYourself();
}
```

2 Вы уже знаете, что элементы интерфейса должны быть открытыми. Проверим это. Добавьте к проекту интерфейс IClown (как вы добавляли классы): щелкните правой кнопкой мыши на имени проекта в окне Solution Explorer, **выберите Add→New Item... и** Interface. Убедитесь, что он называется IClown.cs. IDE создаст интерфейс с объявлением:

```
interface IClown
{
```

Теперь попробуйте объявить внутри интерфейса закрытый метод:

```
private void Honk();
```

Выберите команду Build→Build Solution. Появится сообщение:

1 The modifier 'private' is not valid for this item

Модификатор public внутри интерфейса писать не нужно, доступ ко всем его методам и свойствам имеется по умолчанию.

Удалите **модификатор private**, сообщение об ошибке исчезнет.

3 Перед переходом к следующей странице попробуйте написать остальной код интерфейса IClown и заставить класс TallGuy реализовать этот интерфейс. Интерфейс IClown должен обладать не возвращающим значений и не имеющим параметров методом Honk (Гудок) и предназначенным только для чтения строковым свойством FunnyThingIHave (Смотри, что у меня есть), обладающим методом чтения, которое не имеет метода записи.

4 Вы записали интерфейс вот так?

```
interface IClown
{
    string FunnyThingIHave { get; }
    void Honk();
}
```

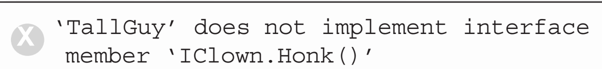
Это пример интерфейса, имеющего метод чтения, но не имеющего метода записи. Помните, что интерфейсы не могут иметь полей, но когда вы реализуете свойство, предназначенное только для чтения, для остальных объектов оно выглядит как поле.

Заставим класс `TallGuy` реализовывать `IClown`. Помните, что после двоеточия сначала ставится имя базового класса (если таковой имеется), а затем список интерфейсов через запятую. В данном случае базовый класс отсутствует, поэтому напишем:

```
class TallGuy : IClown
```

Реализовывать интерфейс IClown будет класс TallGuy.

Убедитесь, что остальной код класса остался без изменений. Выберите команду Build Solution в меню Build, чтобы построить решение. Появятся два сообщения об ошибке. Вот одно из них:



Объявив, что класс TallGuy реализует интерфейс IClown, вы пообещали добавить в этот интерфейс все методы и свойства, но не сделали этого!

5 Как только вы добавите все свойства и методы, упомянутые в интерфейсе, сообщение об ошибке пропадет. Поэтому добавьте предназначенное только для чтения свойство `FunnyThingIHave` с методом чтения, возвращающим строку `большие ботинки`. И добавьте метод `Honk()`, который пишет «Honk honk!» в консоль.

Вот как это выглядит:

```
public string FunnyThingIHave {
    get { return "большие ботинки"; }
}

public void Honk() {

    Console.WriteLine("Honk honk!")
}
```

Интерфейс требует у реализующего его класса наличия свойства FunnyThingIHave с методом чтения. Метод чтения может быть любым, даже возвращающим одну и ту же строку.

Интерфейсу нужен открытый метод Honk, не возвращающий значения, но при этом совершенно не важно, что будет делать этот метод. Если метод присутствует и сигнатура совпадает, программа будет компилироваться.

6 Теперь ничто не мешает компиляции кода! Сделайте так, чтобы кнопка вызывала метод `Honk()` объекта `TallGuy`.

Ссылки на интерфейс

Можно создать массив ссылок IWorker, но получить новые объекты из интерфейса вы не сможете. Впрочем, вам достаточно будет сослаться на новые экземпляры классов, реализующих интерфейс IWorker. Результатом будет массив, хранящий набор различных объектов!

Предположим, у вас есть метод, которому требуется объект, умеющий выполнять метод `FindFlowers()`. Под это условие подходит любой объект, реализующий интерфейс `INectarCollector`. Это может быть объект `Worker`, объект `Robot` или объект `Dog`.

Вы можете сослаться на этот объект и быть уверенными в наличии нужных вам методов.

Попытка создать экземпляр интерфейса приведет к ошибкам компиляции.

Это не работает...

```
IStingPatrol dennis = new IStingPatrol();
```

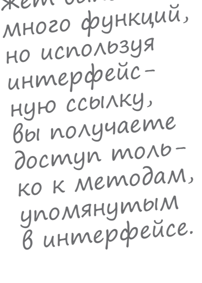
```
1  Cannot create an instance of the abstract class or interface
```

Для интерфейсов ключевое слово `new` не работает, и это имеет смысл, ведь методы и свойства не имеют реализации. Объектам просто неоткуда было бы узнать, как себя вести.

...зато работает это:

```
NectarStinger fred = new NectarStinger();
IStingPatrol george = fred;
```

Помните, вы могли передать ссылку BLT любому классу, который должен ссылаться на сэндвичи, так как класс BLT является производным от класса Sandwich? То же самое происходит и в данном случае: вы можете использовать объект NectarStinger в любом методе или операторе, ожидающем IStingPatrol.



В первой строчке при помощи оператора `new` создается ссылка с именем Fred, указывающая на объект `NectarStinger`.

Со второй строчкой все намного интереснее, так как здесь **при помощи интерфейса `IStingPatrol` создается новая ссылочная переменная**. На первый взгляд код выглядит несколько странно. Но взгляните:

```
NectarStinger ginger = fred;
```

Третий оператор создает новую ссылку на объект `NectarStinger`. Имя этой ссылки `ginger`, и она указывает на тот же самый объект, что и ссылка `fred`. Оператор `george` использует интерфейс `IStingPatrol` аналогичным способом.

Что же случилось?

Тут только один оператор `new`, так что появляется **только один новый объект**. Второй оператор создает ссылочную переменную `george`, которая может указывать на экземпляр **любого класса, реализующий интерфейс `IStingPatrol`**.

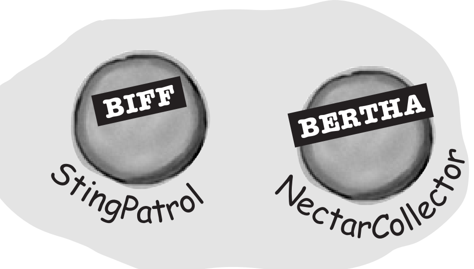
Ссылка на интерфейс аналогична ссылке на объект

Вы уже знаете, как выглядят объекты в куче. Интерфейсная ссылка является всего лишь еще одним способом обратиться к уже знакомым объектам. Это очень просто!

Объекты создаются как всегда.
Вы уже не раз это делали. Оба этих класса реализованы в IStingPatrol

```
StingPatrol biff = new StingPatrol();
NectarCollector bertha = new NectarCollector();
```

Пусть класс StingPatrol реализует интерфейс IStingPatrol, а класс NectarCollector — интерфейс INectarCollector.



Добавьте ссылки на IStingPatrol и INectarCollector.
Интерфейсные ссылки ничем не отличаются от ссылок любого другого типа.

```
IStingPatrol defender = biff;
INectarCollector cutiePie = bertha;
```

Эти два оператора используют интерфейсы для создания новых ссылок на существующие объекты. Интерфейсные ссылки могут указывать только на экземпляры классов, реализующих интерфейс.

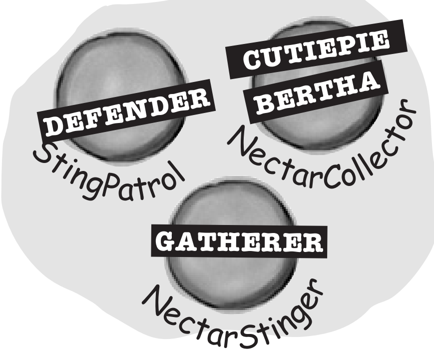
Интерфейсная ссылка позволяет сохранить объект
Объект исчезает, как только на него не остается ссылок. Но никто не говорит, что все ссылки должны принадлежать одному типу! Интерфейсные ссылки позволяют спасти объект от удаления.

```
biff = null;
```

Этот объект не исчезает из-за ссылки defender.

Назначьте интерфейсной ссылке новый экземпляр
На самом деле вам *не нужны* ссылки на объекты, можно создать новый объект и сопоставить его с ссылочной интерфейсной переменной.

```
INectarCollector gatherer = new NectarStinger();
```



Оператор is

Иногда требуется понять, реализуется ли интерфейс определенным классом. Предположим, что все рабочие пчелы представлены в виде массива Bees. В массиве можно хранить переменные типа Worker, так как все рабочие пчелы принадлежат классу Worker или производным от него классам.

Но какая из рабочих пчел может собирать нектар? Для ответа на этот вопрос нужно узнать, реализует ли класс интерфейс INectarCollector. Это можно сделать при помощи оператора **is**.

Рабочие представлены в виде массива Workers. Оператор is позволяет определить тип пчелы.

```
Worker[] bees = new Worker[3];
bees[0] = new NectarCollector();
bees[1] = new StingPatrol();
bees[2] = new NectarStinger();
for (int i = 0; i < bees.Length; i++)
{
    if (bees[i] is INectarCollector)
    {
        bees[i].DoThisJob("Nectar Collector", 3);
    }
}
```

Массив рабочих пчел просматривается в цикле, и оператор is определяет наличие методов и свойств, нужных для выполнения указанной работы.

is сравнивает интерфейсы и другие типы данных.

Эта запись означает, что если данная пчела реализует интерфейс INectarCollector... нужно выполнить приведенный ниже код.

Теперь, когда мы знаем, что эта пчела — сборщик нектара, ее можно отправить за нектаром.

Класс, не производный от класса Worker, но реализующий интерфейс INectarCollector, может выполнять работу по сборке нектара! Но так как класс Worker не является для него базовым, вы не можете поместить его в массив с другими пчелами. Подумайте, каким образом можно получить массив из пчел обоего вида?

Часто Задаваемые Вопросы

В: Свойства, добавляемые в интерфейс, выглядят как автоматически реализуемые. Неужели при реализации интерфейса я могу использовать только такие свойства?

О: Вовсе нет. Свойства внутри интерфейсов действительно напоминают своим видом автоматически реализуемые — посмотрите на свойства Job и ShiftsLeft в IWorker на следующей странице. Реализовать свойство Job можно так:

```
public Job { get;
private set; }
```

Модификатор private set ставится, так как автоматические свойства требуют наличия как метода чтения, так и метода записи (пусть даже и закрытого). Но вы можете написать и другой код:

```
public job { get {
return "Бухгалтер"; } }
```

и программа все равно будет компилироваться. По желанию можно добавить и метод записи. (При реализации же через автоматическое свойство вы всего лишь решаете, будет метод записи открытым или закрытым.)

Интерфейсы и наследование

Когда один класс наследует от другого, он получает все его методы и свойства. **Наследование интерфейсов** происходит еще проще. Так как в интерфейсах отсутствуют тела методов, вам уже не придется заботиться о вызове конструкторов и методов базового класса. Наследующие интерфейсы просто накапливают в себе методы и свойства своих родителей.

```
interface IWorker
{
  string Job { get; }
  int ShiftsLeft { get; }
  void DoThisJob(string job, int shifts)
  void WorkOneShift()
}
```

От созданного нами интерфейса IWorker могут наследовать все остальные интерфейсы.

На диаграмме классов наследование интерфейсов обозначается пунктирной линией.

(interface) IWorker
Job ShiftsLeft
DoThisJob() WorkOneShift()

(interface) IStingPatrol
StingerLength EnemyAlert
SharpenStinger() LookForEnemies()

(interface) INectarCollector
Nectar
FindFlowers() GatherNectar() ReturnToHive()

Класс реализует все методы и свойства

Класс, реализующий интерфейс, должен включать в себя все свойства и методы этого интерфейса. В ситуации, когда один интерфейс наследует от другого, все *их* свойства и методы также должны быть реализованы.

```
interface IStingPatrol : IWorker
{
    int AlertLevel { get;}
    int StingerLength { get; set;}
    bool LookForEnemies();
    int SharpenStinger(int length);
}
```

Уже знакомый вам интерфейс IStingPatrol теперь наследует от интерфейса IWorker. Изменение кажется совсем небольшим, но для всех классов, реализующих IStingPatrol, ситуация коренным образом изменилась.

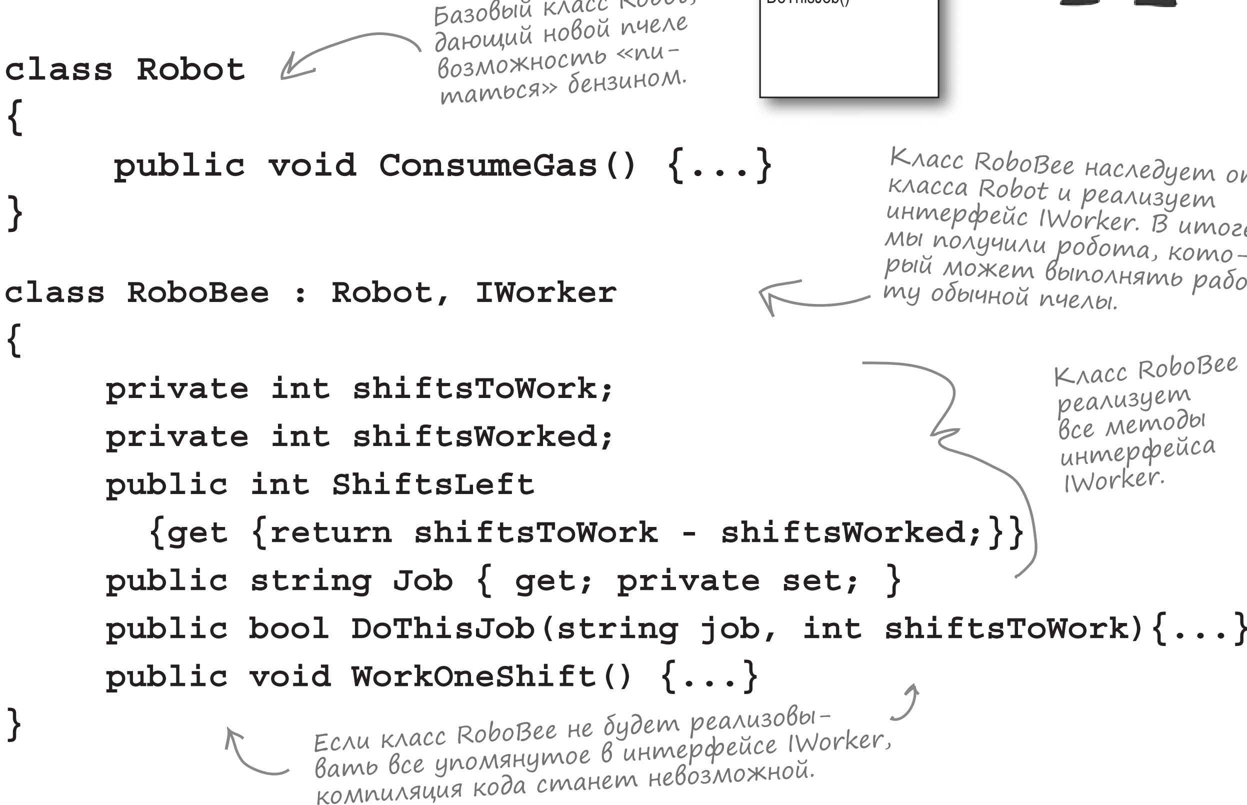
Класс, реализующий интерфейс IStingPatrol, должен реализовывать не только эти методы...

(interface) IWorker
Job ShiftsLeft
DoThisJob() WorkOneShift()

...но и методы интерфейса IWorker, от которого происходит наследование.

RoboBee 4000 функционирует без меда

Создадим пчелу новой формации, RoboBee 4000, работающую на топливе. «Привив» ее интерфейсу интерфейс `IWorker`, вы даете ей возможность делать все то, что делает обычная пчела.




```
class Robot
{
     public void ConsumeGas() {...}
}

class RoboBee : Robot, IWorker
{
     private int shiftsToWork;
     private int shiftsWorked;
     public int ShiftsLeft
       {get {return shiftsToWork - shiftsWorked;}}
     public string Job { get; private set; }
     public bool DoThisJob(string job, int shiftsToWork){...}
     public void WorkOneShift() {...}
}
```

Остальные классы нашего приложения не «увидят» функциональной разницы между пчелой-роботом и обычной пчелой. Оба этих класса реализуют интерфейс `IWorker` и с точки зрения программы действуют как рабочие пчелы.

Отличить объекты друг от друга позволит оператор `is`:

```
if (workerBee is Robot) {
  // мы узнали, что workerBee
  // это объект Robot
}
```

Оператор is показывает, какой класс или интерфейс реализует workerBee и каково его положение в иерархии наследования.

Любой класс может реализовывать ЛЮБОЙ интерфейс, если он реализует все методы и свойства этого интерфейса.

is показывает вам, что именно объект реализует
as показывает компилятору, как обработать этот объект

Иногда требуется вызвать метод, полученный объектом в процессе реализации интерфейса. Но что делать, если вы не знаете, нужному ли типу принадлежит объект? На помощь вам придет оператор **is**. А оператор **as** позволит преобразовать один совместимый ссылочный тип в другой.

```
IWorker[] bees = new IWorker[3];
    bees[0] = new NectarStinger();
    bees[1] = new RoboBee();
    bees[2] = new Worker();

for (int i = 0; i < bees.Length; i++) {
  if (bees[i] is INectarCollector) {
      INectarCollector thisCollector;
      thisCollector = bees[i] as INectarCollector;
      thisCollector.GatherNectar();
      ...
```

Все эти пчелы реализуют интерфейс IWorker, но мы не знаем, какие из них реализуют другие интерфейсы, например INectarCollector.

Мы перебираем пчел...

Мы не можем вызывать для пчел методы интерфейса INectarCollector. Ведь пчелы принадлежат типу IWorker и ничего не знают о методах INectarCollector.

...и проверяем, реализует ли пчела интерфейс INectarCollector.

Теперь можно вызывать методы интерфейса INectarCollector.

Оператор as заставляет использовать указанный объект как реализацию интерфейса INectarCollector.

Возьми в руку карандаш

Для каждого из показанных справа операторов напишите, при каком значении счетчика пчел `i` он будет иметь значение `true`. Зачеркните слева две строчки, которые не компилируются.

```
IWorker[] Bees = new IWorker[8];
Bees[0] = new NectarStinger();
Bees[1] = new RoboBee();
Bees[2] = new Worker();
Bees[3] = Bees[0] as IWorker;
Bees[4] = IStingPatrol;
Bees[5] = null;
Bees[6] = Bees[0];
Bees[7] = new INectarCollector();
```

1. (Bees[i] is INectarCollector)

....

2. (Bees[i] is IStingPatrol)

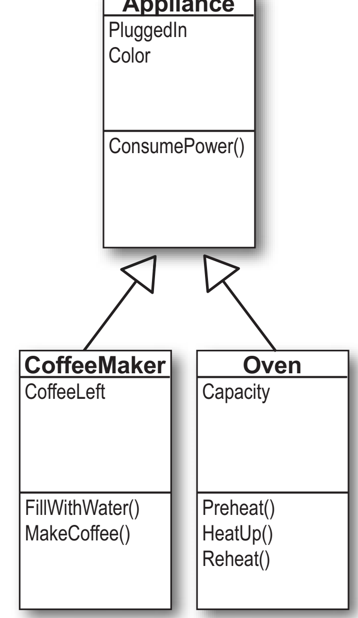
....

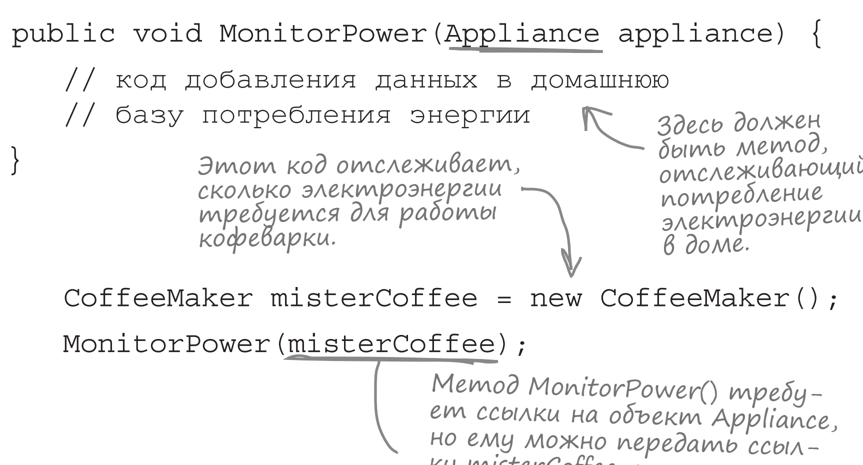
3. (Bees[i] is IWorker)

....

Кофеварка относится к Приборам

Для задачи экономии электроэнергии функции отдельных приборов не имеют значения. Вас заботит только то, что все они потребляют электричество. Поэтому при написании программы учета электроэнергии можно ограничиться классом `Appliance` (Прибор). Но чтобы отличить кофеварку от духовки, потребуется иерархия классов. Методы и свойства, описывающие поведение кофеварки и духовки, будут помещены в классы `CoffeeMaker` и `Oven`. Эти классы будут производными от класса `Appliance`, содержащего общие для них методы и свойства.





Такое поведение вы уже наблюдали в предыдущей главе, когда передавали методу, ожидающему ссылки на Sandwich, ссылку на BLT.

Возьми в руку карандаш — Решение

Вот при каких значениях счетчика пчел `i` операторы справа возвращают значение `true`. Слева зачеркнуты две строчки, препятствующие компиляции.

```
IWorker[] Bees = new IWorker[8];
Bees[0] = new NectarStinger();
Bees[1] = new RoboBee();
Bees[2] = new Worker();
Bees[3] = Bees[0] as IWorker;
Bees[4] = IStingPatrol;
Bees[5] = null;
Bees[6] = Bees[0];
Bees[7] = new INectarCollector();
```

(Строки `Bees[4]` и `Bees[7]` зачеркнуты.)

Метод NectarStinger() реализует интерфейс IStingPatrol.

1. `(Bees[i] is INectarCollector)`

   0, 3 и 6

2. `(Bees[i] is IStingPatrol)`

   0, 3 и 6

3. `(Bees[i] is IWorker)`

   0, 1, 2, 3 и 6

Восходящее приведение

Когда вы используете производный класс вместо базового, например, ссылаясь на кофеварку вместо прибора, — это называется **восходящим приведением (upcasting)**. Это очень мощный инструмент, который вы получаете, построив иерархию классов. К сожалению, он работает только с методами и свойствами базового класса. Другими словами, рассматривая кофеварку как прибор, вы не можете заставить ее сварить кофе MakeCoffee() или налить воду FillWithWater(). Зато вы *можете* определить, включена ли она в розетку, так как это состояние относится ко всем приборам (и именно поэтому свойство PluggedIn помещено в класс Appliance).

1 Создадим объекты

Классы CoffeeMaker и Oven создаются обычным способом:

```
CoffeeMaker misterCoffee = new CoffeeMaker();

Oven oldToasty = new Oven();
```

Для начала получим экземпляры объектов Oven и CoffeeMaker.

2 А вдруг нам потребуется массив приборов?

Объект CoffeeMaker нельзя поместить в массив Oven[], а объекту Oven не место в массиве CoffeeMaker[]. Но они прекрасно уживутся в массиве Appliance[]:

```
Appliance[] kitchenWare = new Appliance[2];

kitchenWare[0] = misterCoffee;

kitchenWare[1] = oldToasty;
```

Восходящее приведение позволяет создать массив, в котором найдется место как для духовки, так и для кофеварки.

3 Не любой прибор является духовкой

Ссылаясь на объект Appliance, вы получаете доступ **только** к методам и свойствам приборов. Вы **не можете** при этом воспользоваться методами и свойствами объекта CoffeeMaker, ***даже если вы знаете, что речь и в самом деле идет о кофеварке.*** Поэтому корректны следующие операторы:

```
Appliance powerConsumer = new CoffeeMaker();

powerConsumer.ConsumePower();
```

powerConsumer — это ссылка класса Appliance на объект CoffeeMaker.

POWER CONSUMER

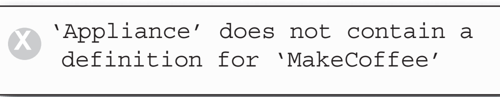
Объект CoffeeMaker

Но написав вот такую строчку:

```
powerConsumer.MakeCoffee();
```

Строчка не будет компилироваться, так как метод powerConsumer работает только со свойствами объекта Appliance.

вы получите сообщение об ошибке:

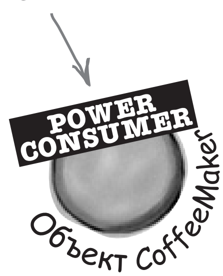


так как после восходящего приведения можно пользоваться только методами и свойствами **одного уровня с ссылкой**, которую вы используете для доступа к объекту.

Нисходящее приведение

Теперь вы знаете, что, рассматривая кофеварку и духовку как приборы, вы лишаетесь доступа к их собственным методам и свойствам. К счастью, существует процедура **нисходящего приведения (downcasting)**. Узнать, относится ли рассматриваемый объект Appliance к классу CoffeeMaker, можно при помощи оператора **is**. Теперь ничто не мешает вам вернуться от класса Appliance к классу CoffeeMaker при помощи оператора **as**.

Вот ссылка Appliance, указывающая на объект CoffeeMaker.



1 **Начнем с объекта CoffeeMaker**

Вот код, который мы использовали для восходящего приведения:

```
Appliance powerConsumer = new CoffeeMaker();
powerConsumer.ConsumePower();
```

2 **Но как превратить Appliance обратно в класс CoffeeMaker?**

Для начала воспользуйтесь оператором is для проверки совместимости.

```
if (powerConsumer is CoffeeMaker)
    // мы можем осуществить нисходящее приведение!
```

Теперь, когда вы точно знаете, что ваш прибор — кофеварка...

... вы можете воспользоваться оператором as для нисходящего приведения, чтобы снова получить доступ к методам и свойствам класса CoffeeMaker. Так как класс CoffeeMaker наследует от класса Appliance, доступ к методам и свойствам базового класса у него тоже сохранится.

```
if (powerConsumer is CoffeeMaker) {
    CoffeeMaker javaJoe = powerConsumer as CoffeeMaker;
    javaJoe.MakeCoffee();
}
```

Ссылка javaJoe указывает на тот же объект, что и ссылка powerConsumer. Но она относится к классу CoffeeMaker и дает возможность вызвать метод MakeCoffee().

Неудачное нисходящее приведение возвращает null

А что произойдет, если при помощи оператора as попытаться преобразовать объект Oven в объект CoffeeMaker? Вы получите нулевой результат, и программа прекратит работу.

```
if (powerConsumer is CoffeeMaker) {
    Oven foodWarmer = powerConsumer as Oven;
    foodWarmer.Preheat();
}
```

Они же не совпадают!

Объект powerConsumer не относится к классу Oven. Поэтому при попытке осуществить нисходящее приведение ссылка foodWarmer даст null. Вот что произойдет при попытке использовать пустую ссылку...

An unhandled exception of type 'System.NullReferenceException' occurred in UpcastingDowncastingExample.exe

Нисходящее и восходящее приведение интерфейсов

Вы уже видели, что операторы is и as работают с интерфейсами. Значит, для них возможны операции восходящего и нисходящего приведения. Добавим интерфейс ICooksFood к любому классу, который умеет подогревать еду. Добавим также класс Microwave (Микроволновка). Наряду с классом Oven он будет реализовывать интерфейс ICooksFood. В результате вы получите три способа доступа к объекту Oven. А функция IntelliSense подскажет, какие операции вы можете производить в каждом из этих трех случаев:

```
Oven misterToasty = new Oven();
misterToasty.
```

Capacity
Color
ConsumePower
Equals
GetHashCode
GetType
HeatUp
PluggedIn
Preheat
Reheat

int Oven.Capacity

Сразу после ввода точки функция IntelliSense выведет список всех возможных членов.

misterToasty — это **ссылка класса Oven**, указывающая на объект Oven, так что у вас есть доступ ко всем свойствам и методам этого класса... но это ссылка менее общего типа, поэтому указывать она может только на объекты класса Oven.

Любой класс, реализующий интерфейс ICooksFood, относится к приборам, которые могут разогревать HeatUp() еду.

(interface) ICooksFood: Capacity; HeatUp(), Reheat()

Oven: Capacity; Preheat(), HeatUp(), Reheat()

Microwave: Capacity; HeatUp(), Reheat(), MakePopcorn()

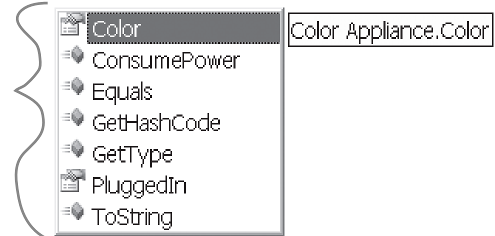
```
ICooksFood cooker;
if (misterToasty is ICooksFood)
    cooker = misterToasty as ICooksFood;
    cooker.
```

Capacity
Equals
GetHashCode
GetType
HeatUp
Reheat
ToString

int ICooksFood.Capacity

Ссылка cooker принадлежит **интерфейсу ICooksFood** и указывает на тот же объект Oven. Она дает доступ только к членам интерфейса ICooksFood, но при этом может указывать на объект Microwave.

```
Appliance powerConsumer;
if (misterToasty is Appliance)
    powerConsumer = misterToasty;
    powerConsumer.
```



Ссылка powerConsumer принадлежит **классу Appliance**. Она дает доступ к открытым полям, методам и свойствам этого класса. Но при желании вы можете с ее помощью сослаться на объект CoffeeMaker.

Три ссылки на один и тот же объект дают доступ к различным методам и свойствам, в зависимости от своего типа.

Часто Задаваемые Вопросы

В: Почему восходящее приведение можно осуществлять всегда, а нисходящее нет?

О: Компилятор может предупредить вас о том, что восходящее приведение происходит неправильно. Более того, эта операция не работает, только когда вы пытаетесь сопоставить объект классу, от которого не происходит наследования, или интерфейсу, который этим объектом не реализуется. Компилятор распознает невозможность подобной операции и выводит сообщение об ошибке.

При этом компилятор не может проверить, допустимо ли нисходящее присваивание, которое вы пытаетесь осуществить. Справа от оператора `as` может располагаться любое имя класса или интерфейса. В случае, когда нисходящее присваивание невозможно, оператор `as` возвращает значение `null`. Компилятор допускает такое поведение, потому что бывают случаи, когда именно оно и требуется.

В: Кто-то говорил мне, что интерфейс подобен контракту, но я не понимаю почему.

О: Да, в определенной степени это действительно так. Заставляя класс реализовывать интерфейс, вы как бы обещаете компилятору поместить в него определенные методы. И компилятор следит, чтобы вы это обещание выполнили.

Хотя намного нагляднее представить интерфейс в виде списка. По этому списку компилятор проверяет присутствие в классе всех методов, упомянутых в интерфейсе.

В: Могу ли я поместить тело метода в интерфейс?

О: Нет, компилятор не позволит вам это сделать. Интерфейс не должен содержать операторы. Оператор в виде двоеточия, реализующий интерфейс, не имеет отношения к оператору, использующемуся при наследовании классов. Реализация интерфейса ничего не меняет в классе. Она всего лишь гарантирует присутствие в классе методов, перечисленных в интерфейсе.

В: Интерфейс выглядит как наложение ограничений, ничего не меняющих в самом классе. Зачем мне его использовать?

О: Если класс реализует интерфейс, интерфейсная ссылка может указывать на любой экземпляр этого класса. Это позволяет обойтись одним ссылочным типом, который работает с набором объектов различного вида.

Вот маленький пример. Лошадь, буйвол, мул и вол могут тащить телегу. Но в нашем симуляторе зоопарка `Horse`, `Ox`, `Mule` и `Steer` — разные классы. Для катания в парке аттракционов вы хотите создать массив животных, которые могут тащить тележку. Но поместить в один массив животных из разных классов можно только в случае, когда все они наследуют от общего базового класса. В нашем случае это условие не соблюдается. Что же делать?

Вам потребуется интерфейс `IPuller` с методами, отвечающими за перемещение тележки. Теперь вы можете объявить массив:

```
IPuller[] pullerArray;
```

В этот массив можно поместить ссылку на любое животное, реализующее интерфейс `IPuller`.

В: Можно ли реализовать интерфейс, не вводя много кода?

О: Конечно! Средства IDE позволяют реализовать интерфейс автоматически. Начните вводить код класса:

```
class
   Microwave : ICooksFood
   { }
```

Щелкните на ICooksFood — под буквой I появится маленькая полоска. Если задержать на ней курсор, появится значок:

interface ICooksFood

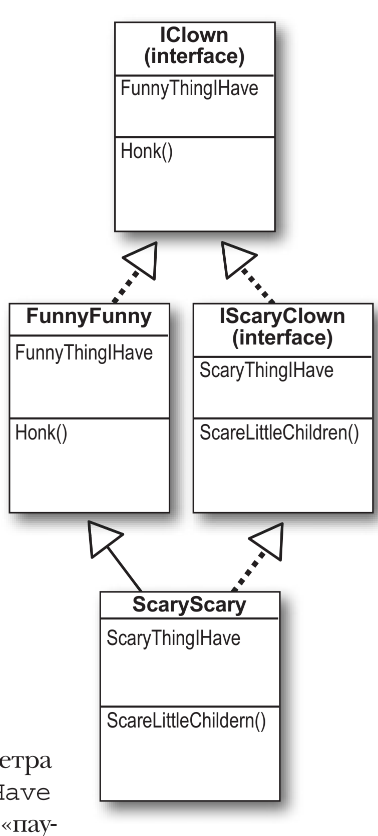
ICooksFood

Если щелкнуть на значке не удается, используйте комбинацию Ctrl-точка.

Щелкните на значке и выберите команду Implement Interface 'ICooksFood'. Это автоматически добавит все члены, которые еще не реализуют интерфейс. Каждый из них снабжен оператором `throws`, о котором мы подробно поговорим в главе 10.

Интерфейс напоминает список, с которым сверяется компилятор, проверяя, реализует ли ваш класс определенный набор методов.

Расширьте интерфейс IClown при помощи реализующих его классов.



Начните с интерфейса `IClown` из Задания! на с. 320:

```
interface IClown {
    string FunnyThingIHave { get; }
    void Honk();
}
```

2 Создайте производный интерфейс `IScaryClown` со свойством `ScaryThingIHave` типа `string` и методом чтения, но без метода записи, а также с не возвращающим значения методом `ScareLittleChildren()`.

3 Создайте классы для забавных и страшных клоунов:

★ Класс `FunnyFunny` с закрытой строковой переменной, хранящей список забавного. На основе параметра `FunnyThingIHave` конструктор задает значение закрытого поля. Метод `Honk()` выводит сообщение «Привет! У меня есть », далее следует возвращаемое значение метода записи `FunnyThingIHave`.

★ Класс `ScaryScary` хранит в закрытой переменной целое число, переданное конструктором в виде параметра `numberOfScaryThings`. Метод чтения `ScaryThingIHave` возвращает строку с числом из конструктора и словом «пауков». Метод `ScareLittleChildren()` вызывает окно с текстом «Ага! Попался!»

4 А это неработающий код. Вам нужно найти ошибки и заставить его работать.

```
static void Main(string[] args) {
    ScaryScary fingersTheClown = new ScaryScary("big shoes", 14);
    FunnyFunny someFunnyClown = fingersTheClown;
    IScaryClown someOtherScaryClown = someFunnyClown;
    someOtherScaryClown.Honk();
    Console.
```

Пальцы клоуна ужасны.

Если не найдешь ошибку... то...

Упражнение
Решение

Вот как выглядит расширение интерфейса IClown с реализующими его классами.

```
interface IClown {
    string FunnyThingIHave { get; }
    void Honk();
}

interface IScaryClown : IClown {
string ScaryThingIHave { get; }
    void ScareLittleChildren();
}

class FunnyFunny : IClown {
    public FunnyFunny(string funnyThingIHave) {
        this.funnyThingIHave = funnyThingIHave;
    }
    private string funnyThingIHave;
    public string FunnyThingIHave {
        get { return "Привет! У меня есть " + funnyThingIHave; }
    }

    public void Honk() {
        Console.WriteLine(this.FunnyThingIHave);
    }
}

class ScaryScary : FunnyFunny, IScaryClown {
    public ScaryScary(string funnyThingIHave, int numberOfScaryThings)
        : base(funnyThingIHave) {
        this.numberOfScaryThings = numberOfScaryThings;
    }

    private int numberOfScaryThings;
    public string ScaryThingIHave {
        get { return "У меня " + numberOfScaryThings + " пауков"; }
    }

    public void ScareLittleChildren() {
        Console.WriteLine("Ага! Попался!");
    }
}

static void Main(string[] args) {
    ScaryScary fingersTheClown = new ScaryScary("большие ботинки", 14);
    FunnyFunny someFunnyClown = fingersTheClown;
    IScaryClown someOtherScaryclown = someFunnyClown as ScaryScary;
    someOtherScaryclown.Honk();
    Console.ReadKey()
}
```

Метод Honk() использует этот метод записи для отображения сообщения, что избавляет вас от дублирующегося кода.

Можно еще раз реализовать этот метод и свойство интерфейса IClown, но почему бы не унаследовать их от FunnyFunny?

Так как ScaryScary — это производный класс от FunnyFunny, который реализует интерфейс IClown, ScaryScary также будет реализовывать интерфейс IClown.

Ссылка FunnyFunny может указывать на объект ScaryScary, так как он принадлежит к производному от FunnyFunny классу. Но ссылка IScaryClown не может указывать на произвольного клоуна, так как клоун может оказаться нестрашным. Поэтому вы используете оператор as.

Ссылку someOtherScaryClown можно использовать для вызова метода ScareLittleChildren(), но вы не сможете вызвать этот метод ссылкой someFunnyClown.

Модификаторы доступа

Вы уже знаете, насколько важным является ключевое слово private, как его нужно использовать и чем оно отличается от ключевого слова public. В C# подобные ключевые слова называются **модификаторами доступа (access modifiers)**. Меняя модификатор свойства, поля, метода или даже всего класса, вы меняете способ доступа других классов к указанным элементам. В этом разделе мы вспомним про уже известные вам модификаторы и познакомимся с новыми.

Методы, поля и свойства класса называются его **членами (members)**. Любой член может быть помечен модификатором доступа public или private.

(До тех пор, пока существует доступ к объявленному классу.)

- **public означает свободный доступ**
  Пометив класс или его члены модификатором public, вы объявляете открытый доступ для всех экземпляров всех классов. Это наименее ограничивающий из модификаторов. И вы уже видели, причиной каких проблем он может стать. Используйте его только тогда, когда это действительно необходимо.

- **private означает доступ только для других членов этого же класса**
  Пометив члены класса модификатором private, вы оставляете доступ к ним только для других членов этого же класса или **экземпляров этого же класса**. Сам класс можно пометить словом private, только если он **находится внутри другого класса**. После этого доступ к нему сохранится только у экземпляров этого внешнего класса.

Отсутствие модификатора доступа при объявлении члена класса означает, что будет использован вариант Private.

- **protected означает открытый только для производных классов**
  Вы уже видели, что из производных классов не всегда имеется доступ к полям базовых, что не всегда удобно. Но любой член класса с модификатором protected доступен как в рамках его собственного класса, так и из методов производных классов.

- **internal означает открытый для других классов в сборке**
  Встроенные классы .NET Framework являются **сборками (assemblies)** — библиотеками классов, на которые можно ссылаться из вашего проекта. Их список можно увидеть, щелкнув правой кнопкой мыши на пункте References в окне Solution Explorer и выбрав команду Add Reference... Если при построении сборки воспользоваться модификатором internal, доступ к классам будет осуществляться только изнутри сборки. Существует вариация этого модификатора protected internal, воспользовавшись которой вы ограничите доступ текущей сборкой и типами, которые являются производными от содержащего класса.

Отсутствие модификатора доступа при объявлении класса или интерфейса означает, что будет использован вариант internal. При этом класс становится доступен для любого другого класса в составе сборки. Если сборка всего одна, модификатор internal является аналогом модификатора public для классов и интерфейсов. Откройте какой-нибудь старый проект, поменяйте доступ некоторых классов на internal и посмотрите, что получится.

- **sealed означает, что от данного класса нельзя наследовать**
  Существуют классы, наследование от которых невозможно. К ним относятся многие классы .NET Framework: попробуйте, к примеру, создать класс, наследующий от класса String (метод этого класса IsEmptyOrNull() вы использовали в предыдущей главе). Компилятор выдаст сообщение об ошибке «cannot derive from sealed type 'string'». Чтобы запретить наследование от созданного вами класса, достаточно добавить ключевое слово sealed после модификатора доступа.

Ключевое слово Sealed не относится к модификаторам доступа.

Изменение видимости при помощи модификаторов доступа

Посмотрим, как модификаторы доступа влияют на **видимость** различных членов класса. Пометим вспомогательное поле funnyThingIHave словом protected и отредактируем метод ScareLittleChildren(), использующий поле funnyThingIHave:

Внесите эти изменения в решение. Затем верните модификатору доступа значение private и посмотрите, к каким ошибкам это приведет.

1 Перед вами два интерфейса. IClown для клоуна с набором смешных вещей и производный от него IScaryClown. Страшный клоун не только имеет все функции обычного клоуна, но еще и пугает маленьких детей.

```
interface IClown {
    string FunnyThingIHave { get; }
    void Honk();
}

interface IScaryClown : IClown {
    string ScaryThingIHave { get; }
    void ScareLittleChildren();
}
```

Слово this указывает, на какую именно переменную вы ссылаетесь. Оно говорит «Посмотри на текущий экземпляр класса».

Ключевое слово this позволяет отличить вспомогательное поле от одноименного параметра. Имя funnyThingIHave относится к параметру, в то время как запись this.funnyThingIHave обозначает вспомогательное поле.

2 Класс FunnyFunny реализует интерфейс IClown. Поле funnyThingIHave помечено модификатором protected, значит, доступ к нему имеют все экземпляры производного класса.

```
class FunnyFunny : IClown {
    public FunnyFunny(string funnyThingIHave) {
        this.funnyThingIHave = funnyThingIHave;
    }
    protected string funnyThingIHave;
    public string FunnyThingIHave {
        get { return "Привет! У меня есть " + funnyThingIHave;
}
    }

    public void Honk() {
        Console.WriteLine(this.FunnyThingIHave);
    }
}
```

Посмотрите, как модификатор protected повлиял на метод ScaryScary.ScareLittleChildren().

Употребив рядом со свойством ключевое слово this, вы запускаете метод чтения или метод записи.

Ключевое слово this указывает, что в данном случае имеется в виду вспомогательное поле, а не одноименный параметр.

3 Класс ScaryScary реализует интерфейс IScaryClown и наследует от класса FunnyFunny, реализующего интерфейс IClown. Посмотрите, как именно метод ScareLittle Children() осуществляет доступ к вспомогательному полю funnyThingIHave. Такое поведение связано с модификатором protected. При модификаторе private компиляция кода стала бы невозможной.

Модификаторы доступа под увеличительным стеклом

```
class ScaryScary : FunnyFunny, IScaryClown {
    public ScaryScary(string funnyThingIHave,
                      int numberOfScaryThings)
        : base(funnyThingIHave) {
        this.numberOfScaryThings = numberOfScaryThings;
    }

    private int numberOfScaryThings;
    public string ScaryThingIHave {
        get { return "У меня " + numberOfScaryThings + " пауков"; }
    }

    public void ScareLittleChildren() {
         Console.WriteLine("Ты не можешь забрать "
                              + base.funnyThingIHave);
    }
}
```

Параметр numberOfScaryThings закрыт, как это обычно бывает со вспомогательными полями. Соответственно он доступен только для экземпляров класса ScaryScary.

Ключевое слово protected оставляет доступ к элементу **только** со стороны экземпляров производного класса.

Ключевое слово base заставляет использовать значение из базового класса. Но в данном случае можно воспользоваться также ключевым словом this. Вы понимаете почему?

Наличие при переменной funnyThing IHave модификатора private привело бы к сообщению об ошибке. Ключевое слово protected делает ее видимой из классов, производных от класса FunnyFunny.

4 Этот метод создает экземпляры FunnyFunny и ScaryScary. Обратите внимание, как с помощью оператора as осуществляется нисходящее приведение someFunnyClown к ссылке IScaryClown.

```
static void Main(string[] args) {
    ScaryScary fingersTheClown = new ScaryScary("большие ботинки", 14);
    FunnyFunny someFunnyClown = fingersTheClown;
    IScaryClown someOtherScaryclown = someFunnyClown as ScaryScary;
    someOtherScaryclown.Honk();
    Console.ReadKey();
}
```

Программа искусственно удлинена, чтобы показать возможность восходящего приведения от ScaryScary к FunnyFunny, а затем — нисходящего приведения к IScaryClown. Вместо этих трех строчек можно написать всего одну. Попробуйте сделать это самостоятельно.

Так как метод Main() не является частью классов FunnyFunny или ScaryScary, он не может обратиться к защищенному полю funnyThingIHave.

Кнопка не принадлежит ни одному из классов, поэтому операторы имеют доступ только к открытым членам объектов FunnyFunny или ScaryScary.

Часто задаваемые вопросы

В: Зачем нужен интерфейс, если все необходимые методы можно написать непосредственно в теле класса?

О: По мере усложнения программ классов становится слишком много. Интерфейсы позволяют сгруппировать эти классы в соответствии с выполняемыми задачами. Это гарантирует, что для выполнения определенной работы классы будут использовать одни и те же методы. Благодаря интерфейсу вам не придется беспокоиться о том, как именно выполняется эта работа.

Представим, что у вас есть классы грузовиков и парусников, реализующие интерфейс перевозки пассажиров `ICarryPassenger`. Он требует у реализующих его классов наличия метода `ConsumeEnergy()`. Программа может использовать оба класса для перевозки пассажиров, хотя метод `ConsumeEnergy()` для парусников использует силу ветра, а для грузовиков — дизельное топливо.

Без интерфейса `ICarryPassenger` программе сложно объяснить, какие транспортные средства могут перевозить пассажиров, а какие — нет. Вам потребовалось бы просматривать все классы, чтобы определить в них наличие методов, пригодных для перевозки пассажиров, а потом вызывать эти методы вручную. Без стандартного интерфейса они, скорее всего, назывались бы слишком разнородно, кроме того, могли оказаться скрытыми в теле других методов. Запутаться в этом очень легко.

В: Зачем нужны свойства, если есть поля?

О: Интерфейс определяет способ, которым класс выполняет конкретную работу. При этом он не является объектом, поэтому вы не можете создавать экземпляры и хранить в них информацию. Добавив поле путем объявления переменной, вы ставите перед программой задачу сохранить данные. Свойство же, с точки зрения остальных объектов, выглядит как поле, но по сути является методом и не нуждается в хранении данных.

В: Чем обычная ссылка отличается от интерфейсной?

О: Как работают обычные ссылки, вы уже знаете. Если создать экземпляр `Skateboard` с именем `VertBoard` и ссылку на него с именем `HalfPipeBoard`, они будут указывать на один объект. Но если `Skateboard` реализует интерфейс `IStreetTricks`, а вы создаете ссылку на `Skateboard` с именем `StreetBoard`, она будет знать только методы класса `Skateboard`, являющиеся частью интерфейса `IStreetTricks`.

Все три ссылки указывают на один объект. Но если ссылки `HalfPipeBoard` и `VertBoard` дают доступ ко всем свойствам и методам объекта, ссылка `StreetBoard` видит только те методы и свойства, которые указаны в интерфейсе.

В: Получается, интерфейсные ссылки ограничивают мои возможности. Зачем же мне их тогда использовать?

О: Интерфейсные ссылки позволяют работать с различными объектами, выполняющими одну функцию. С их помощью вы можете создать массив, обменивающийся информацией с методами интерфейса `ICarryPassenger`, неважно, работаете вы с грузовиком, лошадью или автомобилем. Способы, которыми эти объекты выполняют работу, различаются, но благодаря интерфейсным ссылкам вы знаете, что все они имеют одни и те же методы, одинаковые параметры и возвращают значения одинаковых типов. Это дает вам возможность единообразно вызывать их и передавать им информацию.

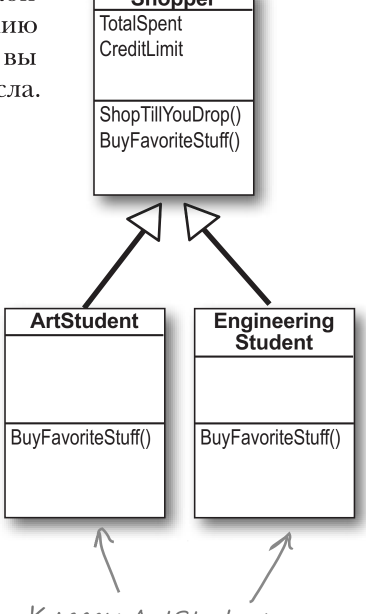
В: Зачем мне может понадобиться ключевое слово protected?

О: Оно позволяет инкапсулировать класс. Иногда требуется доступ из производного класса к внутренней части базового. Но при построении классов поля оставляют открытыми, только если без этого совсем нельзя обойтись. Модификатор доступа protected позволяет открыть поля, доступ к которым нужен производному классу, оставив их закрытыми для остальных объектов.

Интерфейсные ссылки «знают» только о тех свойствах и методах, которые упомянуты в интерфейсе.

Классы, для которых недопустимо создание экземпляров

Помните иерархию классов, использованную для симулятора зоопарка? Закончили вы ее множеством экземпляров бегемотов, собак и львов. Но в иерархию входят и классы `Canine` и `Feline`, а также базовый класс `Animal`. Думаем, вы уже поняли, что создание экземпляров некоторых классов не имеет смысла. Рассмотрим дополнительный пример:



Начнем с базового класса, описывающего поведение студентов в магазине.

```
class Shopper {
    public void ShopTillYouDrop()
        while (TotalSpent < CreditLimit)
            BuyFavoriteStuff();
    }
    public virtual void BuyFavoriteStuff () {
        // Реализация здесь неуместна - мы не знаем,
        // что любит покупать наш студент!
    }
}
```

Класс `ArtStudent` производный от класса `Shopper`:

```
class ArtStudent : Shopper {
    public override void BuyFavoriteStuff () {
        BuyArtSupplies();
        BuyBlackTurtlenecks();
        BuyDepressingMusic();
    }
}
```

И класс `EngineeringStudent` наследует от класса `Shopper`:

```
class EngineeringStudent : Shopper {
    public override void BuyFavoriteStuff () {
        BuyPencils();
        BuyGraphingCalculator();
        BuyPocketProtector();
    }
}
```

Что произойдет с появлением экземпляра класса Shopper? Имеет ли смысл его создавать?

Абстрактный класс. Перепутье между классом и интерфейсом

Предположим, вам требуется интерфейс, чтобы заставить классы реализовывать определенные методы и свойства. Но при этом вы хотите включить в этот интерфейс некий код, чтобы определенные методы не приходилось реализовывать во всех наследующих классах. На помощь вам придет **абстрактный класс (abstract class)**. Этот элемент имеет свойства интерфейса, но позволяет записать внутри себя код.

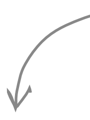
Метод, не имеющий тела, называется **абстрактным (abstract method)**. Наследующие классы должны реализовывать все абстрактные методы, как и в случае наследования от интерфейса.

Абстрактные методы могут находиться только внутри абстрактных классов. Если поместить внутрь класса абстрактный метод и не пометить сам класс словом abstract, программа перестанет компилироваться.

Абстрактный класс напоминает обычный
Как уже знакомый обычный класс, абстрактный класс имеет поля и методы и даже позволяет осуществлять наследование. Фактически вы уже знаете, что умеет делать абстрактный класс!

Абстрактный класс напоминает интерфейс
Создавая класс, реализующий интерфейс, вы соглашаетесь реализовывать все определенные в рамках интерфейса свойства и методы. Абстрактный класс работает аналогичным способом — все объявленные в нем свойства и методы должны реализовываться в производных классах.

Противоположностью абстрактному является конкретное. Конкретный метод обладает телом. Конкретными являются все классы, с которыми вы работали до этого момента.



Создавать экземпляры абстрактного класса нельзя
Самым большим отличием абстрактного класса является невозможность создать его экземпляр при помощи оператора new. Попытавшись это сделать, вы получите сообщение об ошибке.

Ошибка связана с наличием абстрактных методов, не содержащих кода! Компилятор не позволяет создать экземпляр класса, который не содержит кода, точно так же как не позволял создавать экземпляры интерфейсов.

```
Cannot create an instance of the
abstract class or interface 'MyClass'
```

Что? Класс, от которого я не могу получить экземпляры? Зачем он вообще нужен?

Иногда источником части кода являются производные классы.

Бывает так, что создание ненужных объектов ***имеет плохие последствия***. Поля самого верхнего класса иерархии обычно задаются в производных классах. В классе Animal могут находиться вычисления, зависящие от значения логической переменной HasTail или Vertebrate, но в нем невозможно задать эту переменную.

Вот еще один пример...

Класс PlanetMission клуб астрофизиков использует для отправки ракет к различным планетам.

```
class PlanetMission {
  public long RocketFuelPerMile;
  public long RocketSpeedMPH;
  public int MilesToPlanet;

  public long UnitsOfFuelNeeded() {
    return MilesToPlanet * RocketFuelPerMile;
  }

  public int TimeNeeded() {
    return MilesToPlanet / (int) RocketSpeedMPH;
  }

  public string FuelNeeded() {
    return "You'll need "
      + MilesToPlanet * RocketFuelPerMile
      + " units of fuel to get there. It'll take "
      + TimeNeeded() + " hours.";
  }
}
```

Присваивать этим полям значения в базовом классе бессмысленно, потому что мы не знаем, какая ракета куда полетит.

Один полет совершается на Венеру, другой — на Марс.

```
class Venus : PlanetMission {
    public Venus() {
        MilesToPlanet = 40000000;
        RocketFuelPerMile = 100000;
        RocketSpeedMPH = 25000;
    }
}

class Mars : PlanetMission {
    public Mars() {
        MilesToPlanet = 75000000;
        RocketFuelPerMile = 100000;
        RocketSpeedMPH = 25000;
    }
}
```

Конструкторы производных классов Mars и Venus задают значения трех полей, унаследованных от PlanetMission. Но поля **не могут получить значения** от экземпляра PlanetMission. Что произойдет, если их попытается использовать метод FuelNeeded()?

```
private void button1_Click(object s, EventArgs e) {
  Mars mars = new Mars();
  MessageBox.Show(mars.FuelNeeded());
}
```

You'll need 7500000000000 units of fuel to get there. It'll take 3000 hours.

OK

```
private void button2_Click(object s, EventArgs e) {
  Venus venus = new Venus();
  MessageBox.Show(venus.FuelNeeded());
}
```

You'll need 4000000000000 units of fuel to get there. It'll take 1600 hours.

OK

```
private void button3_Click(object s, EventArgs e) {
  PlanetMission planet = new PlanetMission();
  MessageBox.Show(planet.FuelNeeded());
}
```

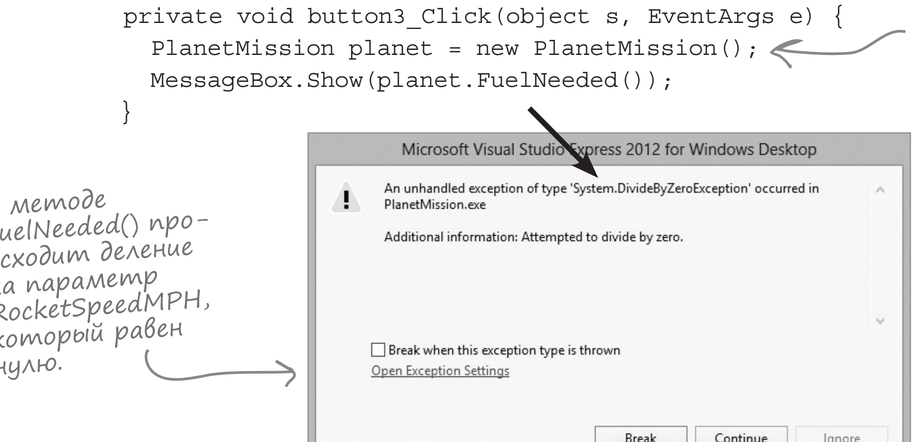
Перед тем как перевернуть страницу, подумайте, что произойдет после щелчка на третьей кнопке...

Как уже было сказано, создавать экземпляры некоторых классов недопустимо

В программе с предыдущей страницы проблемы начинаются с появлением экземпляра `PlanetMission`. Метод `FuelNeeded()` этого класса ожидает, что значения его полей будут заданы в производном классе. Если этого не происходит, им по умолчанию присваивается нулевое значение. А при попытке поделить на ноль...

```
private void button3_Click(object s, EventArgs e) {
  PlanetMission planet = new PlanetMission();
  MessageBox.Show(planet.FuelNeeded());
}
```

Класс PlanetMission не был предназначен для создания экземпляров на его основе. Предполагалось, что от него можно будет только наследовать. Но мы создали экземпляр, вот тут-то и начались проблемы...



В методе FuelNeeded() происходит деление на параметр RocketSpeedMPH, который равен нулю.

Решение: абстрактный класс

Для классов, помеченных словом `abstract`, C# не позволяет создавать экземпляры. Как и в случае интерфейса, возможно только наследование.

Модификатор abstract указывает, что класс может играть роль базового класса.

```
abstract class PlanetMission {
  public long RocketFuelPerMile;
  public long RocketSpeedMPH;
  public int MilesToPlanet;

  public long UnitsOfFuelNeeded() {
    return MilesToPlanet * RocketFuelPerMile;
  }

  // здесь определяется остальная часть класса
}
```

Теперь компиляция программы невозможна, пока мы не уберем строчку, создающую экземпляр PlanetMission.

Вернитесь к программе для планирования вечеринок, которую мы писали для Кэтлин в прошлой главе, и еще раз посмотрите на иерархию классов. Имело ли смысл создавать экземпляр класса `Party` или лучше запретить это, пометив класс ключевым словом abstract?.

Абстрактный метод не имеет тела

Как вы знаете, в интерфейсе объявляются методы и свойства, но отсутствует их код. Это потому, что каждый метод в составе интерфейса является **абстрактным (abstract method)**. Давайте его реализуем! Сообщение об ошибке должно исчезнуть. Продолжая абстрактный класс, нужно гарантировать перекрытие всех его абстрактных методов. Достаточно напечатать «public override», и как только вы нажмете пробел, появится список доступных для перекрытия методов. Выберите вариант `SetMissionInfo()` и напишите:

Интерфейс содержит только абстрактные методы, поэтому ключевое слово abstract применяется, только когда речь идет об абстрактных классах. Такие классы могут иметь в своем составе не только абстрактные, но и конкретные методы.

```
abstract class PlanetMission {

    public abstract void SetMissionInfo(
        int milesToPlanet, int rocketFuelPerMile,
        long rocketSpeedMPH);

    // остальной код класса...
```

По виду абстрактный метод напоминает интерфейс, у него нет тела, но при этом любой класс, наследующий от PlanetMission, должен реализовать метод SetMissionInfo(). Или программа не будет компилироваться.

Попытавшись построить программу, вы получите сообщение об отсутствии реализации унаследованного абстрактного члена:

```
'VenusMission' does not implement inherited abstract
member 'PlanetMission.SetMissionInfo(int, int, long)'
```

Так давайте реализуем его! Ошибка сразу исчезнет.

```
class Venus : PlanetMission {

    public Venus() {
        SetMissinInfo(40000000, 100000, 25000);
    }
    public override SetMissionInfo(int milesToPlanet, long rocketFuelPerMile,
     int rocketSpeedMPH) {
        this.MilesToPlanet = milesToPlanet;
        this.RocketFuelPerMile = rocketFuelPerMile;
        this.RocketSpeedMPH = rocketSpeedMPH;
    }
}
```

Наследуя от абстрактного класса, нужно перекрыть все его абстрактные методы.

Класс Mars отличается от класса Venus только параметрами. Что вы думаете об этой иерархии классов? Имеет ли смысл сделать SetMissionInfo() абстрактным? Или вместо этого нужен конкретный метод в классе PlanetMission?

Возьми в руку карандаш

Продемонстрируйте свое искусссство. Слева вы видите набор классов и объявлений интерфейсов. Нарисуйте справа соответствующие диаграммы классов, как показано в примере номер один. Не забудьте, что пунктирная линия показывает наследование от интерфейса, в то время как сплошная — наследование от класса.

Дано:

1)

```
interface Foo { }
class Bar : Foo { }
```

2)

```
interface Vinn { }
abstract class Vout : Vinn { }
```

3)

```
abstract class Muffie : Whuffie { }
class Fluffie : Muffie { }
interface Whuffie { }
```

4)

```
class Zoop { }
class Boop : Zoop { }
class Goop : Boop { }
```

5)

```
class Gamma : Delta, Epsilon { }
interface Epsilon { }
interface Beta { }
class Alpha : Gamma,Beta { }
class Delta { }
```

Диаграмма

1) (interface) Foo ← Bar

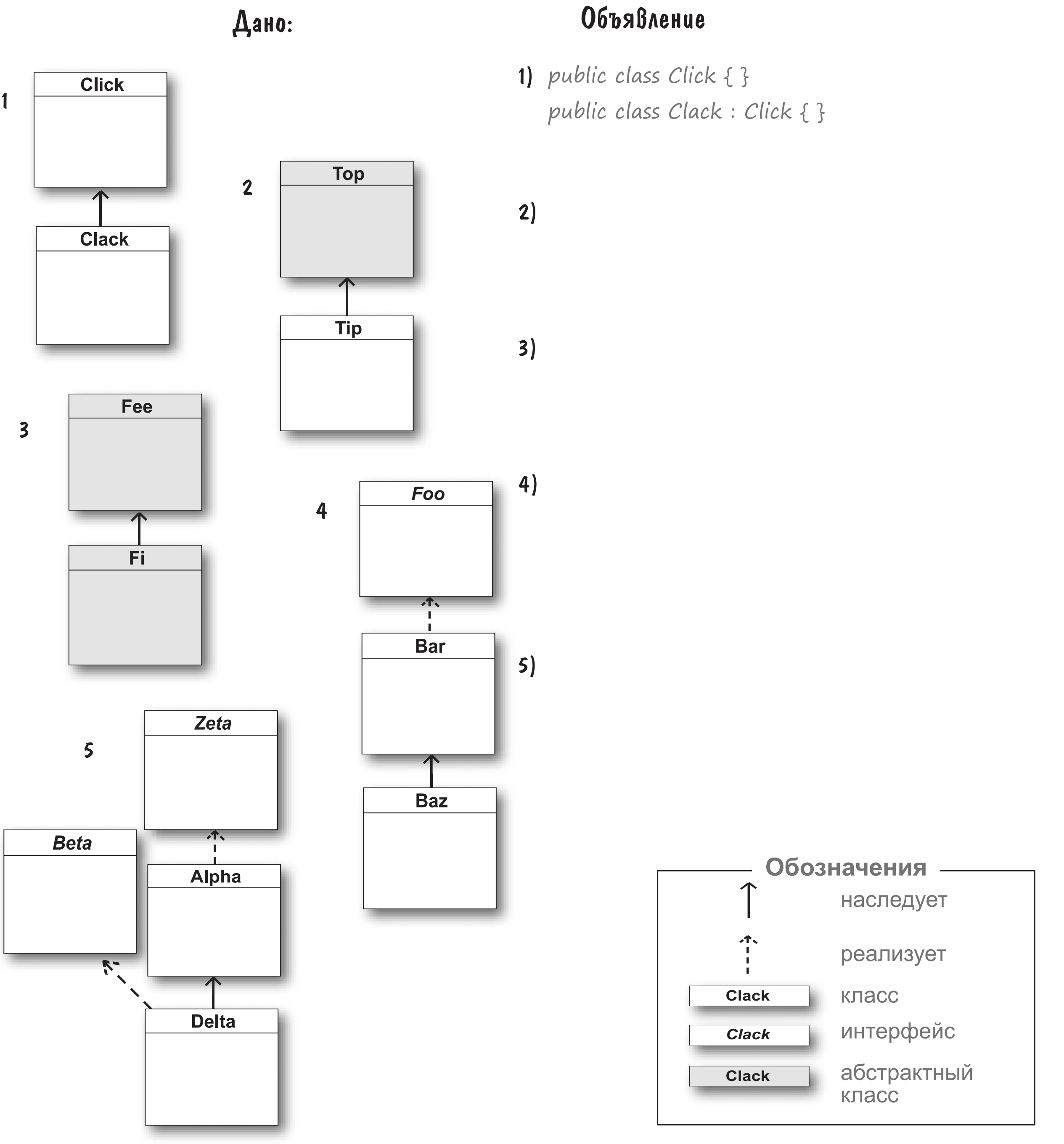
2)

3)

4)

5)

А в этом задании слева приведены диаграммы классов, вам же нужно превратить их в объявления, как показано в первом примере.



Кто важнее: абстрактный класс или интерфейс?

Абстрактный класс: Я думаю, абсурдна сама постановка вопроса: кто из нас важнее. Без меня программист не выполнит свою работу. Посмотрим правде в глаза: ты и близко ко мне не можешь подойти.

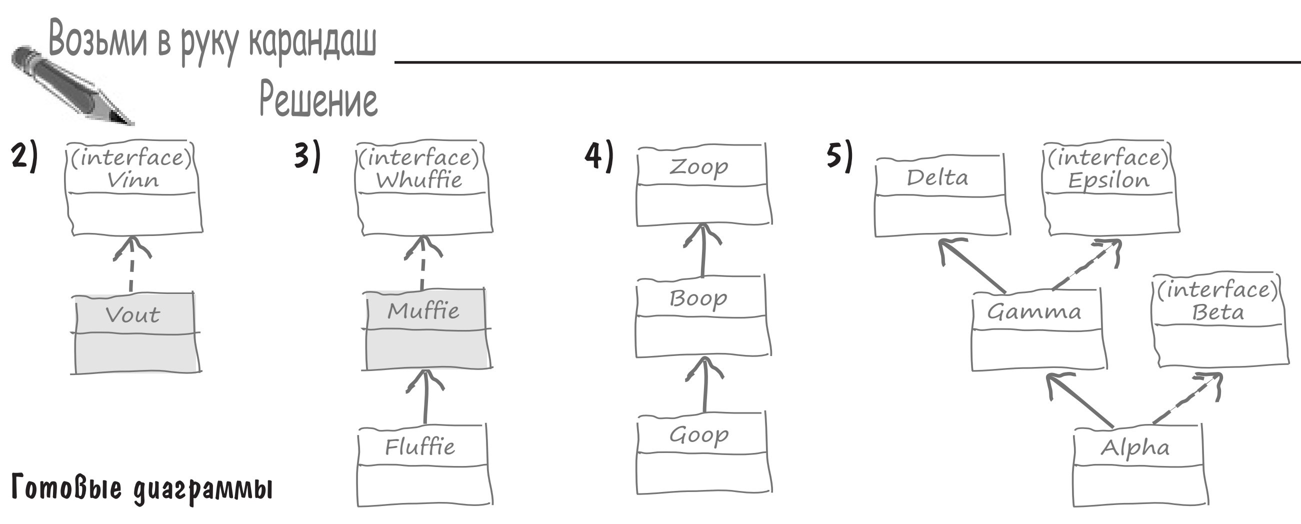
Интерфейс: Прекрасно. Другого я от тебя и не ожидал!

Абстрактный класс: Как ты вообще мог подумать, что можешь быть важнее меня? Ты даже не в состоянии наследовать как полагается, тебя можно только реализовать.

Интерфейс: Только полный невежда может гордиться тем, что интерфейсы не используют механизм наследования, а реализуются. Реализация ничем не хуже наследования, а в чем-то даже превосходит его!

Абстрактный класс: Превосходит? Да ты сошел с ума. Я намного более гибок. Я могу иметь как обычные, так и абстрактные методы. И даже виртуальные методы, если захочу. Да, я не создаю экземпляры, но этого не можешь и ты. Зато моя функциональность ничем не хуже, чем у обычного класса.

Интерфейс: Да? А если класс захочет наследовать у тебя ***и*** у твоего товарища? **Наследовать у 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.



Абстрактный класс

Кажется, ты несколько переоцениваешь свою роль.

Интерфейс

Думаешь, раз ты содержишь код, лучше тебя никого нет? Но ты не поспоришь с фактом, что наследовать можно только от одного класса за раз. Так что ты ограничен. Да, я не содержу кода, но роль кода сильно преувеличена.

Абстрактный класс

Такую чепуху можно услышать только от интерфейса. Нет ничего важнее кода! Именно он заставляет программу работать.

Интерфейс

Девять из десяти, что программисту нужны определенные свойства и методы, но его при этом не волнует, как именно они реализуются.

Абстрактный класс

Да ну? По моим наблюдениям, программистов очень даже волнует содержимое свойств и методов.

Интерфейс

Да, конечно. Только вспомни, как часто программисты пишут методы с объектами в качестве параметров, которые просто должны включать в себя определенные методы. При этом никого не волнует, как именно эти методы построены. Главное, чтобы они были. В этой ситуации достаточно написать интерфейс, и проблема решена!

Абстрактный класс

Да, конечно... расскажи кодеру, что он не может писать код.

Интерфейс

Не имеет значения!

---

2)
```
abstract class Top { }
class Tip : Top { }
```

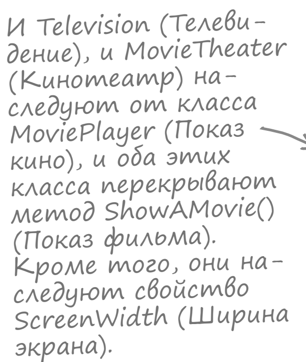
3)
```
abstract class Fee { }
abstract class Fi : Fee { }
```

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.

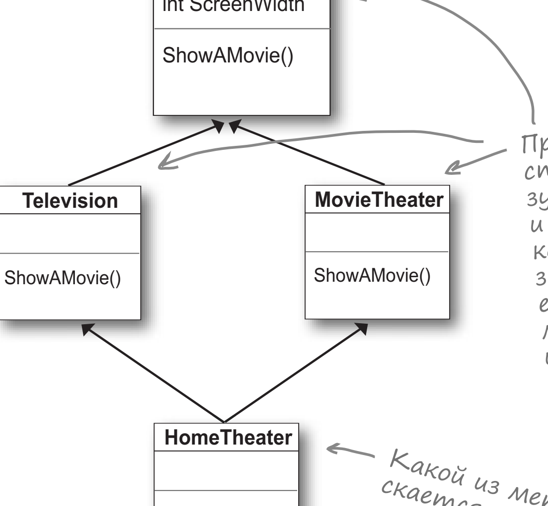
Корректные объявления



Это не недостаток, а защита.

Разрешить существование нескольких базовых классов — все равно что открыть банку с червями. Существуют языки программирования, допускающие **множественное наследование (multiple inheritance)**. Но предоставив вместо этой функции интерфейсы, C# спасает вас от большой путаницы, которую мы хотели бы назвать...

Смертельным ромбом!



Избегайте неопределенности!

В языках, допускающих смертельный ромб, возможны крайне неприятные ситуации, так как для работы с подобными неопределенностями требуются специальные правила... А это означает дополнительные усилия при написании программы! C# позволяет этого избежать. Сделайте `Television` и `MovieTheater` интерфейсами, и вам хватит одного метода `ShowAMovie()`. Главное, чтобы этот метод присутствовал в указанном вами месте.

Возьмите фрагменты кода из бассейна и поместите их на пустые строчки. Каждый фрагмент можно использовать несколько раз. В бассейне есть и лишние фрагменты. Вам нужно получить набор классов, которые будут компилироваться, запускаться и давать показанный ниже результат.

```
..................... Nose {
  ..................;
    string Face { get; }
}

abstract class ..............:..............{
    public virtual int Ear()
    {
        return 7;
    }
    public Picasso(string face)
    {
     .................... = face;
    }
    public virtual string Face {
     ..........{ ......................; }
    }
    string face;
}

class ............: ..............{
    public Clowns() : base("Clowns") { }
}
```

```
class ............: .................{
    public Acts() : base("Acts") { }
    public override ..................{
        return 5;
    }
}

class ...........: ............ {
    public override string Face {
        get { return "Of76"; }
    }
    public static void Main(string[] args) {
        string result = "";
        Nose[] i = new Nose[3];
        i[0] = new Acts();
        i[1] = new Clowns();
        i[2] = new Of76();
        for (int x = 0; x < 3; x++) {
            result += ( ..................+ " "
                      + .................) + "\n";
        }
        Console.WriteLine(result);
        Console.ReadKey();
    }
}
```

Точка входа находится здесь.

Результат

```
file:///C:/Users/Pul
5 Acts
7 Clowns
7 Of76
```

Каждый фрагмент кода можно использовать несколько раз.

Acts( );
Nose( );
Of76( );
Clowns( );
Picasso( );

Of76 [ ] i = new Nose[3];
Of76 [3] i;
Nose [ ] i = new Nose( );
Nose [ ] i = new Nose[3];

:
;
class
abstract
interface
int Ear()
this
this.
face
this.face

i
i( )
i(x)
i[x]

get
set
return

class
5 class
7 class
7 public class

i.Ear(x)
i[x].Ear()
i[x].Ear(
i[x].Face

Acts
Nose
Of76
Clowns
Picasso

Ответ на с. 367.

Думаю, что теперь я хорошо умею управлять объектами!

Идея объединить данные и код в классы на момент своего появления была революционной, но теперь это вполне обычная практика программирования.

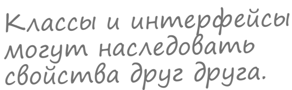
Вы объектно-ориентированный программист.

То, чем вы занимаетесь, называется **объектно-ориентированным программированием** (ООР). До появления таких языков, как C#, объекты и методы при написании кода не использовались. Применялись функции (которые в тех языках назывались методами), сосредоточенные в одном месте. Можно сказать, что каждая программа имела один статический класс, наполненный статическими методами. Писать программы было намного сложнее. К счастью, вам не придется писать программы без ООР, так как это ключевая часть языка C#.

Четыре принципа объектно-ориентированного программирования

Объектно-ориентированное программирование опирается на четыре принципа, которые вам уже знакомы. Вы пользовались ими при написании программ: **наследование**, **абстракция** и **инкапсуляция**. Последний принцип называется немного странно — **полиморфизм**, но и с ним вы на самом деле уже сталкивались.

Так называется создание объектов, которые отслеживают свое состояние при помощи закрытых полей, а для других классов предоставляют открытые свойства и методы. Таким образом, другие классы видят только ту часть данных, которую им нужно видеть.



Инкапсуляция

Это слово буквально означает «множество форм». Можете ли вы представить ситуацию, когда объект принимает разные формы?

Абстракция

Создание модели, начинающейся с более общих — абстрактных — классов, от которых наследуют более подробные классы.

Полиморфизм

Различные формы объекта

Помните, как вы использовали *Пересмешник* вместо *Животное и выдержанный Вермонт* вместо *Сыр*? Именно такое поведение называется **полиморфизмом**. И именно его вы используете при восходящем и нисходящем приведении. Другими словами, вы вызываете методы и свойства объекта независимо от их реализации.

О полиморфизме можно говорить, когда вы берете экземпляр класса и используете его в операторе или методе, которые ожидают значение другого типа, например, из родительского класса или реализуемого интерфейса.

Пример полиморфизма

Вам предстоит выполнить очень большое (как никогда ранее) упражнение, в котором вам придется то и дело использовать полиморфизм, так что будьте внимательны. Ниже показаны четыре типичных способа применения этого явления. Пытайтесь их отслеживать по мере выполнения упражнения:

- [ ] Взять ссылочную переменную одного класса и присвоить ей экземпляр другого класса.

```
NectarStinger bertha = new NectarStinger();
INectarCollector gatherer = bertha;
```

- [ ] Восходящее приведение путем использования производного класса в операторе или методе, ожидающем значение из базового класса.

```
spot = new Dog();
zooKeeper.FeedAnAnimal(spot);
```

Для метода FeedAnAnimal(), ожидающего объект класса Animal, допустимо передавать объект класса Dog, который наследует от класса Animal.

- [ ] Создание ссылочной переменной интерфейсного типа и нацеливание ее на объект, реализующий интерфейс.

```
IStingPatrol defender = new StingPatrol();
```

Это тоже восходящее приведение!

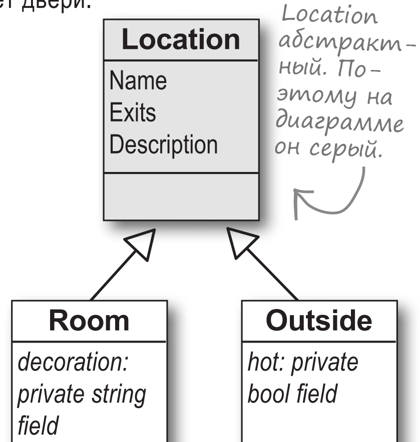
- [ ] Нисходящее приведение при помощи оператора as.

```
void MaintainTheHive(IWorker worker) {
    if (worker is HiveMaintainer) {
        HiveMaintainer maintainer = worker as HiveMaintainer;
        ...
```

Метод MaintainTheHive() в качестве параметра использует интерфейс IWorker. Оператор as позволяет нацелить ссылку HiveMaintainer на объект worker.

Длинные упражнения

Давайте построим дом! В модели дома классы будут представлять комнаты и прочие помещения, а интерфейс пусть соответствует двери.



1 Начнем с модели классов

Каждая комната и помещение должны быть представлены отдельным объектом. Внутренние комнаты наследуют от класса `Room` (Комната), а внешние от класса `Outside` (Снаружи). Для этих классов, в свою очередь, имеется базовый класс `Location` (Помещение) с двумя полями: `Name` для названия помещения (Кухня) и `Exits` (Выходы) массив объектов, связанный с отдельными помещениями. В итоге запись `diningRoom.Name` будет иметь значение `Dining Room` (Столовая), а запись `diningRoom.Exits` будет соответствовать массиву `{ LivingRoom, Kitchen }`.

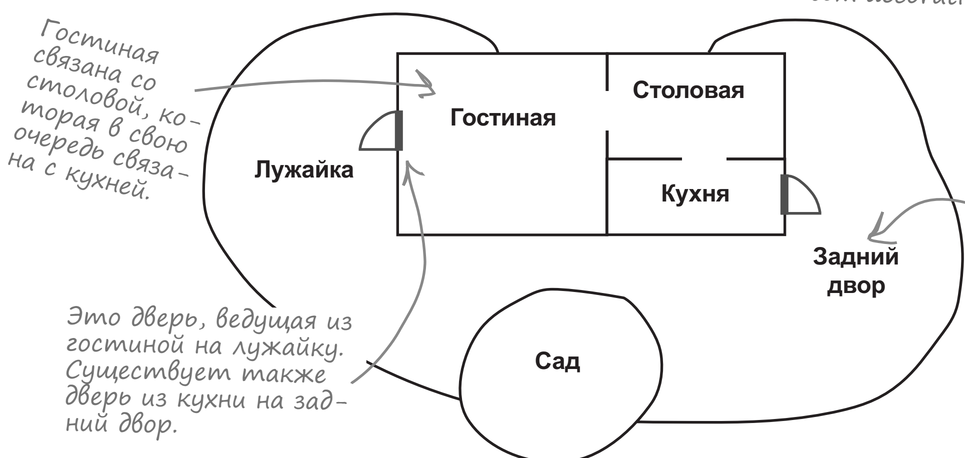
✦ **Создайте проект Windows Forms Application и добавьте к нему классы `Location`, `Room` и `Outside`.**

Интерьер помещений описывается предназначенным только для чтения свойством decoration.

2 Нарисуем план дома

В доме три комнаты, лужайка, задний двор и сад. Внешних дверей две: одна ведет из гостиной на лужайку, а вторая — из кухни на задний двор.

Определить, жарко ли снаружи, поможет предназначенное только для чтения булево свойство Hot.



Переместиться с лужайки на задний двор можно через сад.

Все комнаты имеют двери, но только некоторые двери ведут наружу.

IHasExteriorDoor
DoorDescription DoorLocation

3 Для комнат с внешней дверью создадим интерфейс IHasExteriorDoor

Помещения с ведущими наружу дверями (лужайка, задний двор, гостиная и кухня) должны реализовывать интерфейс `IHasExteriorDoor`. Предназначенное только для чтения свойство `DoorDescription` содержит описание двери (передняя дверь «Дубовая с латунной ручкой», в то время как сзади у нас «Калитка»). Свойство `DoorLocation` содержит ссылку на помещение, в которое ведет дверь (Кухня).

4 **Класс Location**

Вот код для абстрактного класса Location:

Конструктор задает значение поля name, которое является предназначенным только для чтения полем свойства Name.

Открытое поле Exits является массивом ссылок Location, который отслеживает, какие помещения связаны с тем, в котором находитесь вы.

Метод Description виртуальный, его нужно перекрыть.

```
abstract class Location {
    public Location(string name) {
        Name = name;
    }
    public Location[] Exits;

    public string Name { get; private set; }

    public virtual string Description {
        get {
            string description = "Вы находитесь в " + name
                + ". Вы видите двери, ведущие в: ";
            for (int i = 0; i < Exits.Length; i++) {
                description += " " + Exits[i].Name;
                if (i != Exits.Length - 1)
                    description += ",";
            }
            description += ".";
            return description;
        }
    }
}
```

Свойство Description возвращает строку с описанием комнаты и всех прилегающих к ней помещений (их список содержится в поле Exits[]). Так как в производных классах описание будет меняться, свойство требуется перекрыть.

Класс Room перекрывает и расширяет метод Description, добавляя к нему интерьер. К методу Outside он добавит температуру.

Помните, что от класса Location можно наследовать, объявлять ссылочные переменные типа Location, но нельзя создавать экземпляры.

5 **Создание классов**

Начнем с классов Room и Outside. Затем создадим еще два класса: OutsideWithDoor, наследующий от класса Outside и реализующий интерфейс IHasExteriorDoor, и RoomWithDoor, который является производным от класса Room и также реализует интерфейс IHasExteriorDoor.

Вот как выглядит описание этих классов:

Дополнительную информацию вы получите на следующей странице.

```
class OutsideWithDoor : Outside, IHasExteriorDoor
{
   // Тут будет свойство DoorLocation (Положение двери)
   // А здесь свойство DoorDescription (Описание двери)
}

class RoomWithDoor : Room, IHasExteriorDoor
{
   // Тут будет свойство DoorLocation (Положение двери)
   // А здесь свойство DoorDescription (Описание двери)
}
```

Это будет очень большое упражнение… но мы обещаем, что вы получите удовольствие! И обязательно запомните новый материал.

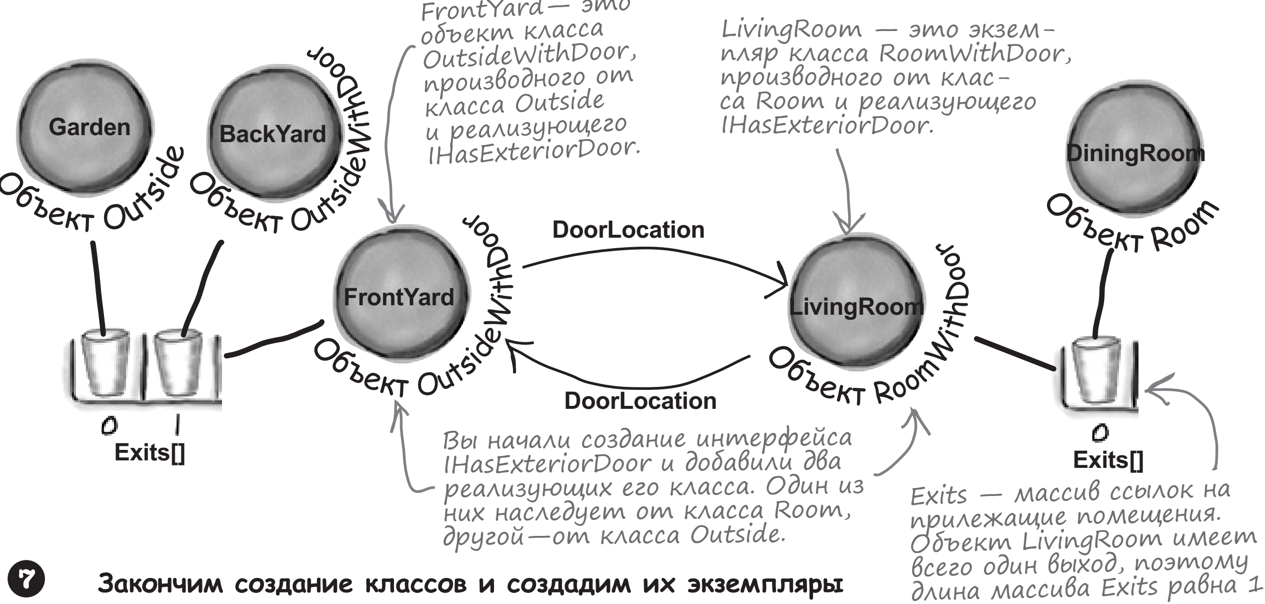
Переверните страницу и продолжим!

Длинные упражнения

Пришло время создать объекты, представляющие различные части дома, и добавить форму для работы с ними.

6 **Как работают объекты**

Рассмотрим архитектуру объектов frontYard и livingRoom. Из-за наличия дверей они должны быть экземплярами класса, реализующего IHasExteriorDoor. Свойство DoorLocation хранит ссылку на помещение, расположенное по другую сторону двери.



7 **Закончим создание классов и создадим их экземпляры**

Практически все готово для построения объектов. Вам осталось:

★ Убедиться, что конструктор класса Outside задает предназначенное только для чтения свойство Hot и перекрывает свойство Description, добавляя текст «Тут очень жарко», когда переменная Hot имеет значение true. Жарко должно быть на заднем дворе, но не на лужайке и не в саду.

★ Конструктор класса Room должен задавать свойство Decoration и перекрывать свойство Description, добавляя «Здесь вы видите *(интерьер)*». В гостиной находится старинный ковер, в столовой — хрустальная люстра, а на кухне — плита из нержавеющей стали и сетчатая дверь, ведущая на задний двор.

★ Форма должна создавать объекты и хранить на них ссылки. Добавьте метод CreateObjects(), который будет вызываться конструктором формы.

★ Создайте по экземпляру для шести помещений дома. Вот пример для гостиной:

```
RoomWithDoor livingRoom = new RoomWithDoor("Гостиная",
     "старинный ковер" , "дубовая дверь с латунной ручкой");
```

Метод CreateObjects() должен добавлять поле Exits[] к каждому объекту:

```
frontYard.Exits = new Location[] { backYard, garden };
```

Каждое помещение будет иметь собственное поле в классе формы.

Exits создает массив из двух строк.

Здесь нужны фигурные скобки.

8 **Построение формы**

Создадим простую форму для экскурсии по дому. Потребуется большое текстовое поле `description`, в котором будут появляться описания помещения. Элемент `ComboBox` с именем `exits` содержит список выходов из комнаты. Кнопка `goHere` перемещает вас в помещение, выбранное в `ComboBox`, а кнопка `goThroughTheDoor` появляется при наличии выхода наружу.

Содержимое списка ComboBox задается здесь.

Explore the House

Это **многострочное** текстовое поле с именем **description**, отображающее описание текущего помещения.

Элемент ComboBox содержит список выходов, поэтому присвоим ему имя **exits**. Выберите для свойства DropDownStyle вариант DropDownList.

Кнопка **goHere** позволяет перейти в другое помещение.

Go here:

Элемент ComboBox

Go through the door

Кнопка **goThroughTheDoor** появляется, если вы находитесь в комнате с дверью на улицу. Для настройки видимости присвойте свойству Visible значение true или false.

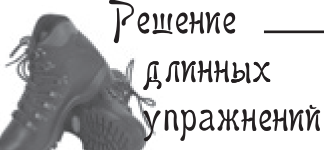
9 **Заставим форму работать!**

Осталось соединить друг с другом отдельные части.

- ★ Форме потребуется поле с именем `currentLocation`, определяющее текущее положение.
- ★ Добавьте метод `MoveToANewLocation()` с параметром `Location`, присваивающий свойству `currentLocation` новое значение. Затем он будет очищать раскрывающийся список при помощи метода `Items.Clear()` и добавлять новые имена из массива `Exits[]` при помощи метода `Items.Add()`. Наконец, останется отобразить первый пункт раскрывающегося списка, присвоив его свойству `SelectedIndex` значение ноль.
- ★ В текстовом поле должно появиться описание текущего помещения.
- ★ Оператор **`is`** проверяет наличие дверей в помещении. При их обнаружении нужно отобразить кнопку Go through the door, воспользовавшись свойством Visible.
- ★ Щелчок на кнопке Go here должен перемещать в помещение, выбранное в раскрывающемся списке.
- ★ Щелчок на кнопке Go through the door должен приводить к перемещению через дверь.

Подсказка: индекс элемента, выбранного в раскрывающемся списке, совпадает с индексом соответствующего помещения в массиве Exits[].

Поле формы currentLocation — это ссылка класса Location. Хотя она указывает на объект, реализующий интерфейс IHasExteriorDoor, написать «currentLocation.DoorLocation» нельзя, так как DoorLocation не является полем класса Location. Вам потребуется нисходящее приведение.



Вот код для модели дома. Комнаты и другие помещения представлены с помощью классов, в то время как интерфейс соответствует дверям.

```
interface IHasExteriorDoor {
    string DoorDescription { get; }
    Location DoorLocation { get; set; }
}

class Room : Location {
    private string decoration;

    public Room(string name, string decoration)
        : base(name) {
        this.decoration = decoration;
    }

    public override string Description {
        get {
            return base.Description + " Вы видите " + decoration + ".";
        }
    }
}

class RoomWithDoor : Room, IHasExteriorDoor {
    public RoomWithDoor(string name, string decoration, string doorDescription)
        : base(name, decoration)
    {
        DoorDescription = doorDescription;
    }

    public string DoorDescription { get; private set; }

    public Location DoorLocation { get; set; }
}
```

Это интерфейс IHasExteriorDoor.

Класс Room наследует от класса Location и добавляет поле для предназначенного только для чтения свойства Decoration. Значение данного поля задает конструктор класса.

Класс RoomWithDoor наследует от класса Room и реализует интерфейс IHasExteriorDoor. В дополнение к функциям класса Room в конструктор добавляется описание внешней двери. Появляется ссылка DoorLocation, указывающая, куда именно ведет дверь. Интерфейс IHasExteriorDoor требует методов DoorDescription и DoorLocation.

Вы использовали вспомогательные поля вместо автоматически реализуемых свойств? Это тоже корректное решение.

```
class Outside : Location {
    private bool hot;
    public bool Hot { get { return hot; } }

    public Outside(string name, bool hot)
        : base(name)
    {
        this.hot = hot;
    }

    public override string Description {
        get {
            string NewDescription = base.Description;
            if (hot)
                NewDescription += " Очень жарко.";
            return NewDescription;
        }
    }
}

class OutsideWithDoor : Outside, IHasExteriorDoor {
    public OutsideWithDoor(string name, bool hot, string doorDescription)
        : base(name, hot)
    {
        this.doorDescription = doorDescription;
    }

    public string DoorDescription { get; private set; }

    public Location DoorLocation { get; set; }

    public override string Description {
        get {
            return base.Description + " Вы видите " + DoorDescription + ".";
        }
    }
}
```

Класс Outside во многом подобен классу Room. Он тоже является производным от класса Location и добавляет вспомогательное поле для свойства Hot. Это свойство используется в методе Description().

Класс OutsideWithDoor наследует от класса Outside и реализует интерфейс IHasExteriorDoor. По виду он напоминает класс RoomWithDoor.

Свойство Description базового класса получает значение в зависимости от того, будет ли жарко в рассматриваемом помещении. Оно зависит от исходного свойства Description класса Location и поэтому включает информацию о помещении и выходах.

Переверните страницу и продолжим!

Решение длинных упражнений

Это код формы, расположенный в файле Form1.cs, внутри объявления Form1.

```
public partial class Form1 : Form
{
    Location currentLocation;

    RoomWithDoor livingRoom;
    Room diningRoom;
    RoomWithDoor kitchen;

    OutsideWithDoor frontYard;
    OutsideWithDoor backYard;
    Outside garden;

    public Form1() {
       InitializeComponent();
       CreateObjects();
       MoveToANewLocation(livingRoom);
    }

    private void CreateObjects() {
       livingRoom = new RoomWithDoor("Гостиная", "старинный ковер",
               "дубовая дверь с латунной ручкой");
       diningRoom = new Room("Столовая", "хрустальная люстра");
       kitchen = new RoomWithDoor("Кухня", "плита из нержавеющей стали", "сетчатая дверь");

       frontYard = new OutsideWithDoor("лужайка", false, "дубовая дверь с латунной ручкой");
       backYard = new OutsideWithDoor("Задний двор", true, "сетчатая дверь");
       garden = new Outside("Сад", false);

       diningRoom.Exits = new Location[] { livingRoom, kitchen };
       livingRoom.Exits = new Location[] { diningRoom };
       kitchen.Exits = new Location[] { diningRoom };
       frontYard.Exits = new Location[] { backYard, garden };
       backYard.Exits = new Location[] { frontYard, garden };
       garden.Exits = new Location[] { backYard, frontYard };

       livingRoom.DoorLocation = frontYard;
       frontYard.DoorLocation = livingRoom;

       kitchen.DoorLocation = backYard;
       backYard.DoorLocation = kitchen;
    }
```

Форма отслеживает, в какой комнате вы находитесь в данный момент.

При помощи этих ссылочных переменных форма следит за каждым помещением в доме.

Конструктор формы создает объект и использует метод MoveToANewLocation для перехода в другое помещение.

Создание объектов начинается с создания экземпляров и передачи конструкторам этих экземпляров нужной информации.

Здесь мы передаем описание дверей конструкторам OutsideWithDoor.

Здесь заполняется массив Exits[] для каждого из экземпляров. Данная процедура возможна только после создания всех экземпляров!

Для объектов IHasExteriorDoor нужно указать положение дверей.

`Exits` является полем открытого строкового массива в классе `Location`. Это не очень хороший пример инкапсуляции! Посторонний объект может легко внести изменения в массив `Exits`. В следующей главе вы узнаете более удачный способ представления набора строк или других объектов.

```
    private void MoveToANewLocation(Location newLocation) {
        currentLocation = newLocation;

        exits.Items.Clear();
        for (int i = 0; i < currentLocation.Exits.Length; i++)
            exits.Items.Add(currentLocation.Exits[i].Name);
        exits.SelectedIndex = 0;

        description.Text = currentLocation.Description;

        if (currentLocation is IHasExteriorDoor)
            goThroughTheDoor.Visible = true;
        else
            goThroughTheDoor.Visible = false;
    }
    private void goHere_Click(object sender, EventArgs e) {
        MoveToANewLocation(currentLocation.Exits[exits.SelectedIndex]);
    }

    private void goThroughTheDoor_Click(object sender, EventArgs e) {
        IHasExteriorDoor hasDoor = currentLocation as IHasExteriorDoor;
        MoveToANewLocation(hasDoor.DoorLocation);
    }
}
```

Метод MoveToANewLocation() показывает в форме новое помещение.

Сначала нужно очистить раскрывающийся список, затем его можно будет заново заполнить названиями помещений. Присвоение переменной SelectedIndex значения ноль отображает первый пункт списка. Не забудьте присвоить свойству DropDownStyle значение DropDownList, чтобы пользователи не смогли добавлять в список свои значения.

Делает кнопку Go through the door невидимой, если текущее помещение не реализует IHasExteriorDoor.

Щелчок на кнопке Go here: перемещает в выбранное помещение.

Оператор **as** осуществляет нисходящее приведение currentLocation к IHasExteriorDoor, что дает нам доступ к полю DoorLocation.

Работа еще не закончена!

Нашу замечательную модель дома можно превратить в игру! Хотите поиграть с компьютером в прятки? Нам потребуется класс Opponent (Соперник) и возможность прятать его в комнатах. Ну и конечно, большой дом, в котором удобно прятаться! Мы добавим новый интерфейс, описывающий укромные местечки. И обновим форму, чтобы получить возможность проверять наличие укромных мест и отслеживать, сколько ходов вы сделали, пытаясь найти соперника!

Итак, начнем!

*Настолько сложного задания у вас еще не было. Внимательно прочитайте инструкции! **Можете** подглядывать в решение.*

Упражнение

Время сыграть в прятки! Создадим дополнительные комнаты, укромные местечки и соперника в игре.

Создайте новый проект и воспользуйтесь командой Add Existing Item, чтобы добавить классы из первой части.

На этот раз мы не приводим диаграмму классов, поэтому следует взять лист бумаги и нарисовать ее самостоятельно. Это поможет вам понять программу, которую вы собираетесь создать.

1 Интерфейс IHidingPlace

Вам не потребуется ничего особенного. Любой класс, производный от `Location` и реализующий `IHidingPlace`, имеет место, где может спрятаться соперник. Потребуется только строка, хранящая информацию о том, где он прячется («в шкафу», «под кроватью»...)

- ★ Добавьте только метод чтения, так как при наличии в комнате укромного места вам уже не потребуется ничего менять.

2 Классы, реализующие IHidingPlace

Потребуются два класса: `OutsideWithHidingPlace` (наследующий от класса `Outside`) и `RoomWithHidingPlace` (наследующий от класса `Room`). Укромные места должны быть в любой комнате с наружной дверью, поэтому наследование будет осуществляться от класса `RoomWithHidingPlace`, а не от класса `Room`.

Итак, укромное место есть в каждой комнате с наружной дверью: в кухне стоит шкаф, а в гостиной есть чулан.

3 Класс, описывающий соперника

Объект Opponent будет прятаться, а вы его искать.

- ★ Ему потребуется закрытое поле `Location` (`myLocation`) для отслеживания его положения и закрытое поле `Random` (`random`) для поиска случайного укромного места.
- ★ Конструктор присваивает начальное положение переменной `myLocation`, а переменную `random` — новому экземпляру `Random`. Игра начинается на переднем дворе, а затем случайным образом выбирается место для укрытия. Соперник делает 10 ходов. Оказываясь перед внешней дверью, он подбрасывает монету, чтобы определить, нужно ли через нее проходить.
- ★ Метод `Move()` перемещает соперника. Если `random.Next(2)` имеет значение 1, соперник проходит в дверь. Затем он случайным образом выбирает один из выходов из нового помещения и проходит сквозь него. Если в помещении негде спрятаться, соперник снова случайным образом выбирает дверь и уходит в другое место.
- ★ Метод `Check()` с параметром location возвращает значение true, когда соперник прячется.

4 Дополнительные комнаты

Обновите метод `CreateObjects()`, чтобы создать больше комнат:

- ★ Добавьте лестницу с деревянными перилами, соединяющую гостиную с **коридором второго этажа**, где висит картина с собакой и стоит шкаф.
- ★ Верхний коридор ведет в три комнаты: **главную спальню** с большой кроватью, **вторую спальню** с маленькой кроватью и **ванную** с раковиной и туалетом. Прятаться можно как под кроватями, так и в душе.
- ★ Передний и задний дворы соединены **проездом** с гаражом, пригодным для укрытия. Прятаться можно и в сарае.

Обновляем форму

Создадим несколько новых кнопок. Они будут появляться и исчезать, в зависимости от того, на какой стадии находится игра.

Верхние две кнопки и раскрывающийся список будут видимы только в игре.

В начале игры отображается только кнопка Hide (Прячься!). После щелчка на ней в текстовом поле идет отсчет до 10 для соперника и десять раз вызывается метод Move(). Затем кнопка становится невидимой.

Средняя кнопка называется check (проверка) Для нее не нужно задавать свойство Text.

Эта кнопка проверяет укромные местечки в каждой комнате. Она видна, только когда вы находитесь в помещении, где можно спрятаться. При этом ее свойство Text изменяется. Ему присваивается слово Check (Смотрим), за которым следует название места. Например, если вы находитесь в спальне, на кнопке появится надпись Check under the bed (Смотрим под кроватью).

Заставим кнопки работать

Появились две новые кнопки.

В главе 2 вы уже встречали методы DoEvents() и Sleep(), которые будут использоваться в данном случае.

★ Центральная кнопка становится видимой, только когда вы находитесь в комнате с местом для укрытия. Она ищет соперника при помощи метода `Check()`. Если вы его находите, игра перезапускается.

★ Нижняя кнопка запускает игру. В текстовом поле с задержкой 200 миллисекунд появляются цифры от 1 до 10. После каждой из них соперник перемещается при помощи метода `Move()`. Затем на полсекунды появляется надпись «Я иду искать!», и игра начинается.

7 Метод, перерисовывающий форму, и метод, перезапускающий игру

Метод `RedrawForm()` отвечает за появление нужного текста в текстовом поле, видимость кнопок и ярлык на средней кнопке. Метод `ResetGame()` запускается после обнаружения соперника. Он возвращает соперника на передний двор и позволяет начать игру заново после щелчка на кнопке «Hide!» Результатом его работы является пустая форма с текстовым полем и кнопкой «Hide!» В текстовом поле должна быть информация, где и за сколько ходов вы обнаружили соперника.

8 Отслеживание числа ходов

В текстовом поле должно отображаться количество проверенных укромных мест и переходов из одного помещения в другое. После обнаружения соперника должно появляться сообщение «Ты нашел меня за X ходов!»

9 Вид формы в начале работы программы

Изначально форма должна содержать пустое текстовое поле и кнопку «Hide!», щелчок на которой начинает игру!

Упражнение Решение

Вот как выглядит программа после создания новых комнат, укромных мест и соперника, с которым вы будете играть в прятки.

Новый интерфейс IHidingPlace содержит всего одно поле типа string с методом чтения, возвращающим название места, где можно спрятаться.

```
interface IHidingPlace {
    string HidingPlaceName { get; }
}

class RoomWithHidingPlace : Room, IHidingPlace {
    public RoomWithHidingPlace(string name, string decoration, string hidingPlaceName)
        : base(name, decoration)
    {
        HidingPlaceName = hidingPlaceName;
    }
public string HidingPlaceName { get; private set; }
    }
    public override string Description {
        get {
            return base.Description + " Спрятаться можно " + hidingPlaceName + ".";
        }
    }
}
```

Класс RoomWithHidingPlace наследует от класса Room и реализует интерфейс IHidingPlace, добавляя свойство HidingPlaceName. Данное вспомогательное поле задается конструктором.

```
class RoomWithDoor : RoomWithHidingPlace, IHasExteriorDoor {
    public RoomWithDoor(string name, string decoration,
                        string hidingPlaceName, string doorDescription)
        : base(name, decoration, hidingPlaceName)
    {
        DoorDescription = doorDescription;
    }

    public string DoorDescription { get; private set; }

    public Location DoorLocation { get; set; }
}
```

Так как укрытия было решено поместить в комнаты с внешними дверями, мы сделали класс RoomWithDoor производным от RoomWithHidingPlace. Теперь конструктор этого производного класса передает название укромного места конструктору RoomWithHidingPlace.

Вам потребуется класс OutsideWithDoor, такой же как в программе «знакомство с домом».

```
class OutsideWithHidingPlace : Outside, IHidingPlace {
    public OutsideWithHidingPlace(string name, bool hot, string hidingPlaceName)
        : base(name, hot)
    {
       HidingPlaceName = hidingPlaceName;
    }

    public string HidingPlaceName { get; private set; }

    public override string Description {
        get {
            return base.Description + " Можно спрятаться " + HidingPlaceName + ".";
        }
    }
}
```

Класс OutsideWithHidingPlace наследует от класса Outside и реализует метод IHidingPlace аналогично классу RoomWithHidingPlace.

```
class Opponent {
    private Random random;
    private Location myLocation;
    public Opponent(Location startingLocation) {
        myLocation = startingLocation;
        random = new Random();
    }
    public void Move() {
       bool hidden = false;
        while (!hidden) {
            if (myLocation is IHasExteriorDoor) {
                IHasExteriorDoor locationWithDoor =
                                myLocation as IHasExteriorDoor;
                if (random.Next(2) == 1)
                    myLocation = locationWithDoor.DoorLocation;
            }
            int rand = random.Next(myLocation.Exits.Length);
            myLocation = myLocation.Exits[rand];
            if (myLocation is IHidingPlace)
                hidden = true;
        }
    }
    public bool Check(Location locationToCheck) {
        if (locationToCheck != myLocation)
            return false;
        else
            return true;
    }
}
```

Конструктор класса Opponent в качестве параметра берет начальное помещение. Он создает экземпляр Random, при помощи которого случайным образом осуществляется перемещение из одного места в другое.

Метод Move() при помощи оператора is проверяет наличие внешней двери в комнате. При ее наличии с 50%-й вероятностью он через нее проходит. Таким способом он случайным образом перемещается по дому, пока не находит место, где можно спрятаться.

Этот цикл while продолжается, пока переменная hidden не примет значение true. Это произойдет при попадании в помещение, пригодное для укрытия.

Метод Check() сравнивает значение переменной location класса opponent со значением ссылки Location. Если обе ссылки указывают на один объект, значит, соперник найден!

Переверните страницу и продолжим!

Упражнение Решение

Это код формы. Неизменными в нем остались только методы `goHere_Click()` и `goThroughTheDoor_Click()`.

Список полей класса Form1. Именно с их помощью отслеживаются помещение, соперник и количество перемещений, сделанных игроком.

```
int Moves;

Location currentLocation;

RoomWithDoor livingRoom;
RoomWithHidingPlace diningRoom;
RoomWithDoor kitchen;
Room stairs;
RoomWithHidingPlace hallway;
RoomWithHidingPlace bathroom;
RoomWithHidingPlace masterBedroom;
RoomWithHidingPlace secondBedroom;

OutsideWithDoor frontYard;
OutsideWithDoor backYard;
OutsideWithHidingPlace garden;
OutsideWithHidingPlace driveway;

Opponent opponent;
```

Конструктор Form1 создает объекты, определяет положение соперника и перезагружает игру. Добавленный к методу ResetGame() булев параметр отвечает за появление сообщения только после вашего выигрыша.

```
public Form1() {
    InitializeComponent();
    CreateObjects();
    opponent = new Opponent(frontYard);
    ResetGame(false);
}
private void MoveToANewLocation(Location newLocation) {
    Moves++;
    currentLocation = newLocation;
    RedrawForm();
}

private void RedrawForm() {
    exits.Items.Clear();
    for (int i = 0; i < currentLocation.Exits.Length; i++)
        exits.Items.Add(currentLocation.Exits[i].Name);
    exits.SelectedIndex = 0;
    description.Text = currentLocation.Description + "\r\n(перемещение #" + Moves +
")";
    if (currentLocation is IHidingPlace) {
        IHidingPlace hidingPlace = currentLocation as IHidingPlace;
        check.Text = "Check " + hidingPlace.HidingPlaceName;
        check.Visible = true;
    }
    else
        check.Visible = false;
    if (currentLocation is IHasExteriorDoor)
        goThroughTheDoor.Visible = true;
    else
        goThroughTheDoor.Visible = false;
}
```

Метод MoveToANewLocation() задает новое положение и перерисовывает форму.

Нам нужно название укромного места, но в наличии имеется только объект CurrentLocation, не обладающий свойством HidingPlaceName. Воспользуемся оператором as, чтобы скопировать ссылку на переменную IHidingPlace.

Метод RedrawForm() формирует раскрывающийся список, задает текст (добавляя номер перемещения), а также управляет видимостью кнопок в зависимости от наличия наружной двери или укромного места.

Добавив всего пару строк, вы пристроите к дому целое крыло! Видите, как полезна инкапсуляция классов и объектов?

```
private void CreateObjects() {
    livingRoom = new RoomWithDoor("Гостиная", "старинный ковер",
              "в гардеробе", "дубовая дверь с латунной ручкой");
    diningRoom = new RoomWithHidingPlace("Столовая", "хрустальная люстра",
               "в высоком шкафу");
    kitchen = new RoomWithDoor("Кухня", "приборы из нержавеющей стали",
              "в сундуке", "сетчатая дверь");
    stairs = new Room("Лестница", "деревянные перила");
    hallway = new RoomWithHidingPlace("Верхний коридор", "картина с собакой",
              "в гардеробе");
    bathroom = new RoomWithHidingPlace("Ванная", "раковина и туалет",
              "в душе");
    masterBedroom = new RoomWithHidingPlace("Главная спальня", "большая кровать",
              "под кроватью");
    secondBedroom = new RoomWithHidingPlace("Вторая спальня", "маленькая кровать",
              "под кроватью");

    frontYard = new OutsideWithDoor("лужайка", false, "тяжелая дубовая дверь");
    backYard = new OutsideWithDoor("Задний двор", true, "сетчатая дверь");
    garden = new OutsideWithHidingPlace("Сад", false, "в сарае");
    driveway = new OutsideWithHidingPlace("Подъезд", true, "в гараже");

    diningRoom.Exits = new Location[] { livingRoom, kitchen };
    livingRoom.Exits = new Location[] { diningRoom, stairs };
    kitchen.Exits = new Location[] { diningRoom };
    stairs.Exits = new Location[] { livingRoom, hallway };
    hallway.Exits = new Location[] { stairs, bathroom, masterBedroom, secondBedroom };
    bathroom.Exits = new Location[] { hallway };
    masterBedroom.Exits = new Location[] { hallway };
    secondBedroom.Exits = new Location[] { hallway };
    frontYard.Exits = new Location[] { backYard, garden, driveway };
    backYard.Exits = new Location[] { frontYard, garden, driveway };
    garden.Exits = new Location[] { backYard, frontYard };
    driveway.Exits = new Location[] { backYard, frontYard };

    livingRoom.DoorLocation = frontYard;
    frontYard.DoorLocation = livingRoom;

    kitchen.DoorLocation = backYard;
    backYard.DoorLocation = kitchen;
}
```

Новый метод CreateObjects() создает объекты, из которых состоит дом. От старого метода он отличается большим количеством вариантов.

Переверните страницу и продолжим!

Упражнение Решение

Это остальной код формы. Обработчики событий кнопок goHere и goThroughTheDoor идентичны своим собратьям из первой части упражнения. Вы найдете их, вернувшись на несколько страниц назад.

```
private void ResetGame(bool displayMessage) {
    if (displayMessage) {
        MessageBox.Show("Меня нашли за " + Moves + " ходов!");
        IHidingPlace foundLocation = currentLocation as IHidingPlace;
        description.Text = "Соперник найден за " + Moves
             + " ходов! Он прятался " + foundLocation.HidingPlaceName + ".";
    }
    Moves = 0;
    hide.Visible = true;
    goHere.Visible = false;
    check.Visible = false;
    goThroughTheDoor.Visible = false;
    exits.Visible = false;
}

private void check_Click(object sender, EventArgs e) {
    Moves++;
    if (opponent.Check(currentLocation))
        ResetGame(true);
    else
        RedrawForm();
}

private void hide_Click(object sender, EventArgs e) {
    hide.Visible = false;

    for (int i = 1; i <= 10; i++) {
        opponent.Move();
        description.Text = i + "... ";
        Application.DoEvents();
        System.Threading.Thread.Sleep(200);
    }

    description.Text = "Я иду искать!";
    Application.DoEvents();
    System.Threading.Thread.Sleep(500);

    goHere.Visible = true;
    exits.Visible = true;
    MoveToANewLocation(livingRoom);
}
```

Метод ResetGame() перезагружает игру. Он отображает финальное сообщение, а затем делает все кнопки, кроме кнопки «Hide!», невидимыми.

Нам нужно отобразить имя укромного места, но ссылка CurrentLocation принадлежит классу Location и не имеет доступа к полю HidingPlaceName. К счастью, можно воспользоваться оператором as для нисходящего приведения к ссылке IHidingPlace, указывающей на нужный нам объект.

Кнопка check проверяет, не прячется ли соперник в комнате, где вы находитесь. При его обнаружении перезагружает игру. Не обнаружив его, перерисовывает форму (чтобы обновить число ходов).

Помните обработчик событий DoEvents() из главы 2? Именно он отвечает за обновление информации в текстовом поле.

Игра начинается с кнопки «Hide!» Сначала кнопка становится невидимой. Затем включается счет до 10 и сопернику говорится, что нужно спрятаться. Напоследок видимыми становятся первая кнопка и раскрывающийся список, и игрок помещается в гостиную. Метод MoveToANewLocation() вызывает метод RedrawForm().

Решение ребуса в бассейне со с. 349

со с. 349

Вам требовалось взять фрагменты кода из бассейна и поместить их на пустые строки таким образом, чтобы получить показанный ниже результат.

```
interface Nose {
    int Ear() ;
    string Face { get; }
}

abstract class Picasso : Nose {
    public virtual int Ear()
    {
        return 7;
    }
    public Picasso(string face)
    {
        this.face = face;
    }
    public virtual string Face {
        get { return face ; }
    }
    string face;
}

class Clowns : Picasso {
    public Clowns() : base("Clowns") { }
}
```

Свойства могут появиться в произвольном месте класса! Код проще читать, если они сосредоточены сверху, но в данном случае мы поместили свойство face в нижнюю часть класса Picasso.

Класс Acts вызывает конструктор базового для него класса Picasso. Он передает конструктору переменную Acts, которая сохраняется в свойстве face.

```
class Acts : Picasso {
    public Acts() : base("Acts") { }
    public override int Ear() {
        return 5;
    }
}

class Of76 : Clowns {
    public override string Face {
      get { return "Of76"; }
    }
    public static void Main(string[] args) {
        string result = "";
        Nose[] i = new Nose[3];
        i[0] = new Acts();
        i[1] = new Clowns();
        i[2] = new Of76();
        for (int x = 0; x < 3; x++) {
            result += ( i[x].Ear() + " "
                     + i[x].Face ) + "\n";
        }
        Console.WriteLine(result);
        Console.ReadKey();
    }
}
```

Face — это метод записи, возвращающий значения свойства face. И метод, и свойство определены в классе Picasso и наследуются производными классами.

```
file:///C:/Users/Pul
5 Acts
7 Clowns
7 Of76
```

Вывод

Большие объемы данных

Пришла беда — отворяй ворота.

В реальном мире данные, как правило, не хранятся маленькими кусочками. Данные поступают **вагонами, штабелями и кучами**. Для их систематизации нужны мощные инструменты, и тут вам на помощь приходят **коллекции**. Они позволяют **хранить, сортировать и редактировать** данные, которые обрабатывает программа. В результате вы можете сосредоточиться на основной идее программирования, оставив задачу отслеживания данных коллекциям.

Категории данных не всегда можно сохранять в переменных типа string

Предположим, у вас есть несколько рабочих пчел из класса `Worker`. Как написать конструктор, который в качестве параметра берет работу? Если названия работ помещать в переменные типа `string`, получится примерно такой код:

Наше приложение для управления пчелами отслеживало работу каждой из них при помощи строк вида Sting Patrol или Nectar Collector.

Код позволяет передать эти значения в конструктор, даже если программа поддерживает только такие занятия, как Sting Patrol (Охранник), Nectar Collector (Сборщик нектара) и другие, привычные для пчел варианты работы.

```
Worker buzz = new Worker("Прокурор");
Worker clover = new Worker("Кинолог");
Worker gladys = new Worker("Диктор");
```

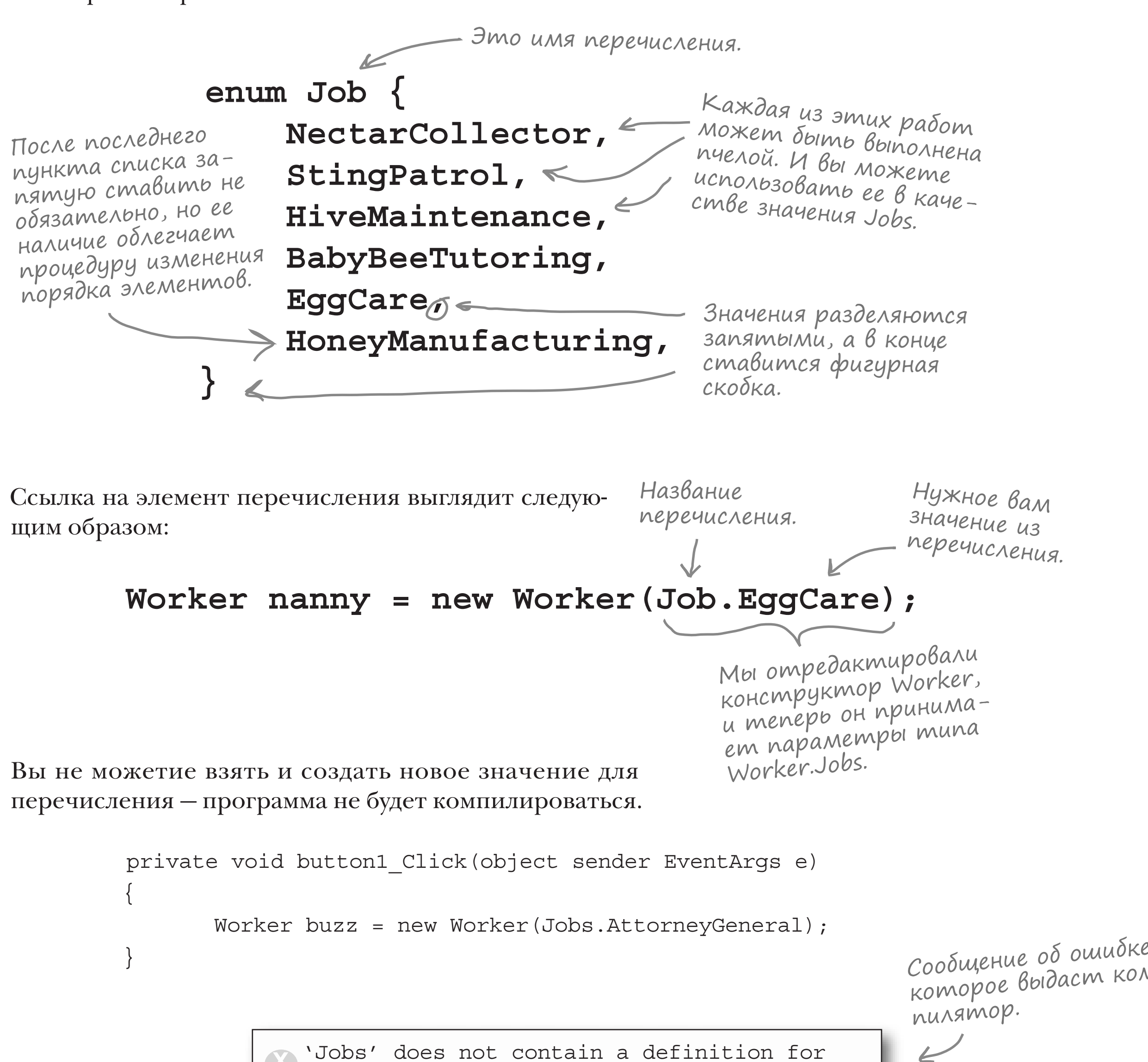
Этот код без проблем компилируется. Но с точки зрения пчел эти профессии не имеют никакого смысла. Поэтому хорошо бы сделать подобные данные недействительными для класса Worker.

К конструктору `Worker` можно добавить код, проверяющий корректность каждой строчки. Но тогда, чтобы расширить список доступных для пчел занятий, вам придется отредактировать данный код и перекомпилировать класс `Worker`. Так что это недальновидное решение. А что делать при наличии других классов, проверяющих типы работ, которые могут выполнять рабочие пчелы? Вы получите дублирующийся код, затрудняющий вашу работу.

Фактически нам нужно объявить: «Здесь могут использоваться только строго определенные значения». И **перечислить** эти значения.

Перечисления

Такой тип данных, как **перечисление (enum),** позволяет переменным принимать только конечное множество возможных значений. Поэтому вы можете определить перечисление Jobs, указав доступные варианты работ:




```
enum Job {
    NectarCollector,
    StingPatrol,
    HiveMaintenance,
    BabyBeeTutoring,
    EggCare,
    HoneyManufacturing,
}
```

Ссылка на элемент перечисления выглядит следующим образом:

```
Worker nanny = new Worker(Job.EggCare);
```

Вы не можетие взять и создать новое значение для перечисления — программа не будет компилироваться.

```
private void button1_Click(object sender EventArgs e)
{
       Worker buzz = new Worker(Jobs.AttorneyGeneral);
}
```

```
'Jobs' does not contain a definition for
'AttorneyGeneral'
```

Присвоим числам имена

Иногда удобнее работать с именованными числами — сопоставить их элементам перечисления и обращаться к ним по именам. Это позволит избежать непонятных цифр, разбросанных по всему коду. Вот перечисление для оценки качества выполнения собаками различных команд:

```
enum TrickScore {
    Sit = 7,
    Beg = 25,
    RollOver = 50,
    Fetch = 10,
    ComeHere = 5,
    Speak = 30,
}
```

Порядок следования не имеет значения. Кроме того, одно и то же число можно сопоставить разным именам.

Укажите имя, затем знак "=", а затем число, которому это имя соответствует.

По умолчанию базовым для элементов перечисления является тип int.

Можно объявить и перечисление другого типа, например byte или long, как это сделано в нижней части страницы.

Такая запись заставляет компилятор использовать число, которому соответствует элемент перечисления. В данном случае 10.

Вот фрагмент метода, использующего перечисление `TrickScore` путем приведения к значению типа `int` и обратно:

Этот оператор присваивает переменной value значение 30.

```
int value = (int)TrickScore.Fetch * 3;
MessageBox.Show(value.ToString());
TrickScore score = (TrickScore)value;
MessageBox.Show(score.ToString());
```

Можно выполнить обратное приведение int к TrickScore. Так как значение равно 30, результатом станет TrickScore.Speak. Соответственно метод score.ToString() возвращает "Speak".

Элементы перечисления можно использовать как числа и производить с ними разнообразные вычисления, а можно воспользоваться методом `ToString()` и рассматривать их как строки. При отсутствии явного присвоения элементы перечисления получают значения по умолчанию. Первому элементу присваивается 0, второму — 1 и т. д.

А что делать, если нужно присвоить элементу перечисления очень большое значение? По умолчанию все элементы принадлежат к типу `int`, поэтому нужно поменять тип перечисления при помощи оператора `:`

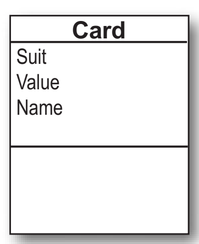
```
enum TrickScore : long {
        Sit = 7,
        Beg = 2500000000025
}
```

Этот оператор заставляет компилятор относить элементы перечисления TrickScore к типу long.

Если принадлежность элементов перечисления к типу long не указана явно, при компиляции появится сообщение: `Cannot implicitly convert type 'long' to 'int'.`

Упражнение

Воспользуемся полученными сведениями для построения класса, описывающего игральные карты



1 Создайте новый проект и добавьте класс Card

Вам нужны два открытых поля: `Suit` (`Spades`, `Clubs`, `Diamonds`, `Hearts`) и `Value` (`Ace`, `Two`, `Three...Ten`, `Jack`, `Queen`, `King`), а также предназначенные только для чтения поля `Name` (`Ace of Spades` (туз пик), `Five of Diamonds` (пятерка бубен)).

2 Задайте масти и карты с помощью двух перечислений

Воспользуйтесь уже знакомой вам командой Add >> Class, заменяя слово `class` на `enum`. Убедитесь, что `(int)Suits.Spades` = 0, `Clubs` = 1, `Diamonds` = 2, а `Hearts` = 3. Во втором перечислении: `(int)Values.Ace` должно быть = 1, `Two` = 2, `Three` = 3 и т. д. `Jack` (валет) = 11, `Queen` (дама) = 12, а `King` (король) = 13.

3 Добавим картам свойство

Свойство `Name` доступно только для чтения. Метод чтения должен возвращать строку с названием карты. Этот код вызывает свойство `Name` и отображает результат:

```
Card card = new Card(Suits.Spades, Values.Ace);
string cardName = card.Name;
```

Чтобы это заработало, в класс Card нужно поместить конструктор с двумя параметрами.

Переменная `cardName` должна иметь значение `Ace of Spades`.

4 Кнопка, отображающая название случайной карты

Будем выбирать карту, случайным образом присваивая число от 0 до 3 переменной `Suits`, а числа от 1 до 13 — переменной `Values`. Воспользуемся встроенным классом `Random`, позволяющим вызвать метод `Next()` тремя способами:

*Возможность вызывать один метод разными способами называется **перегрузкой (overloading)**. О ней мы поговорим позже...*

```
Random random = new Random();
int numberBetween0and3 = random.Next(4);
int numberBetween1and13 = random.Next(1, 14);
int anyRandomInteger = random.Next();
```

Здесь случайным образом выбирается число от 1 до 14.

Часто задаваемые вопросы

В: При наборе метода `Random.Next()` появилось окно IntelliSense с надписью «3 of 3». Что это значит?

О: Вы увидели метод, который был **перегружен**. Возможность вызывать один метод несколькими способами называется перегрузкой. При работе с классом, в составе которого присутствуют подобные методы, IDE показывает вам все возможные варианты. В рассматриваемом случае класс `Random` имеет три метода `Next()`. Когда вы печатаете random.Next(), появляется окно IntelliSense с параметрами всех вариантов. Слева и справа от записи «3 of 3» находятся стрелки, которые позволяют выбирать варианты. Это полезно в случаях, когда метод имеет множество определений. Вам же важно выбрать правильный вариант метода `Next()`. Более подробно об этом мы поговорим чуть позже.

```
random.Next(
▲ 3 of 3 ▼ int Random.Next(int minValue, int maxValue)
Returns a random number within a specified range.
minValue: The inclusive lower bound of the random number returned.
```

Упражнение Решение

Колода карт позволяет проиллюстрировать важность ограничения значений. Было бы странно обнаружить джокера треф или 13 червей. Вот как выглядит код класса `Card`.

```
enum Suits {
    Spades,
    Clubs,
    Diamonds,
    Hearts
}
```

По умолчанию первый элемент списка приравнивается к нулю, второй — к единице, третий — к двойке и т. д.

```
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
}
```

В данном случае значения по умолчанию переопределяются и Values.Ace равно 1.

Счет начинается с тузов.

Для перечислений мы выбрали имена `Suits` (Масти) и Values (Достоинство), в то время как свойства класса `Card`, использующие эти перечисления в качестве типов, называются `Suit` и `Value`. Что вы думаете по поводу этих имен? Посмотрите на примеры перечислений, встречающиеся в книге. Может быть, следовало назвать их `Suit` и `Value`?

Класс Card имеет свойство Suit типа Suits и свойство Value типа Values.

```
class Card {
    public Suits Suit { get; set; }
    public Values Value { get; set; }

    public Card(Suits suit, Values value) {
        this.Suit = suit;
        this.Value = value;
    }

    public string Name {
        get { return Value.ToString() + " of " + Suit.ToString(); }
    }
}
```

Метод чтения свойства Name использует строку, полученную из имени при помощи метода ToString().

Код кнопки, щелчок на которой вызывает окно с названием случайной карты.

Перегруженный метод Random.Next() генерирует случайное число, которое мы присваиваем перечислению.

```
Random random = new Random();
private void button1_Click(object sender, EventArgs e) {
    Card card = new Card((Suits)random.Next(4), (Values)random.Next(1, 14));
    MessageBox.Show(card.Name);
}
```

Создать колоду карт можно было при помощи массива...

Как создать класс, представляющий собой колоду карт? Нужно отслеживать каждую карту в колоде и их порядок. Данная задача решается при помощи массива Card: первой карте присваивается значение 0, следующей — 1 и т. д. Отправной точкой сделаем класс Deck, описывающий полную колоду (52 карты).

```
class Deck {
    private Card[] cards =  {
        new Card(Suits.Spades, Values.Ace),
        new Card(Suits.Spades, Values.Two),
        new Card(Suits.Spades, Values.Three),
        // ...
        new Card(Suits.Diamonds, Values.Queen),
        new Card(Suits.Diamonds, Values.King),
    };

    public void PrintCards() {
        for (int i = 0; i < cards.Length; i++)
            Console.WriteLine(cards[i].Name());
    }
}
```

Объявление массива сокращено для экономии места на странице. Но здесь перечислены все 52 карты в колоде.

... но что, если вам захочется большего?

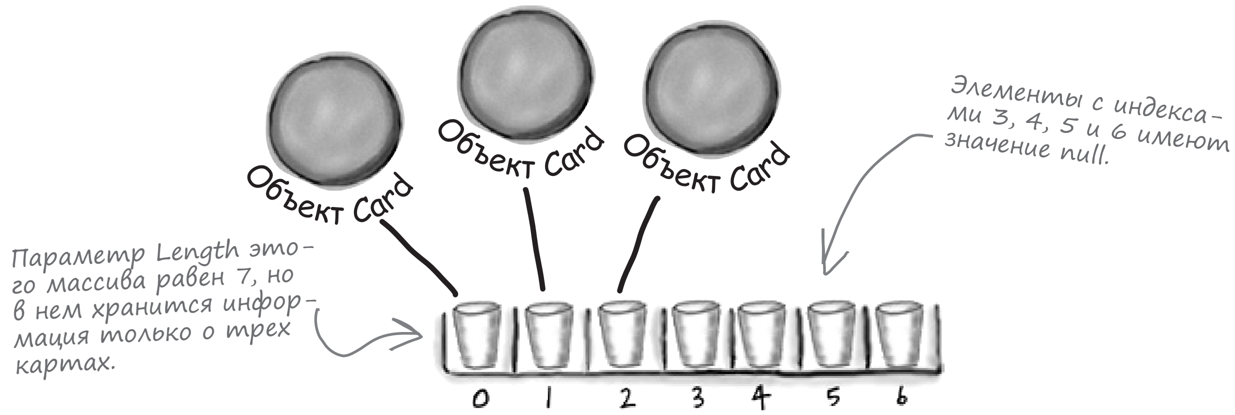
Представьте все, что можно делать с карточной колодой. Например, для моделирования игры нужно менять порядок карт, а также добавлять их в колоду и убирать из нее. Эти задачи уже не решить при помощи массива.

Как бы вы добавили метод Shuffle(), моделирующий процесс тасования колоды? Как смоделировать процесс сдачи карт? Каким образом добавить карты в колоду?

Проблемы работы с массивами

Массивы прекрасно подходят для хранения фиксированного списка значений или ссылок. Но как только возникает необходимость поменять элементы местами или добавить новые значения, ситуация осложняется.

1. Каждый массив имеет длину, которую вы должны знать. Для получения пустых элементов можно воспользоваться ссылкой на неопределенное значение null:



2. Чтобы отследить количество имеющихся карт, создадим поле `topCard` типа `int` для хранения информации об индексе последней карты. Параметр `Length` нашего массива имеет значение 7, в то время как `topCard` равно 3.

3. Можно добавить метод `Peek()`, возвращающий ссылку на верхнюю карту, сымитировав взятие верхней карты из колоды. А как добавить карту? Если параметр `topCard` меньше длины массива, карта добавляется в массив, а значение `topCard` увеличивается на 1. В противном случае нужно создать новый, большний массив и скопировать туда имеющиеся карты. Удаляя карту, вы вычитаете 1 из `topCard` и возвращаете элементу массива значение null. А как удалить карту **из середины**? После удаления карты 4 нужно сместить вниз карту 5, затем карту 6 и т. д.

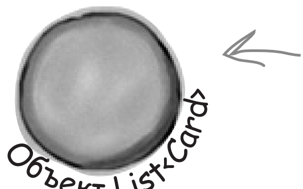
Коллекции

В .NET Framework существует множество классов **коллекций (collection)**, позволяющих легко решить вопросы с добавлением и удалением элементов массива. Чаще всего используется коллекция `List<T>`. `List<T>` позволяет легко добавить, удалить, выбрать элемент и даже поменять порядок следования элементов.

Начнем с создания нового экземпляра List<T>

Коллекции, как и массивы, принадлежат к определенному типу. Тип объектов или значений, которые должны там храниться, указывается в момент создания коллекции в угловых скобках <> с помощью оператора `new`.

```
List<Card> cards = new List<Card>();
```



Тип <Card> был указан в момент создания коллекции, так что теперь там хранятся только ссылки на объекты типа Card.

Иногда для экономии места и упрощения код <T> будет опускаться.

РАССЛАБЬТЕСЬ

Символ <T> указывает на *обобщенную* коллекцию.

Символ T заменяется типом, `List<int>` означает `List` значений типа `int`.

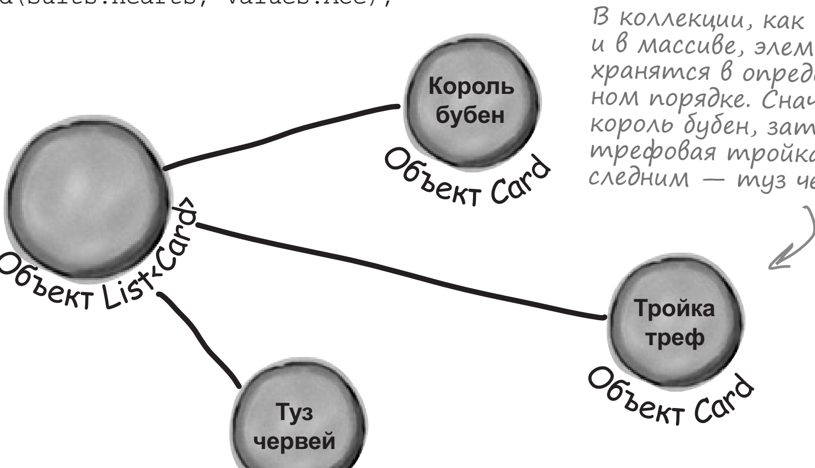
Добавление в коллекцию List<T>

В коллекцию `List<T>` можно добавить произвольное количество элементов (главное, чтобы они были ***полиморфны*** с типом, указанным при создании `List<T>`).

```
cards.Add(new Card(Suits.Diamonds, Values.King);
cards.Add(new Card(Suits.Clubs, Values.Three);
cards.Add(new Card(Suits.Hearts, Values.Ace);
```

Они могут быть назначены: интерфейсам, абстрактным классам, базовым классам и т. п.

В коллекции, как и в массиве, элементы хранятся в определенном порядке. Сначала король бубен, затем трефовая тройка и последним — туз червей.



В коллекцию List можно добавить произвольное число карт, достаточно воспользоваться методом Add(). Если количество объектов превышает количество свободных мест, размер коллекции увеличивается.

Коллекции List

Класс `List` встроен в .NET Framework и позволяет делать вещи, о которых вы не могли даже мечтать, имея в арсенале только старые добрые массивы. Перечислим новые возможности:

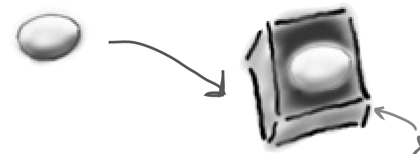
1 Можно создать коллекцию

```
List<Egg> myCarton = new List<Egg>();
```

Под созданный объект List в куче выделяется память. Но в объекте пока ничего нет.

2 Добавить в нее объект

```
Egg x = new Egg();
myCarton.Add(x);
```

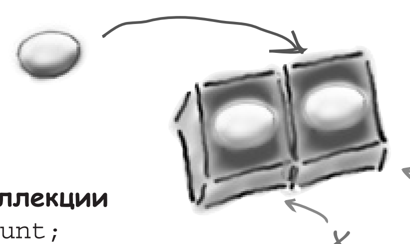


Теперь в коллекции имеется объект Egg...

x

3 Добавить еще один объект

```
Egg y = new Egg();

myCarton.Add(y);
```



...она увеличивается, чтобы принять второй объект Egg.

x y

4 Узнать количество объектов в коллекции

```
int theSize = myCarton.Count;
```

5 Узнать, содержатся ли в коллекции объекты определенного типа

```
bool Isin = myCarton.Contains(x);
```

Теперь коллекцию можно проверять на наличие объектов Egg. В данном случае оператор вернет значение true.

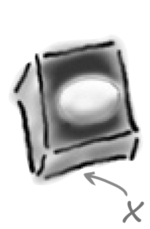
6 Определить их положение

```
int idx = myCarton.IndexOf(y);
```

Элемент x имеет индекс 0, в то время как элемент y — индекс 1.

7 Удалить оттуда элемент

```
myCarton.Remove(y);
```



Бах!

x

После удаления элемента y в коллекции остается только элемент x (размер коллекции уменьшается).

Заполните таблицу, сверяясь с расположенным слева кодом. Вам нужно показать, как выглядел бы код, если бы вместо коллекции использовался массив. Мы не ожидаем от вас 100% правильных результатов, просто попытайтесь догадаться.

Предположим, что эти операторы выполняются по порядку.

Мы начали выполнять упражнение...

Коллекция	Обычный массив
`List<String> myList = new List <String>();`	*String [] myList = new String[2];*
`String a = "Yay!";`	*String a = "Yay!";*
`myList.Add(a);`	
`String b = "Bummer";`	*String b = "Bummer";*
`myList.Add(b);`	
`int theSize = myList.Count;`	
`Guy o = guys[1];`	
`bool isIn = myList.Contains(b);`	

Подсказка: потребуется более одной строки кода.

Итак, вот как выглядит правильный вариант кода с использованием вместо коллекций обычных массивов.

Коллекция	Обычный массив
`List<String> myList = new List <String>();`	`String[] myList = new String[2];`
`String a = "Yay!"`	`String a = "Yay!";`
`myList.Add(a);`	*myList[0] = a;*
`String b = "Bummer";`	`String b = "Bummer";`
`myList.Add(b);`	*myList[1] = b;*
`int theSize = myList.Count;`	*int theSize = myList.Length;*
`Guy o = guys[1];`	*Guy o = guys[1];*

`bool isIn = myList.Contains(b);`	*bool isIn = false;
	for (int i = 0; i < myList.Length; i++) {
	if (b == myList[i]) {
	isIn = true;
	}
	}*

Коллекции используют методы, как и уже знакомые вам классы. Чтобы увидеть список доступных методов, введите `.` после имени `List`. Параметры методам передаются так же, как это делалось для созданных вами классов.

Массивы довольно сильно вас ограничивают. В момент создания массива требуется указать его размер, а код для нужных процедур приходится писать вручную.

В .NET Framework существует класс Array, упрощающий некоторые операции, но коллекции все равно более просты в использовании, поэтому мы сконцентрируемся в основном на них.

Динамическое изменение размеров

Упражнение!

В момент создания коллекции вы не указываете информацию о ее длине. Она растет или сжимается в соответствии с количеством объектов внутри. Рассмотрим это на примере. **Создайте консольное приложение** и добавьте код в метод `Main()`. Мы будем **использовать отладчик** для пошагового просмотра кода и отслеживания происходящего.

Мы объявляем коллекцию объектов Shoe (Ботинок) с именем ShoeCloset (ОбувнойШкаф).

```
List<Shoe> shoeCloset = new List<Shoe>();

shoeCloset.Add(new Shoe()
     { Style = Style.Sneakers, Color = "Черный" });
shoeCloset.Add(new Shoe()
     { Style = Style.Clogs, Color = "Коричневый" });
shoeCloset.Add(new Shoe()
     { Style = Style.Wingtips, Color = "Черный" });
shoeCloset.Add(new Shoe()
     { Style = Style.Loafers, Color = "Белый" });
shoeCloset.Add(new Shoe()
     { Style = Style.Loafers, Color = "Красный" });
shoeCloset.Add(new Shoe()
     { Style = Style.Sneakers, Color = "Зеленый" });

int numberOfShoes = shoeCloset.Count;
foreach (Shoe shoe in shoeCloset) {
    shoe.Style = Style.Flipflops;
    shoe.Color = "Оранжевый";
}

shoeCloset.RemoveAt(4);

Shoe thirdShoe = shoeCloset[2];
Shoe secondShoe = shoeCloset[1];
shoeCloset.Clear();

shoeCloset.Add(thirdShoe);
if (shoeCloset.Contains(secondShoe))
    Console.WriteLine("Удивительно!");
```

Оператор **new** можно использовать внутри метода List.Add().

Возвращает число объектов Shoe в перечислении.

Цикл foreach просматривает всю обувь в шкафу.

Метод Remove() удаляет объект по ссылке, метод RemoveAt() — по индексу.

Метод Clear() удаляет из перечисления все объекты.

До удаления всех элементов перечисления мы сохранили две ссылки на обувь. Одна из них была возвращена, а вторая — нет.

Эта строчка не будет запускаться, потому что метод Contains() возвращает значение false. В пустое перечисление мы добавили только элемент thirdShoe, но не элемент secondShoe.

`foreach` — это специальный цикл, работающий с коллекциями. Он использует операторы для каждого элемента коллекции. В данном случае создается идентификатор с именем `shoe`, которому присваиваются элементы коллекции по очереди.

Цикл foreach работает и с массивами тоже.

Это класс Shoe и перечисление Style.

```
class Shoe {
    public Style Style;
    public string Color;
}

enum Style {
    Sneakers,
    Loafers,
    Sandals,
    Flipflops,
    Wingtips,
    Clogs,
}
```

Обобщенные коллекции

Вы уже видели, что коллекция может состоять как из строк, так и из ботинок. Можно поместить туда целые числа или произвольные созданные вами объекты. Именно поэтому класс List является **обобщенной коллекцией**. В момент его создания нужно указать тип значений, которые могут храниться внутри.

Эта запись не означает, что вы добавляете букву T. Она всего лишь показывает, что класс или интерфейс может работать со значениями любого типа, (достаточно указать этот тип в скобках). Запись List<Shoe> означает, что коллекция содержит только элементы класса Shoe.

```
List<T> name = new List<T>();
```

Класс List может быть или очень гибким (позволяющим любой тип), или очень ограниченным. Коллекции могут все то, что могут массивы (плюс еще кое-что).

В .NET Framework существуют обобщенные интерфейсы, позволяющие созданным вами коллекциям работать с любыми типами. Класс List реализует эти интерфейсы, именно поэтому работа с коллекцией целых чисел практически не отличается от работы с коллекцией объектов класса Shoe.

→ ***Убедитесь самостоятельно.*** Наберите **List** в IDE, щелкните правой кнопкой мыши и выберите команду Go To Definition. Вы увидите объявление класса List. Он реализует несколько интерфейсов:

Отсюда берутся методы RemoveAt(), IndexOf() и Insert().

```
class List<T> : IList<T>,
ICollection<T>, IEnumerable<T>, IList,
ICollection, IEnumerable
```

Отсюда берутся методы Add(), Clear(), CopyTo() и Remove(). (Это верно для всех обобщенных коллекций).

Этот интерфейс позволяет использовать цикл foreach.

КЛЮЧЕВЫЕ МОМЕНТЫ

- **List** — это класс .NET Framework.
- Класс List **динамически меняет свой размер**.
- Для добавления элементов в класс List используйте метод **Add().** Удалить элементы можно с помощью метода **Remove().**
- Метод **RemoveAt()** позволяет удалять объекты, начиная с заданного **индекса.**
- Тип значений, которые могут храниться в коллекции, указывается в угловых скобках. Запись List<Frog> означает, что в коллекции List могут храниться только объекты класса Frog.
- Метод **IndexOf()** определяет индекс для заданного объекта.
- Узнать количество элементов класса List можно при помощи свойства **Count**.
- Метод **Contains()** проверяет коллекцию на наличие объектов.
- **foreach** — это цикл, перебирающий элементы коллекции и выполняющий для каждого указанный код. Его синтаксис: **foreach(string s in StringList).** Вам не нужно заботиться об инкрементном увеличении на единицу, перебор элементов коллекции выполняется автоматически.

Будьте осторожны!

Нельзя редактировать коллекцию, когда она стоит в цикле foreach!

Попытка редактирования приведет к сообщению об ошибке. К счастью, можно сделать копию коллекции. IEnumerable обладает методом ToList(), который позволит создать копию и спокойно просматривать ее в цикле.

Магниты с кодом

Воспользуйтесь фрагментами кода для реконструкции формы с кнопкой, нажатие которой будет выводить показанное ниже окно с сообщением.

```
private void button1_Click(object
sender, EventArgs e){

}
```

```
a.RemoveAt(2);
```

```
List<string> a = new List<string>();
```

```
public void printL (List<string> a){
```

```
if (a.Contains("two")) {
    a.Add(twopointtwo);
}
```

```
a.Add(zilch);
a.Add(first);
a.Add(second);
a.Add(third);
```

```
}
```

```
string result = "";
```

```
if (a.Contains("three")){
    a.Add("four");
}
```

```
foreach (string element in a)
{
    result += "\n" + element;
}
```

```
MessageBox.Show(result);
```

```
}
```

```
if (a.IndexOf("four") != 4) {
    a.Add(fourth);
}
```

```
printL(a);
```

```
string zilch = "zero";
string first = "one";
string second = "two";
string third = "three";
string fourth = "4.2";
string twopointtwo = "2.2";
```

zero
one
three
four
4.2

OK

Решение упражнения с магнитами

Помните, в главе 3 говорилось об интуитивно понятных именах? Сейчас мы их сознательно избегаем, так как это слишком упрощает решение ребуса. Но в жизни такие имена, как printL(), использовать не стоит!

```
private void button1_Click(object sender, EventArgs e)
{
    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);
    }
    printL(a);
}

public void printL (List<string> a){
    string result = "";
    foreach (string element in a)
    {
        result += "\n" + element;
    }
    MessageBox.Show(result);
}
}
```

zero
one
three
four
4.2

OK

Вы понимаете, почему «2.2» никогда не попадет в коллекцию, несмотря на то что этот элемент был объявлен?

Метод RemoveAt() удаляет элемент, следующий за элементом #2. То есть третий элемент коллекции.

Метод printL() использует цикл foreach для просмотра коллекции строк, добавления каждого элемента в общую строку и отображения результата в окне диалога.

Цикл foreach по очереди отображает все элементы коллекции.

Часто задаваемые вопросы

В: Разве перечисления и класс `List` выполняют не одну и ту же задачу?

О: Основным и главным отличием перечислений является то, что они относятся к **типам**, в то время как коллекции `List` — это **объекты**.

Перечисление является удобным способом хранения списков констант, дающим возможность обращаться к ним по имени. Перечисления увеличивают читабельность кода и гарантируют использование корректных имен для доступа к нужным вам значениям.

В коллекцию List можно записать практически все. Так как речь идет о списке объектов, каждый элемент может иметь собственные методы и свойства. Перечислениям же можно назначать только значимые типы (например, из перечисленных в начале главы 4), хранить в них ссылочные переменные нельзя.

Перечисления не могут динамически менять размер. Они не реализуют интерфейсы, не имеют методов, а для сохранения значения из перечисления в переменной другого типа потребуется операция приведения. Как видите, это разные способы хранения данных.

В: Зачем при таком мощном инструменте, как `List`, мне могут потребоваться массивы?

О: Бывают случаи, когда работать приходится с фиксированным количеством элементов или с последовательностью значений фиксированной длины.

Вы можете преобразовать коллекцию в массив при помощи метода ToArray()... обратное преобразование совершается при помощи одного из перегруженных конструкторов объекта List<T>.

В: Почему коллекция называется обобщенной?

О: Обобщенная коллекция — это объект (или встроенный объект, позволяющий хранить множество других объектов), который настраивается под хранение одного типа данных.

В: Вы объяснили, что такое «коллекция». Но почему «обобщенная»?

О: В супермаркеты товар доставляют в однотипных коробках, на которых написано название содержимого («Чипсы», «Кола», «Мыло» и т. п.). Важно, что внутри коробки, а не то, как это выглядит.

Так же и с обобщенными типами данных. Объект List<T>, наполненный объектами Shoe, объектами Card, целыми числами или даже другими коллекциями, служит таким же контейнером. Вы можете добавлять, удалять, вставлять элементы вне зависимости от их типа.

В: Возможна ли коллекция, не имеющая типа?

О: Нет. Любая обобщенная коллекция должна принадлежать определенному типу. В C# имеются и коллекции `ArrayLists`, хранящие объекты произвольного типа. Чтобы воспользоваться ими, нужно включить в верхнюю часть кода строчку `using System.Collections;`. Но вряд ли это вам когда-либо потребуется, так как коллекция `List<object>` в большинстве случаев замечательно работает!

Создавая объект List, вы всегда указываете тип данных, которые будут в нем храниться. Сохранять можно значимые типы (int, bool или decimal) или класс.

Массивы занимают меньше места в памяти и быстрее обрабатываются процессором, впрочем, выигрыш в производительности получается незначительным. Если ваша программа работает слишком медленно, вряд ли проблему можно решить переходом от коллекций к массивам.

Термин «обобщенный» указывает, что хотя каждый экземпляр List может сохранять данные только одного типа, класс List работает с любыми типами данных.

Именно эту функцию выполняет символ <T>. Вместо него вы указываете, какому типу принадлежит рассматриваемый объект List. Этим обобщенные коллекции отличаются от всего, что вы использовали раньше.

Инициализаторы коллекций похожи на инициализаторы объектов

C# позволяет уменьшить количество вводимого текста при создании коллекции. Вы можете воспользоваться **инициализатором коллекций (collection initializer)**, который добавляет элементы в коллекцию непосредственно в момент ее создания.

Этот код вы видели несколько страниц назад. Он создает объект List<Shoe> и заполняет его объектами Shoe.

```
List<Shoe> shoeCloset = new List<Shoe>();
shoeCloset.Add(new Shoe() { Style = Style.Sneakers, Color = "Черный" });
shoeCloset.Add(new Shoe() { Style = Style.Clogs, Color = "Коричневый" });
shoeCloset.Add(new Shoe() { Style = Style.Wingtips, Color = "Черный" });
shoeCloset.Add(new Shoe() { Style = Style.Loafers, Color = "Белый" });
shoeCloset.Add(new Shoe() { Style = Style.Loafers, Color = "Красный" });
shoeCloset.Add(new Shoe() { Style = Style.Sneakers, Color = "Зеленый" });
```

Код, полученный при помощи инициализатора

Обратите внимание: каждому объекту Shoe присваивается начальное значение при помощи инициализатора.

Инициализатор коллекции может быть создан добавлением каждого полученного при помощи метода Add() элемента к оператору, формирующему коллекцию.

```
List<Shoe> shoeCloset = new List<Shoe>() {
    new Shoe() { Style = Style.Sneakers, Color = "Черный" },
    new Shoe() { Style = Style.Clogs, Color = "Коричневый" },
    new Shoe() { Style = Style.Wingtips, Color = "Черный" },
    new Shoe() { Style = Style.Loafers, Color = "Белый" },
    new Shoe() { Style = Style.Loafers, Color = "Красный" },
    new Shoe() { Style = Style.Sneakers, Color = "Зеленый" },
};
```

За оператором создания коллекции следуют фигурные скобки, в которых находятся разделенные запятыми операторы `new`.

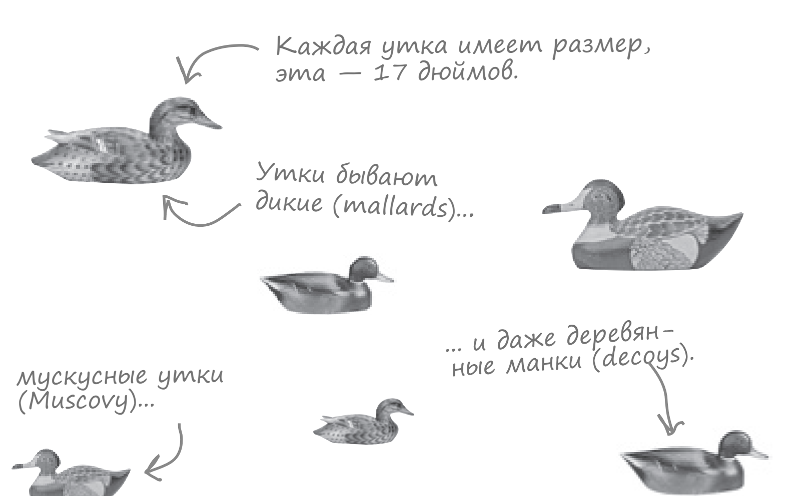
Инициализатор может содержать не только набор операторов `new`, но и переменные.

Инициализатор коллекций делает код компактнее, комбинируя создание коллекции с добавлением начального набора элементов.

Коллекция уток

Рассмотрим класс Duck, следящий за вашей коллекцией уток. (Вы же коллекционируете уток, правда?) **Создайте консольное приложение** и добавьте в него класс Duck и перечисление KindOfDuck (Виды уток).

Duck
Size Kind
Quack() Swim() Eat() Walk()




```
class Duck {
    public int Size;
    public KindOfDuck Kind;
}
```

Класс имеет два открытых поля и методы, которые здесь не показываются.

```
enum KindOfDuck {
    Mallard,
    Muscovy,
    Decoy,
}
```

Перечисление KindOfDuck хранит информацию о видах уток в коллекции.

Добавьте к проекту класс Duck и перечисление KindOfDuck.

Инициализатор коллекции уток

У нас шесть уток, поэтому мы создадим коллекцию List<Duck> с состоящим из шести операторов инициализатором. Каждый из этих операторов создает новую утку, указывая значения поля Size и Kind. **Добавьте этот код** в метод Main() в файле Program.cs:

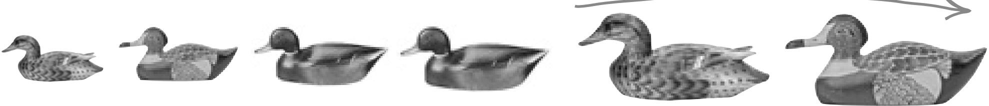
```
List<Duck> ducks = new List<Duck>() {
    new Duck() { Kind = KindOfDuck.Mallard, Size = 17 },
    new Duck() { Kind = KindOfDuck.Muscovy, Size = 18 },
    new Duck() { Kind = KindOfDuck.Decoy, Size = 14 },
    new Duck() { Kind = KindOfDuck.Muscovy, Size = 11 },
    new Duck() { Kind = KindOfDuck.Mallard, Size = 14 },
    new Duck() { Kind = KindOfDuck.Decoy, Size = 13 },
};
// Этот метод не позволяет результату исчезнуть, пока вы его не прочитали
Console.ReadKey();
```

Вы добавляете к методу Main() код, выводящий результаты. Эта строчка оставляет результат видимым, пока вы не нажмете клавишу.

Сортировка элементов коллекции

Сортировка чисел или букв — вполне обычное дело. Но вот как отсортировать объекты, особенно при наличии нескольких полей? Можно расположить объекты по именам, в других случаях имеет смысл сортировать по длине или по дате рождения. Существует много способов расположить объекты по порядку, и коллекции поддерживают все.

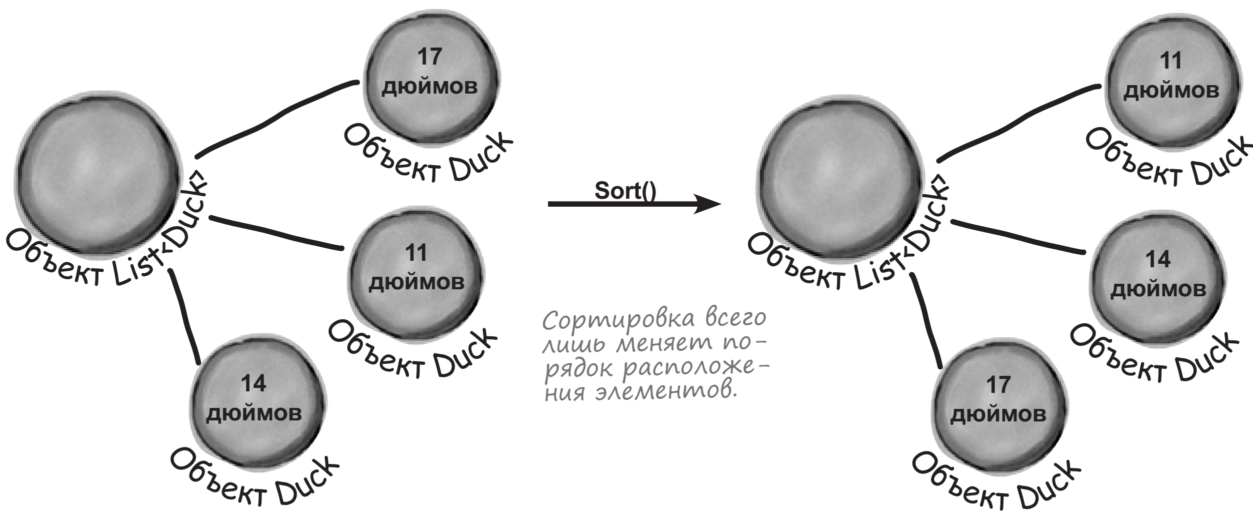
от самой маленькой к самой большой...



Сортировка объектов осуществляется с помощью интерфейса IComparer<T>, о котором мы поговорим чуть позже...

Встроенные методы сортировки

Каждая коллекция снабжена методом `Sort()`, меняющим порядок элементов. Существуют заданные способы сортировки большинства встроенных типов и классов, кроме того, можно написать способ сортировки вашего собственного класса.



Интерфейс IComparable<Duck>

Метод `List.Sort()` умеет сортировать объекты любого типа и классы, которые **реализуют интерфейс IComparable<T>**. В составе этого интерфейса находится только метод `CompareTo()`. `Sort()`, который использует метод `CompareTo()` объекта для сравнения с другими объектами. Возвращаемое значение (типа int) показывает, кто должен быть первым.

Для коллекций объектов, которые не реализуют интерфейс `IComparable<T>`, в .NET имеется другой способ. Метод `Sort()` можно передать экземпляру класса, реализующего `IComparer<T>`. Этот интерфейс тоже имеет всего один метод и позволяет задавать специальные способы сортировки.

Любой класс может работать со встроенным методом Sort( ) объекта List с помощью IComparable<T> и CompareTo( ).

Метод CompareTo() сравнивает два объекта

Можно отредактировать класс `Duck` таким образом, чтобы он начал реализовывать интерфейс `IComparable<Duck>`. Для этого добавим метод `CompareTo()`, использующий ссылку на объект `Duck` в качестве параметра. Если утка, взятая для сравнения, должна оказаться после анализируемой утки, метод `CompareTo()` возвращает положительное число.

Обновите класс `Duck` путем реализации интерфейса `IComparable<Duck>`, чтобы отсортировать уток по размеру:

```
class Duck : IComparable<Duck> {
    public int Size;
    public KindOfDuck Kind;

    public int CompareTo(Duck duckToCompare) {
        if (this.Size > duckToCompare.Size)
            return 1;
        else if (this.Size < duckToCompare.Size)
            return -1;
        else
            return 0;
    }
}
```

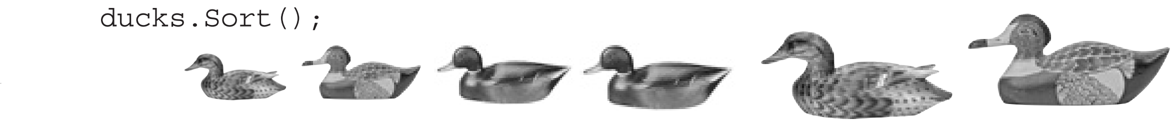
Реализуя IComparable<T>, вы указываете тип параметра, по которому осуществляется сравнение.

Так выглядят большинство методов CompareTo(). В данном случае сравниваются значения полей Size двух уток. Если утка, на которую ссылается слово this, больше, возвращается 1. В противном случае, –1. А в случае совпадения размеров — ноль.

Чтобы получить ряд от самой маленькой до самой большой утки, метод CompareTo() должен возвращать положительное число при сравнении с уткой меньшего размера.

Добавьте этот код в конец метода `Main()` до вызова метода `Console.ReadKey()`, и коллекция уток будет отсортирована по размеру. Поместите **точку останова** в метод `CompareTo()` и воспользуйтесь отладчиком, чтобы понять, как все работает.

```
ducks.Sort();
```



Способы сортировки

Для коллекций существует специальный встроенный в .NET Framework интерфейс, позволяющий создать отдельный класс для сортировки составляющих объекта `List<T>`. **Реализуя интерфейс `IComparer<T>`**, вы объясняете коллекции, каким способом нужно упорядочить ее содержимое. Задача выполняется средствами метода `Compare()`, который берет параметры двух объектов *x* и *y* и возвращает целое число. Если *x* меньше, чем *y*, возвращается отрицательное число. В случае их равенства возвращается ноль. Ну а если *x* больше, чем *y*, будет возвращено положительное число.

Способ сортировки зависит от способа реализации интерфейса IComparer<T>.

Вот пример объявления класса, сравнивающего объекты `Duck` по размеру. Добавьте этот класс к своему проекту.

Этот класс реализует интерфейс IComparer и указывает тип сортируемых объектов: Duck.

Тип сравниваемых значений всегда будет совпадать.

```
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;
        }
}
```

Метод Compare() возвращает целое число и имеет два параметра, принадлежащих типу сортируемых объектов.

Метод позволяет осуществлять любые типы сравнений.

Отрицательное число означает, что x должен стоять перед y, так как x «меньше, чем» y.

Положительное число означает, что x должен следовать за объектом y, так как x «больше» y.

Ноль означает совпадение размеров объектов.

Это метод вывода уток в виде коллекции List<Duck>.

Добавьте метод PrintDucks в класс Program, чтобы обеспечить вывод данных.

Обновите метод Main() таким образом, чтобы он вызывался до и после сортировки.

```
public static void PrintDucks(List<Duck> ducks)
{
    foreach (Duck duck in ducks)
        Console.WriteLine(duck.Size.ToString() + "-дюймов " + duck.Kind.ToString());
    Console.WriteLine("Утки кончились!");
}
```

Создадим экземпляр объекта-компаратора

Для сортировки с помощью `IComparer<T>` требуется новый экземпляр реализующего этот интерфейс класса. Этот объект помогает методу `List.Sort()` упорядочить массив. Но как и в случае с другими (нестатическими) классами, вам нужно указать начальные значения.

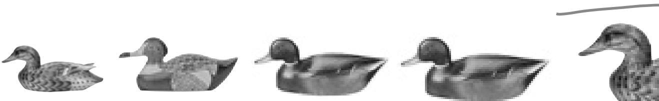
Здесь не написан код, присваивающий элементам коллекции начальные значения, вы найдете его несколькими страницами ранее. Если этого не сделать, ссылки будут указывать на пустые значения.

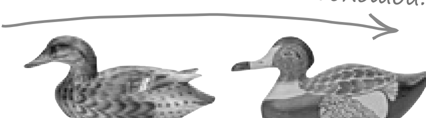
```
DuckComparerBySize sizeComparer = new DuckComparerBySize();
ducks.Sort(sizeComparer);
PrintDucks(ducks);
```

Методу Sort() передается ссылка на новый объект DuckComparerBySize как на параметр метода.

Добавьте этот код в метод Main(), чтобы видеть, как были отсортированы утки.

От самой маленькой к самой большой...





Множественные реализации интерфейса IComparer как разные способы сортировки

Можно создать набор классов `IComparer<Duck>`, каждый из которых будет сортировать уток по-своему. Вам будет достаточно только выбрать нужный компаратор. Добавим к проекту еще одну реализацию процедуры сравнения:

Этот компаратор сортирует уток по виду. Помните, что сравнение элементов перечисления Kind осуществляется по их **индексу**.

В итоге дикая утка окажется перед мускусной, которая в свою очередь оказывается впереди манка.

```
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 сравниваем свойство уток Kind, поэтому сортировка происходит по индексам перечисления KindOfDuck.

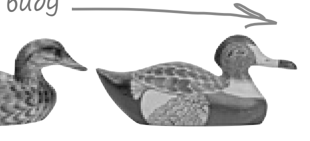
Пример совместного использования перечислений и коллекций.

В данном случае операторы «больше, чем» и «меньше, чем» имеют другое значение. Мы использовали логические операторы < и > для сравнения индексов перечисления при сортировке уток.

```
DuckComparerByKind kindComparer = new DuckComparerByKind();
ducks.Sort(kindComparer);
PrintDucks(ducks);
```

Сортировка по виду утки...

Дополнительный код для метода Main().



Если метод Sort() для объекта IComparer<T> не указан, будет использован заданный по умолчанию метод сравнения значимых типов или ссылок.

Сложные схемы сравнения

Отдельный класс для сортировки уток позволяет применить более сложную логику сравнения, добавив члены, определяющие метод сортировки коллекции.

```
enum SortCriteria {
    SizeThenKind,
    KindThenSize,
}

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;
    }
}

DuckComparer comparer = new DuckComparer();

comparer.SortBy = SortCriteria.KindThenSize;
ducks.Sort(comparer);
PrintDucks(ducks);

comparer.SortBy = SortCriteria.SizeThenKind;
ducks.Sort(comparer);
PrintDucks(ducks);
```

Перечисление указывает объекту, каким именно способом будут сортироваться утки. Первый вариант «По размеру, затем по виду», второй — наоборот.

Класс, выполняющий сравнение уток, более сложен. Метод Compare() использует те же самые параметры, но сначала проверяет значение открытого поля SortBy, чтобы определить способ сортировки.

Оператор if проверяет поле SortBy. Если оно имеет значение SizeThenKind, утки сначала будут упорядочиваться по размеру, а затем в пределах каждого размера пройдет сортировка по виду.

Раньше, если утки имели один и тот же размер, возвращалось нулевое значение, теперь же проверяется еще и видовая принадлежность уток. В итоге 0 возвращается не только когда утки имеют одинаковый размер, но и когда относятся к одному и тому же виду.

Если значение поля SortBy отлично от SizeThenKind, то сначала сортировка выполняется по виду утки. Обнаружив двух уток одного вида, компаратор сравнивает их размер.

Сначала мы, как обычно, создаем экземпляр компаратора. Затем присваиваем значение полю SortBy, после чего можно вызвать метод ducks.Sort(). Теперь вы можете менять метод сортировки уток, редактируя значение одного поля. Добавьте этот код в конец метода Main(), и вы получите возможность сортировать коллекцию много раз!

Упражнение

Создайте пять случайных карт и расположите их в определенном порядке.

1 Код, моделирующий набор перемешанных карт

Создайте консольное приложение и добавьте в метод `Main()` код, создающий пять случайных объектов Card. Используйте встроенный метод `Console.WriteLine()` для записи имени каждого объекта в окне вывода. Добавьте в конец программы метод `Console.ReadKey()`, чтобы это окно не исчезало после окончания работы программы.

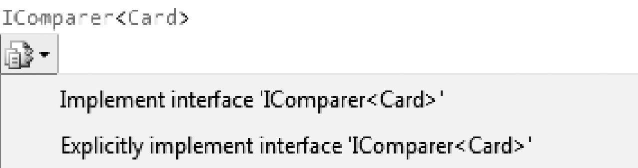
2 Класс, который реализует сортирующий карты интерфейс IComparer<Card>

Воспользуйтесь средствами IDE, сокращающими трудозатраты. Введите:

```
class CardComparer_byValue : IComparer<Card>
```

Щелкните на `IComparer<Card>` и задержите курсор над символом I. Появится черта, щелчок на которой откроет вот такое окно:

Альтернативным способом вызова этого меню является клавиатурная комбинация ***Ctrl-точка****.*

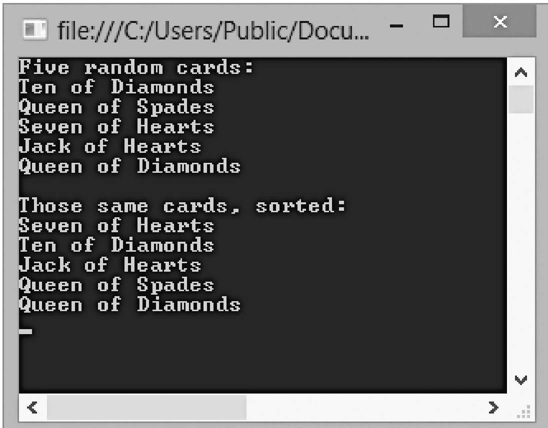


Выберите вариант Implement interface IComparer<Card>, и IDE введет за вас все методы и свойства, необходимые для реализации данного интерфейса. Будет создан пустой метод `Compare()`, сравнивающий карты `x` и `y`. Пусть метод возвращает 1 (если `x` больше `y`), –1 (если меньше) и 0 (если карты одинаковы). Убедитесь, что король следует после валета, который следует за четверкой, следующей за тузом.

3 Конечный результат

Вот как выглядит окно вывода:

Встроенный метод Console.WriteLine() добавляет строки в результат, в то время как метод Console.ReadKey() ждет нажатия клавиши, чтобы завершить программу.



Объект IComparer должен осуществлять сортировку таким образом, чтобы первыми в списке оказывались карты с наименьшим значением.

Упражнение Решение

Вот как выглядит сортировка случайного набора карт.

```
class CardComparer_byValue : IComparer<Card> {
    public int Compare(Card x, Card y) {
        if (x.Value < y.Value) {
            return -1;
        }
        if (x.Value > y.Value) {
            return 1;
        }
        if (x.Suit < y.Suit) {
            return -1;
        }
        if (x.Suit > y.Suit) {
            return 1;
        }
        return 0;
    }
}
```

Встроенный метод List.Sort() берет объект IComparer, имеющий один метод: Compare(). Эта реализация проводит сравнение двух карт сначала по старшинству, потом по масти.

Если x имеет большее значение, метод возвращает 1. Если значение меньше, возвращается -1. Но оба оператора, возвращающие значение, завершают работу метода.

Эти операторы выполняются только при совпадении значений x и y, ведь первые два оператора, возвращающие значение, в этом случае не выполняются.

Если ни один из четырех операторов, возвращающих значение, не сработал, значит, карты одинаковые. В этом случае возвращается 0.

```
static void Main(string[] args)
{
    Random random = new Random();
    Console.WriteLine("Пять случайных карт:");
    List<Card> cards = new List<Card>();
    for (int i = 0; i < 5; i++)
    {
        cards.Add(new Card((Suits)random.Next(4),
                          (Values)random.Next(1, 14)));
        Console.WriteLine(cards[i].Name);
    }

    Console.WriteLine();
    Console.WriteLine("Те же карты, отсортированные:");
    cards.Sort(new CardComparer_byValue());
    foreach (Card card in cards)
    {
        Console.WriteLine(card.Name);
    }
    Console.ReadKey();
}
```

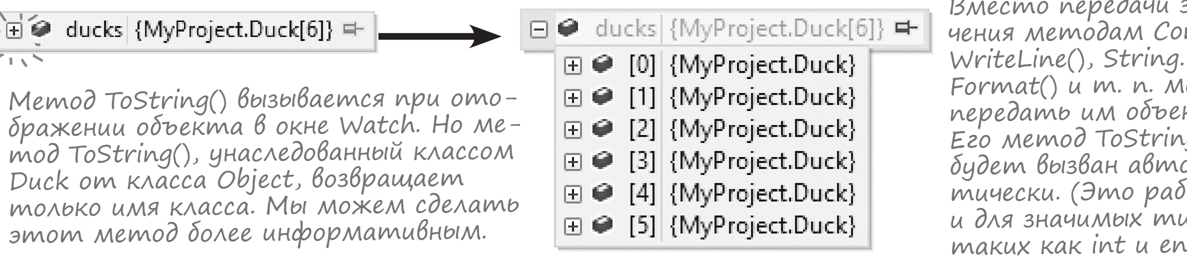
Это обобщенная коллекция объектов Card. Ее легко сортировать с помощью интерфейса IComparer.

Метод Console.ReadKey() нужен, чтобы программа не закрывалась после завершения. Для реальных приложений он не подходит. При запуске программы по Ctrl-F5 она начинает работать без отладки. После завершения появляется строка Press any key to continue..., и приложение ждет нажатия клавиши. Но отладки не происходит, и точки останова с контрольными значениями не работают.

Перекрытие метода ToString()

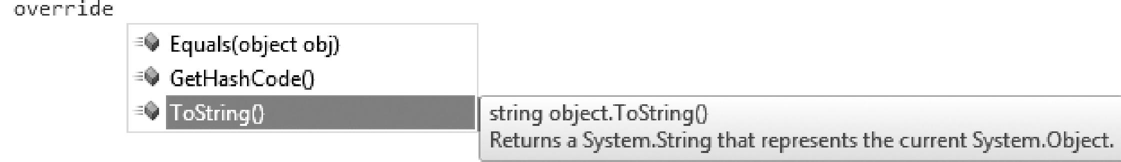
Все объекты .NET имеют **метод ToString(), преобразующий объект в строку**. По умолчанию он возвращает имя класса (MyProject.Duck). Этот метод унаследован от класса Object (который является базовым для любого объекта). Оператор +, соединяющий строки **автоматически, вызывает метод ToString()**. Методы Console.WriteLine() или String.Format() так же автоматически вызывают его при передаче им объектов.

В программе сортировки уток поместите точку останова в метод Main() после инициализации коллекции и запустите отладку программы. Затем **наведите указатель мыши на любую переменную ducks**, чтобы знать ее значение. Увидеть при отладке содержимое коллекции можно, щелкнув на кнопке со знаком + слева от имени переменной:



Итак, вы видите, что коллекция содержит шесть объектов Duck (MyProject — это пространство имен, в котором мы находимся). Щелчок на кнопке + (слева от номера утки) показывает значения параметров Kind и Size. Но нельзя ли сделать так, чтобы вся информация показывалась одновременно?

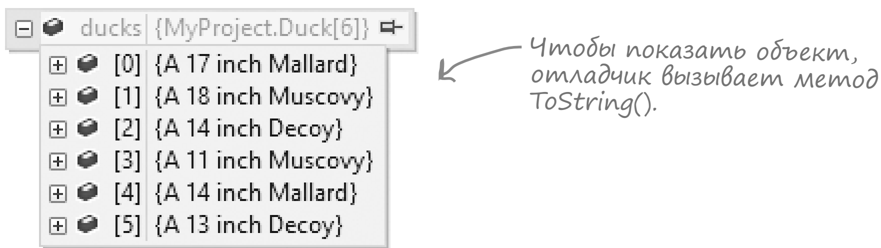
ToString() — это виртуальный метод класса Object, который является базовым по отношению к любому объекту. Так что вам остается только **перекрыть метод ToString()**, и вы увидите результаты в окне watch! Откройте класс Duck и добавьте новый метод с ключевым словом **override**. После нажатия Пробела появится список доступных для перекрытия методов:



Выберите вариант ToString() и замените содержимое метода вот этим:

```
public override string ToString()
{
    return "A " + Size + " inch " + Kind.ToString();
}
```

Запустите программу и снова посмотрите на коллекцию. Теперь вы видите все сведения о Duck!



Обновим цикл foreach

Вы видели уже два примера циклического вызова метода `Console.WriteLine()` для вывода информации об элементах коллекции на консоль. Вот так выводились сведения о коллекции `List<Card>`:

```
foreach (Card card in cards)
{
    Console.WriteLine(card.Name);
}
```

Метод `PrintDucks()` выполнял аналогичную функцию для объектов `Duck`:

```
foreach (Duck duck in ducks)
{
    Console.WriteLine(duck.Size.ToString() + "-inch " + duck.Kind.ToString());
}
```

Упоминание о методе .ToString() в данном случае можно опустить, оператор + вызовет его автоматически.

Теперь, когда объект `Duck` получил свой метод `ToString()`, метод `PrintDucks()` должен использовать это преимущество:

```
public static void PrintDucks(List<Duck> ducks) {
    foreach (Duck duck in ducks) {
        Console.WriteLine(duck);
    }
Console.WriteLine("Утки кончились!");
}
```

При передаче методу Console.WriteLine() ссылки на объект он автоматически вызовет метод ToString() этого объекта.

Введите этот код в программу `Ducks` и запустите ее. Список вывода не изменится. Но если вы захотите добавить свойство `Gender` для учета пола утки, достататочно обновить метод `ToString()`, и все методы, которые его используют (включая метод `PrintDucks()`), изменятся соответствующим образом.

Добавим метод ToString() к объекту Card

Свойство `Name` объекта `Card` возвращает имя карты:

```
public string Name
{
    get { return Value.ToString() + " of " + Suit.ToString(); }
}
```

Вы до сих пор можете вызывать метод ToString() таким способом, но теперь вы знаете, что в этом нет необходимости, так как оператор + вызывает его автоматически.

Вот что должен делать метод `ToString()`. Поэтому добавьте его к классу `Card`:

```
public override string ToString()
{
    return Name;
}
```

Теперь ваши программы, использующие объекты `Card`, можно легко отладить.

Функции метода ToString() не ограничиваются идентификацией ваших объектов в IDE. В следующих главах мы рассмотрим преимущества, которые дает преобразование объектов в строки.

Цикл foreach под увеличительным стеклом

Интерфейс IEnumerable<T>

Найдите переменную List<Duck> и воспользуйтесь функцией IntelliSense для рассмотрения метода GetEnumerator(). Напечатайте «.GetEnumerator» и посмотрите на открывшееся меню:

Инициализаторы коллекций работают с ЛЮБЫМ объектом IEnumerable<T>!

```
ducks.GetEnumerator
```

List<Duck>.Enumerator List<Duck>.GetEnumerator()
Returns an enumerator that iterates through the System.Collections.Generic.List<T>.

Создайте новый массив объектов Duck:

```
Duck[] duckArray = new Duck[6];
```

Напечатайте duckArray.GetEnumerator, ведь у массива имеется данный метод. Это связано с тем, что объекты List, как и массивы, реализуют интерфейс **IEnumerable<T>**, содержащий единственный метод GetEnumerator(), возвращающий **элемент перечисления**.

Это объект Enumerator, обеспечивающий механизм перебора коллекции. Рассмотрим цикл foreach, просматривающий коллекцию List<Duck> с переменной duck:

```
foreach (Duck duck in ducks) {
   Console.WriteLine(duck);
}
```

Код, скрывающийся за этим циклом:

```
IEnumerator<Duck> enumerator = ducks.GetEnumerator();
while (enumerator.MoveNext()) {
     Duck duck = enumerator.Current;
     Console.WriteLine(duck);
}
IDisposable disposable = enumerator as IDisposable;
if (disposable != null) disposable.Dispose();
```

Реализуя интерфейс IEnumerable<T>, коллекция предоставляет цикл, перебирающий ее элементы по очереди.

(Не беспокойтесь, что вы не понимаете последних двух строк. С интерфейсом IDisposable вы познакомитесь в главе 9.)

Здесь написан не весь код, но в данном случае главное, чтобы вы поняли основную идею...

Эти два кода выводят один и тот же список уток. Можете проверить самостоятельно.

При циклическом просмотре коллекции или массива метод MoveNext() возвращает значение true при наличии следующего элемента и значение false, если достигнут последний элемент. Свойство Current всегда возвращает ссылку на текущий элемент. А цикл foreach объединяет указанные функции!

Поэкспериментируйте, заставив метод ToString() объекта Duck увеличивать свойство Size на единицу. Запустите отладку и наведите указатель мыши на имя Duck. Проделайте это несколько раз. Помните, что каждый раз будет вызываться метод ToString().

Как вы думаете, что будет происходить при выполнении цикла foreach, если метод ToString() меняет одно из полей объекта?

Восходящее приведение с помощью IEnumerable

Помните о возможности восходящего приведения объектов к базовому классу? При работе с объектами `List` эту операцию можно проделать для всей коллекции. Это называется **ковариацией (covariance)**, и вам потребуется только ссылка на интерфейс `IEnumerable<T>`.

Создайте консольное приложение и в нем базовый класс `Bird` (его продолжением будет класс `Duck`) и производный класс `Penguin`. Воспользуемся методом `ToString()`, чтобы понять, где какой класс.

Bird: Name; Fly()

Duck: Size, Kind

Penguin

Упражнение!

```
class Bird {
    public string Name { get; set; }
    public virtual void Fly() {
        Console.WriteLine("Полетели!");
    }
    public override string ToString() {
        return "Имя птицы " + Name;
    }
}
```

```
class Penguin : Bird
{
    public override void Fly() {
       Console.WriteLine("Пингвины не летают!");
    }
    public override string ToString() {
        return "Имя пингвина " + base.Name;
    }
}
```

Это класс Bird и наследующий от него класс Penguin. Добавьте их в новый проект типа Console Application, затем скопируйте туда же существующий класс Duck, поменяв соответствующим образом его объявление.

```
class Duck : Bird, IComparable<Duck> {
    // Остальной код без изменений
}
```

Вот первые строки метода `Main()`, инициализирующие коллекцию **и осуществляющие ее восходящее приведение**.

Скопируйте инициализатор для коллекции уток.

```
List<Duck> ducks = new List<Duck>() { /* инициализируйте объект как обычно */ }
IEnumerable<Bird> upcastDucks = ducks;
```

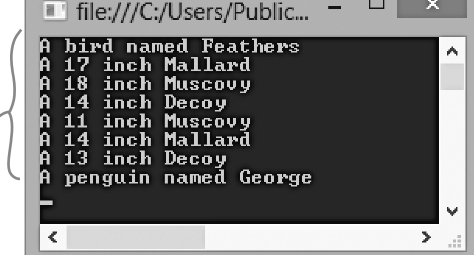
Посмотрите на последнюю строчку. Ссылку на коллекцию `List<Duck>` вы присваиваете интерфейсной переменной `IEnumerable<Bird>`. Запустите отладку и убедитесь, что обе ссылки указывают на один и тот же объект.

Объединим птиц в единую коллекцию

Ковариация позволяет добавлять частную коллекцию к более общей. Скажем, в коллекцию объектов `Bird` можно добавить коллекцию `Duck`. Вам пригодится метод `List.AddRange()`.

```
List<Bird> birds = new List<Bird>();

birds.Add(new Bird() { Name = "Пернатые" });
birds.AddRange(upcastDucks);
birds.Add(new Penguin() { Name = "Джордж" });

foreach (Bird bird in birds) {
    Console.WriteLine(bird);
}
```

Благодаря восходящему приведению уток к IEnumerable<Bird> их можно добавить в коллекцию объектов Bird.



Создание перегруженных методов

Вместо редактирования пространства имен можно использовать оператор.

Вы уже использовали не только **перегруженные методы,** но и перегруженный конструктор, который был частью встроенного класса .NET. Словом, вы уже знаете, насколько полезной является эта функция. Хотели бы вы встраивать перегруженные методы в ваши собственные классы? Это легко можно сделать! Достаточно написать два и более одноименных метода с различными параметрами.

1. **Создайте новый проект и добавьте в него класс Card.**
   Это легко сделать, щелкнув правой кнопкой мыши на имени проекта в окне Solution Explorer и выбрав вариант Existing Item в меню Add. IDE добавит в проект копию класса. Но при этом **он останется в пространстве имен старого проекта**, поэтому откройте файл `Card.cs` и отредактируйте строчку `namespace`. Аналогичную операцию проделайте для `Values` и `Suits`.

Без этой операции для доступа к классу Card потребуется указывать пространство имен в явном виде (например, oldnamespace.Card).

2. **Теперь добавим в класс Card новые перегруженные методы.**
   Создайте два статических метода `DoesCardMatch()`. Первый будет проверять масть карты, а второй — старшинство. Оба метода возвращают значение true при совпадении карт.

```
public static bool DoesCardMatch(Card cardToCheck, Suits suit) {
   if (cardToCheck.Suit == suit) {
      return true;
   } else {
      return false;
   }
}
```

Перегруженные методы не обязаны быть статическими, но мы решили, что вам будет полезно попрактиковаться в написании статических методов.

```
public static bool DoesCardMatch(Card cardToCheck, Values value) {
   if (cardToCheck.Value == value) {
      return true;
   } else {
      return false;
   }
}
```

Как работает процедура перегрузки, вы уже видели в программе для расчета стоимости вечеринок из главы 6, там вы добавляли перегруженный метод CalculateCost() в класс DinnerParty.

3. **Добавьте к форме кнопку, использующую оба метода.**
   Вот код этой кнопки:

```
Card cardToCheck = new Card(Suits.Clubs, Values.Three);
bool doesItMatch = Card.DoesCardMatch(cardToCheck, Suits.Hearts);
Console.WriteLine(doesItMatch);
```

Как только будет напечатано `DoesCardMatch()`, IDE покажет, что вы действительно написали перегруженный метод:

```
Card.DoesCardMatch(
▲ 1 of 2 ▼ bool Card.DoesCardMatch(Card cardToCheck, Suits suit)
```

Поэкспериментируйте с этими методами, чтобы привыкнуть к работе.

Упражнение

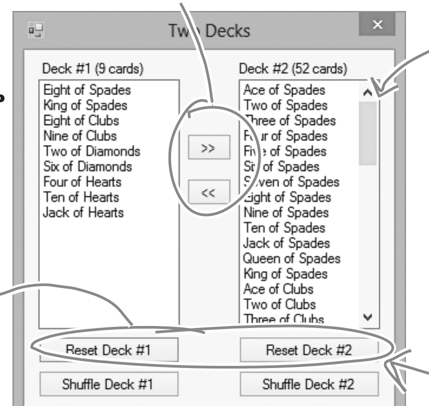
Попрактикуемся в применении объектов `List`, создав класс для хранения колоды карты и форму, которая будет его использовать.

1 Форма, позволяющая перемещать карты между колодами

Создадим класс `Deck` (Колода) для хранения произвольного количества карт. В реальной колоде 52 карты, но в классе `Deck` может быть сколько угодно карт (или не быть ни одной).

Затем вам потребуется форма, показывающая содержимое двух объектов `Deck`. При первом запуске программы в колоде #1 может быть до 10 случайных карт, а в колоде #2 — полный набор (52 карты). Обе колоды отсортированы по масти и по старшинству. В это положение колоду можно вернуть в любой момент щелчком на кнопке Reset. Также форма снабжена кнопками << и >>, которые перемещают карты из одной колоды в другую.

Помните, что с помощью свойства Name элементу управления можно присвоить имя, улучшив тем самым читабельность кода. При двойном щелчке на кнопке обработчику событий будет дано соответствующее имя.



Добавить к форме нужно не только обработчики событий для шести кнопок, но и два метода. Начните с метода `ResetDeck()`, возвращающего колоду в исходное состояние. В качестве параметра он использует значение типа int: если ему передать 1, он превращает первый объект `Deck` в пустую колоду, а затем случайным образом назначает ей 10 карт; передав значение 2, вы превратите второй объект `Deck` в упорядоченную колоду из 52 карт:

```
private void RedrawDeck(int DeckNumber) {
    if (DeckNumber == 1) {
        listBox1.Items.Clear();
        foreach (string cardName in deck1.GetCardNames())
            listBox1.Items.Add(cardName);
        label1.Text = "Deck #1 (" + deck1.Count + " cards)";
    } else {
        listBox2.Items.Clear();
        foreach (string cardName in deck2.GetCardNames())
            listBox2.Items.Add(cardName);
        label2.Text = "Deck #2 (" + deck2.Count + " cards)";
    }
}
```

Метод RedrawDeck() тасует колоду, вытаскивает из нее случайные карты и обновляет содержимое текстовых полей в соответствии с проделанным обменом.

*Обратите внимание, как карты добавляются в коллекцию при помощи цикла **foreach**.*

2 Создание класса Deck

«Основой» мы назвали объявление класса без его реализации.

Вот основа класса Deck. Мы написали за вас несколько методов. Вам осталось добавить методы `Shuffle()` и `GetCardNames()` и заставить работать метод `Sort()`. Мы добавили два **перегруженных конструктора**: создающий колоду из 52 карт и загружающий в колоду содержимое массива объектов Card.

Deck
Count
Add() Deal() GetCardNames() Shuffle() Sort()

Класс Deck хранит объекты в коллекции List, при этом они являются закрытыми.

Полная колода из 52 карт будет создана без передачи параметров конструктору.

Принадлежность параметра к типу IEnumerable<Card> позволяет передавать конструктору любую коллекцию, а не только List<T> или массив.

Перегруженный конструктор в качестве параметра берет массив карт, загружая его в исходную колоду.

Метод Deal (Раздача) выдает одну карту из колоды, он удаляет объект Card и возвращает ссылку на него. Передав 0, вы сдадите верхнюю карту колоды. Чтобы сдать любую другую карту, укажите ее индекс.

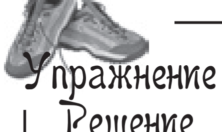
Хотя метод GetCardNames() возвращает массив, мы даем доступ к IEnumerable<string>.

```
class Deck {
    private List<Card> cards;
    private Random random = new Random();

    public Deck() {
        cards = new List<Card>();
        for (int suit = 0; suit <= 3; suit++)
            for (int value = 1; value <= 13; value++)
                cards.Add(new Card((Suits)suit, (Values)value));
    }

    public Deck(IEnumerable<Card> initialCards) {
        cards = new List<Card>(initialCards);
    }

    public int Count { get { return cards.Count; } }

    public void Add(Card cardToAdd) {
        cards.Add(cardToAdd);
    }

    public Card Deal(int index) {
        Card CardToDeal = cards[index];
        cards.RemoveAt(index);
        return CardToDeal;
    }

    public void Shuffle() {
        // метод тасует карты, меняя их порядок случайным образом
    }

    public IEnumerable<string> GetCardNames() {
        // метод возвращает массив типа string с именами всех карт
    }

    public void Sort() {
        cards.Sort(new CardComparer_bySuit());
    }
}
```

Подсказка: *свойство SelectedIndex элемента управления ListBox совпадает с индексом карты в коллекции. Его можно передать методу Deal(). Если карта не выбрана, значение меньше нуля, и метод moveToDeck не работает.*

Вам нужно написать методы Shuffle() и GetCardNames(), а также добавить класс, реализующий IComparer, чтобы заставить работать метод Sort(). Кроме того, потребуется добавить уже написанный класс Card. Если вы будете делать это командой Add Existing Item, не забудьте отредактировать пространство имен.

Еще одна подсказка: *Протестировать метод Shuffle() можно с помощью формы. Щелкайте на кнопке "Reset Deck #1", пока не получите колоду из трех карт. В этом случае легко убедиться, что метод перетасовки работает.*



Упражнение Решение

Вот код для класса, хранящего информацию о колоде карт, а также для использующей этот класс формы.

```
class Deck {
    private List<Card> cards;
    private Random random = new Random();
    public Deck() {
        cards = new List<Card>();
        for (int suit = 0; suit <= 3; suit++)
            for (int value = 1; value <= 13; value++)
                cards.Add(new Card((Suits)suit, (Values)value));
    }
    public Deck(IEnumerable<Card> initialCards) {
        cards = new List<Card>(initialCards);
    }
    public int Count { get { return cards.Count; } }
    public void Add(Card cardToAdd) {
        cards.Add(cardToAdd);
    }
    public Card Deal(int index) {
        Card CardToDeal = cards[index];
        cards.RemoveAt(index);
        return CardToDeal;
    }
    public void Shuffle() {
        List<Card> NewCards = new List<Card>();
        while (cards.Count > 0) {
            int CardToMove = random.Next(cards.Count);
            NewCards.Add(cards[CardToMove]);
            cards.RemoveAt(CardToMove);
        }
        cards = NewCards;
    }
    public IEnumerable<string> GetCardNames() {
        string[] CardNames = new string[cards.Count];
        for (int i = 0; i < cards.Count; i++)
            CardNames[i] = cards[i].Name;
        return CardNames;
    }
    public void Sort() {
        cards.Sort(new CardComparer_bySuit());
    }
}
```

Этот конструктор создает колоду из 52 карт. Он использует вложенный цикл for. Внешний цикл осуществляет перебор четырех мастей. Соответственно внутренний цикл, перебирающий 13 значений карт, запускается четыре раза, то есть по разу на каждую масть.

Это второй конструктор. Данный класс содержит два перегруженных конструктора с различными параметрами.

Методы Add и Deal довольно прозрачны. Метод Deal убирает карту из коллекции, а метод Add, наоборот, добавляет в нее.

Метод Shuffle() создает экземпляр List<Cards> с именем NewCards. Затем он берет случайную карту из поля Cards и помещает в коллекцию NewCards, пока коллекция Cards не опустеет. После этого он перенаправляет поле Cards на новый экземпляр. Так как ссылок на старый экземпляр в итоге не остается, он удаляется.

Методу GetCardNames() нужно создать достаточно большой массив, чтобы в него поместились имена всех карт. Для этого используется цикл for, хотя задача решается и с помощью цикла foreach.

```
class CardComparer_bySuit : 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;
    }
}
```

Сортировка по масти напоминает сортировку по старшинству. Единственное отличие в том, что масти сравниваются первыми, а сравнение значений карт происходит только при совпадении мастей.

Вместо конструкции if/else if мы использовали набор операторов if. Это работает, так как каждый следующий оператор if выполняется только в случае невыполнения предыдущего.

```
    Deck deck1;
    Deck deck2;
    Random random = new Random();

    public Form1() {
        InitializeComponent();
        ResetDeck(1);
        ResetDeck(2);
        RedrawDeck(1);
        RedrawDeck(2);
    }

    private void ResetDeck(int deckNumber) {
        if (deckNumber == 1) {
            int numberOfCards = random.Next(1, 11);
            deck1 = new Deck(new Card[] { });
            for (int i = 0; i < numberOfCards; i++)
                deck1.Add(new Card((Suits)random.Next(4),
                                   (Values)random.Next(1, 14)));
            deck1.Sort();
        } else
            deck2 = new Deck();
    }
```

Конструктор формы сначала возвращает колоды в исходное состояние, потом перерисовывает их.

Для восстановления колоды #1 сначала используется метод random. Next(), после чего создается пустая колода. При помощи цикла for туда добавляются случайные карты. Напоследок остается провести сортировку. Восстановить колоду #2 еще проще — достаточно создать экземпляр Deck().

Метод RedrawDeck() вам уже встречался.

Переверните страницу и продолжим!

Упражнение Решение

Присвоение элементам управления значимых имен облегчает чтение кода. Если бы эти кнопки назывались button1_Click, button2_Click и т. д., вы не смогли бы сразу определить их назначение!

Это остаток кода формы.

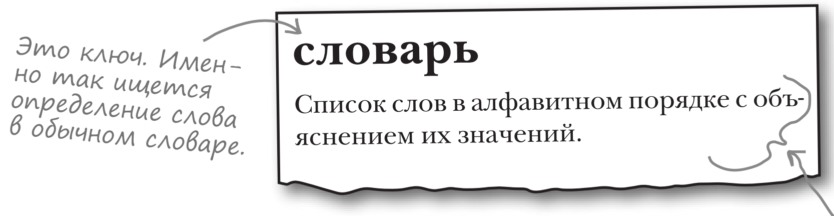
```
    private void reset1_Click(object sender, EventArgs e) {
        ResetDeck(1);
        RedrawDeck(1);
    }

    private void reset2_Click(object sender, EventArgs e) {
        ResetDeck(2);
        RedrawDeck(2);
    }

    private void shuffle1_Click(object sender, EventArgs e) {
        deck1.Shuffle();
        RedrawDeck(1);
    }

    private void shuffle2_Click(object sender, EventArgs e) {
        deck2.Shuffle();
        RedrawDeck(2);
    }

    private void moveToDeck1_Click(object sender, EventArgs e) {
        if (listBox2.SelectedIndex >= 0)
            if (deck2.Count > 0) {
                deck1.Add(deck2.Deal(listBox2.SelectedIndex));
            }
        RedrawDeck(1);
        RedrawDeck(2);
    }

    private void moveToDeck2_Click(object sender, EventArgs e) {
        if (listBox1.SelectedIndex >= 0)
            if (deck1.Count > 0)
                deck2.Add(deck1.Deal(listBox1.SelectedIndex));
        RedrawDeck(1);
        RedrawDeck(2);
    }
}
```

Функция этих кнопок проста — сначала восстановить или перетасовать колоду, а затем перерисовать ее.

Свойство SelectedIndex элемента управления ListBox позволяет понять, какую именно карту выбрал пользователь, чтобы переместить ее. (Если оно имеет отрицательное значение, значит, карта не выбрана, и кнопка ничего не делает.) После перемещения карты требуется перерисовать обе колоды.

Словари

Коллекции подобны длинным страницам, заполненным именами. А теперь представьте, что вы хотите сопоставить каждому имени адрес. Или снабдить каждый автомобиль в вашей коллекции описанием. Для этого вам потребуется **словарь (dictionary)**. Эта структура позволяет взять некий **ключ (key)** и связать его со **значением (value)**. Каждый ключ при этом появляется в словаре **только один раз**.



Это значение, то есть данные, связанные с определенным ключом.

Словарь `Dictionary` в C# объявляется так:

```
Dictionary <Tkey, TValue> kv = new Dictionary <TKey, TValue>();
```

Как и в случае с List<T>, символ <T> соответствует типу. Как видите, вы можете объявить для ключа один тип, а для значений — другой.

Это снова типы. Первый тип в угловых скобках всегда соответствует ключу, а второй — значению.

А вот пример работы со словарем:

```
private void button1_Click(object sender, EventArgs e)
{
    Dictionary<string, string> wordDefinition =
        new Dictionary<string, string>();

    wordDefinition.Add ("Словарь", "Книга, содержащая список слов "
                     + "в алфавитном порядке и объясняющая их значения");
    wordDefinition.Add ("Ключ", “средство, дающее нам доступ "
                     + "к пониманию чего бы то ни было.");
    wordDefinition.Add ("Значение", "Размер, количество, строка или ссылка.");

    if (wordDefinition.ContainsKey("Ключ")) {
        MessageBox.Show(wordDefinition["Ключ"]);
    }
}
```

Ключи и значения в этом словаре принадлежат к типу string, ведь мы моделируем обычный словарь, в котором за термином идет определение.

Ключи и значения добавляются в словарь при помощи метода Add().

Метод Add() сначала берет ключ, а потом значение.

Метод ContainsKey() позволяет определить наличие в словаре указанного в качестве параметра слова.

Этот метод позволяет узнать значение указанного ключа.

Функциональность словарей

Словари во многом напоминают коллекции. Они гибки, позволяют вам работать с данными произвольного типа и имеют множество встроенных функций. Рассмотрим основные методы класса `Dictionary`:

★ **Добавление элементов.**
Добавление осуществляется путем передачи ключа и значения методу `Add()`.

```
Dictionary<string, string> myDictionary = new Dictionary<string, string>();
myDictionary.Add("какой-то ключ", "какое-то значение");
```

★ **Поиск значения по ключу.**
Самым важным при работе со словарем является просмотр значений — собственно, за этим вы их там и храните. Для словаря `Dictionary<string, string>` доступ к значению осуществляется по ключу типа string, возвращает он также строку.

```
string lookupValue = myDictionary["какой-то ключ"];
```

★ **Удаление элементов.**
Как и при работе с объектами `List`, для удаления из словаря используется метод `Remove()`. Достаточно передать ему ключ, как сам ключ и значение будут удалены.

```
myDictionary.Remove("какой-то ключ");
```

Ключи уникальны, а значения могут появляться произвольное количество раз. Двум ключам может соответствовать одно значение. Именно поэтому удаление осуществляется по ключу.

★ **Получение списка ключей.**
Свойство `Keys` в комбинации с циклом `foreach` позволяет получить перечень ключей словаря.

```
foreach (string key in myDictionary.Keys) { ... };
```

Ключи относятся к свойствам словаря. В рассматриваемом случае они принадлежат типу string, поэтому Keys — это набор строк.

★ **Подсчет пар.**
Свойство `Count` возвращает число пар «ключ–значение», имеющихся в словаре:

```
int howMany = myDictionary.Count;
```

Ключ и значение могут принадлежать разным типам

Словари фактически универсальны и могут содержать значения любых типов (от строк до объектов). Вот пример словаря, в котором ключи относятся к типу int, а значения — к объектам Duck.

```
Dictionary<int, Duck> duckDictionary = new Dictionary<int, Duck>();
duckDictionary.Add(376, new Duck()
    { Kind = KindOfDuck.Mallard, Size = 15 });
```

Словари, сопоставляющие целые числа и объекты, используются в случаях присвоения объектам уникальных идентификаторов.

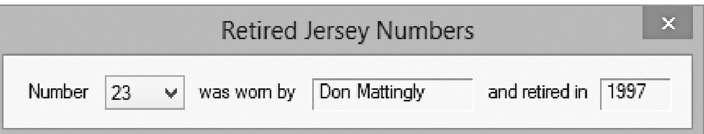
Практическое применение словаря

Вот небольшая программа, которая понравится фанатам бейсбола из Нью-Йорка. Как только игрок перестает выступать за команду, футболка с его номером перестает использоваться. Создадим программу, которая показывает, какие номера были у знаменитых игроков и когда эти игроки ушли из большого спорта.

```
class JerseyNumber {
    public string Player { get; private set; }
    public int YearRetired { get; private set; }

    public JerseyNumber(string player, int numberRetired) {
        Player = player;
        YearRetired = numberRetired;
    }
}
```

Йоги Берра играл под № 8 за одну команду, а Карл Рипкин-младший... под тем же номером за другую. Но в словаре одному значению может соответствовать только один ключ, поэтому в программе на данный момент оказались номера игроков одной команды. Попытайтесь придумать способ сохранять информацию об игроках нескольких команд.

Вот форма:



А это код формы:

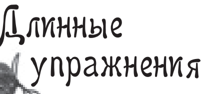
```
public partial class Form1 : Form {
   Dictionary<int, JerseyNumber> retiredNumbers = new Dictionary<int, JerseyNumber>() {
       {3, new JerseyNumber("Babe Ruth", 1948)},
       {4, new JerseyNumber("Lou Gehrig", 1939)},
       {5, new JerseyNumber("Joe DiMaggio", 1952)},
       {7, new JerseyNumber("Mickey Mantle", 1969)},
       {8, new JerseyNumber("Yogi Berra", 1972)},
       {10, new JerseyNumber("Phil Rizzuto", 1985)},
       {23, new JerseyNumber("Don Mattingly", 1997)},
       {42, new JerseyNumber("Jackie Robinson", 1993)},
       {44, new JerseyNumber("Reggie Jackson", 1993)},
  };

   public Form1() {
       InitializeComponent();

       foreach (int key in retiredNumbers.Keys) {
           number.Items.Add(key);
       }
   }

   private void number_SelectedIndexChanged(object sender, EventArgs e) {
       JerseyNumber jerseyNumber = retiredNumbers[(int)number.SelectedItem] as JerseyNumber;
       nameLabel.Text = jerseyNumber.Player;
       yearLabel.Text = jerseyNumber.YearRetired.ToString();
   }
}
```

Объекты JerseyNumber передаются в словарь при помощи инициализатора коллекции.

Ключи из словаря добавляются в коллекцию элементов ComboBox.

Свойство SelectedItem элемента ComboBox является объектом. Так как ключ принадлежит типу int, вам потребуется операция приведения к этому типу.

Событие SelectedIndexChanged элемента ComboBox обновляет две метки на форме, выводя значение объекта JerseyNumber, взятое из словаря.



Длинные упражнения

Создадим симулятор карточной игры.

Это упражнение слегка отличается...
Допускаем, что вы изучаете C# , чтобы потом найти работу. Программисты работают в команде, и никто не занимается созданием программ от начала до конца. Каждый выполняет *кусок* задания. Словом, мы решили предоставить вам пазл, в котором отдельные фрагменты уже собраны. Код формы приведен в шаге #3. Вам остается только ввести его в IDE. Но при этом все классы, которые вы пишете, ***должны работать с этим кодом***. А вот этого добиться не так-то просто!

Спецификация
Работа над любым профессиональным программным обеспечением начинается со спецификации. Вам предстоит построить симулятор классической карточной игры ***Go Fish!*** Существуют различные варианты правил, вот те, которые будете использовать вы:

- Используется колода из 52 карт. Игрокам раздается по пять карт. Карты, оставшиеся после раздачи, называются **запасом**. Игроки по очереди спрашивают про наличие карт («Есть ли у кого семерки?»). Игрок, имеющий нужную карту, отдает ее. Если такой карты ни у кого нет, берется одна карта из запаса.
- Целью игры является сбор взяток. Взяткой считается набор из четырех одинаковых карт. Для выигрыша нужно собрать максимальное количество взяток. Собранный набор из четырех карт выкладывается на стол.
- Если выложенная взятка оставляет игрока без карт, он берет пять карт из запаса. Если в запасе осталось меньше пяти карт, игрок берет их все. Игра заканчивается, как только запас иссякает. Победитель определяется по максимальному количеству взяток.
- В нашей версии человек выступает против двух компьютерных игроков. Каждая партия начинается с выбора человеком карты, о наличии которой он будет спрашивать. Затем два компьютерных игрока спрашивают про свои карты. Результаты каждой партии отображаются в текстовом поле.
- Процедуры сдачи карт и взяток автоматизированы. Как только кто-то побеждает, игра заканчивается и выводится имя победителя (или победителей в случае ничьей). Больше никаких действий выполнить нельзя. Игроку остается только перезагрузить программу, чтобы начать новую партию.

Если задачу не сформулировать заранее, как узнать, что работа закончена? Именно поэтому разработка профессиональных приложений начинается со спецификации, которая сообщает вам, что должно получиться в итоге.

Построение формы

Форме нужен элемент управления `ListBox` для отображения карт игрока, два элемента управления `TextBox` для отображения процесса игры и кнопка, с помощью которой игрок будет спрашивать о наличии карт.

Присвойте свойству Name этого элемента TextBox-значение textName. На этом снимке экрана оно отключено, но при запуске программы должно отображаться.

Свойству Name этой кнопки присвойте значение buttonStart. На рисунке она отключена, потому что игра уже началась.

Go Fish!

Your name

Ed

Start the game!

Game progress

Ed asks if anyone has a Ten
Joe has 0 Tens
Bob has 0 Tens
Ed must draw from the stock.
Joe asks if anyone has a King
Ed has 0 Kings
Bob has 0 Kings
Joe must draw from the stock.
Bob asks if anyone has a Four
Ed has 0 Fours
Joe has 0 Fours
Bob must draw from the stock.
Ed has 9 cards.
Joe has 7 cards.
Bob has 9 cards.
The stock has 7 cards left.

Books

Bob has a book of Aces
Ed has a book of Sixes
Ed has a book of Nines
Bob has a book of Eights
Joe has a book of Queens

Your hand

Two of Spades
Two of Diamonds
Two of Hearts
Seven of Spades
Seven of Diamonds
Seven of Hearts
Ten of Spades
Ten of Clubs
Ten of Diamonds

Ask for a card

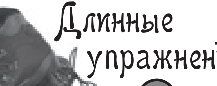
Карты на руках у игрока отображаются в поле ListBox с именем listHand.

Эти элементы TextBox называются textProgress и textBooks.

Присвойте свойству ReadOnly этих элементов TextBox значение True, что сделает их доступными только для чтения. Также присвойте значение True свойству Multiline.

Присвойте свойству Name этой кнопки значение buttonAsk, а свойству Enabled — значение False. В результате кнопку можно будет нажать только после начала игры.

Переверните страницу и продолжим!



Это код формы

Введите его в IDE.

```
public partial class Form1 : Form {
    public Form1() {
        InitializeComponent();
    }

    private Game game;

    private void buttonStart_Click(object sender, EventArgs e) {
        if (String.IsNullOrEmpty(textName.Text)){
            MessageBox.Show("Please enter your name", "Can't start the game yet");
            return;
        }
        game = new Game(textName.Text, new List<string> { "Joe", "Bob" }, textProgress);
        buttonStart.Enabled = false;
        textName.Enabled = false;
        buttonAsk.Enabled = true;
        UpdateForm();
    }

    private void UpdateForm() {
        listHand.Items.Clear();
        foreach (String cardName in game.GetPlayerCardNames())
            listHand.Items.Add(cardName);
        textBooks.Text = game.DescribeBooks();
        textProgress.Text += game.DescribePlayerHands();
        textProgress.SelectionStart = textProgress.Text.Length;
        textProgress.ScrollToCaret();


    private void buttonAsk_Click(object sender, EventArgs e) {
        textProgress.Text = "";
        if (listHand.SelectedIndex < 0) {
            MessageBox.Show("Please select a card");
            return;
        }
        if (game.PlayOneRound(listHand.SelectedIndex)) {
            textProgress.Text += "The winner is... " + game.GetWinnerName();
            textBooks.Text = game.DescribeBooks();
            buttonAsk.Enabled = false;
        } else
            UpdateForm();
    }
}
```

Это единственный класс, с которым взаимодействует форма. Именно он управляет всей игрой.

Свойство Enabled включает и отключает элементы управления формы.

При начале новой игры создается экземпляр класса Game, становится доступной кнопка Ask, пропадает доступ к кнопке Start Game, после чего форма перерисовывается.

Этот метод очищает текстовое поле от предыдущего списка и заполняет его набором карт для новой игры.

Свойство SelectionStart и метод ScrollToCaret() позволяют прокручивать текст, если он не помещается в текстовом поле.

Свойство SelectionStart определяет начало выделенного фрагмента, после этого метод ScrollToCaret() прокручивает содержимое текстового поля до места нахождения курсора.

Игрок выбирает карту и щелкает на кнопке Ask, чтобы узнать, есть ли данная карта у кого-нибудь из соперников. Класс Game играет с помощью метода PlayOneRound().

Этот код вам тоже понадобится

Вам потребуется уже написанный код для класса Card, перечислений Suits и Values, класса Deck и класса CardComparer_byValue. В классы нужно будет добавить несколько методов... суть которых вы должны понимать.

```
public Card Peek(int cardNumber) {
    return cards[cardNumber];
}
```

Метод Peek() позволяет взять карту из колоды.

```
public Card Deal() {
    return Deal(0);
}
```

Кто-то перегрузил метод Deal(), сделав его легче для восприятия. Если не передавать ему параметры, будет сдана верхняя карта в колоде.

```
public bool ContainsValue(Values value) {
    foreach (Card card in cards)
        if (card.Value == value)
            return true;
    return false;
}
```

Метод ContainsValue() ищет в колоде карты определенного старшинства и, находя их, возвращает значение true. Вы догадываетесь, как он будет использоваться в нашей игре?

```
public Deck PullOutValues(Values value) {
    Deck deckToReturn = new Deck(new Card[] { });
    for (int i = cards.Count - 1; i >= 0; i--)
        if (cards[i].Value == value)
            deckToReturn.Add(Deal(i));
    return deckToReturn;
}
```

Метод PullOutValues() позволяет получить наборы по четыре одинаковые карты. Он ищет карты, совпадающие по старшинству, извлекает их из колоды и возвращает новый вариант колоды, в который включена и взятка.

```
public bool HasBook(Values value) {
    int NumberOfCards = 0;
    foreach (Card card in cards)
        if (card.Value == value)
            NumberOfCards++;
    if (NumberOfCards == 4)
        return true;
    else
        return false;
}
```

Метод HasBook(), получив в качестве параметра карту, начинает искать взятки. Обнаружив четыре одинаковые карты, он возвращает значение True.

```
public void SortByValue() {
    cards.Sort(new CardComparer_byValue());
}
```

Метод SortByValue() сортирует колоду с помощью класса Comparer_byValue.

→ Переверните страницу и продолжим!

5 **Это СЛОЖНАЯ часть: создание класса Player**

Экземпляры класса `Player` существуют для всех игроков. Они создаются обработчиком событий кнопки `buttonStart`.

Внимательно читайте комментарии, там сказано, какие именно действия должны выполнять методы, для которых вы пишете код.

```
class Player
{
    private string name;
    public string Name { get { return name; } }
    private Random random;
    private Deck cards;
    private TextBox textBoxOnForm;

    public Player(String name, Random random, TextBox textBoxOnForm) {
       // Конструктор класса Player инициализирует четыре закрытых поля, а затем
       // добавляет элементу управления TextBox строку "Joe has just
       // joined the game", используя имя закрытого поля. Не забудьте поставить
       // знак переноса в конец каждой строки, добавляемой в TextBox.
    }

    public IEnumerable<Values> PullOutBooks() {  } // код на следующей странице

    public Values GetRandomValue() {
       // Этот метод получает случайное значение, но из числа карт колоды!
    }

    public Deck DoYouHaveAny(Values value) {
       // Соперник спрашивает о наличии у меня карты нужного достоинства
       // Используйте метод Deck.PullOutValues() для взятия карт. Добавьте в TextBox
       // строку "Joe has 3 sixes", используйте новый статический метод Card.Plural()
    }

    public void AskForACard(List<Player> players, int myIndex, Deck stock) {
       // Это перегруженная версия AskForACard() — выберите случайную карту с помощью
       // метода GetRandomValue() и спросите о ней методом AskForACard()
    }
```

Уникальный случай — последнюю карту противника берет другой игрок, а на момент вызова метода AskForACard() у него не осталось карт. Как следует поступить в подобной ситуации?

```
    public void AskForACard(List<Player> players, int myIndex, Deck stock, Values value) {
       // Спросите карту у соперников. Добавьте в TextBox текст: "Joe asks if anyone has
       // a Queen". В качестве параметра вам будет передана коллекция игроков
       // спросите (с помощью метода DoYouHaveAny()), есть ли у них карты
       // указанного достоинства. Переданные им карты добавьте в свой набор.
       // Следите за тем, сколько карт было добавлено. Если ни одной, вам нужно
       // взять карту из запаса (передается как параметр), в текстовое
       // поле нужно добавить строку TextBox: "Joe had to draw from the stock"
    }
    // Перечень свойств и коротких методов, которые уже были написаны
    public int CardCount { get { return cards.Count; } }
    public void TakeCard(Card card) { cards.Add(card); }
    public IEnumerable<string> GetCardNames() { return cards.GetCardNames(); }
    public Card Peek(int cardNumber) { return cards.Peek(cardNumber); }
    public void SortHand() { cards.SortByValue(); }
}
```

6 Добавим этот метод в класс Player.

Метод `PullOutBooks()` для класса `Player` циклически просматривает все 13 достоинств карт. Для каждого из них подсчитываются все карты в поле cards аналогичного достоинства. Если совпадают все четыре карты, значение добавляется в возвращаемую переменную books, а из карт игрока данная взятка изымается.

Вот о чем еще следует подумать.

```
public IEnumerable<Values> PullOutBooks() {
    List<Values> books = new List<Values>();
    for (int i = 1; i <= 13; i++) {
        Values value = (Values)i;
        int howMany = 0;
        for (int card = 0; card < cards.Count; card++)
            if (cards.Peek(card).Value == value)
                howMany++;
        if (howMany == 4) {
            books.Add(value);
            cards.PullOutValues(value);
        }
    }
    return books;
}
```

Здесь пригодится добавленный в класс Deck метод Peek(). Он позволяет увидеть карту в колоде по ее индексу, но в отличие от метода Deal() не удаляет карту.

Нужно построить ДВЕ перегруженные версии метода theAskForACard(). Первый использует противник, когда просим карту: метод просматривает его карты и определяет, какую карту следует попросить. Второй применяется, когда карту просит игрок. Оба метода просят у ВСЕХ остальных игроков любые карты указанного достоинства.

7 Добавим этот метод в класс Card.

Это статический метод, который берет значение и возвращает название карты во множественном числе (по правилам английского языка), то есть для 10 возвращается "Tens", а для 6 — "Sixes" (с "es" на конце). Для вызова статического метода следует указать имя класса — `Card.Plural()`, — и от него нельзя получить экземпляры.

```
public partial class Card {
    public static string Plural(Values value) {
        if (value == Values.Six)
            return "Sixes";
        else
            return value.ToString() + "s";
    }
}
```

Для добавления статического метода мы воспользовались разделяемым классом, чтобы вам было проще понять происходящее. Но необходимости в разделяемом классе не было, поэтому вы можете добавить метод непосредственно в класс Card.

После создания обработчика события "Ask for a card" с ним можно связать событие DoubleClick элемента ListBox. В результате вы сможете запросить карту **двойным щелчком** на ней.

Properties	
listHand System.Windows.Forms.ListBox	
DoubleClick	buttonAsk_Click
MouseCaptureChanged	buttonStart_Click
MouseClick	buttonAsk_Click
MouseDoubleClick	

Скоро закончим, переверните страницу!

Длинные упражнения

8 **Создание класса Game**

Форма сохраняет экземпляр Game, управляющий игрой. Посмотрите, как он используется.

В классах Player и Game имеются ссылки на элемент формы TextBox, в котором появляются сообщения о ходе игры. Убедитесь, что в верхней части файлов есть строка «using System.Windows.Forms;».

```
class Game {
    private List<Player> players;
    private Dictionary<Values, Player> books;
    private Deck stock;
    private TextBox textBoxOnForm;
    public Game(string playerName, IEnumerable<string> opponentNames, TextBox textBoxOnForm) {
        Random random = new Random();
        this.textBoxOnForm = textBoxOnForm;
        players = new List<Player>();
        players.Add(new Player(playerName, random, textBoxOnForm));
        foreach (string player in opponentNames)
            players.Add(new Player(player, random, textBoxOnForm));
        books = new Dictionary<Values, Player>();
        stock = new Deck();
        Deal();
        players[0].SortHand();
    }
    private void Deal() {
       // Именно здесь начинается игра.
       // Тасуется колода, раздается по пять карт каждому игроку, затем с помощью
       // цикла foreach вызывается метод PullOutBooks() для каждого игрока.
    }
    public bool PlayOneRound(int selectedPlayerCard) {
       // Сыграйте один раз. Параметром является выбранная игроком карта из имеющихся на руках
       // Вызовите метод AskForACard() для каждого из игроков, начиная с человека
       // с нулевым индексом. Затем вызовите метод PullOutBooks() —
       // если он вернет значение true, значит, у игрока кончились
       // карты. Закончив со всеми игроками, отсортируйте карты
       // человека (чтобы список в форме выглядел красиво). Проверьте, не закончились
       // ли карты в запасе. В случае положительного результата очистите поле TextBox
       // и выведите фразу "The stock is out of cards. Game over!".
    }
    public bool PullOutBooks(Player player) {
       // Игроки выкладывают взятки. Метод возвращает значение true, если карты
       // у игрока закончились. Каждая взятка добавляется в словарь Books.
    }
    public string DescribeBooks() {
       // Этот метод возвращает длинную строку с описанием взяток каждого игрока,
       // взяв за основу содержание словаря Books: "Joe has a book of sixes.
       // (перенос строки) Ed has a book of Aces."
    }
```

Интерфейс IEnumerable<T> делает классы более <u>гибкими</u>. Об этом следует помнить при дальнейшем редактировании кода. В данный момент для получения экземпляра класса Game можно написать string[].List<string>.

Это полезно и для инкапсуляции. Если использовать IEnumerable<T> вместо List<T>, вы не сможете случайно отредактировать код.

Для написания метода GetWinnerName() нужно создать новый словарь Dictionary<string, int> с именем winners и поместить его на самый верх. По имени игрока словарь будет предоставлять информацию о количестве сделанных им за игру взяток. Сначала с помощью цикла foreach нужно будет записать в словарь все сделанные взятки. Затем второй цикл foreach должен найти их максимальное значение. Нужно помнить, что возможна и ничья — ситуация, когда максимальное количество взяток присутствует у более чем одного игрока. Так что вам потребуется еще один цикл foreach, ищущий всех игроков в словаре winners, которые имеют максимальное количество взяток. Затем этот цикл создает строку с информацией о том, кто же выиграл.

```
public string GetWinnerName() {
       // Этот метод вызывается в конце игры. Он использует собственный словарь
       // (Dictionary<string, int> winners) для отслеживания количества взяток
       // каждого игрока. Сначала цикл foreach (Values value in books.Keys)
       // заполняет словарь winners информацией о взятках. Затем
       // словарь просматривается на предмет поиска максимального
       // количества взяток. Напоследок словарь просматривается еще один раз, чтобы
       // сформировать список победителей в виде строки ("Joe and Ed"). Если победитель
       // один, возвращается строка "Ed with 3 books". В противном
       // случае возвращается строка "A tie between Joe and Bob with 2 books."
}

// Пара коротких методов, которые были написаны раньше:

public IEnumerable<string> GetPlayerCardNames() {
    return players[0].GetCardNames();
}

public string DescribePlayerHands() {
    string description = "";
    for (int i = 0; i < players.Count; i++) {
        description += players[i].Name + " has " + players[i].CardCount;
        if (players[i].CardCount == 1)
            description += " card." + Environment.NewLine;
        else
            description += " cards." + Environment.NewLine;
    }
    description += "The stock has " + stock.Count + " cards left.";
    return description;
}
```

Введите в окно Watch (int)'\r', чтобы присвоить символ \r числу. В результате вы получите 13, в то время как '\n' превращается в 10. Каждый символ превращается в символ Юникод. Дополнительную информацию по этой теме вы получите в следующей главе.

В этой книге для переноса строк в окнах диалога вы пользовались символом \n. В .NET для этой цели существует удобная константа Environment.NewLine. Она содержит символы \r\n. Если посмотреть на текст, отоформатированный в Windows, в конце каждой строки вы найдете символы '\r' и '\n'. Другие операционные системы (например, Unix) используют только '\n'. Метод MessageBox.Show() автоматически преобразовывает '\n' в разрыв строки, но константа Environment.NewLine облегчает восприятие кода. Кстати, она добавляется в конец каждой строки и при работе с методом Console.WriteLine().

Решение длинных упражнений

Вот как полностью выглядят методы для класса `Game`.

```
private void Deal() {
    stock.Shuffle();
    for (int i = 0; i < 5; i++)
        foreach (Player player in players)
            player.TakeCard(stock.Deal());
    foreach (Player player in players)
        PullOutBooks(player);
}
```

Метод Deal() вызывается в начале игры, тасует колоду и раздает каждому игроку по пять карт. Затем он собирает взятки, если таковые появляются.

```
public bool PlayOneRound(int selectedPlayerCard) {
    Values cardToAskFor = players[0].Peek(selectedPlayerCard).Value;
    for (int i = 0; i < players.Count; i++) {
        if (i == 0)
            players[0].AskForACard(players, 0, stock, cardToAskFor);
        else
            players[i].AskForACard(players, i, stock);
        if (PullOutBooks(players[i])) {
            textBoxOnForm.Text += players[i].Name
                        + " drew a new hand" + Environment.NewLine;
            int card = 1;
            while (card <= 5 && stock.Count > 0) {
                players[i].TakeCard(stock.Deal());
                card++;
            }
        }
        players[0].SortHand();
        if (stock.Count == 0) {
            textBoxOnForm.Text =
                "The stock is out of cards. Game over!" + Environment.NewLine;
            return true;
        }
    }
    return false;
}
```

Если после того как игрок спросил карту, образовалась взятка, она у него забирается. Если взяток нет, он берет новые пять карт из запаса.

После щелчка на кнопке Ask for a card игра вызывает метод AskForACard() с выбранной картой в качестве параметра. Затем метод AskForACard() вызывается для каждого из игроков.

После каждого тура карты игрока сортируются, чтобы упорядочить отображаемый список. Затем проверяется, не закончилась ли игра. В случае ее окончания метод PlayOneRound() возвращает значение true.

```
public bool PullOutBooks(Player player)
{
    IEnumerable<Values> booksPulled = player.PullOutBooks();
    foreach (Values value in booksPulled)
        books.Add(value, player);
    if (player.CardCount == 0)
        return true;
    return false;
}
```

Метод PullOutBooks() проверяет карты игрока на наличие взяток. Обнаруженная взятка добавляется в словарь. Если карт не осталось, возвращается значение true.

Так как в форме должен отображаться список взяток, воспользуемся методом DescribeTheBooks(), чтобы превратить записи словаря в строки.

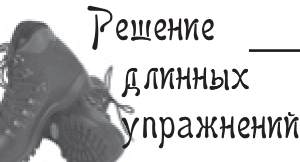
```
public string DescribeBooks() {
    string whoHasWhichBooks = "";
    foreach (Values value in books.Keys)
        whoHasWhichBooks += books[value].Name + " has a book of "
            + Card.Plural(value) + Environment.NewLine;
    return whoHasWhichBooks;
}

public string GetWinnerName() {
    Dictionary<string, int> winners = new Dictionary<string, int>();
    foreach (Values value in books.Keys) {
        string name = books[value].Name;
        if (winners.ContainsKey(name))
            winners[name]++;
        else
            winners.Add(name, 1);
    }
    int mostBooks = 0;
    foreach (string name in winners.Keys)
        if (winners[name] > mostBooks)
            mostBooks = winners[name];
    bool tie = false;
    string winnerList = "";
    foreach (string name in winners.Keys)
        if (winners[name] == mostBooks)
        {
            if (!String.IsNullOrEmpty(winnerList))
            {
                winnerList += " and ";
                tie = true;
            }
            winnerList += name;
        }
    winnerList += " with " + mostBooks + " books";
    if (tie)
        return "A tie between " + winnerList;
    else
        return winnerList;
}
```

После взятия последней карты требуется определить победителя. Именно этим занимается метод GetWinnerName() на основе информации из словаря winners. Ключом является имя игрока, а значением — количество взяток.

Затем определяется максимальное количество взяток. Оно помещается в переменную mostBooks.

Теперь, когда мы знаем игрока с максимальным количеством взяток, можно вывести строку с именем победителя (или победителей).

Переверните страницу и продолжим!



Так выглядят полностью написанные методы класса `Player`.

```
public Player(String name, Random random, TextBox textBoxOnForm) {
    this.name = name;
    this.random = random;
    this.textBoxOnForm = textBoxOnForm;
    this.cards = new Deck( new Card[] {} );
    textBoxOnForm.Text += name +
                " has just joined the game" + Environment.NewLine;
}
```

Это конструктор класса Player. Он задает значения приватных полей и добавляет в текстовое поле строку с информацией о присоединившемся игроке.

```
public Values GetRandomValue() {
    Card randomCard = cards.Peek(random.Next(cards.Count));
    return randomCard.Value;
}
```

Метод GetRandomValue() с помощью метода Peek() выбирает случайную карту среди имеющихся у игрока.

```
public Deck DoYouHaveAny(Values value) {
    Deck cardsIHave = cards.PullOutValues(value);
    textBoxOnForm.Text += Name + " has " + cardsIHave.Count + " "
           + Card.Plural(value) + Environment.NewLine;
    return cardsIHave;
}
```

Метод DoYouHaveAny() использует метод PullOutValues() для извлечения карт, которые соответствуют заданным параметрам.

```
public void AskForACard(List<Player> players, int myIndex, Deck stock) {
    if (stock.Count > 0) {
        if (cards.Count == 0)
            cards.Add(stock.Deal());
    Values randomValue = GetRandomValue();
    AskForACard(players, myIndex, stock, randomValue);
}
```

Противник отдал последнюю карту, и метод GetRandomValue() пытается вызвать Deal() для пустой колоды. Операторы if пытаются это предотвратить.

В программе два перегруженных метода AskForACard(). Первый используется соперниками — выбирает случайную карту из имеющихся и вызывает второй метод AskForACard().

```
public void AskForACard(List<Player> players, int myIndex,
      Deck stock, Values value) {
    textBoxOnForm.Text += Name + " asks if anyone has a "
                              + value  + Environment.NewLine;
    int totalCardsGiven = 0;
    for (int i = 0; i < players.Count; i++) {
        if (i != myIndex) {
            Player player = players[i];
            Deck CardsGiven = player.DoYouHaveAny(value);
            totalCardsGiven += CardsGiven.Count;
            while (CardsGiven.Count > 0)
                cards.Add(CardsGiven.Deal());
        }
    }
    if (totalCardsGiven == 0) && stock.Count > 0) {
        textBoxOnForm.Text += Name +
              " must draw from the stock." + Environment.NewLine;
        cards.Add(stock.Deal());
    }
}
```

Метод AskForACard() проверяет всех игроков (за исключением спрашивающего), вызывает их метод DoYouHaveAny() и добавляет найденные подходящие карты.

При отсутствии у соперников подходящих карт игрок берет карту из запаса при помощи метода Deal().

Дополнительное мини-упражнение: Улучшите инкапсуляцию класса Player, заменив в этих методах List<Player> на IEnumerable<Player>, не повлияв на работу программы.

Дополнительные типы коллекций...

Объекты List и Dictionary относятся к **встроенным обобщенным коллекциям** и являются частью .NET Framework. Они очень гибки, доступ к их данным осуществляется в произвольном порядке. Но иногда работу программы с данными требуется ограничить, так как *явление*, которое вы моделируете, должно работать как в реальности. В ситуациях используются **очередь (Queue)** или **стек (Stack)**. Они относятся к обобщенным коллекциям, например List<T>, но гарантируют обработку данных в определенном порядке.

Здесь перечислены не все типы коллекций, а только те, с которыми вам, вероятнее всего, придется работать.

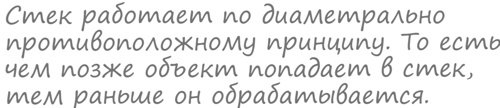
Используйте очередь, когда первый сохраненный объект будет обрабатываться первым, как в случае:

- автомобилей, движущихся по улице с односторонним движением;
- людей, стоящих в очереди;
- клиентов, ждущих обслуживания по телефону;
- других ситуаций, когда первым обслуживается тот, кто первым пришел.

Очередь работает по принципу «раньше вошел, раньше вышел». То есть объект, первым помещенный в очередь, первым и обрабатывается.

Используйте стек, когда первым вы собираетесь обрабатывать последний из сохраненных объектов, как в случае:

- мебели, загруженной в кузов грузовика;
- стопки книг, из которых вы хотите сначала прочитать верхнюю;
- людей, выходящих из самолета;
- пирамиды из людей. Первым спрыгивать вниз должен тот, кто находится на самом верху. Только представьте, что произойдет, если начать уходить из пирамиды снизу!



Обобщенные коллекции являются важной частью NET Framework

Они настолько полезны, что IDE автоматически добавляет оператор в верхнюю часть каждого класса, который создается в рамках проекта:

```
using System.Collections.Generic;
```

Обобщенные коллекции встречаются почти во всех проектах, ведь вам нужно где-то хранить данные. Группы одинаковых объектов в реальном мире практически всегда можно объединить в категории, которые в той или иной степени напоминают какую-то из коллекций.

Цикл foreach позволяет осуществлять перечисления в очереди и стеке, так как они реализуют интерфейс **IEnumerable**!

Очередь подобна списку. Новые объекты помещаются в его конец, а читается он с начала. Стек же дает доступ только к последнему помещенному в него объекту.

Очередь: первый вошел, первый вышел

Очередь отличается от списка тем, что вы не можете добавлять и удалять элементы с произвольным индексом. Вы **добавляете объект в очередь (enqueue)** и **удаляете из нее (dequeue)**. В последнем случае оставшиеся объекты сдвигаются на один элемент.

```
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");
string takeALook = myQueue.Peek();    ①
string getFirst = myQueue.Dequeue();  ②
string getNext = myQueue.Dequeue();   ③
int howMany = myQueue.Count;          ④
myQueue.Clear();
MessageBox.Show("Peek() returned: " + takeALook + "\n"
    + "The first Dequeue() returned: " + getFirst + "\n"
    + "The second Dequeue() returned: " + getNext + "\n"
    + "Count before Clear() was " + howMany + "\n"
    + "Count after Clear() is now " + myQueue.Count);
                                                ⑤
```

Создаем очередь строк.

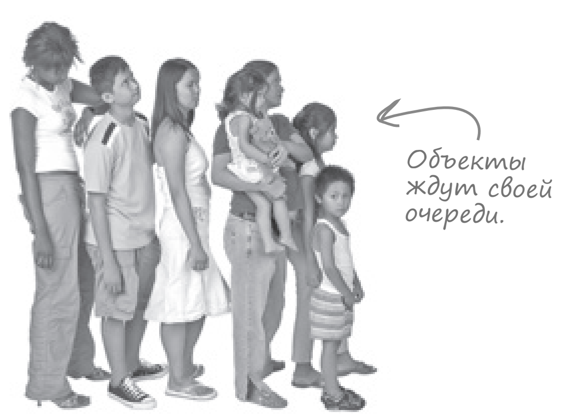
В очередь добавляются четыре элемента.

Метод Peek() позволяет выделить первый элемент в очереди, не удаляя его.

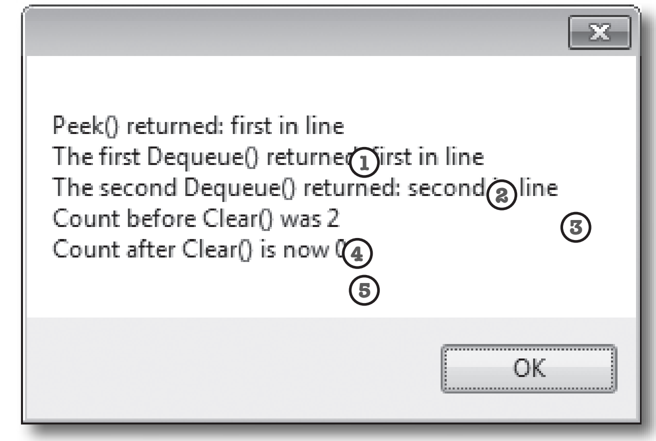
Первый метод Dequeue() убирает из очереди первый элемент. После чего второй элемент сдвигается на первое место, и следующий вызов метода Dequeue() удаляет уже его.

*Метод Clear() **удаляет** из очереди все объекты.*

Свойство Count возвращает число элементов в очереди.



Объекты ждут своей очереди.



Стек: последним вошел, первым вышел

Стек имеет всего одно, но значительное отличие от очереди. Вы **помещаете** в него каждый элемент и можете в любой момент взять последний элемент стека. Стек напоминает бусы. Сначала вы снимаете бусину, которая была нанизана последней, потом следующую и т. д. Нельзя взять четвертую с конца бусину, не сняв три предыдущие.

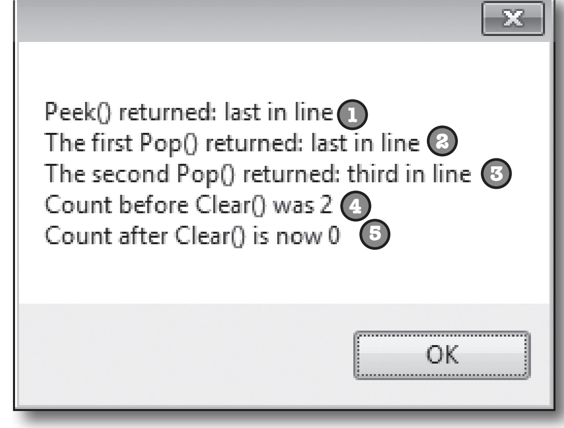
Помещаемый в стек элемент сдвигает все прочие элементы на единицу вниз и оказывается на самом верху.

Создание стека ничем не отличается от создания любой другой обобщенной коллекции.

```
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");
(1) string takeALook = myStack.Peek();
(2) string getFirst = myStack.Pop();
(3) string getNext = myStack.Pop();
(4) int howMany = myStack.Count;
myStack.Clear();
MessageBox.Show("Peek() returned: " + takeALook + "\n"
    + "The first Pop() returned: " + getFirst + "\n"
    + "The second Pop() returned: " + getNext + "\n"
    + "Count before Clear() was " + howMany + "\n"
    + "Count after Clear() is now " + myStack.Count);
                                               (5)
```

Метод Pop() удаляет из стека последний добавленный туда элемент.

Вместо символов \n можно воспользоваться константой Environment.NewLine, но мы предпочли сократить код.



Последний объект, помещенный в стек, становится первым объектом, который будет оттуда взят.

Бессмыслица какая-то... А что я могу сделать со стеком и очередью такого, чего не могу сделать с объектами List? Они всего лишь позволяют сделать код на пару строк короче при том что я не имею доступа к их средним элементам. Ну и зачем такие ограниченные объекты могут мне понадобиться?

Не волнуйтесь, вам не придется ни в чем себя ограничивать.

Скопировать объект `Queue` в объект `List` очень легко. Так же легко, как скопировать объект `List` в объект `Queue`, а объект `Queue` в объект `Stack`... более того, вы можете создать объекты `List`, `Queue` и `Stack` из любого другого объекта, реализующего интерфейс **`IEnumerable`**. Достаточно воспользоваться перегруженным конструктором, который позволяет передавать копируемую коллекцию в качестве параметра. То есть вы можете легко представить свои данные в виде коллекции, которая максимально соответствует вашим нуждам. (Но не забывайте, что при копировании вы создаете новый объект, который будет занимать место в памяти.)

Заполним стек четырьмя строками.

```
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");

Queue<string> myQueue = new Queue<string>(myStack);
List<string> myList = new List<string>(myQueue);
Stack<string> anotherStack = new Stack<string>(myList);
MessageBox.Show("myQueue has " + myQueue.Count + " items\n"
    + "myList has " + myList.Count + " items\n"
    + "anotherStack has " + anotherStack.Count + " items\n");
```

Стек можно легко превратить в очередь, затем трансформировать ее в объект list и вернуть в состояние стека.

Все четыре элемента были скопированы в новые коллекции.

```
myQueue has 4 items
myList has 4 items
anotherStack has 4 items
OK
```

...для доступа ко всем членам очереди или стека достаточно воспользоваться циклом foreach!

Упражнение

Напишите программу, которая помогает хозяину кафе кормить лесорубов лепешками. Начните с класса `Lumberjack` (Лесоруб). Сконструируйте форму и добавьте к кнопкам обработчики событий.

1 Добавьте в класс **Lumberjack** метод чтения для свойства FlapjackCount и методы TakeFlapjacks и EatFlapjacks.

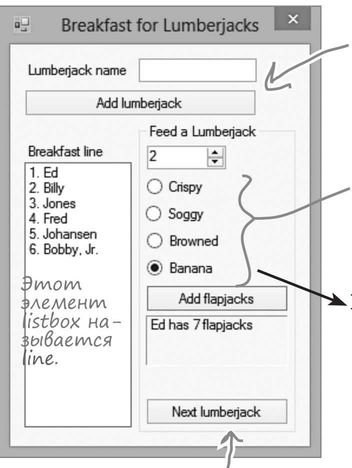
```
class Lumberjack {
    private string name;
    public string Name { get { return name; } }
    private Stack<Flapjack> meal;
    public Lumberjack(string name) {
        this.name = name;
        meal = new Stack<Flapjack>();
    }
    public int FlapjackCount { get { // возвращает число } }
    public void TakeFlapjacks(Flapjack Food, int HowMany) {
       // Добавляет в стек Meal указанное количество лепешек
    }
    public void EatFlapjacks() {
       // Выведите эти сведения на консоль
    }
}
```

```
enum Flapjack {
    Crispy,
    Soggy,
    Browned,
    Banana
}
```

```
Output
Ed's eating flapjacks
Ed ate a banana flapjack
Ed ate a banana flapjack
Ed ate a crispy flapjack
Ed ate a soggy flapjack
Ed ate a soggy flapjack
Ed ate a soggy flapjack
Ed ate a browned flapjack
```

Заметили, что перечисление Flapjack использует строчные буквы (Soggy), но на выходе вы получаете прописные (soggy)? Подсказываем, как исправить ситуацию. Метод ToString() возвращает строку, одним из открытых членов которой является метод ToLower(), возвращающий только прописные буквы.

2 Форма позволяет ввести имя лесоруба в текстовое поле, чтобы поместить его в очередь. Выдав первому в очереди лесорубу лепешки, вы отправляете его есть, щелкнув на кнопке Next lumberjack. Мы написали обработчик событий кнопки Add flapjacks. Отслеживайте состояние лесорубов при помощи очереди **breakfastLine**.



Щелчок на кнопке Add Lumberjack добавляет имя лесоруба из текстового поля name в очередь breakfastLine.

Перетаскивая элементы RadioButton внутрь элемента GroupBox, вы автоматически соединяете их и оставляете возможность выбрать только один переключатель за один раз. Узнать имена переключателей можно в методе addFlapjacks_Click.

```
private void addFlapjacks_Click(...) {
    if (breakfastLine.Count == 0) return;
    Flapjack food;
    if (crispy.Checked == true)
        food = Flapjack.Crispy;
    else if (soggy.Checked == true)
        food = Flapjack.Soggy;
    else if (browned.Checked == true)
        food = Flapjack.Browned;
    else
        food = Flapjack.Banana;

    Lumberjack currentLumberjack = breakfastLine.Peek();
    currentLumberjack.TakeFlapjacks(food,
                         (int)howMany.Value);
    RedrawList();
```

Обратите внимание на особый синтаксис: else if.

Метод Peek() возвращает ссылку на первого лесоруба в очереди.

Элемент управления NumericUpDown называется howMany, а label — nextInLine.

Эта кнопка убирает из очереди следующего лесоруба, вызывает его метод EatFlapjacks() и перерисовывает содержимое текстового поля.

Метод RedrawList() выводит в текстовое поле содержимое очереди. Его вызывают все кнопки. ***Подсказка***: *он использует цикл foreach.*

Эта программа выводит строки на консоль, поэтому для просмотра результатов нужно открыть окно Output.

Упражнение Решение

```
private Queue<Lumberjack> breakfastLine = new Queue<Lumberjack>();
private void addLumberjack_Click(object sender, EventArgs e) {
    if (String.IsNullOrEmpty(name.Text)) return;
    breakfastLine.Enqueue(new Lumberjack(name.Text));
    name.Text = "";
    RedrawList();
}
private void RedrawList() {
    int number = 1;
    line.Items.Clear();
    foreach (Lumberjack lumberjack in breakfastLine) {
        line.Items.Add(number + ". " + lumberjack.Name);
        number++;
    }
    if (breakfastLine.Count == 0) {
        groupBox1.Enabled = false;
        nextInLine.Text = "";
    } else {
        groupBox1.Enabled = true;
        Lumberjack currentLumberjack = breakfastLine.Peek();
        nextInLine.Text = currentLumberjack.Name + " has "
            + currentLumberjack.FlapjackCount + " flapjacks";
    }
}
private void nextLumberjack_Click(object sender, EventArgs e) {
    if (breakfastLine.Count == 0) return;
    Lumberjack nextLumberjack = breakfastLine.Dequeue();
    nextLumberjack.EatFlapjacks();
    nextInLine.Text = "";
    RedrawList();
}
class Lumberjack {
    private string name;
    public string Name { get { return name; } }
    private Stack<Flapjack> meal;

    public Lumberjack(string name) {
        this.name = name;
        meal = new Stack<Flapjack>();
    }
    public int FlapjackCount { get { return meal.Count; } }

    public void TakeFlapjacks(Flapjack food, int howMany) {
        for (int i = 0; i < howMany; i++) {
            meal.Push(food);
        }
    }

    public void EatFlapjacks() {
        Console.WriteLine(name + "'s eating flapjacks");
        while (meal.Count > 0) {
            Console.WriteLine(name + " ate a "
              + meal.Pop().ToString().ToLower() + " flapjack");
        }
    }
}
```

Этот элемент ListBox называется line (очередь), а элемент Label между двумя кнопками носит имя nextInLine (следующий в очереди).

Метод RedrawList() при помощи цикла foreach убирает лесорубов из очереди и помещает сведения о них в список.

Этот оператор if обновляет метку в соответствии с информацией о следующем лесорубе в очереди.

Метод TakeFlapjacks обновляет содержимое стека Meal (Прием пищи).

Именно здесь перечисление Flapjack преобразуется в строчные буквы. Постарайтесь понять, как это происходит.

Метод EatFlapjacks использует цикл while для вывода информации о трапезе каждого лесоруба.

Метод meal.Pop() возвращает перечисление, метод ToString() которого возвращает строку, метод ToLower() которой возвращает другую строку.

Прибереги последний байт для меня

Иногда настойчивость окупается.

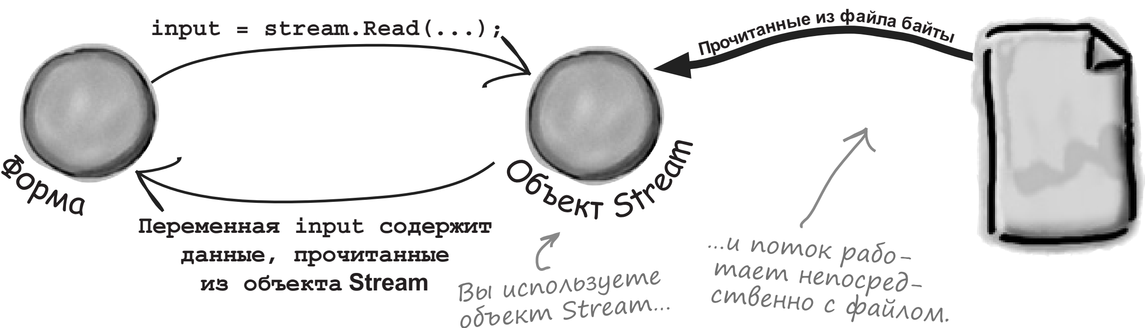
Пока что все ваши программы жили недолго. Они запускались, некоторое время работали и закрывались. Но этого недостаточно, когда имеешь дело с важной информацией. Вы должны уметь **сохранять свою работу**. В этой главе мы поговорим о том, как **записать данные в файл**, а затем о том, как **прочитать эту информацию**. Вы познакомитесь с **потоковыми классами** .NET и узнаете о тайнах **шестнадцатеричной** и **двоичной** систем счисления.

Для чтения и записи данных в .NET используются потоки

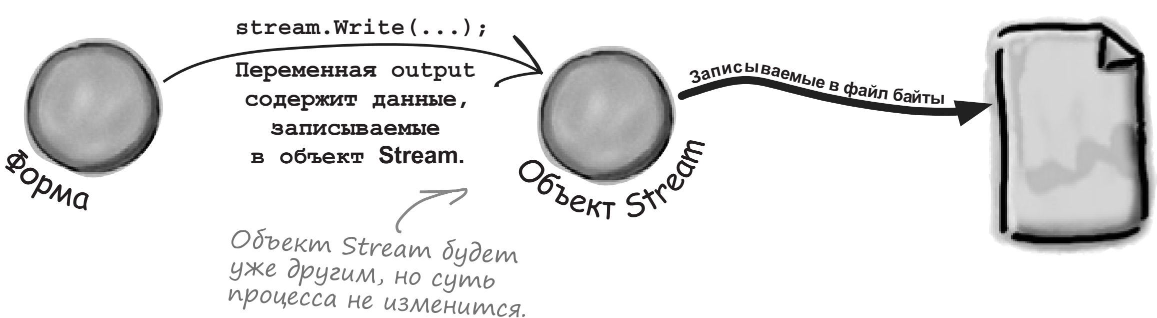
Поток (stream) — это способ, которым .NET Framework обменивается данными с программой. Каждый раз, когда программа читает или записывает файл, соединяется с другим компьютером сети или выполняет какие-либо действия, связанные **с передачей байтов** из одного места в другое, речь идет о потоке данных.

Для чтения данных из файла и записи их в файл используется объект Stream.

Представьте простую программу — форму с обработчиком событий, читающим данные из файла. Эта операция осуществляется с помощью объекта Stream.

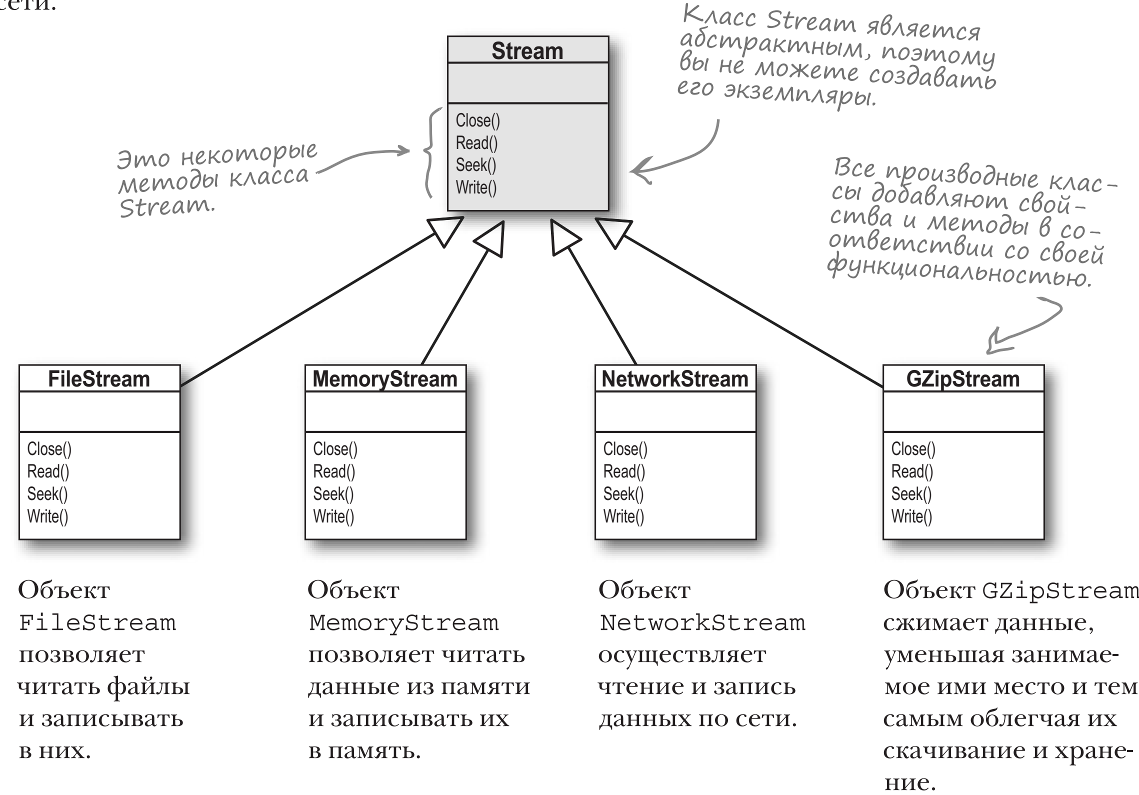


Для записи данных в файл используется другой объект Stream.



Различные потоки для различных данных

Каждый поток является производным от абстрактного класса **Stream**, и существует множество встроенных классов stream, предназначенных для различных операций. В данной главе будут рассмотрены чтение и запись в обычные файлы, но все эти сведения легко применить к сжатым и зашифрованным файлам и даже к передаче данных по сети.



Объект `FileStream` позволяет читать файлы и записывать в них.

Объект `MemoryStream` позволяет читать данные из памяти и записывать их в память.

Объект `NetworkStream` осуществляет чтение и запись данных по сети.

Объект `GZipStream` сжимает данные, уменьшая занимаемое ими место и тем самым облегчая их скачивание и хранение.

Вы можете:

1. **Записывать в поток.**
   Эта операция осуществляется при помощи метода `Write()`.

2. **Читать из потока.**
   Метод `Read()` дает доступ к чтению данных из файла, сети или из памяти, используя поток. Вы можете читать данные из очень больших файлов — даже настолько огромных, что они не помещаются в памяти.

3. **Менять свое положение в потоке.**
   Большинство потоков поддерживают метод `Seek()`, устанавливающий вашу позицию в потоке.

Потоки позволяют читать и записывать данные. Выбор потока осуществляется в соответствии с типом данных.

Объект FileStream

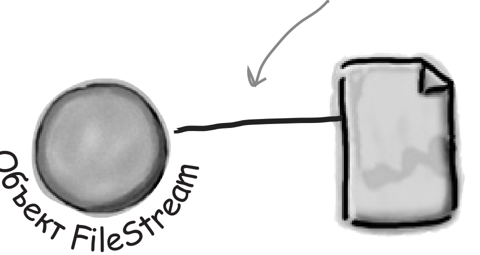
Вот что происходит при записи в файл нескольких строк текста:

В верхней части любой программы, работающей с потоками, должна присутствовать строчка `using System.IO;`

1. Создается объект `FileStream`, получающий команду записывать в файл.

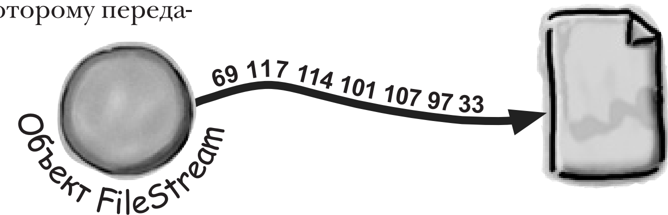
Присоединить объект FileStream можно только к одному объекту за раз.

2. Объект `FileStream` присоединяется к файлу.



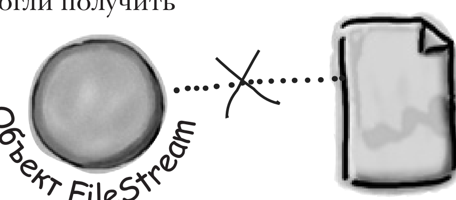
3. Строки, которые требуется записать, следует преобразовать в массив типа `byte`.

4. Вызывается метод `Write()`, которому передается массив типа `byte`.



5. Поток закрывается, чтобы другие программы могли получить доступ к файлу.

Забыв перекрыть поток, вы блокируете доступ к файлу всем прочим программам.



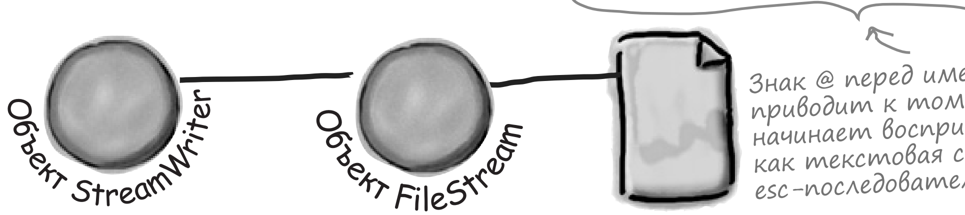
Трехшаговая процедура записи текста в файл

В C# существует удобный класс **StreamWriter**, выполняющий описанный в предыдущем разделе алгоритм за один шаг. Вам нужно только создать объект StreamWriter и присвоить ему имя. Он **автоматически** создаст объект FileStream и откроет файл. После чего остается только воспользоваться методами Write() и WriteLine().

Класс StreamWriter автоматически создает объект FileStream и управляет им.

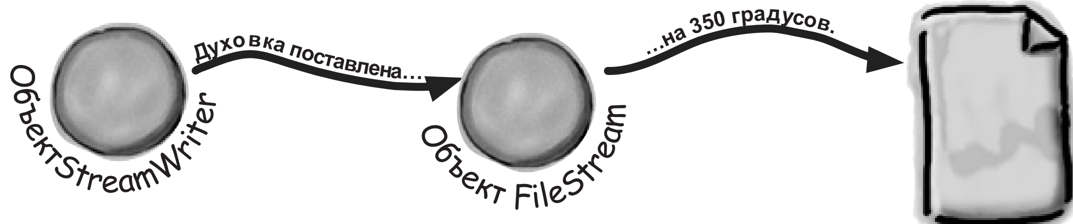
1. **Для открытия и создания файлов используйте конструктор класса StreamWriter**
Имя файла можно передать конструктору StreamWriter(). При этом файл открывается автоматически. В классе StreamWriter существует также перегруженный конструктор, работающий с логическими параметрами: true соответствует добавлению текста в конец существующего файла, а false — удалению существующего файла и созданию нового с аналогичным именем.

```
StreamWriter writer = new StreamWriter(@"C:\newfiles\toaster oven.txt", true);
```



2. **Для записи в файл используйте методы Write() и WriteLine()**
Метод Write() записывает текст, а метод WriteLine() добавляет к тексту знак переноса строки. Символы {0}, {1}, {2} и т. д. в записываемой строке позволяют включить в текст параметры: {0} заменяется первым параметром после записанной строки, {1} — вторым и т. д.

```
writer.WriteLine("The {0} is set to {1} degrees.", appliance, temp);
```



3. **Для освобождения от файла используйте метод Close()**
Оставив поток открытым и соединенным с файлом, вы заблокируете файл для всех остальных программ. Поэтому не забывайте писать:

```
writer.Close();
```

Дьявольский план Жулика

Жители Объектвиля долгое время боялись Жулика. И вот он решил воспользоваться объектом StreamWriter для реализации зловещего плана. Посмотрим на него поближе. Создайте консольное приложение и добавьте этот код в метод Main():

Записывать файл в корневой каталог — не очень хорошая идея, и операционная система может не позволить это сделать. Поэтому укажите любой другой адрес по вашему выбору.

Знак @ перед маршрутом доступа объясняет объекту StreamWriter, что символ \ не является началом esc-последовательности.

Эта строчка создает объект StreamWriter и указывает его адрес.

```
StreamWriter sw = new StreamWriter(@"C:\secret_plan.txt");
sw.WriteLine("How I'll defeat Captain Amazing");
sw.WriteLine("Another genius secret plan by The Swindler");
sw.Write("I'll create an army of clones and ");
sw.WriteLine("unleash them upon the citizens of
Objectville.");
string location = "the mall";
for (int number = 0; number <= 6; number++){
    sw.WriteLine("Clone #{0} attacks {1}", number, location);
    if (location == "the mall") { location = "downtown"; }
    else { location = "the mall"; }
}
sw.Close();
```

Метод WriteLine() переносит строки после записи, в то время как метод Write() отправляет обычный текст.

Вы понимаете, что происходит с переменной location?

Скобки {} внутри текста передают переменные в записываемую строку. {0} замещается первым параметром после строки, {1} — вторым и т. д.

Метод Close() разрывает связь с файлом и прочими ресурсами, с которыми работает объект StreamWriter. Если не закрыть поток, запись текста осуществлена не будет.

Это важно!

Объект StreamWriter находится в пространстве имен System.IO, поэтому в верхней части программы должна быть строка using System.IO;

Вот такой результат получается на выходе.

secret_plan - Notepad

File Edit Format View Help

```
How I'll defeat Captain Amazing
Another genius secret plan by The Swindler
I'll create an army of clones and unleash them upon the citizens of Objectville.
Clone #0 attacks the mall
Clone #1 attacks downtown
Clone #2 attacks the mall
Clone #3 attacks downtown
Clone #4 attacks the mall
Clone #5 attacks downtown
Clone #6 attacks the mall
```

Магниты для объекта StreamWriter

У вас есть код для кнопки `button1_Click()`. Расположите магниты таким образом, чтобы создать класс Flobbo. При этом обработчик событий должен привести к результату, показанному внизу страницы. Удачи!

```
static void Main(string[] args) {
    Flobbo f = new Flobbo("blue yellow");
    StreamWriter sw = f.Snobbo();
    f.Blobbo(f.Blobbo(f.Blobbo(sw), sw), sw);
}
```

```
sw.WriteLine(Zap);
Zap = "red orange";
return true;
```

```
}
```

```
sw.WriteLine(Zap);
sw.Close();
return false;
```

```
public bool Blobbo
    (bool Already, StreamWriter sw) {
```

```
public bool Blobbo(StreamWriter sw) {
```

```
sw.WriteLine(Zap);
Zap = "green purple";
return false;
```

```
}
```

```
return new
  StreamWriter("macaw.txt");
```

```
}
```

```
}
```

```
}
```

```
private string Zap;

public Flobbo(string Zap) {
    this.Zap = Zap;
}
```

```
class Flobbo {
```

```
if (Already) {
```

```
} else {
```

```
public StreamWriter Snobbo() {
```

Предположим, что у всех кодов в начале есть `using System.IO;`

Результат:

```
macaw - Notepad
File Edit Format View Help
blue yellow
green purple
red orange
```

Решение задачи с магнитами

Вам требовалось сконструировать класс Flobbo, работающий определенным образом.

```
static void Main(string[] args) {
    Flobbo f = new Flobbo("blue yellow");
    StreamWriter sw = f.Snobbo();
    f.Blobbo(f.Blobbo(f.Blobbo(sw), sw), sw);
}
```

Если ввести это в IDE, файл *macaw.txt* будет записан в папку *bin\Debug*, вложенную в папку вашего проекта, так как именно отсюда запускается исполняемый файл.

```
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;
        }
    }
}
```

Еще раз напоминаем, что для ребусов мы намеренно используем произвольные имена переменных и методов, потому что значимые имена делают задачу слишком легкой! Но пожалуйста, не нужно брать с нас пример, когда вы пишете программы.

Предположим, что у всех кодов в начале есть `using System.IO;`

Метод Blobbo() перегружен, он имеет два объявления с двумя различными параметрами.

После завершения работы не забудьте закрыть файлы.

Если запустить этот код в IDE, файл macaw.txt появится в папке bin\Debug.

Результат:

macaw - Notepad

File Edit Format View Help

```
blue yellow
green purple
red orange
```

Чтение и запись при помощи двух объектов

Секретный план Жулика мы прочитаем при помощи потока `StreamReader`. Именно его конструктору передается имя файла, который требуется прочитать. Метод `ReadLine()` возвращает строку с текстом из файла. Для прочтения всех строк используйте цикл, который работает, пока поле `EndOfStream` не получит значение true, то есть пока не закончатся строки:

В данном случае мы слишком вольно используем слово «поток». Класс `StreamReader` (наследующий от `TextReader`) читает символы из потока, но это не поток. Поток создается, когда вы передаете имя файла в его конструктор, и закрывается с помощью метода `Close()`. Он имеет также перегруженный конструктор, которому можно передать объект `Stream`. Теперь вы поняли, как это работает?

```
string folder =
    Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
StreamReader reader =
    new StreamReader(folder + @"\secret_plan.txt");
StreamWriter writer =
    new StreamWriter(folder + @"\emailToCaptainAmazing.txt");
```

Передайте файл, который требуется прочитать конструктору класса StreamReader. На этот раз мы не записываем его в папку C:\!

С помощью класса StreamReader программа читает план Жулика, а средства класса StreamWriter позволяют написать файл, который будет отправлен по электронной почте супергерою Капитану Великолепному.

```
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 plan:");
while (!reader.EndOfStream) {
    string lineFromThePlan = reader.ReadLine();
    writer.WriteLine("The plan -> " + lineFromThePlan);
}
writer.WriteLine();
writer.WriteLine("Can you help us?");
writer.Close();
reader.Close();
```

Пустой метод WriteLine() записывает пустую строку.

Свойство EndOfStream, позволяющее определить, остались ли в файле непрочитанные данные.

Цикл читает строку при помощи считывающего устройства и записывает ее при помощи устройства записи.

Вы должны закрыть все открытые потоки, даже если всего лишь читаете файл.

Объекты StreamReader и StreamWriter после создания их экземпляров открыли собственные потоки. Чтобы закрыть их, мы два раза вызываем метод Close().

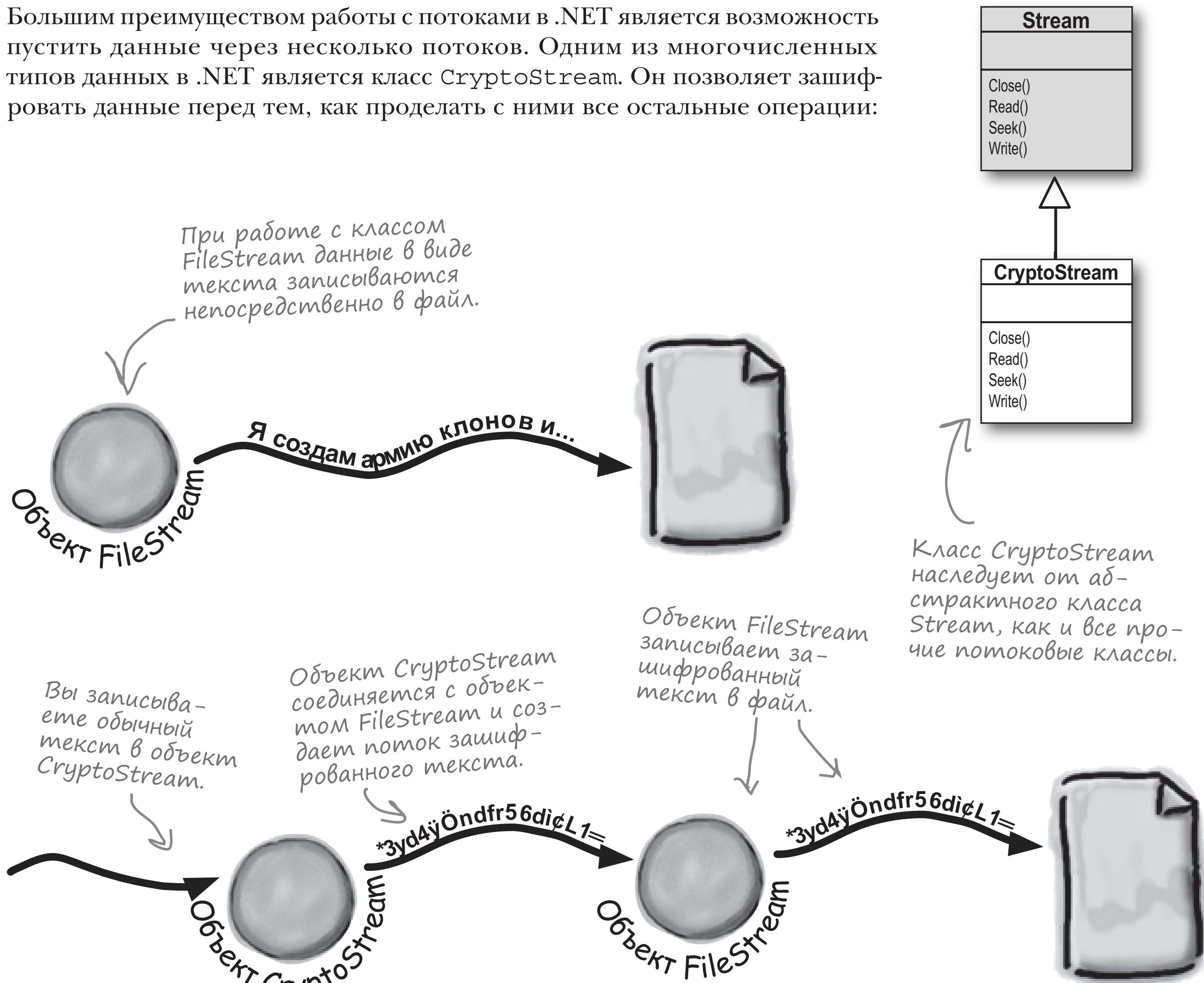
emailToCaptainAmazing - Notepad

File Edit Format View Help

```
To: CaptainAmazing@objectville.net
From: Commissioner@objectiville.net
Subject: Can you save the day... again?

We've discovered the Swindler's plan:
The plan -> How I'll defeat Captain Amazing
The plan -> Another genius secret plan by The Swindler
The plan -> I'll create an army of clones and unleash them upon the citizens of Objectville.
The plan -> Clone #0 attacks the mall
The plan -> Clone #1 attacks downtown
The plan -> Clone #2 attacks the mall
The plan -> Clone #3 attacks downtown
The plan -> Clone #4 attacks the mall
The plan -> Clone #5 attacks downtown
The plan -> Clone #6 attacks the mall

Can you help us?
```

Данные могут проходить через несколько потоков

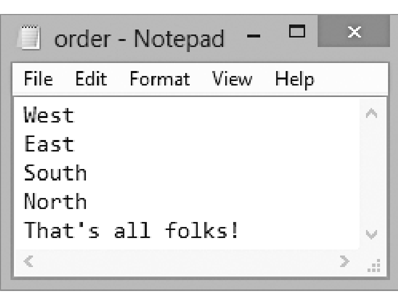
Большим преимуществом работы с потоками в .NET является возможность пустить данные через несколько потоков. Одним из многочисленных типов данных в .NET является класс `CryptoStream`. Он позволяет зашифровать данные перед тем, как проделать с ними все остальные операции:



Потоки можно ПЕРЕДАВАТЬ ПО ЦЕПОЧКЕ. Один поток может быть записан в другой, который в свою очередь записывается в еще один поток... концом цепочки часто является файл.

Ребус в бассейне

Вам нужно взять фрагменты кода из бассейна и поместить их на пустые строчки. Любой фрагмент можно использовать несколько раз. В бассейне есть и лишние фрагменты. В результате нужно получить окно с текстом, показанное справа.




```
class Pineapple {
  const _______ d = "delivery.txt";
  public ______ _______
      { North, South, East, West, Flamingo }
  public static void Main(string[] args) {
    ___________ o = new _____________("order.txt");
    Pizza pz = new Pizza(new ___________(d, true));
    pz._________(Fargo.Flamingo);
    for (_____ w = 3; w >= 0; w--) {
      Pizza i = new Pizza
             (new ____________(d, false));
      i.Idaho((Fargo)w);
      Party p = new Party(new ___________(d));
      p.____________(o);
    }
    o.____________("That's all folks!");
    o.___________();
  }
}
```

```
class Pizza {
  private _____________ ________;
  public Pizza(___________ ________) {
    _______.writer = writer;
  }
  public void ______(_______.Fargo f) {
    writer.__________(f);
    writer.___________();
  }
}

class Party {
  private _____________ reader;
  public Party(_____________ reader) {
    ___________.reader = reader;
  }
  public void HowMuch(___________ q) {
    q.__________(reader.__________());
    reader.___________();
  }
}
```

Каждый фрагмент можно использовать несколько раз!

HowMany
HowMuch
HowBig
HowSmall

int
long
string
enum
class

ReadLine
WriteLine

Stream
reader
writer
StreamReader
StreamWriter
Open
Close

public
private
this
class
static

for
while
foreach

=
>=
<=
!=
==
++
--

Fargo
Utah
Idaho
Dakota
Pineapple

Решение ребуса в бассейне

Это перечисление (особенно метод ToString()) используется для вывода конечного результата.

Это точка входа программы. Здесь создается объект StreamWriter, который передается в класс Party. Затем члены перечисления Fargo в цикле передаются методу Pizza.Idaho() для вывода в форму.

Класс Pizza использует объект StreamWriter как закрытое поле, а его метод Idaho() записывает в файл элементы перечисления Fargo, используя их методы ToString(), автоматически вызываемые методом WriteLine().

В классе Party имеется поле StreamReader, и метод HowMuch() читает оттуда строки, записывая их в поле StreamWriter.

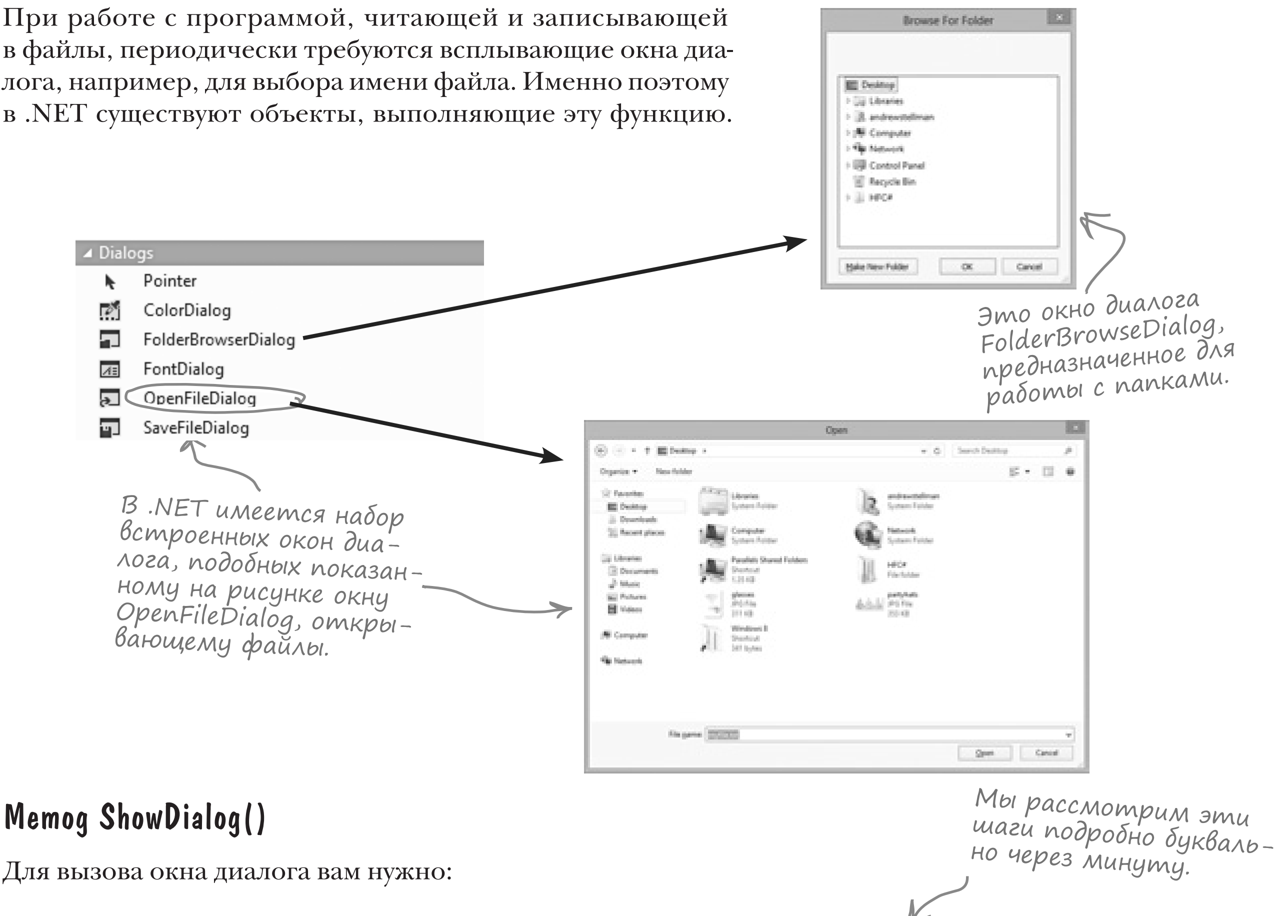
```
class Pineapple {
    const string d = "delivery.txt";
    public enum Fargo { North, South, East, West, Flamingo }

    public static void Main(string[] args) {
        StreamWriter o = new StreamWriter("order.txt");
        Pizza pz = new Pizza(new StreamWriter(d, true));
        pz.Idaho(Fargo.Flamingo);
        for (int w = 3; w >= 0; w--) {
            Pizza 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();
    }
}
```

order - Notepad
File Edit Format View Help

```
West
East
South
North
That's all folks!
```

Встроенные объекты для вызова стандартных окон диалога

При работе с программой, читающей и записывающей в файлы, периодически требуются всплывающие окна диалога, например, для выбора имени файла. Именно поэтому в .NET существуют объекты, выполняющие эту функцию.



Метод ShowDialog()

Для вызова окна диалога вам нужно:

Мы рассмотрим эти шаги подробно буквально через минуту.

1. Создать экземпляр окна диалога. Это можно сделать при помощи оператора new или путем перетаскивания из Toolbox.

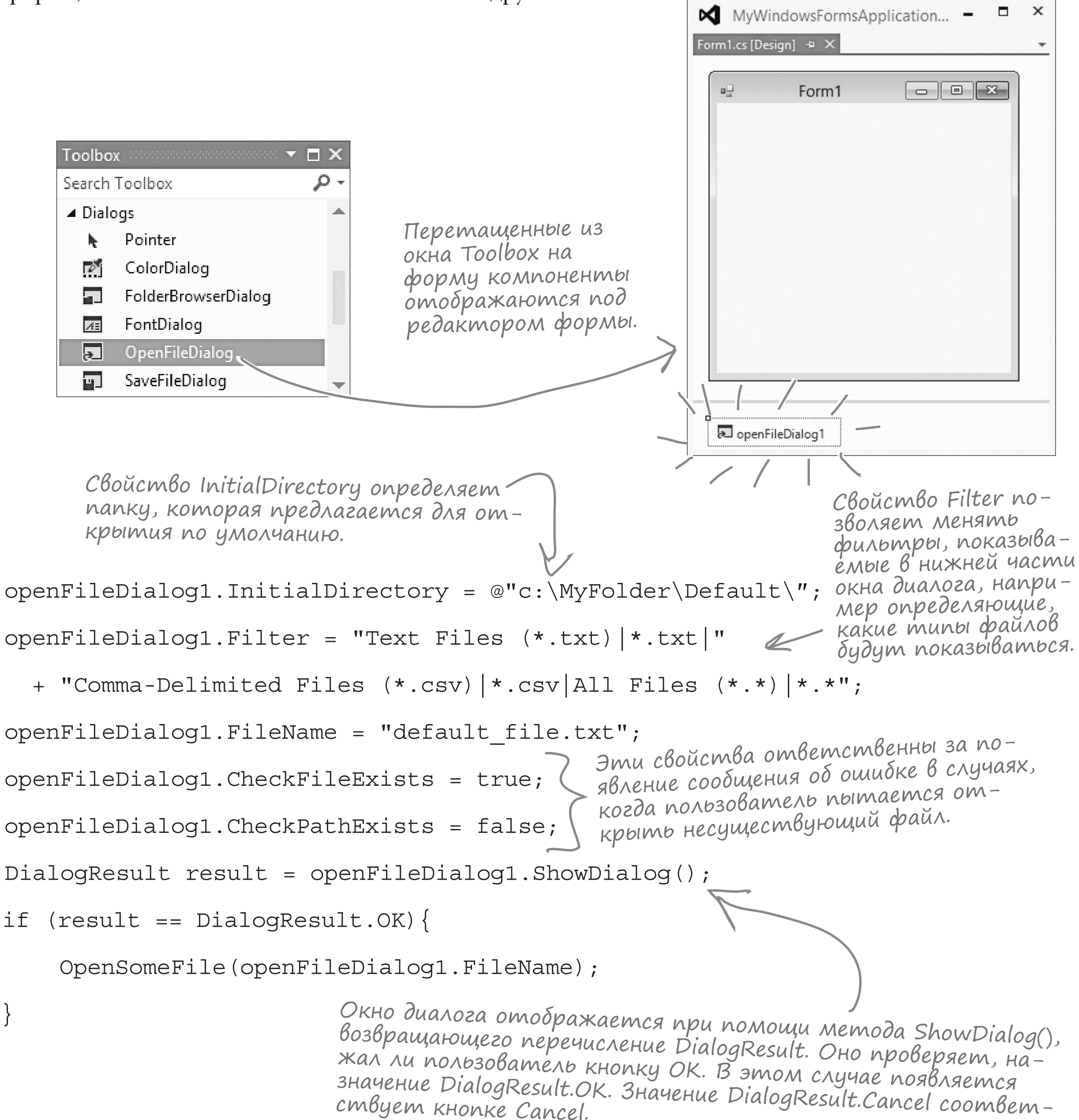
2. Задайте свойства окна. Например, `Title` (текст в строке заголовка), `InitialDirectory` (адрес открываемой по умолчанию папки) и `FileName` (для окон диалога Open и Save).

3. Вызовите метод `ShowDialog()`. Он вызывает окно диалога и не возвращает значение, пока пользователь не щелкнет на кнопке OK или Cancel или другим способом не закроет окно.

4. Метод `ShowDialog()` возвращает перечисление `DialogResult`. Его члены принимают значения `OK` (означает, что пользователь щелкнул на кнопке OK), `Cancel`, `Yes` и `No` (для окон диалога Yes/No).

Окна диалога — это тоже объекты

Чтобы добавить стандартное окно диалога Windows, достаточно перетащить на форму элемент управления `OpenFileDialog` из окна Toolbox. Он появляется в пространстве под формой, так как это **компонент**. Так называется специальный вид **невизуальных элементов управления**. Они не появляются на форме, но их можно использовать как любые другие элементы.




```
openFileDialog1.InitialDirectory = @"c:\MyFolder\Default\";
openFileDialog1.Filter = "Text Files (*.txt)|*.txt|"
  + "Comma-Delimited Files (*.csv)|*.csv|All Files (*.*)|*.*";
openFileDialog1.FileName = "default_file.txt";
openFileDialog1.CheckFileExists = true;
openFileDialog1.CheckPathExists = false;
DialogResult result = openFileDialog1.ShowDialog();
if (result == DialogResult.OK){
    OpenSomeFile(openFileDialog1.FileName);
}
```

Окна диалога — это объекты

Объект **OpenFileDialog** показывает стандартное окно Windows Open, а объект **SaveFileDialog** — стандартное окно Save. Их можно отобразить, создав при помощи оператора new экземпляр, задав его свойства и вызвав его метод ShowDialog(). Этот метод возвращает перечисление DialogResult (простой логической переменной в данном случае недостаточно, так как некоторые окна диалога имеют более двух кнопок).

```
saveFileDialog1 = new SaveFileDialog();

saveFileDialog1.InitialDirectory = @"c:\MyFolder\Default\";

saveFileDialog1.Filter = "Text Files (*.txt)|*.txt|"
  + "Comma-Delimited Files (*.csv)|*.csv|All Files (*.*)|*.*";

DialogResult result = saveFileDialog1.ShowDialog();

if (result == DialogResult.OK){
    SaveTheFile(saveFileDialog1.FileName);
}
```

После перетаскивания элемента SaveFileDialog из окна Toolbox на форму к методу формы InitializeComponent() добавляется эта строка.

Понять смысл свойства Filter несложно. Сравните написанное между символами | с тем, что показывается в раскрывающемся списке внизу окна.

Метод ShowDialog () и свойство FileName имеют те же функции, что и объект OpenFileDialog.

Предполагается, что в программе есть метод SaveTheFile(), принимающий имя файла в качестве параметра.

Объект SaveFileDialog соответствует стандартному окну диалога Save as...

Свойство Title меняет текст в строке заголовка.

*Редактирование раскрывающегося списка Save as type осуществляется при помощи свойства **Filter**.*

Маршрут доступа к выбранному пользователем файлу сохраняется в свойстве FileName.

Метод ShowDialog() вызывает окно диалога, открытое на папке, заданной свойством InitialDirectory.

Перечисление DialogResult, возвращаемое методом ShowDialog(), указывает, какую кнопку нажал пользователь.

Встроенные классы File и Directory

Подобно классу StreamWriter, класс File создает потоки, позволяющие работать с файлами в фоновом режиме. Методы этого класса дают возможность выполнять большинство операций без предварительного создания объекта FileStream. Объект Directory предназначен для работы с папками.

Класс File позволяет:

1. **Проверять, существует ли файл**
   Метод Exists() возвращает значение true при наличии указанного вами файла и false — в противном случае.

2. **Читать из файла и записывать в него**
   Метод OpenRead() дает доступ к чтению файла, а методы Create() и OpenWrite() — к записи в файл.

3. **Добавлять в файл текст**
   Метод AppendAllText() добавляет текст в существующий файл и создает файл, если он отсутствует на момент запуска метода.

4. **Получает информацию о файле**
   Методы GetLastAccessTime() и GetLastWriteTime() возвращают время и дату последнего доступа и последнего редактирования файла.

При больших объемах работы с файлом имеет смысл создать экземпляр класса FileInfo вместо работы со статическими методами класса File.

Класс FileInfo отличается от класса File тем, что для работы вам потребуется его экземпляр.

Кроме того, 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, то есть представляет собой набор методов для работы с файлами. После создания экземпляра FileInfo вы получаете доступ к тому же списку методов, что и при работе с классом File.

Класс Directory позволяет:

1. **Создать новую папку**
   При работе с методом CreateDirectory() вам требуется указать маршрут доступа к папке.

2. **Получить список файлов в папке**
   Метод GetFiles() создает массив файлов, содержащихся в папке. Вам нужно только указать маршрут доступа к ней.

3. **Удалить папку**
   Эта несложная процедура выполняется методом Delete().

Часто задаваемые вопросы

В: И все-таки зачем нужны символы {0} и {1} в объявлении объекта StreamWriter?

О: При вводе строк в файл иногда возникает необходимость ввести туда и содержимое многочисленных переменных. К примеру, можно написать:

```
writer.WriteLine("Меня зовут " + name +
    "мне " + age + " лет.");
```

Но комбинировать строки таким образом долго, кроме того, возрастает вероятность опечаток. Намного проще писать код:

```
writer.WriteLine(
  "Меня зовут {0} мне {1} лет",
   name, age);
```

Такой вариант легче читается, особенно когда в одной строке оказывается много переменных.

В: Зачем нужен знак @ перед именем файла?

О: При добавлении в программу строковых констант компилятор преобразует esc-последовательности (`\n` или `\r`) в специальные символы. Но косая черта присутствует и в маршрутах доступа к файлам. Поместив в начало строки знак `@`, вы говорите C#, что в строке отсутствуют esc-последовательности. Кроме того, в нее начинают включаться знаки переноса (то есть нажатия клавиши Enter фиксируются автоматически):

```
string twoLine = @"это строка,
занимающая две строчки.";
```

В: Напомните еще раз, что означают символы `\n` и `\t`.

О: Это так называемые esc-последовательности. `\n` — перенос строки, `\t` — табуляция, `\r` — символ возврата. В текстовых файлах Windows строки должны заканчиваться символами `\r\n` (об этом мы говорили в главе 8 при знакомстве с константой `Environment.NewLine`). Чтобы использовать в строке обратную косую черту, которую компилятор не воспринимает как esc-последовательность, делайте ее ***двойной***: `\\`.

В: А что там говорилось про преобразование строки в массив байтов? Как это работает?

О: Наверное, вы уже слышали, что файлы на диске представлены в виде битов и байтов. Другими словами, при записи файла на жесткий диск операционная система воспринимает его как набор байтов. Объекты `StreamReader` и `StreamWriter` преобразуют эти байты в понятные вам символы, то есть выполняют кодирование и раскодирование. 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. При открытии файла, скажем, в приложении Блокнот каждый байт преобразуется в символ: например, `E` соответствует 69, а `a` — 97 (впрочем, все зависит от кодировки... но мы поговорим об этом чуть позже). Соответственно при сохранении введенного вами в Блокнот текста символы преобразуются обратно в байты. Это преобразование нужно и для записи переменной типа `string` в поток.

В: Разве для записи в файл недостаточно объекта StreamWriter? Зачем создавать объект FileStream?

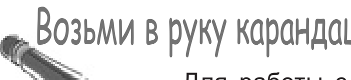
О: Для записи строк в текстовый файл и их дальнейшего чтения действительно достаточно объектов `StreamReader` и `StreamWriter`. Но для более сложных операций требуется задействовать другие потоки. Записывать в файл числа, массивы, коллекции и объекты `StreamWriter` не умеет. Впрочем, эта тема будет подробно рассмотрена чуть позже.

В: Как мне создать собственные диалоговые окна?

О: Вы можете добавить в проект форму и придать ей нужный вид. Затем при помощи оператора `new` создается ее экземпляр (именно так вы поступали с объектом `OpenFileDialog`). После чего остается добавить метод `ShowDialog()`, и новое окно диалога готово.

В: Зачем закрывать потоки 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. Именно это происходит с вашей программой при открытии файла. Файл остается заблокированным, пока вы не воспользуетесь методом `Close()`.



Для работы с файлами и папками в .NET существуют два встроенных класса с многочисленными статическими методами. Класс **`File`** содержит методы работы с файлами, а класс **`Directory`** дает возможность работать с папками. Напишите, как, с вашей точки зрения, работают представленные слева строки кода.

Код	Его функция

`if (!Directory.Exists(@"c:\SYP")) {`
`    Directory.CreateDirectory(@"c:\SYP");`
`}`	

`if (Directory.Exists(@"c:\SYP\Bonk")) {`
`    Directory.Delete(@"c:\SYP\Bonk");`
`}`	

`Directory.CreateDirectory(@"c:\SYP\Bonk");`	
`Directory.SetCreationTime(@"c:\SYP\Bonk",` `        new DateTime(1976, 09, 25));`	
`string[] files = Directory.GetFiles(@"c:\windows\",` `        "*.log", SearchOption.AllDirectories);`	

`File.WriteAllText(@"c:\SYP\Bonk\weirdo.txt",`
`        @"Это первая строчка`
`а это вторая строчка`
`а это последняя строчка");`	

`File.Encrypt(@"c:\SYP\Bonk\weirdo.txt");` *Это вы еще не проходили... сможете ли вы угадать назначение этого метода?*	
`File.Copy(@"c:\SYP\Bonk\weirdo.txt",` `        @"c:\SYP\copy.txt");`	
`DateTime myTime =` `        Directory.GetCreationTime(@"c:\SYP\Bonk");`	
`File.SetLastWriteTime(@"c:\SYP\copy.txt", myTime);`	
`File.Delete(@"c:\SYP\Bonk\weirdo.txt");`	

Открытие и сохранение файлов при помощи окон диалога

Построим программу, открывающую текстовый файл. Она должна позволять редактировать файл и сохранять сделанные изменения при помощи стандартных элементов управления .NET.

Упражнение!

Чтобы развернуть текстовое поле на всю форму, перетащите на нее элемент TableLayoutPanel из окна Containers, присвойте свойству Dock значение Fill, а при помощи редактора свойств Rows и Columns создайте две строки и один столбец. В верхнюю ячейку перетащите элемент TextBox, в нижнюю — элемент FlowLayoutPanel. Свойству Dock присвойте значение Fill, а свойству FlowDirection — значение RightToLeft и перетащите туда две кнопки. Размер верхнего ряда элемента TableLayoutPanel задайте равным 100%, а размер нижнего ряда поменяйте таким образом, чтобы там поместились две кнопки.

1. **Создание простой формы.**
Вам потребуется текстовое поле и две кнопки. Перетащите на форму также элементы OpenFileDialog и SaveFileDialog. Двойным щелчком на кнопках создайте обработчики событий и **добавьте закрытое строковое поле name**. Добавьте оператор using для System.IO.

Simple Text Editor

Это текстовое поле, свойство Multiline которого имеет значение true.

Open Save

2. **Привязка кнопки Open к элементу openFileDialog.**
Кнопка Open отображает объект OpenFileDialog и использует метод File.ReadAllText() для чтения файла в текстовом поле:

```
private void open_Click(object sender, EventArgs e) {
  if (openFileDialog1.ShowDialog() == DialogResult.OK) {
    name = openFileDialog1.FileName;
    textBox1.Clear();
    textBox1.Text = File.ReadAllText(name);
  }
}
```

Щелчок на кнопке Open делает видимым элемент управления OpenFileDialog.

3. **Действия для кнопки Save.**
Кнопка Save использует для сохранения файла метод File.WriteAllText():

```
private void save_Click(object sender, EventArgs e) {
  if (saveFileDialog1.ShowDialog() == DialogResult.OK) {
    name = saveFileDialog1.FileName;
    File.WriteAllText(name, textBox1.Text);
  }
}
```

Методы ReadAllText() и WriteAllText() являются частью класса File. Более подробно мы поговорим о них через несколько страниц.

4. **Другие свойства окна диалога.**
   - ★ Поменяйте текст в строке заголовка с помощью свойства Title объекта saveFileDialog.
   - ★ Укажите открываемую по умолчанию папку с помощью свойства initialFolder.
   - ★ Укажите, что объект OpenFileDialog с помощью свойства Filter должен показывать только текстовые файлы.

Если не задать это свойство, раскрывающийся список в нижней части окон диалога Open и Save будет пустым. В данном случае используйте фильтр: Text Files (*.txt)|*.txt.

Возьми в руку карандаш
Решение

Вот какие действия производят строки кода.

Код	Функция

if (!Directory.Exists(@"c:\SYP")) {
    Directory.CreateDirectory(@”c:\SYP”);
}
	Проверяет существование папки C:\SYP и создает ее, если она отсутствует.

if (Directory.Exists(@”c:\SYP\Bonk")) {
    Directory.Delete(@"c:\SYP\Bonk");
}
	Проверяет существование папки C:\SYP\Bonk. Если папка существует, она удаляется.

Directory.CreateDirectory(@"c:\SYP\Bonk");	Создает папку C:\SYP\Bonk.
Directory.SetCreationTime(@"c:\SYP\Bonk", new DateTime(1976, 09, 25));	Задает время создания папки C:\SYP\Bonk, 25 сентября 1976.
string[] files = Directory.GetFiles(@"c:\windows\", "*.log", SearchOption.AllDirectories);	Получает список всех файлов с расширением *.log в папке C:\Windows, включая файлы в подпапках.
File.WriteAllText(@"c:\SYP\Bonk\weirdo.txt", @"Это первая строчка а это вторая строчка а это последняя строчка");	Создает файл weirdo.txt (если он еще не создан) в папке C:\SYP\Bonk и записывает в него три строки текста.
File.Encrypt(@"c:\SYP\Bonk\weirdo.txt"); Это альтернативный способ использования объекта CryptoStream.	Зашифровывает файл weirdo.txt при помощи встроенной системы шифрования Windows.
File.Copy(@"c:\SYP\Bonk\weirdo.txt", @"c:\SYP\copy.txt");	Копирует содержимое файла C:\SYP\Bonk\weirdo.txt в файл C:\SYP\Copy.txt.
DateTime myTime = Directory.GetCreationTime(@"c:\SYP\Bonk");	Объявляет переменную myTime и присваивает ей время создания папки C:\SYP\Bonk.
File.SetLastWriteTime(@"c:\SYP\copy.txt", myTime);	Меняет последнее время записи файла copy.txt в папке C:\SYP\, присваивая ему значение переменной myTime.
File.Delete(@"c:\SYP\Bonk\weirdo.txt");	Удаляет файл C:\SYP\Bonk\weirdo.txt.

Интерфейс IDisposable

Множество классов .NET реализует крайне полезный интерфейс `IDisposable`. Он содержит **всего один** метод **`Dispose()`**. Этот интерфейс объясняет программе, что есть важные вещи, которые требуется сделать для завершения работы. Ведь **распределенные ресурсы** не освобождаются самостоятельно. Для этого им требуется метод `Dispose()`.

Воспользуемся функцией IDE Go To Definition для просмотра определения интерфейса `IDisposable`. Введите «IDisposable» в произвольном месте внутри класса, щелкните на этой строчке правой кнопкой мыши и выбрите в меню команду Go To Definition. Откроется вкладка с кодом. Вот что вы увидите:

При объявлении объектов в разделе using вызов метода Dispose( ) таких объектов будет осуществляться автоматически.

```
namespace System
{
    // Краткое описание:
    // Дает метод освобождения распределенных ресурсов.
    public interface IDisposable
    {
        // Краткое описание:
        //     Выполняет определяемые приложением задачи,
        //     связанные с освобождением или сбросом
        //     неуправляемых ресурсов.
        void Dispose();
    }
}
```

Многие классы распределяют между собой такие важные ресурсы, как память, файлы и другие объекты. Они соединяются с этими ресурсами и не разрывают связь, пока не получат сигнал, что работа закончена.

Любой класс, реализующий интерфейс IDisposable, немедленно освобождает любые задействованные ресурсы после вызова его метода Dispose(). Это последнее, что делается перед завершением работы с объектом.

В IDE имеется полезная функция, позволяющая автоматически перейти к определению любой переменной, объекта или метода. Достаточно щелкнуть на имени правой кнопкой мыши и выбрать в появившемся меню команду Go To Definition. Аналогичный результат достигается нажатием клавиши F12.

рас-пре-де-лять, гл. раздавать ресурсы или обязанности для определенных целей. *Управляющий **распределил** все конференц-залы под бесполезный семинар по менеджменту.*

Операторы using как средство избежать системных ошибок

На протяжении главы вам твердили о необходимости **закрывать потоки**. Ведь именно с этим связана одна из самых распространенных ошибок. К счастью, в C# имеется замечательный инструмент, позволяющий избежать подобной ситуации, это интерфейс `IDisposable` с его методом `Dispose()`. Если **поместить код потока в оператор `using`**, поток начнет закрываться автоматически. Вам нужно только **объявить потоковую ссылку** с этим оператором, поместив следом в фигурных скобках использующий эту ссылку код. Оператор `using` после завершения работы с этим кодом будет **автоматически вызывать метод `Dispose()` потока**. Вот как это работает:

В данном случае речь идет вовсе не о тех операторах using, которые располагаются в верхней части кода.

За оператором using всегда следует объявление объекта...

...и блок кода в фигурных скобках.

```
using (StreamWriter sw = new StreamWriter("secret_plan.txt")) {
    sw.WriteLine("Как победить Капитана Великолепного");
    sw.WriteLine("Еще один гениальный секретный план");
    sw.WriteLine("от Жулика");
}
```

Эти операторы используют объект, созданный оператором using, как и любой другой.

После завершения работы оператора using вызывается метод Dispose() используемого объекта.

В данном случае на используемый объект указывает ссылка sw, объявленная внутри оператора using. Поэтому будет запущен метод Dispose() класса Stream... который и закроет поток.

Все потоки имеют закрывающий метод `Dispose()`. При этом поток, объявленный внутри оператора using, всегда закрывает себя сам!

По одному оператору using на объект

Операторы `using` можно помещать друг за другом, при этом вам не потребуется дополнительный набор фигурных скобок.

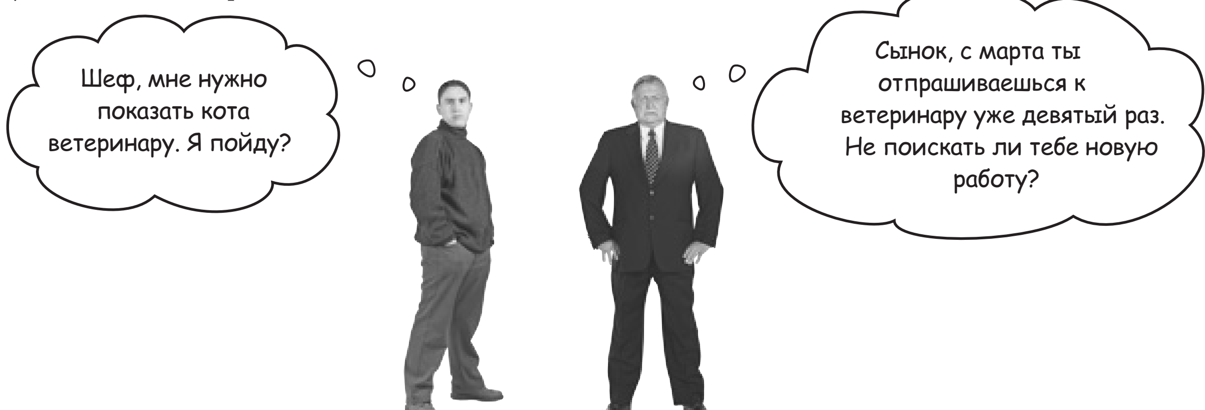
```
using (StreamReader reader = new StreamReader("secret_plan.txt"))
using (StreamWriter writer = new StreamReader("email.txt"))
{
    // операторы, использующие устройства чтения и записи
}
```

Вам больше не требуется метод Close(), чтобы закрыть поток, так как оператор using закроет его автоматически.

Потоки ВСЕГДА следует объявлять внутри оператора using. Это гарантирует, что после завершения работы они будут закрыты!

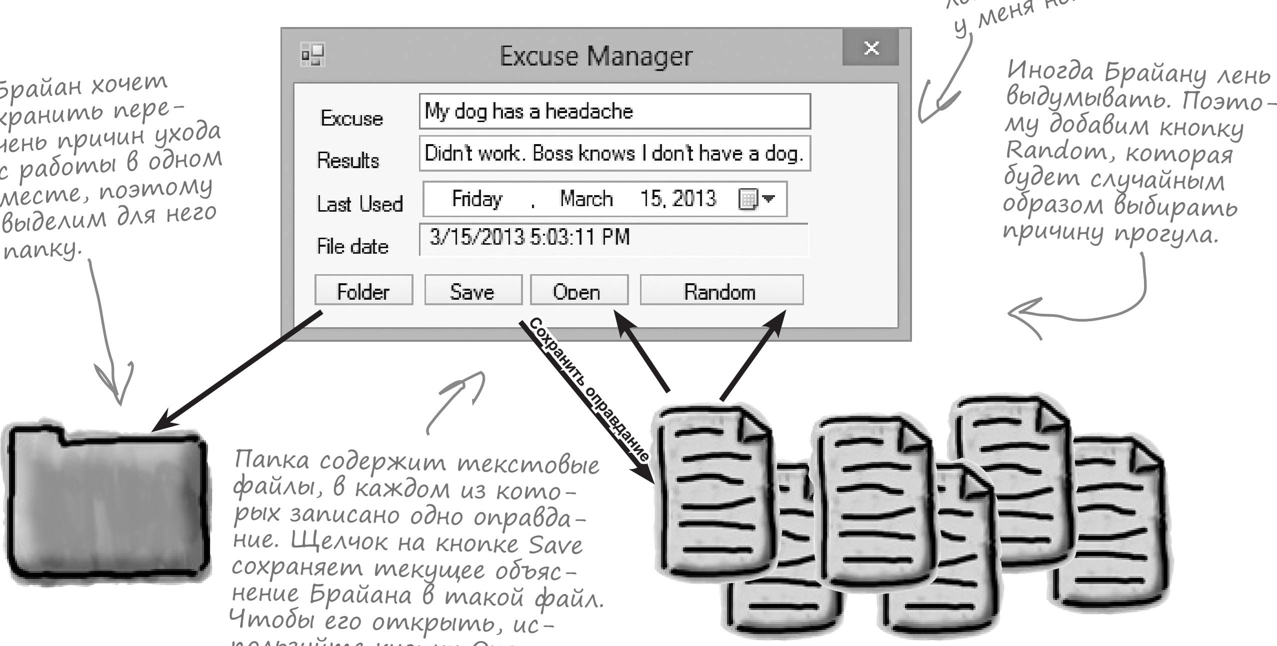
Проблемы на работе

Перед вами Брайан. Он разработчик программ на C# и любит свою работу, но *обожает* устраивать незапланированные выходные. Его начальник **ненавидит** такие ситуации, поэтому Брайану приходится придумывать веские причины.



Создадим программу учета оправданий Брайана

Используйте свои знания о чтении файлов и записи в них для создания программы по поиску подходящего оправдания. Брайану нужно отслеживать, какие причины ухода с работы он использовал в последнее время и как на них отреагировал его начальник.



Упражнение

Создайте для Брайана программу, выбирающую оправдание.

Excuse
Description: string Results: string LastUsed: DateTime ExcusePath: string
OpenFile(string) Save(string)

Создание формы

Вам требуется форма со следующими отличительными признаками:

- ★ При первой загрузке должна быть доступна **только кнопка Folder**.
- ★ При открытии и сохранении причины при помощи элемента `Label` отображается текущая дата. Свойство `AutoSize` этого элемента имеет значение `False`, а свойство `BorderStyle` — значение `Fixed3D`.
- ★ После сохранения оправдания появляется окно Excuse Written.
- ★ Кнопка Folder открывает окно диалога для просмотра папок. После выбора папки появляются кнопки Save, Open и Random Excuse.
- ★ Если пользователь изменил любое из трех полей, в строку заголовка рядом с надписью «Excuse Manager» добавляется звездочка (*). Она исчезает после сохранения данных или открытия нового файла.
- ★ Форма следит за текущей папкой и за тем, было ли сохранено текущее оправдание. Чтобы следить за процессом сохранения, **используйте обработчики событий `Changed`** для трех элементов ввода.

Дважды щелкнув на перетащенном на форму текстовом поле, вы создадите для него обработчик событий Changed.

Создание класса `Excuse` и хранение его экземпляра в форме

Добавим в форму поле `CurrentExcuse` для отображения выбранного оправдания. Вам потребуются **три перегруженных конструктора**: для загрузки формы, для открытия файла и для случайного оправдания. Добавьте методы `OpenFile()` и `Save()` для открытия и сохранения оправдания. А затем еще и метод `UpdateForm()`, обновляющий элементы управления (а вот полезные **подсказки**):

Этот параметр определяет, вносились ли изменения в форму. Вам потребуется поле для его хранения.

Помните, что символ ! означает НЕТ—то есть здесь проверяется, не является ли маршрут доступа к файлу с оправданием пустым или со значением null.

```
private void UpdateForm(bool changed) {
  if (!changed)  {
    this.description.Text = currentExcuse.Description;
    this.results.Text = currentExcuse.Results;
    this.lastUsed.Value = currentExcuse.LastUsed;
    if (!String.IsNullOrEmpty(currentExcuse.ExcusePath))
      FileDate.Text = File.GetLastWriteTime(currentExcuse.ExcusePath).ToString();
      this.Text = "Excuse Manager";
  }
  else
      this.Text = "Excuse Manager*";
  this.formChanged = changed;
}
```

Дважды щелкните на элементах ввода для создания обработчика событий Changed. Три обработчика событий сначала будут менять экземпляр Excuse, а затем вызовут метод UpdateForm(true), дальше способ изменения полей формы выбираете только вы.

Присвойте начальное значение параметру `LastUsed` в конструкторе формы:

```
public Form1() {
    InitializeComponent();
    currentExcuse.LastUsed = lastUsed.Value;
}
```

Кнопка Folder открывает программу просмотра папок

Щелчок на кнопке Folder должен открывать окно диалога Browse for Folder. Форме потребуется поле для хранения информации о папке. При **первом вызове формы** кнопки Save, Open и Random Excuse **отключены**, но кнопка Folder включает их после выбора пользователем папки.

Кнопка Save сохраняет выбранное оправдание в файл

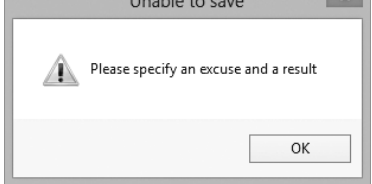
Щелчок на кнопке Save должен вызывать окно диалога Save As.

★ Каждая запись сохраняется в отдельный текстовый файл. В первой строчке фигурирует оправдание, во второй — результат, а в третьей — дата последнего использования (определяется методом `ToString()` объекта `DateTimePicker`). Класс `Excuse` должен иметь метод `Save()` для сохранения оправданий в файл.

★ Окно диалога Save As должно открывать папку, выбранную пользователем при помощи кнопки Folder. В качестве имени файла фигурирует название оправдания с расширением .txt.

★ В окне диалога должны быть фильтры: текстовые файлы (*.txt) и все файлы (*.*).

★ Попытки сохранения при незаполненных полях должны приводить к появлению вот такого окна диалога:



Для отображения значка с восклицательным знаком воспользуйтесь перегруженным методом MessageBox.Show(), который позволяет задавать параметр MessageBoxIcon.

Кнопка Open открывает сохраненные оправдания

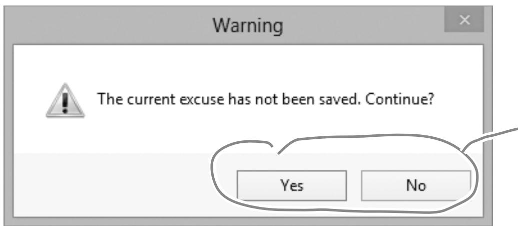
Щелчок на кнопке Open должен вызывать одноименное окно диалога.

★ В окне диалога Open по умолчанию должна быть открыта папка, выбранная пользователем при помощи кнопки Folder.

★ Добавьте метод `Open()` в класс `Excuse`.

★ Метод `Convert.ToDateTime()` загружает сохраненные данные в элемент управления `DateTimePicker`, выбирающий время.

★ Попытка открыть новое оправдание, не сохранив предыдущее, должна приводить к появлению вот такого окна диалога:



Для вызова подобных окон диалога используйте перегруженный метод MessageBox.Show(), позволяющий задавать параметр MessageBoxButtons.YesNo. При щелчке пользователя на кнопке No метод Show() возвращает DialogResult.No.

Ну и наконец, пусть кнопка Random Excuse загружает случайное оправдание

При щелчке на кнопке Random Excuse должен случайным образом открываться файл из папки с оправданиями.

★ Объект `Random` нужно будет сохранить в поле и передать одному из перегруженных конструкторов объекта `Excuse`.

★ Если открытое в данный момент оправдание не сохранено, должно появляться показанное выше окно диалога с предупреждением.

Вот как выглядит код для поиска оправданий отсутствия на работе, который мы написали для Брайана.

```
private Excuse currentExcuse = new Excuse();
private string selectedFolder = "";
private bool formChanged = false;
Random random = new Random();

private void folder_Click(object sender, EventArgs e) {
    folderBrowserDialog1.SelectedPath = selectedFolder;
    DialogResult result = folderBrowserDialog1.ShowDialog();
    if (result == DialogResult.OK) {
        selectedFolder = folderBrowserDialog1.SelectedPath;
        save.Enabled = true;
        open.Enabled = true;
        randomExcuse.Enabled = true;
    }
}

private void save_Click(object sender, EventArgs e) {
    if (String.IsNullOrEmpty(description.Text) || String.IsNullOrEmpty(results.Text)) {
        MessageBox.Show("Please specify an excuse and a result",
             "Unable to save", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
        return;
    }
    saveFileDialog1.InitialDirectory = selectedFolder;
    saveFileDialog1.Filter = "Text files (*.txt)|*.txt|All files (*.*)|*.*";
    saveFileDialog1.FileName = description.Text + ".txt";
    DialogResult result = saveFileDialog1.ShowDialog();
    if (result == DialogResult.OK) {
        currentExcuse.Save(saveFileDialog1.FileName);
        UpdateForm(false);
        MessageBox.Show("Excuse written");
    }
}

private void open_Click(object sender, EventArgs e) {
    if (CheckChanged()) {
        openFileDialog1.InitialDirectory = selectedFolder;
        openFileDialog1.Filter = "Text files (*.txt)|*.txt|All files (*.*)|*.*";
        openFileDialog1.FileName = description.Text + ".txt";
        DialogResult result = openFileDialog1.ShowDialog();
        if (result == DialogResult.OK) {
            currentExcuse = new Excuse(openFileDialog1.FileName);
            UpdateForm(false);
        }
    }
}

private void randomExcuse_Click(object sender, EventArgs e) {
    if (CheckChanged()) {
        currentExcuse = new Excuse(random, selectedFolder);
        UpdateForm(false);
    }
}
```

Форма использует поля для сохранения текущего объекта Excuse в выбранную папку, а также запоминает, был ли изменен этот объект. Кроме того, она сохраняет объект Random для кнопки Random Excuse.

После выбора пользователем папки форма сохраняет ее имя и включает три остальные кнопки.

Это символ оператора ИЛИ, выражение имеет значение true (если не указано оправдание) ИЛИ результат.

Здесь задаются фильтры для окна Save As.

В результате в раскрывающемся списке Files of Type в нижней части окна диалога появятся две строчки: одна для текстовых файлов (.txt), другая для всех прочих файлов (*.*).*

Перечисление DialogResult, возвращаемое окнами диалога Open и Save, гарантирует, что открытие и сохранение файла будет происходить только после щелчка на кнопке OK.

```
private bool CheckChanged() {
    if (formChanged) {
        DialogResult result = MessageBox.Show(
               "The current excuse has not been saved. Continue?",
                "Warning", MessageBoxButtons.YesNo, MessageBoxIcon.Warning);
        if (result == DialogResult.No)
            return false;
    }
    return true;
}

private void description_TextChanged(object sender, EventArgs e) {
    currentExcuse.Description = description.Text;
    UpdateForm(true);
}

private void results_TextChanged(object sender, EventArgs e) {
    currentExcuse.Results = results.Text;
    UpdateForm(true);
}

private void lastUsed_ValueChanged(object sender, EventArgs e) {
    currentExcuse.LastUsed = lastUsed.Value;
    UpdateForm(true);
}

class Excuse {
    public string Description { get; set; }
    public string Results { get; set; }
    public DateTime LastUsed { get; set; }
    public string ExcusePath { get; set; }
    public Excuse() {
        ExcusePath = "";
    }
    public Excuse(string excusePath) {
        OpenFile(excusePath);
    }
    public Excuse(Random random, string folder) {
        string[] fileNames = Directory.GetFiles(folder, "*.txt");
        OpenFile(fileNames[random.Next(fileNames.Length)]);
    }
    private void OpenFile(string excusePath) {
        this.ExcusePath = excusePath;
        using (StreamReader reader = new StreamReader(excusePath)) {
            Description = reader.ReadLine();
            Results = reader.ReadLine();
            LastUsed = Convert.ToDateTime(reader.ReadLine());
        }
    }
    public void Save(string fileName) {
        using (StreamWriter writer = new StreamWriter(fileName))
        {
            writer.WriteLine(Description);
            writer.WriteLine(Results);
            writer.WriteLine(LastUsed);
        }
    }
}
```

Метод MessageBox.Show() возвращает перечисление DialogResult, которое мы можем проверить.

Это три обработчика событий Changed для полей ввода формы. Изменение состояния любого из них говорит о том, что оправдание было отредактировано, поэтому сначала следует обновить экземпляр Excuse, а затем вызвать метод UpdateForm(), добавить звездочку в строку заголовка и присвоить свойству Changed значение true.

Передача значения true методу UpdateForm() заставляет его пометить форму как измененную, но не обновлять состояние полей ввода.

Кнопка Random Excuse использует метод Directory.GetFiles() для чтения текстовых файлов в выбранной папке единым массивом, после чего выбирается случайный индекс, чтобы открыть один из файлов.

Мы гарантировали использование оператора using при каждом открытии потока. В этом случае наши файлы всегда будут закрыты.

Здесь мы объявили объект StreamWriter внутри оператора using, инициировав автоматический вызов его метода Close()!

Вы вызывали метод LastUsed.ToString()? Помните, что метод WriteLine() вызывает его автоматически!

Запись файлов сопровождается принятием решений

Вам предстоит написать множество программ, которые берут входную информацию, например, из файла и решают, что с ней потом делать. Вот крайне типичный код с одним длинным оператором `if`. Он проверяет переменную `part` и выводит в файл различные строчки в зависимости от используемого перечисления. Множество вариантов приводит к множественным комбинациям операторов `else if`:

```
enum BodyPart {
    Head,
    Shoulders,
    Knees,
    Toes
}
```

Перед нами перечисление частей тела — голова, плечи, колени и пальцы ног. Мы собираемся сравнивать переменную с каждым из элементов и выводить строку в зависимости от результата.

```
private void WritePartInfo(BodyPart part, StreamWriter writer) {
  if (part == BodyPart.Head)
    writer.WriteLine("на голове волосы");
  else if (part == BodyPart.Shoulders)
    writer.WriteLine("плечи широкие");
  else if (part == BodyPart.Knees)
    writer.WriteLine("колени узловатые");
  else if (part == BodyPart.Toes)
    writer.WriteLine("пальцы ног маленькие");
  else
    writer.WriteLine("а про эту часть тела мы ничего не знаем");
}
```

Используя операторы if/else, мы вынуждены снова и снова писать строчку if (part ==[option]).

Последний оператор else понадобится, если ни одного соответствия не будет найдено.

Какие проблемы могут возникнуть при написании кода с многочисленными операторами if/else? Подумайте об опечатках, несовпадающих скобках и т. п.

Оператор switch

Оператор switch не имеет никаких особенностей, предназначенных для работы с файлами. Это всего лишь полезный инструмент, который мы можем использовать в текущей ситуации.

Сравнение переменной с набором различных значений — часто встречающаяся ситуация, особенно при чтении и записи файлов. Она настолько распространена, что в C# имеется особый оператор.

Это оператор **switch**. Вот как с его помощью будет выглядеть код с предыдущей страницы, построенный на многочисленных комбинациях опеаторов `if/else`:

Оператор switch сравнивает ОДНУ переменную с МНОЖЕСТВОМ значений.

Ключевое слово break; показывает, где заканчивается один оператор case и начинается следующий.

Завершить оператор case можно знаком переноса строки. Программа все равно будет компилироваться, так как один оператор case заканчивается там, где начинается следующий.

Начните с ключевого слова switch, за которым следует сравниваемая переменная.

```
private void WritePartInfo(BodyPart part, StreamWriter writer)
{
    switch (part) {
        case BodyPart.Head:
            writer.WriteLine("на голове волосы");
            break;
        case BodyPart.Shoulders:
            writer.WriteLine("плечи широкие");
            break;
        case BodyPart.Knees:
            writer.WriteLine("колени узловатые");
            break;
        case BodyPart.Toes:
            writer.WriteLine("пальцы ног маленькие");
            break;
        default:
            writer.WriteLine("а про эту часть тела мы ничего не знаем");
            break;
    }
}
```

Тело оператора switch представляет собой набор операторов case, сравнивающих переменную, следующую за ключевым словом switch, с предлагаемыми значениями.

За ключевым словом case следует значение для сравнения, двоеточие и набор операторов, завершающийся словом break;. Именно эти операторы выполняются при совпадении значений.

Оператор Switch может заканчиваться оператором default:, который выполняется, если совпадений не обнаружено.

Чтение и запись информации о картах в файл

Запись колоды в файл осуществить легко — достаточно цикла, записывающего имя каждой карты. Вот метод, который можно добавить к объекту Deck:

```
public void WriteCards(string filename) {
    using (StreamWriter writer = new StreamWriter(filename)) {
        for (int i = 0; i < cards.Count; i++) {
            writer.WriteLine(cards[i].Name);
        }
    }
}
```

А как осуществить чтение из файла? Именно здесь вам на помощь придет оператор switch.

Оператор switch позволяет сравнивать одно значение с набором переменных.

```
Suits suit;
switch (suitString) (
    case "Spades":
        suit = Suits.Spades;
        break;
    case "Clubs":
        suit = Suits.Clubs;
        break;
    case "Hearts":
        suit = Suits.Hearts;
        break;
    case "Diamonds":
        suit = Suits.Diamonds;
        break;
    default:
        MessageBox.Show(suitString + " не подходит!");
}
```

Оператор switch начинает работу с указания ***значения****, с которым будет осуществляться сравнение. Данный оператор switch вызывается из метода, поэтому переменная suit сохранена в строке.*

Каждая из строчек ***case*** *сравнивает некоторое значение с переменной, указанной в строчке switch. В случае совпадения выполняются все операторы, расположенные до ключевого слова* ***break****.*

Строчка ***default*** *стоит в самом конце. Если значение переменной не совпало ни с одним из предложенных вариантов, будут выполнены операторы, стоящие после ключевого слова default.*

Чтение карт из файла

Оператор switch позволяет построить новый конструктор для класса Deck из предыдущей главы. Он читает из файла и проверяет каждую строчку на предмет сравнения с картами. При удачном результате сравнения карта попадает в колоду.

Вот крайне полезный метод Split(). Он разбивает каждую строку на массив более мелких строк, передавая ей массив char[] разделительных знаков.

```
public Deck(string filename) {
   cards = new List<Card>();
   using (StreamReader reader = new StreamReader(filename)) {
      while (!reader.EndOfStream) {
         bool invalidCard = false;
         string nextCard = reader.ReadLine();
         string[] cardParts = nextCard.Split(new char[] { ' ' });
         Values value = Values.Ace;
         switch (cardParts[0]) {
             case "Ace": value = Values.Ace; break;
             case "Two": value = Values.Two; break;
             case "Three": value = Values.Three; break;
             case "Four": value = Values.Four; break;
             case "Five": value = Values.Five; break;
             case "Six": value = Values.Six; break;
             case "Seven": value = Values.Seven; break;
             case "Eight": value = Values.Eight; break;
             case "Nine": value = Values.Nine; break;
             case "Ten": value = Values.Ten; break;
             case "Jack": value = Values.Jack; break;
             case "Queen": value = Values.Queen; break;
             case "King": value = Values.King; break;
             default: invalidCard = true; break;
         }
         Suits suit = Suits.Clubs;
         switch (cardParts[2]) {
             case "Spades": suit = Suits.Spades; break;
             case "Clubs": suit = Suits.Clubs; break;
             case "Hearts": suit = Suits.Hearts; break;
             case "Diamonds": suit = Suits.Diamonds; break;
             default: invalidCard = true; break;
         }
         if (!invalidCard) {
             cards.Add(new Card(suit, value));
         }
      }
   }
}
```

Здесь строку nextCard разбивают при помощи пробелов. В результате строковая переменная Six of Diamonds превращается в массив {"Six", "of", "Diamonds"}.

Оператор switch сравнивает значение с первым словом строки. В случае совпадения значение присваивается переменной value.

Аналогичная операция проделывается с третьим словом строки, но в этом случае совпавшее значение присваивается переменной suit.

Столько кода для чтения всего одной карты? Не многовато ли? А что делать с объектами с большим количеством значений и полей? Неужели мне потребуется оператор switch для каждого?

Есть более простой способ сохранения объектов в файл — сериализация (serialization).

Вместо скрупулезной записи в файл каждого поля и значения можно сохранить объект сериализацией его в поток. Объект при этом превращается в набор байтов. Обратный процесс называется ***десериализацией***, то есть вы воссоздаете объект из потока байтов.

Заметим на будущее, что существует метод Enum.Parse() (с ним вы познакомитесь в главе 14), который преобразует строку Spades в элемент перечисления Suits.Spades. Впрочем, в рассматриваемом случае лучше всего использовать сериализацию, в чем вы скоро убедитесь...

Что происходит с объектом при сериализации?

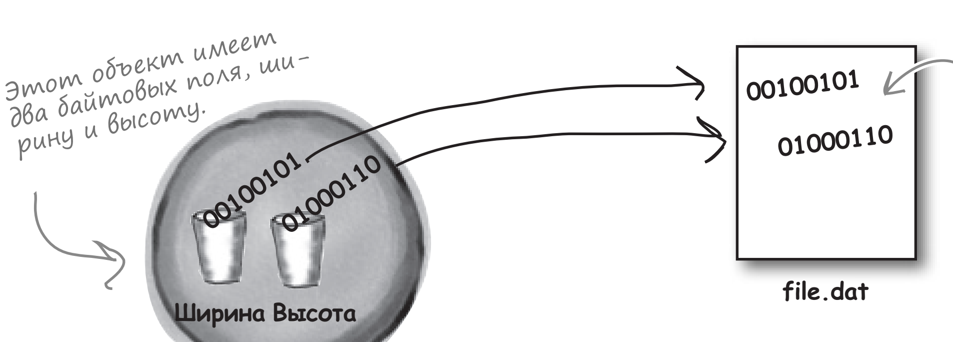
Процесс копирования объекта из кучи и помещения в файл кажется таинственным, но на самом деле там все очень просто.

1 Объект в куче

Экземпляр объекта имеет некое **состояние**. Объект при этом «знает» только то, что делает экземпляр одного класса отличным от другого экземпляра этого же класса.

2 Сериализованный объект

При сериализации **полностью сохраняется состояние объекта**, поэтому каждый экземпляр может быть возвращен в кучу.



Объект снова в куче

3 И затем…

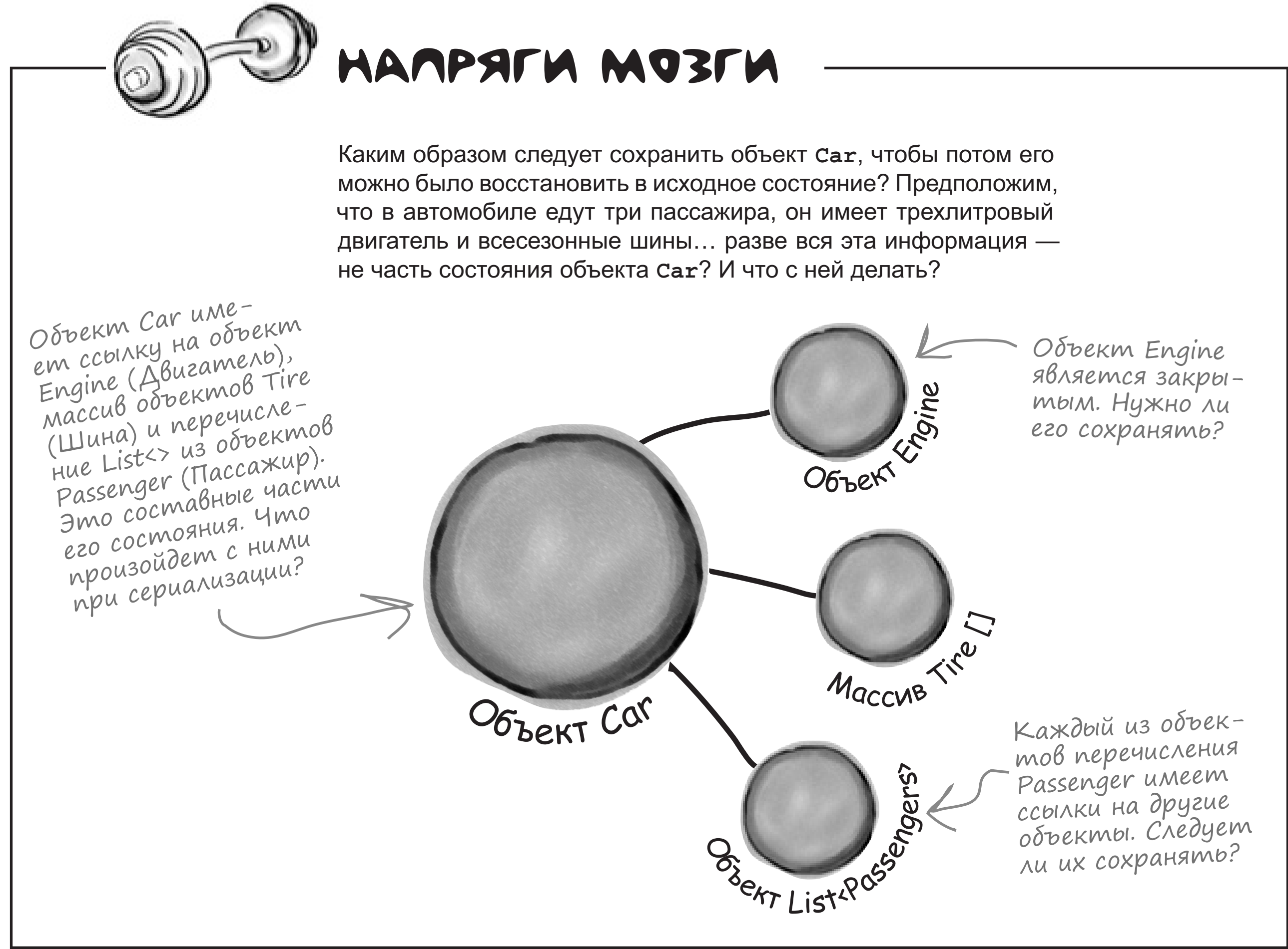
Затем, может быть, через много дней и в другой программе, вы можете провести **десериализацию** объекта. Исходный класс будет восстановлен из файла **в такое же состояние**, в котором он был, со всеми его полями и значениями.

Что именно мы сохраняем

Вы уже знаете, что **объект хранит информацию в полях**. Соответственно при сериализации нужно сохранить каждое из полей.

Чем сложнее, объект, тем сложнее его сериализация. Переменные разных типов — chars, ints, doubles и др. — можно записать в файл без изменений. А как быть, если в составе объекта присутствует *ссылка*? А пять ссылок? А что, если эти ссылки в свою очередь ссылаются на другие ссылки?

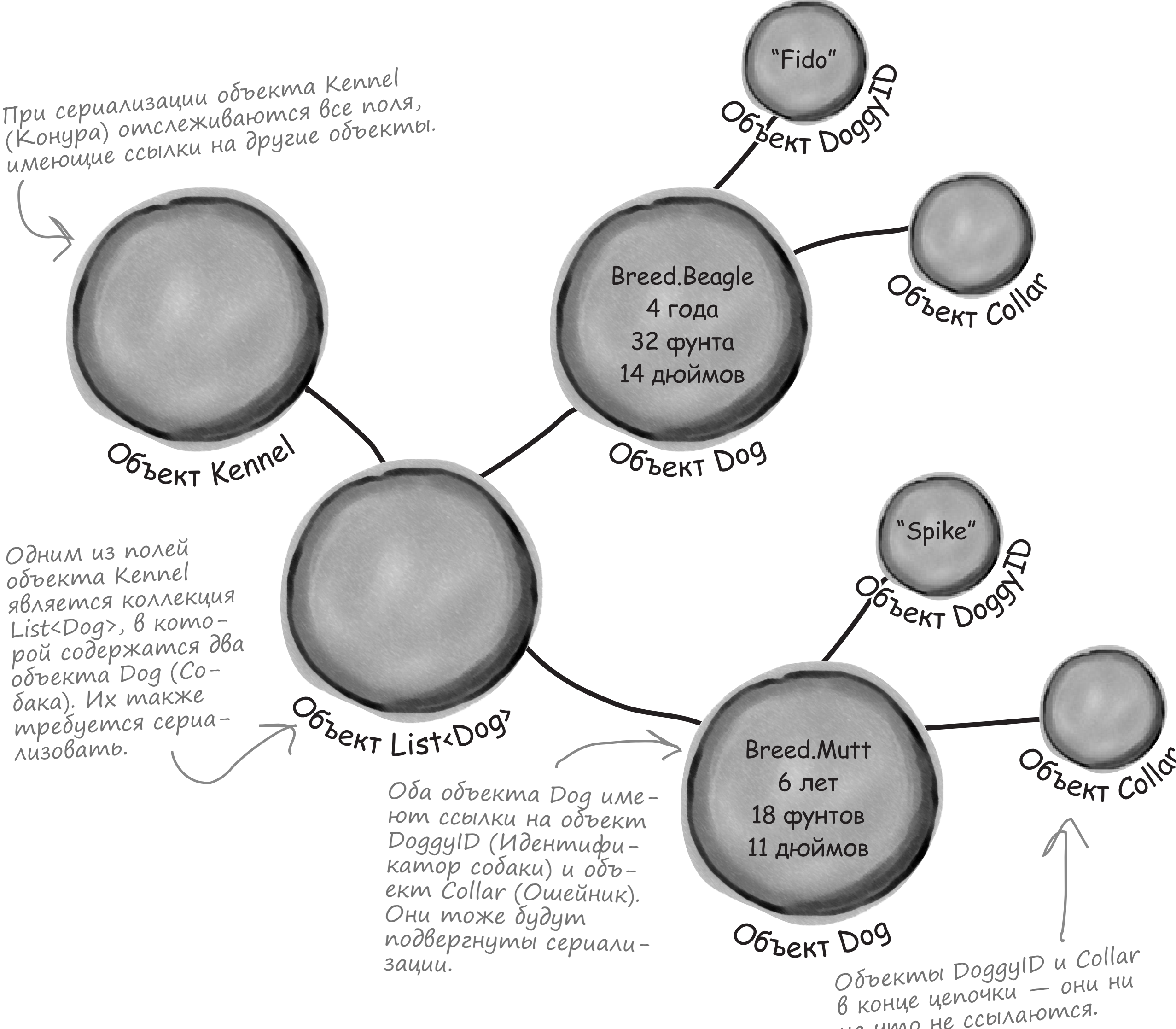
Подумайте об этом. Какая часть объекта является потенциально уникальной? Представьте, что именно нужно восстановить, чтобы получить объект, который был сохранен. Так или иначе, но все содержимое кучи нужно записывать в файл.



Сериализации подвергается не только сам объект...

...но и объекты, *на которые* он ссылается, и даже все объекты, *на которые ссылаются эти объекты* и т. д. Не волнуйтесь, это звучит сложно, но вся процедура производится автоматически. C# начинает с объекта, который вы хотите сериализовать, и смотрит на его поля в поисках связанных объектов. Найдя такой объект, он повторяет аналогичную операцию. Это позволяет сохранить всю нужную информацию.

Подобные группы связанных объектов иногда называют «графами»



Сериализация позволяет читать и записывать объект целиком

В файлы можно записывать не только строчки текста. Процедура **сериализации** позволяет копировать в файл целые объекты и читать их оттуда... а ведь это всего несколько строчек кода! Вам потребуется провести небольшую подготовительную работу — добавить строчку `[Serializable]` в верхнюю часть сериализуемого класса. После этого все будет готово к записи.

Чтение и запись объектов в файл осуществляется быстро. Просто сериализуйте или дисериализуйте объект.

Объект BinaryFormatter

Сериализация *любого* объекта начинается с создания экземпляра объекта `BinaryFormatter`. Это очень просто. Достаточно добавить еще одну строку `using` в верхнюю часть класса:

```
using System.Runtime.Serialization.Formatters.Binary;
...
BinaryFormatter formatter = new BinaryFormatter();
```

Осталось создать поток и приступить к чтению и записи объектов

Метод `Serialize()` объекта `BinaryFormatter` записывает объекты в поток.

Метод File.Create() создает новые файлы. Для открытия существующего файла используйте метод File.OpenWrite().

```
using (Stream output = File.Create(filenameString)) {
   formatter.Serialize(output, objectToSerialize);
}
```

Метод Serialize() берет объект и записывает его в поток. Вы избавлены от необходимости делать это вручную!

Чтобы прочитать сериализованный объект, используйте метод `Deserialize()` объекта `BinaryFormatter`. Он возвращает ссылку, тип которой может не совпадать с переменной, предназначенной для копирования. В этом случае используйте приведение.

```
using (Stream input = File.OpenRead(filenameString)) {
   SomeObj obj = (SomeObj)formatter.Deserialize(input);
}
```

Используя метод Deserialize() для чтения объекта из потока, не забывайте осуществить приведение возвращаемого значения к типу читаемого объекта.

Атрибут [Serializable]

> **Атрибуты — это способ добавить информацию к объявлению класса или члена. Атрибут `[Serializable]` находится в пространстве имен `System`.**

Атрибутом называется специальный тег, добавляемый в верхнюю часть классов. С помощью атрибутов в C# сохраняются **метаданные**, то есть информация о том, как нужно использовать код. **Добавив** атрибут **`[Serializable]` над объявлением класса**, вы показываете, что этот класс можно сериализовать. Подобное допустимо, скажем, для классов с полями значимых типов (например, `int`, `decimal` или `enum`). Отсутствие этого атрибута, как и наличие в классе полей, сериализация которых невозможна, приводит к сообщению об ошибке. ***Убедитесь в этом сами...***

Упражнение!

Создадим и сериализуем класс.

Давайте сериализуем Джо, чтобы сведения о его наличности остались в файле. Откройте проект «Парни» из главы 3 и обновите класс Guy:

```
[Serializable]
class Guy {
```

Этот атрибут должен присутствовать в верхней части любого класса, который вы собираетесь сериализовать.

Добавим на форму кнопки «Save Joe» и «Load Joe». Это код обработчиков событий кнопок, который сериализует объект Joe в файл *Guy_file.dat* и читает его из файла:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
...

private void saveJoe_Click(object sender, EventArgs e)
{
    using (Stream output = File.Create("Guy_File.dat")) {
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Serialize(output, joe);
    }
}
private void loadJoe_Click(object sender, EventArgs e)
{
    using (Stream input = File.OpenRead("Guy_File.dat")) {
        BinaryFormatter formatter = new BinaryFormatter();
        joe = (Guy)formatter.Deserialize(input);
    }
    UpdateForm();
}
```

Эти две строки using обязательны. Первая указывает, в каком пространстве имен происходят процедуры ввода и ввода, вторая делает допустимой процедуру сериализации.

Fun with Joe and Bob
Joe has $50
Bob has $100
The bank has $100
Give $10 to Joe | Receive $5 from Bob
Joe gives $10 to Bob | Bob gives $5 to Joe
Save Joe | Load Joe

2 **Запуск и тестирование программы**

Если Джо в результате обменных операции с Бобом получил 200 долларов, вряд ли он захочет потерять их при выходе из программы. Теперь Джо может сохранить свои капиталы в файл и восстановить их в любой момент.

Что произойдет, если удалить файл Guy_File.dat из папки bin/Debug и щелкнуть на Load Joe?

Сериализуем и десериализуем колоду карт

Давайте еще раз запишем в файл колоду карт. Благодаря сериализации эта задача значительно упростилась. Вам нужно всего лишь создать поток и записать в него объекты.

1 **Создание нового проекта**

Щелкните правой кнопкой мыши на имени нового проекта в окне Solution Explorer, выберите команду Add/Existing Item и добавьте классы `Card` и `Deck` (а также перечисления `Suits` и `Values` и интерфейсы `CardComparer_bySuit` и `CardComparer_byValue`), которые вы использовали в главе 8. Вам также потребуются два класса, сравнивающих карты. Не забудьте отредактировать строку namespace после того, как все файлы будут скопированы в новый проект.

2 **Добавление атрибута**

Добавьте атрибут `[Serializable]` к обоим классам, скопированым в проект.

Без этого сериализация невозможна.

3 **Добавление методов к форме**

Метод `RandomDeck` случайным образом создает колоду карт, а метод `DealCards` раздает их и выводит результат на консоль.

Создается пустая колода, в которую случайным образом добавляются карты из класса Card, созданного в предыдущей главе.

```
Random random = new Random();
private Deck RandomDeck(int number) {
    Deck myDeck = new Deck(new Card[] { });
    for (int i = 0; i < number; i++)
    {
        myDeck.Add(new Card(
            (Suits)random.Next(4),
            (Values)random.Next(1, 14)));
    }
    return myDeck;
}

private void DealCards(Deck deckToDeal, string title) {
    Console.WriteLine(title);
    while (deckToDeal.Count > 0)
    {
        Card nextCard = deckToDeal.Deal(0);
        Console.WriteLine(nextCard.Name);
    }
    Console.WriteLine("------------------");
}
```

Метод DealCards() раздает карты из колоды и выводит результат на консоль.

Не забудьте открыть окно Ouptut для просмотра информации, выводимой на консоль программой WinForms.

4 **Предварительная подготовка закончена... сериализуем колоду**

Добавьте кнопки, управляющие записью и чтением колоды. Сверяйтесь с результатами на консоли. Колода, которую вы записываете в файл, должна совпадать с колодой, которую вы читаете.

Вернитесь на страницу назад, чтобы увидеть операторы using, которые нужно добавить в форму.

```
private void button1_Click(object sender, EventArgs e) {
    Deck deckToWrite = RandomDeck(5);
    using (Stream output = File.Create("Deck1.dat")) {
      BinaryFormatter bf = new BinaryFormatter();
      bf.Serialize(output, deckToWrite);
    }
    DealCards(deckToWrite, "Что было записано в файл");
}
```

Объект BinaryFormatter берет объекты, помеченные атрибутом Serializable, — в нашем случае это объект Deck — и записывает их в поток с помощью метода Serialize().

```
private void button2_Click(object sender, EventArgs e) {
    using (Stream input = File.OpenRead("Deck1.dat")) {
      BinaryFormatter bf = new BinaryFormatter();
      Deck deckFromFile = (Deck)bf.Deserialize(input);
      DealCards(deckFromFile, "Что было прочитано из файла");
   }
}
```

Метод Deserialize() объекта BinaryFormatter возвращает объект обобщенного типа, поэтому требуется осуществить приведение к объекту Deck.

5 **Сериализация набора колод**

После открытия потока в него можно записывать произвольное количество информации. Поэтому добавим две кнопки, позволяющие сохранять в файл произвольное количество колод.

```
private void button3_Click(object sender, EventArgs e) {
    using (Stream output = File.Create("Deck2.dat")) {
      BinaryFormatter bf = new BinaryFormatter();
      for (int i = 1; i <= 5; i++) {
        Deck deckToWrite = RandomDeck(random.Next(1,10));
        bf.Serialize(output, deckToWrite);
        DealCards(deckToWrite, "Колода #" + i + " записана");
      }
    }
}
```

В один поток можно сериализовать несколько объектов.

Обратите внимание, как в строчке, читающей информацию из файла, осуществляется приведение результатов работы метода Deserialize() к объекту Deck.

```
private void button4_Click(object sender, EventArgs e) {
    using (Stream input = File.OpenRead("Deck2.dat")) {
      BinaryFormatter bf = new BinaryFormatter();
      for (int i = 1; i <= 5; i++) {
          Deck deckToRead = (Deck)bf.Deserialize(input);
          DealCards(deckToRead, "Колода #" + i + " прочитана");
      }
    }
}
```

Число сериализуемых объектов может быть произвольным, главное — не забывать приводить читаемые из потока объекты к нужному типу.

6 **Просмотр результата**

Откройте файл `Deck1.dat` в приложении Блокнот. Вы найдете там всю информацию, необходимую для восстановления объекта `Deck`.

Мне не нравится, что когда я открываю файл, в который был записан объект, я вижу какой-то мусор. После записи колоды в виде набора строк я открывала файл в приложении Блокнот и спокойно его читала. Мне казалось, что в C# я должна без проблем понимать все, что делаю.

Сериализованные объекты записываются в двоичном формате.

Они компактны. Поэтому вы можете распознать строки, открыв файл с сериализованным объектом: ведь наиболее компактный способ их записи в файл — именно в виде строк. Но писать в таком виде числа не имеет смысла. Любое число типа `int` можно сохранить в четырех байтах. Поэтому было бы странно хранить, к примеру, число 49 369 144 как 8-символьную строку, удобную для чтения.

Чуть позднее мы познакомим вас с более длинным, но более читабельным (и редактируемым!) форматом сериализации.

За сценой

Для представления символов или строк в виде байтов в .NET используется Юникод. К счастью, в Windows имеется инструмент, позволяющий понять принцип работы Юникода. Откройте приложение Character Map (выберите в меню Start команду Run, введите «charmap.exe» и щелкните на кнопке OK).

При взгляде на символы из разных языков становится понятно, сколько информации требуется записать в файл для сохранения текста. Поэтому .NET **преобразует** все строки и символы в формат Юникод. Берутся данные (например, буква H) и преобразуются в байты (число 72). Ведь буквы, цифры, перечисления и другие данные хранятся в памяти именно в виде байтов. Узнать соответствие между числами и символами можно в Таблице символов (Character Map).

Выберите в списке шрифтов Segoe UI и прокрутите до еврейского алфавита. Щелчком выберите символ Shin.

После выбора символа в строке состояния появится его код. Для буквы Shin — это число 05E9 в шестнадцатеричной системе счисления.

Для преобразования полученного значения в десятичную систему воспользуйтесь калькулятором Windows в режиме Scientific.

Character Map

Font : Segoe UI — Help

Characters to copy : — Select — Copy

Advanced view

U+05E9: Hebrew Letter Shin

Юникод — это индустриальный стандарт. Вы можете посетить сайт **http://unicode.org/**

.NET использует Unicode для хранения символов и текста

Типы, хранящие текст и символы, — `string` и `char` — в памяти хранят информацию в Юникоде. При записи данных в файл сохраняются символы Юникод. Создайте новый проект, постройте форму с тремя кнопками, чтобы посмотреть на работу методов `File.WriteAllBytes()` и `ReadAllBytes()` и понять, как именно осуществляется запись в файл.

1. **Запишем в файл обычную строку и прочитаем ее.**
Воспользуйтесь методом `WriteAllText()`, чтобы заставить первую кнопку записывать строку «Eureka!» в файл «eureka.txt». Затем создайте массив байтов `eurekaBytes`, считайте в него байты из файла и выведите полученный результат:

```
File.WriteAllText("eureka.txt", "Eureka!");
byte[] eurekaBytes = File.ReadAllBytes("eureka.txt");
foreach (byte b in eurekaBytes)
    Console.Write("{0} ", b);
Console.WriteLine();
```

Метод ReadAllBytes() возвращает ссылку на новый массив, содержащий байты, прочитанные из файла.

На консоль будет выведено: `69 117 114 101 107 97 33`. Но если **открыть файл в приложении Simple Text Editor**, вы увидите строку «Eureka!»

2. **Пусть вторая кнопка отображает байты в шестнадцатеричной системе.**
Числа в этой системе показываются не только в приложении Character Map, поэтому имеет смысл научиться с ними работать. Код обработчика событий для второй кнопки должен отличаться методом `Console.Write()`, для которого напишите:

```
    Console.Write("{0:x2} ", b);
```

В шестнадцатеричной системе используются числа от 0 до 9 и буквы от A до F. Так, 6B равно 107.

В итоге метод `Write()` будет выводить параметр 0 (первый после выводимой строки) в виде кода. Поэтому вы увидите семь байтов: `45 75 72 65 6b 61 21`

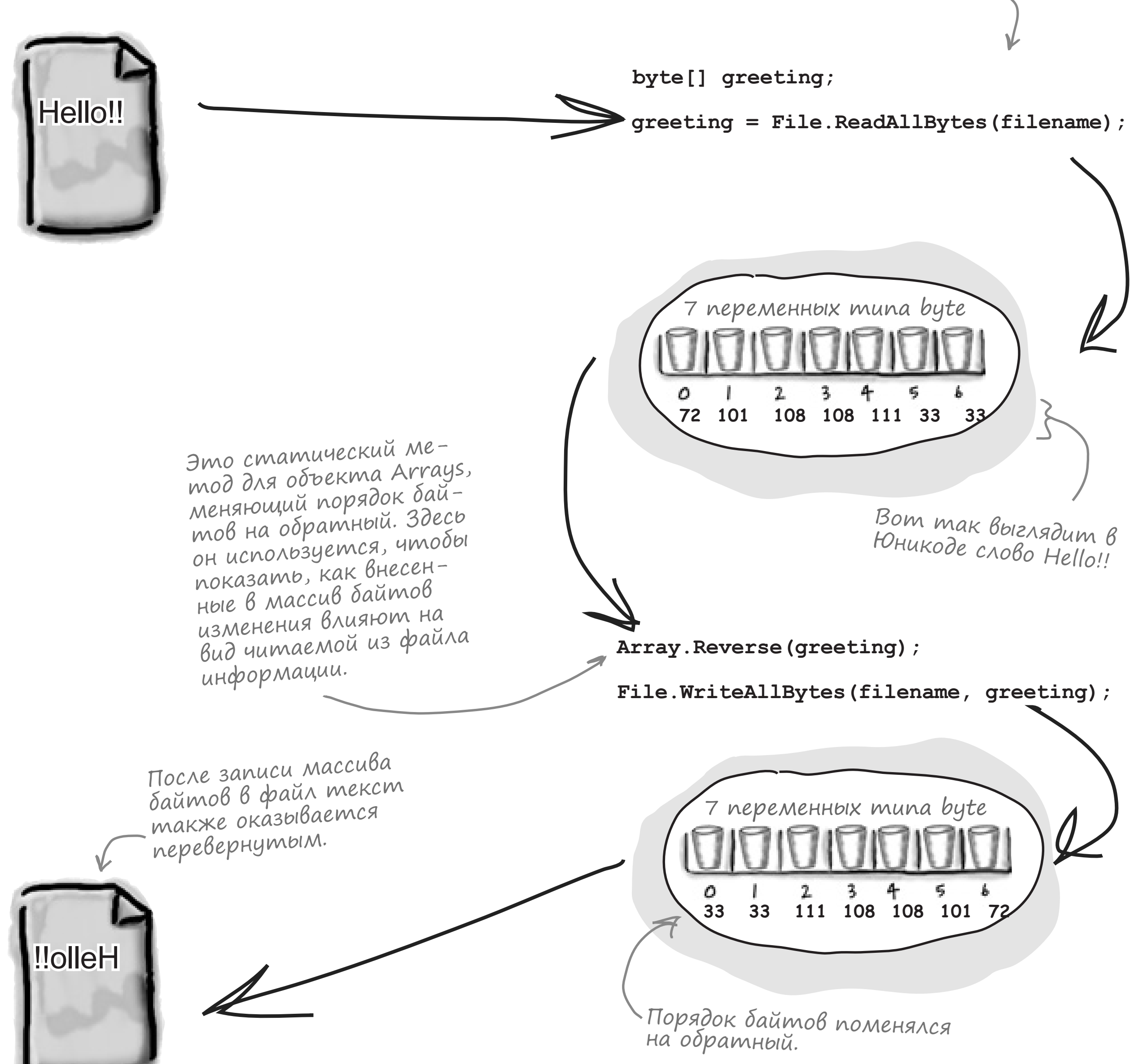
А третья кнопка должна выводить буквы еврейского алфавита
Вернитесь в Таблицу символов (Character Map) и дважды щелкните на символе Shin, чтобы добавить его в поле. Проделайте это для остальных символов в слове «Shalom»: Lamed (U+05DC), Vav (U+05D5) и Mem (U+05DD). В код обработчика событий для третьей кнопки добавьте скопированные буквы и добавьте параметр `Encoding.Unicode`:

```
File.WriteAllText("eureka.txt", "שלום", Encoding.Unicode);
```

Обратили внимание на **обратный порядок букв** после вставки? Дело в том, что еврейские слова читаются справа налево. Запустите код и посмотрите на полученный результат: `ff fe e9 05 dc 05 d5 05 dd 05`. Первые два символа — «FF FE», что означает строку из двухбайтовых символов. Остальные байты представляют собой буквы еврейского алфавита, — но перевернутые: так, U+05E9 будет показано как **`e9 05`**. Но если открыть файл в обычном текстовом редакторе, все будет выглядеть правильно!

Перемещение данных внутри массива байтов

Так как все данные в итоге перекодируются в **байты**, файл имеет смысл представить в виде **большого массива байтов**. Способы чтения таких массивов и записи в них вы уже знаете.



Изменение порядка следования байтов в слове Hello!! работает, так как каждый символ занимает всего один байт. Можете ли вы объяснить, почему это не сработает для слова שלום?

Класс BinaryWriter

Объект StreamWriter так же зашифровывает данные, но он работает только с текстом.

Данные различных типов перед записью в файл *можно* преобразовывать в массивы байтов, но это крайне рутинная работа. Поэтому в .NET появился класс **BinaryWriter**, **автоматически преобразующий данные** и записывающий их в файл. Вам нужно только создать объект FileStream и передать его конструктору класса BinaryWriter. После чего вызывается метод записи данных. Создадим новое приложение Console Application, использующее класс BinaryWriter для записи двоичных данных в файл.

1 Создайте консольное приложение и укажите данные для записи в файл.

```
int intValue = 48769414;
string stringValue = "Hello!";
byte[] byteArray = { 47, 129, 0, 116 };
float floatValue = 491.695F;
char charValue = 'E';
```

Метод File.Create() создает новый файл даже при наличии уже существующего. А метод File.OpenWrite() открывает имеющийся файл и записывает новую информацию поверх старой.

2 При помощи метода File.Create() откроем новый поток:

```
using (FileStream output = File.Create("binarydata.dat"))
using (BinaryWriter writer = new BinaryWriter(output)) {
```

3 При каждом вызове метода Write() в конец файла, в который осуществляется запись данных, будет добавляться байт.

```
  writer.Write(intValue);
  writer.Write(stringValue);
  writer.Write(byteArray);
  writer.Write(floatValue);
  writer.Write(charValue);
}
```

Каждый оператор Write() преобразует в байты одно значение, которое отправляется в объект FileStream. Любой значимый тип будет преобразован автоматически.

Объект FileStream записывает байты до конца файла.

Возьми в руку карандаш

4 Используем написанный ранее код для чтения только что записанного.

```
byte[] dataWritten = File.ReadAllBytes("binarydata.dat");
foreach (byte b in dataWritten)
    Console.Write("{0:x2} ", b);
Console.WriteLine(" - {0} bytes", dataWritten.Length);

Console.ReadKey();
```

Подсказка: каждая строка начинается с числа, указывающего на ее длину. Кроме того, вы можете воспользоваться Таблицей символов для поиска соответствий между строковыми символами и значениями.

Напишите результат в свободные поля внизу. Вы понимаете, **какие байты соответствуют** каждому из пяти операторов Write()? Пометьте каждую группу байтов именем переменной.

__ - ___ bytes

Значения типов float и int занимают по 4 байта, в то время как типы long и double требуют по 8 байт.

Возьми в руку карандаш
Решение

86 29 e8 02 06 48 65 6c 6c 6f 21 2f 81 00 74 f6 d8 f5 43 45 - 20 bytes

intValue | stringValue | byteArray | floatValue | charValue

Первый байт — это длина строки. Посмотреть соответствие вы можете в Таблице символов. Слово «Hello!» начинается с U+0048 и заканчивается U+0021.

Воспользовавшись калькулятором Windows для преобразования значений из шестнадцатеричной системы в десятичную, вы обнаружите числа, составляющие массив.

Переменная типа char 'E' занимает всего один байт, он соответствует символу U+0045.

Класс BinaryReader

Класс BinaryReader работает аналогично классу BinaryWriter. Вы создаете поток, присоединяете к нему объект BinaryReader и вызываете его методы. Но программа для чтения **не знает, что за данные содержатся в файле**. И не может узнать. Значение типа float 491.695F преобразовалось в d8 f5 43 45. Но эти же байты подходят к значению типа int — 1 140 185 334. Поэтому следует в явном виде указывать тип читаемого значения. Добавьте к форме следующий код и займемся чтением только что записанных данных:

Не верьте нам на слово. Замените строку для чтения типа float вызовом метода ReadInt32(). (Вам потребуется заменить тип floatRead на int.) И сами увидите, что читается из файла.

1. Начнем с задания объектов FileStream и BinaryReader:

```
using (FileStream input = File.OpenRead("binarydata.dat"))
using (BinaryReader reader = new BinaryReader(input)) {
```

2. Укажите тип данных, который будет возвращать каждый из методов объекта BinaryReader.

```
int intRead = reader.ReadInt32();
string stringRead = reader.ReadString();
byte[] byteArrayRead = reader.ReadBytes(4);
float floatRead = reader.ReadSingle();
char charRead = reader.ReadChar();
```

У объекта BinaryReader() существует набор методов, каждый из которых возвращает данные определенного типа. Большинству из них не нужны параметры, хотя метод ReadBytes() использует параметр, сообщающий объекту BinaryReader, сколько байтов требуется прочитать.

3. Выведем результат чтения данных на консоль:

```
    Console.Write("int: {0}  string: {1}  bytes: ", intRead, stringRead);
    foreach (byte b in byteArrayRead)
        Console.Write("{0} ", b);
    Console.Write(" float: {0}  char: {1} ", floatRead, charRead);
}
Console.ReadKey();
```

Если вы добавили этот код в конец программы с предыдущей страницы, не забудьте еще один метод ReadKey(), ожидающий нажатия клавиши.

Вот как он будет выглядеть:

```
int: 48769414  string: Hello!  bytes: 47 129 0 116  float: 491.695  char: E
```

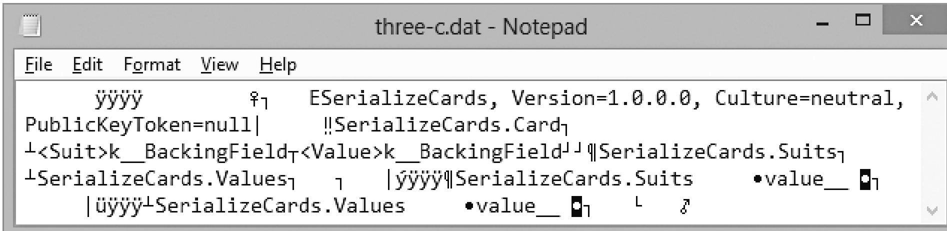
Чтение и запись сериализованных файлов вручную

Открытые в приложении Блокнот сериализованные файлы выглядят не очень красиво. Все записанные вами файлы находятся в папке bin\Debug. Давайте посмотрим на дополнительные приемы работы с ними.

Сериализуем два объекта Card в различные файлы

Используйте написанный ранее код, чтобы сериализовать **тройку крестей** в файл three-c.dat, а **шестерку червей** — в файл six-h.dat. Убедитесь, что оба файла находятся в одной папке и имеют одинаковый размер, и откройте их в Блокноте:

В файле встречаются знакомые слова, но по большей части он нечитаем.



Не забудьте про два оператора using!

Цикл, сравнивающий два двоичных файла

При чтении байтов из потока метод ReadByte() возвращает значение типа int. Поле Length потока позволяет убедиться, что файл прочитан полностью.

```
byte[] firstFile = File.ReadAllBytes("three-c.dat");
byte[] secondFile = File.ReadAllBytes("six-h.dat");
for (int i = 0; i < firstFile.Length; i++)
    if (firstFile[i] != secondFile[i])
        Console.WriteLine("Byte #{0}: {1} versus {2}",
            i, firstFile[i], secondFile[i]);
```

Так как эти файлы были прочитаны в разные массивы, мы имеем возможность сравнить их побайтно. В данном случае в два разных файла были сериализованы объекты одного класса, поэтому они должны быть практически идентичны... но давайте посмотрим, НАСКОЛЬКО они совпадают.

Этот цикл сравнивает первые байты из обоих файлов, потом вторые, потом третьи и т. д. Информация об обнаруженных различиях выводится на консоль.

Будьте осторожны!

Запись в файл не всегда осуществляется с чистого листа!

Будьте осторожны с методом File.OpenWrite(). Он не удаляет имеющийся файл, а начинает запись поверх уже записанной информации. Поэтому мы предпочли метод File.Create(), создающий новый файл.

Переверните страницу и продолжим!

Найдите отличия и отредактируйте файлы

Написанный нами цикл четко указывает на различия между двумя сериализованными файлами Card. Записанные объекты различаются полями Suit и Value, соответственно именно в этом будут различаться файлы. Найдя байты, содержащие информацию о масти и старшинстве карты, мы сможем **отредактировать их**, получив в итоге совершенно новую карту с нужными нам параметрами!

Возможна сериализация объектов в формат XML.

Рассмотрим выведенные на консоль результаты

Они показывают, чем различаются файлы:

```
Byte #307: 1 versus 3
Byte #364: 3 versus 6
```

Вернитесь к перечислению Suits в предыдущей главе, и вы увидите, что трефам соответствует значение 1, а червям — значение 3. Это первое различие. Второе различие, как легко догадаться, — старшинство карт. Различному количеству байтов также имеется объяснение: объекты могли находиться в разных пространствах имен, что изменило длину файла.

Помните, каким образом информация о пространстве имен включается в сериализованный файл? Если пространства имен различаются, размер байтов в сохраненном файле также будет отличаться.

Получается, что если байт #307 в сериализованном файле соответствует масти, мы сможем поменять масть карты, прочитав файл, изменив один байт и снова записав информацию в файл. (Имейте в виду, что в вашем сериализованном файле информация о масти может быть сохранена в другом месте.)

Вручную создадим новый файл с королем пик.

Отредактируем один из прочитанных массивов и снова запишем его в файл.

```
firstFile[307] = (byte)Suits.Spades;
firstFile[364] = (byte)Values.King;
File.Delete("king-s.dat");
File.WriteAllBytes("king-s.dat", firstFile);
```

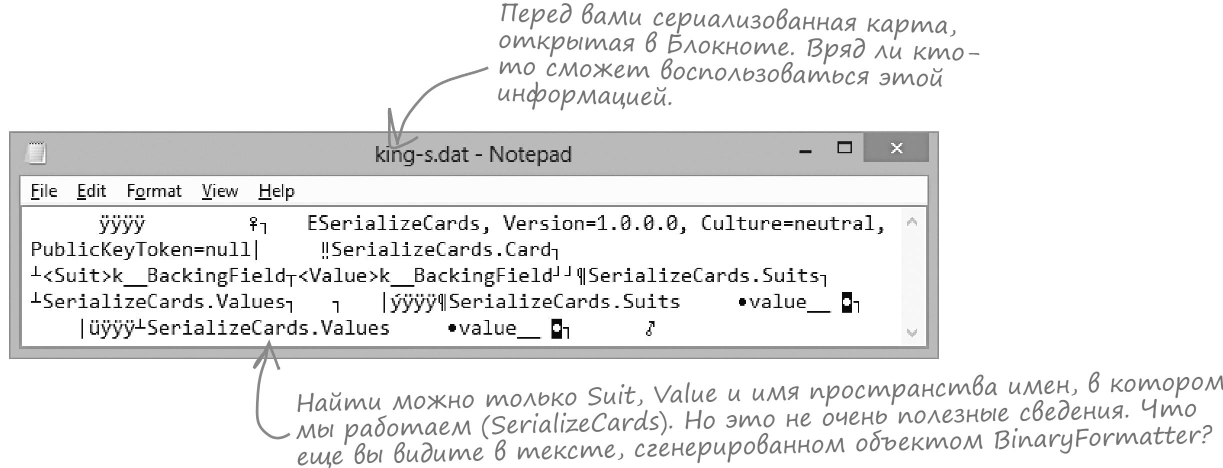
Если числа, которые вы обнаружили в своем файле, отличаются от наших, подставьте их сюда.

Теперь **десериализуйте файл king-s.dat** и убедитесь, что в нем содержится король пик!

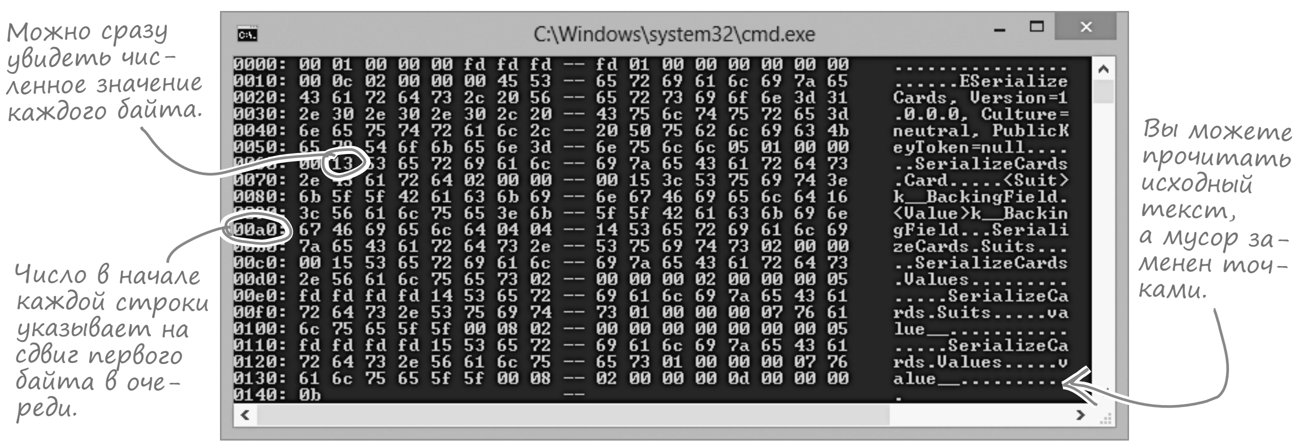
Зная, какие именно байты содержат информацию о масти и значении карты, мы редактируем эти байты перед записью в файл king-s.dat.

Сложности работы с двоичными файлами

Что делать, если вы не знаете точно, какая информация содержится в файле? Неизвестно даже, каким приложением этот файл был создан, а при открытии его в Блокноте вы видите только мусор. Ясно только, что Блокнот не слишком подходит для открытия такого рода файлов.



Существует такой формат, как дамп данных (hex dump), обычно используемый для просмотра двоичной информации. Он намного более информативен. Каждый байт представлен в виде двух символов в шестнадцатеричной системе, что позволяет сконцентрировать многочисленные данные в небольшом объеме. Двоичные данные также хорошо представлять в строках длиной по 8, 16 или 32 байта, так как они делятся именно на такие кусочки. К примеру, формат `int` занимает до 4 байт, соответственно именно такую длину он имеет после сериализации. Вот как наш файл выглядит в виде шестнадцатеричного дампа:



Программа для создания дампа

Дампом данных (hex dump) называется представление содержимого файла в *шестнадцатеричном виде*. Это представление часто используется для получения сведений о внутренней структуре файла. В большинстве операционных систем имеется встроенная программа для работы с дампом данных. К сожалению, Windows в их число не входит, поэтому создадим программу самостоятельно.

Создание дампа данных

Начнем с текста:

```
We the People of the United States, in Order to form a more perfect Union...
```

Вот как он будет выглядеть в дампе данных:

Вы можете видеть численное представление каждого байта.

```
0000: 57 65 20 74 68 65 20 50 -- 65 6f 70 6c 65 20 6f 66    We the People of
0010: 20 74 68 65 20 55 6e 69 -- 74 65 64 20 53 74 61 74     the United Stat
0020: 65 73 2c 20 69 6e 20 4f -- 72 64 65 72 20 74 6f 20    es, in Order to
0030: 66 6f 72 6d 20 61 20 6d -- 6f 72 65 20 70 65 72 66    form a more perf
0040: 65 63 74 20 55 6e 69 6f -- 6e 2e 2e 2e                ect Union...
```

Число в начало каждой строки добавляется, используя смещение первого байта.

Мусор был заменен точками.

Каждое из этих чисел — 57, 65, 6F — значение одного байта в файле в шестнадцатеричном представлении (hex). Если в десятичной системе вы используете числа от 0 до 9, то здесь к числам добавляются еще и буквы от A до F.

Каждая строчка в дампе данных представляет 16 выводимых символов, из которых она, собственно, и была создана. Первые четыре цифры в нашем случае — это смещение внутри файла: первая строчка начинается с 0, следующая — с 16 (в шестнадцатеричной системе это 10), следующая — с 32 и т. д. (Другие шестнадцатеричные дампы могут выглядеть по-другому.)

Работа в шестнадцатеричной системе счисления

Для ввода шестнадцатеричных чисел в программу достаточно добавить перед обычным числом символы 0x (ноль, а затем x):

```
int j = 0x20;
MessageBox.Show("The value is " + j);
```

Оператор +, преобразующий числа в строки, возвращает значение в десятичную систему счисления. Для преобразования к шестнадцатеричной системе пользуйтесь статическим методом String.Format():

```
string h = String.Format("{0:x2}", j);
```

Метод String. Format() использует в качестве параметра метод Console. WriteLine(), так что вы без проблем можете с ним работать.

Достаточно StreamReader и StreamWriter

Наша программа будет писать данные непосредственно в файл. Так как мы записываем всего лишь текст, воспользуемся объектом `StreamWriter`. Нам потребуется также метод **`ReadBlock()`** объекта `StreamReader`, читающий блоки символов в массив типа `char`; нужно только указать размер блока. Так как в строке мы отображаем 16 символов, он будет читать блоки именно такого размера.

Метод называется ReadBlock (), потому что при вызове он «блокируется» (то есть не возвращается в программу, а работает), пока не прочитает или все заказанные вами символы, или файл до конца.

Добавьте форме еще одну кнопку, с ней мы свяжем программу создания дампа данных. Измените маршруты доступа в первых двух строчках, чтобы они указывали на реальные файлы. Начните с сериализованного файла `Card`.

```
using (StreamReader reader = new StreamReader(@"c:\files\inputFile.txt"))
using (StreamWriter writer = new StreamWriter(@"c:\files\outputFile.txt", false))
{
  int position = 0;
  while (!reader.EndOfStream) {
     char[] buffer = new char[16];
     int charactersRead = reader.ReadBlock(buffer, 0, 16);
     writer.Write("{0}: ", String.Format("{0:x4}", position));
     position += charactersRead;
     for (int i = 0; i < 16; i++) {
         if (i < charactersRead) {
            string hex = String.Format("{0:x2}", (byte)buffer[i]);
            writer.Write(hex + " ");
         }
         else
            writer.Write("   ");
       if (i == 7) { writer.Write("-- "); }
       if (buffer[i] < 32 || buffer[i] > 250) { buffer[i] = '.'; }
     }
     string bufferContents = new string(buffer);
     writer.WriteLine("   " + bufferContents.Substring(0, charactersRead));
  }
}
```

Свойство EndOfStream объекта StreamReader возвращает значение false, пока остаются предназначенные для чтения символы.

*Этот метод **ReadBlock()** читает в массив типа char блоки размером до 16 символов.*

*Статический метод **String.Format** преобразует числа в строки. Запись «{0: x4}» указывает методу Format () вывести второй параметр, в нашем случае position, в виде 4-значного шестнадцатеричного числа.*

Этот цикл по очереди выводит все символы.

Некоторые символы со значением до 32 не выводятся, мы заменяем их точками.

Массив char[] можно преобразовать в строку, передав его перегруженному конструктору переменной string.

Метод Substring возвращает фрагмент строки. В данном случае он возвращает первые символы переменной charactersRead, отсчитывая их от начала (position 0). (Посмотрите на процедуру задания переменной charactersRead в цикле while — метод ReadBlock() возвращает в массив число прочитанных им символов.)

Чтение байтов из потока

С текстовыми файлами наша программа прекрасно работает. Но попробуйте воспользоваться методом `File.WriteAllBytes()` для записи в файл массива байтов со значениями больше 127 и посмотрите на дамп данных. Вы обнаружите только набор символов «fd»! Дело в том, что **объект `StreamReader` предназначен для чтения текстовых файлов**, содержащих байты со значениями до 128. Давайте прочитаем байты непосредственно из потока при помощи метода **`Stream.Read()`**. Мы получим практически реальное приложение для работы с дампом данных: вы сможете даже передавать ему имена файлов как **аргументы из командной строки**.

Создайте консольное приложение с именем **`hexdumper`**. Код для него находится на следующей странице, а вот как выглядит результат его работы:

Попытка запустить наше приложение без указания имени файла приводит к сообщению об ошибке.

При попытке передать приложению имя несуществующего файла появляется сообщение об ошибке.

C:\Windows\system32\cmd.exe

```
C:\Users\Public\Projects\HexDumper>HexDumper.exe
usage: HexDumper file-to-dump

C:\Users\Public\Projects\HexDumper>HexDumper.exe does-not-exist.dat
File does not exist: does-not-exist.dat

C:\Users\Public\Projects\HexDumper>HexDumper.exe three-c.dat
0000: 00 01 00 00 00 ff ff ff -- ff 01 00 00 00 00 00 00    ................
0010: 00 0c 02 00 00 00 45 53 -- 65 72 69 61 6c 69 7a 65    ......ESerialize
0020: 43 61 72 64 73 2c 20 56 -- 65 72 73 69 6f 6e 3d 31    Cards, Version=1
0030: 2e 30 2e 30 2e 30 2c 20 -- 43 75 6c 74 75 72 65 3d    .0.0.0, Culture=
0040: 6e 65 75 74 72 61 6c 2c -- 20 50 75 62 6c 69 63 4b    neutral, PublicK
0050: 65 79 54 6f 6b 65 6e 3d -- 6e 75 6c 6c 05 01 00 00    eyToken=null....
0060: 00 13 53 65 72 69 61 6c -- 69 7a 65 43 61 72 64 73    ..SerializeCards
0070: 2e 43 61 72 64 02 00 00 -- 00 15 3c 53 75 69 74 3e    .Card.....<Suit>
0080: 6b 5f 5f 42 61 63 6b 69 -- 6e 67 46 69 65 6c 64 16    k__BackingField.
0090: 3c 56 61 6c 75 65 3e 6b -- 5f 5f 42 61 63 6b 69 6e    <Value>k__Backin
00a0: 67 46 69 65 6c 64 04 04 -- 14 53 65 72 69 61 6c 69    gField...Seriali
00b0: 7a 65 43 61 72 64 73 2e -- 53 75 69 74 73 02 00 00    zeCards.Suits...
00c0: 00 15 53 65 72 69 61 6c -- 69 7a 65 43 61 72 64 73    ..SerializeCards
00d0: 2e 56 61 6c 75 65 73 02 -- 00 00 00 02 00 00 00 05    .Values.........
00e0: fd ff ff ff 14 53 65 72 -- 69 61 6c 69 7a 65 43 61    .....SerializeCa
00f0: 72 64 73 2e 53 75 69 74 -- 73 01 00 00 00 07 76 61    rds.Suits.....va
0100: 6c 75 65 5f 5f 00 08 02 -- 00 00 00 01 00 00 00 05    lue__...........
0110: fc ff ff ff 15 53 65 72 -- 69 61 6c 69 7a 65 43 61    .....SerializeCa
0120: 72 64 73 2e 56 61 6c 75 -- 65 73 01 00 00 00 07 76    rds.Values.....v
0130: 6                           02 00 00 00 03 00 00 00    alue__..........
0140: 0

C:\User
```

Дамп данных существующего файла выводится на консоль.

*Обычно вывод на консоль осуществляется методом Console.WriteLine(). Но мы воспользовались методом **Console.Error.WriteLine()** для вывода сообщений об ошибке.*

*Создание консольного приложения сопровождается построением класса Program с точкой входа, которая объявляется следующим образом: **static void Main(string[] args)**. В данном случае **args** и есть аргументы командной строки. Впрочем, аналогичные вещи вы найдете в любом приложении Windows Forms.*

Для передачи аргументов командной строки в отладчик выберите команду Properties... в меню Project и введите аргументы на вкладке Debug.

Для передачи аргументов командной строки используется параметр args.

Если свойство args.Length не равно 1, значит, в командную строку аргументы не передаются или, наоборот, передается более одного аргумента.

```
static void Main(string[] args)
{
    if (args.Length != 1)
    {
        Console.Error.WriteLine("usage: HexDumper file-to-dump");
        System.Environment.Exit(1);
    }
    if (!File.Exists(args[0]))
    {
        Console.Error.WriteLine("File does not exist: {0}", args[0]);
        System.Environment.Exit(2);
    }
    using (Stream input = File.OpenRead(args[0]))
    {
        int position = 0;
        byte[] buffer = new byte[16];
        while (position < input.Length)
        {
            int charactersRead = input.Read(buffer, 0, buffer.Length);
            if (charactersRead > 0)
            {
                Console.Write("{0}: ", String.Format("{0:x4}", position));
                position += charactersRead;

                for (int i = 0; i < 16; i++)
                {
                    if (i < charactersRead)
                    {
                        string hex = String.Format("{0:x2}", (byte)buffer[i]);
                        Console.Write(hex + " ");
                    }
                    else
                        Console.Write("   ");

                    if (i == 7)
                        Console.Write("-- ");

                    if (buffer[i] < 32 || buffer[i] > 250) { buffer[i] = (byte)'.'; }
                }
                string bufferContents = Encoding.UTF8.GetString(buffer);
                Console.WriteLine("   " + bufferContents);
            }
        }
    }
}
```

Обратите внимание, как здесь используется метод Console.Error.WriteLine().

Метод Exit() завершает программу. В ответ на передачу ему значения типа int он возвращает код ошибки (что полезно при написании командных сценариев и командных файлов).

Если переданного файла не существует, выводится другое сообщение и возвращается другой код ошибки.

Объект StreamReader нам не требуется, так как байты читаются непосредственно из потока.

Метод Stream.Read() читает байты в буфер. В данном случае роль буфера играет массив типа byte, ведь мы читаем байты из текстового файла.

Эта часть программы осталась неизменной, просто буфер теперь содержит байты, а не символы (но метод String.Format() работает в любом случае).

Если запустить консольное приложение командой Start Without Debugging (Ctrl-F5) из меню Debug, после завершения появится приглашение «Press any key to continue...”» (Чтобы продолжить, нажмите любую клавишу).

Это простой способ преобразования массива байтов в строку. Кодировка Encoding.UTF8 указана в явном виде, так как в различных кодировках один и тот же массив байтов даст разные строки.

Часто задаваемые вопросы

В: Почему после методов `File.ReadAllText()` и `File.WriteAllText()` я не пользуюсь методом `Close()`?

О: В классе `File` присутствуют несколько статических методов, которые автоматически открывают файл, читают или пишут данные и затем **автоматически закрывают его**. Вдобавок к уже упомянутым сюда относятся методы `ReadAllBytes()` и `WriteAllBytes()`, работающие с массивами байтов, а также методы `ReadAllLines()` и `WriteAllLines()`, читающие строковые массивы, в которых каждая переменная становится новой строкой в файле. Все эти методы автоматически открывают и закрывают потоки, поэтому дополнительные операторы вам не требуются.

В: Если в классе `FileStream` имеются методы чтения и записи, 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, например наша первая версия программы Excuse Generator. Там мы использовали методы классов `StreamReader` и `StreamWriter`, созданные специально для работы со строками текста. Без них вам приходилось бы сначала читать массив байтов и писать цикл, ищущий в этом массиве знаки переноса строки, так что иногда эти классы сильно облегчают жизнь.

В: Когда используется класс `File`, а когда класс `FileInfo`?

О: Основным их различием является тот факт, что методы класса `File` являются статическими, поэтому вам не нужно создавать их экземпляры. А класс `FileInfo` требует создания экземпляра с указанием имени файла. Его не имеет смысла использовать для единичных операций (например, удаления или перемещения одного файла). Но для многочисленных операций с одним файлом он более эффективен, так как вам достаточно один раз передать имя этого файла. Выбор класса зависит только от конкретной ситуации. Грубо говоря, для единичных операции выбираем класс `File`, а для последовательного редактирования — класс `FileInfo`.

В: Почему символы в слове «Eureka!» записываются как единичные байты, в то время как для записи букв еврейского алфавита требуется по два байта? И что это за символы FF FE в начале?

О: Вы увидели разницу между двумя близко связанными кодировками Юникод. Латинским буквам, цифрам, знакам препинания, а также ряду стандартных символов (фигурным скобкам, амперсандам и другим элементам вашей клавиатуры) соответствуют небольшие значения Юникод — от 0 до 127. (Аналогичная ситуация с символами кодировки ASCII.) Если файл содержит только такие символы, они выводятся в виде одиночных байтов.

Все усложняется, когда на сцену выходят символы с большими значениями. Ведь байт может иметь значение от 0 до 255, и не больше. А вот в двух байтах уже можно хранить числа от 0 до 65,536 — в шестнадцатеричной системе счисления это FFFF. Чтобы сообщить программе, которая будет открывать файл, о наличии символов с высокими значениями в начало файла помещается зарезервированная последовательность символов: FF FE. Это так называемая «отметка порядка байтов». Находя ее, программа узнает, что все символы закодированы в двух байтах.

В: Почему эти символы называются отметкой порядка байтов?

О: Помните, мы меняли порядок байтов? Значение буквы Shin в Юникод U+05E9 было записано в файл как E9 05. Вернитесь к коду, записывающему эти байты, и поменяйте третий параметр на `WriteAllText(): Encoding.BigEndianUnicode`. Порядок следования байтов станет прямым 05 E9. Изменится и отметка порядка байтов: FE FF. Кстати, написанное вами приложение Simple Text Editor может читать как в прямом, так и в обратном порядке!

Символы с небольшими значениями Юникод записываются по одному в байт, в то время как для записи символов Юникод с большими значениями выделяются два байта.

Это используемая в .NET по умолчанию кодировка называется UTF-8. Для смены кодировки нужно передать методу File.WriteAllText() другое значение.

Отредактируем программу Брайана Excuse Manager таким образом, чтобы она начала работать с двоичными файлами, содержащими сериализованные объекты `Excuse`.

1. **Сериализация класса `Excuse`**
   Пометьте класс `Excuse` атрибутом `[Serializable]`. Потребуется также добавить строчку `using`:
   `using System.Runtime.Serialization.Formatters.Binary;`

2. **Отредактируйте метод `Excuse.Save()`**
   Теперь метод `Save()` вместо использования объекта `StreamWriter` должен открывать файл и сериализовывать его. Вам нужно понять, каким образом происходит десериализация текущего класса.

   Подсказка: используйте внутри класса ключевое слово, которое возвращает ссылку на этот же класс.

3. **Отредактируйте метод `Excuse.OpenFile()`**
   Вам потребуется временный объект `Excuse`, в который будет происходить десериализация, после чего его поля скопируются в текущий класс.

4. **Отредактируйте форму**
   Мы больше не работаем с текстовыми файлами, поэтому расширение `.txt` не подходит. Измените окна диалога, заданные по умолчанию имена файлов и код поиска папки, подставив туда разрешение `*.excuse`.

Потрясающе! Весь код для сохранения и открытия оправданий находится внутри класса Excuse. Изменения вносились в класс, форму почти не трогали. И форма работает вне зависимости от того, как класс сохраняет данные. Она просто передает имя файла, и информация сохраняется.

Разумеется! Код так легко редактируется благодаря инкапсуляции класса.

Класс, в котором все внутренние операции скрыты от остальной части программы, а открытым является только то, что необходимо, называется **инкапсулированным**. В программе Excuse Manager форма не имеет информации о том, каким образом оправдания сохраняются в файл. Она всего лишь передает имя файла классу, а уж он делает все необходимое. Именно поэтому вы смогли так легко отредактировать способ работы класса с файлами. Чем лучше инкапсулирован класс, тем проще вам работать с ним в будущем.

Помните, что инкапсуляция является одним из четырех основных признаков объектно-ориентированного программирования? Вот наглядный пример того, каким образом она облегчает нам жизнь.

Упражнение Решение

Вот как выглядит код программы Excuse Manager после редактирования.

В форме требуется отредактировать только три оператора: два в обработчике событий кнопки Save и один для кнопки Open, они меняют расширение файлов и задают имя, под которым файлы сохраняются по умолчанию.

```
private void save_Click(object sender, EventArgs e) {
  // существующий код
  saveFileDialog1.Filter = "Excuse files (*.excuse)|*.excuse|All files (*.*)|*.*";
  saveFileDialog1.FileName = description.Text + ".excuse";
  // существующий код
}
private void open_Click(object sender, EventArgs e) {
  // существующий код
  openFileDialog1.Filter =
    "Excuse files (*.excuse)|*.excuse|All files (*.*)|*.*";
  // существующий код
}
```

Здесь используются стандартные окна диалога Save и Open.

В форме мы поменяли только расширение файлов, которые она передает в класс Excuse.

Это код всего класса Excuse.

```
[Serializable]
class Excuse {
    public string Description { get; set; }
    public string Results { get; set; }
    public DateTime LastUsed { get; set; }
    public string ExcusePath { get; set; }
    public Excuse() {
        ExcusePath = "";
    }
    public Excuse(string excusePath) {
        OpenFile(excusePath);
    }
    public Excuse(Random random, string folder) {
        string[] fileNames = Directory.GetFiles(folder, "*.excuse");
        OpenFile(fileNames[random.Next(fileNames.Length)]);
    }
    private void OpenFile(string excusePath) {
        this.ExcusePath = excusePath;
        BinaryFormatter formatter = new BinaryFormatter();
        Excuse tempExcuse;
        using (Stream input = File.OpenRead(excusePath)) {
            tempExcuse = (Excuse)formatter.Deserialize(input);
        }
        Description = tempExcuse.Description;
        Results = tempExcuse.Results;
        LastUsed = tempExcuse.LastUsed;
    }
    public void Save(string fileName) {
        BinaryFormatter formatter = new BinaryFormatter();
        using (Stream output = File.OpenWrite(fileName)) {
            formatter.Serialize(output, this);
        }
    }
}
```

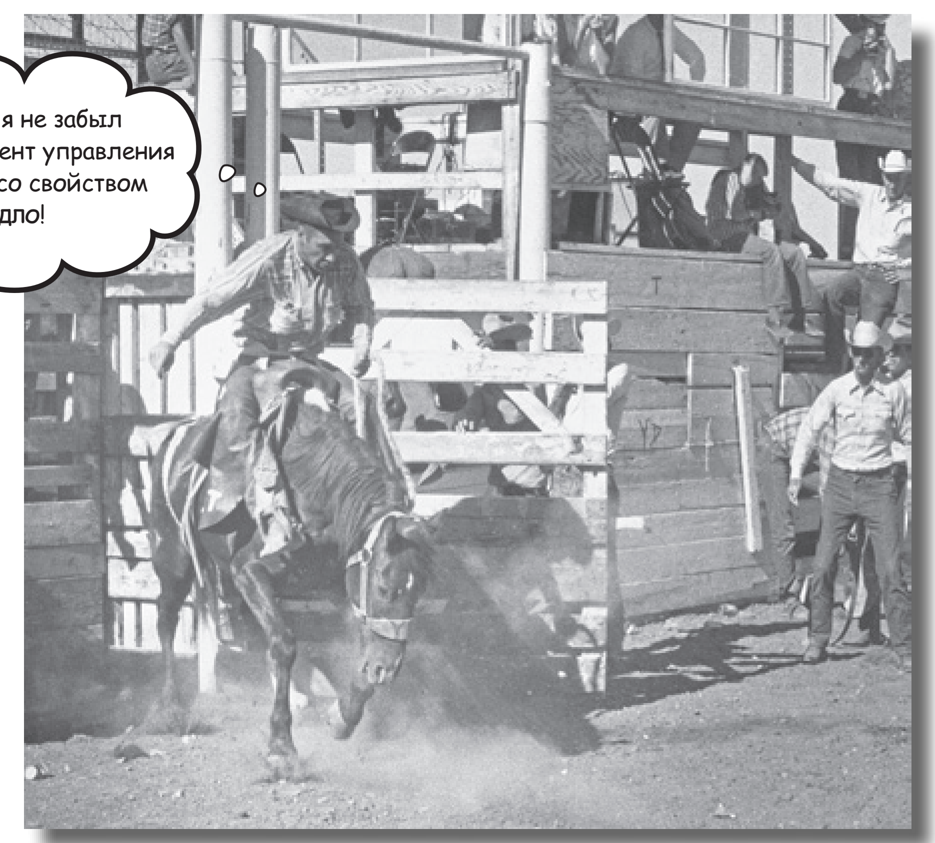
В классе `Excuse` **вам понадобятся строки** `using System.IO;` **и** `using System.Runtime.Serialization.Formatters.Binary;`.

*Конструктор, загружающий случайные оправдания, теперь ищет расширение .excuse вместо расширения *.txt.*

Ключевое слово this тут фигурирует, так как требуется сериализовать именно этот класс.

10 Приложения для магазина windows на языке xaml

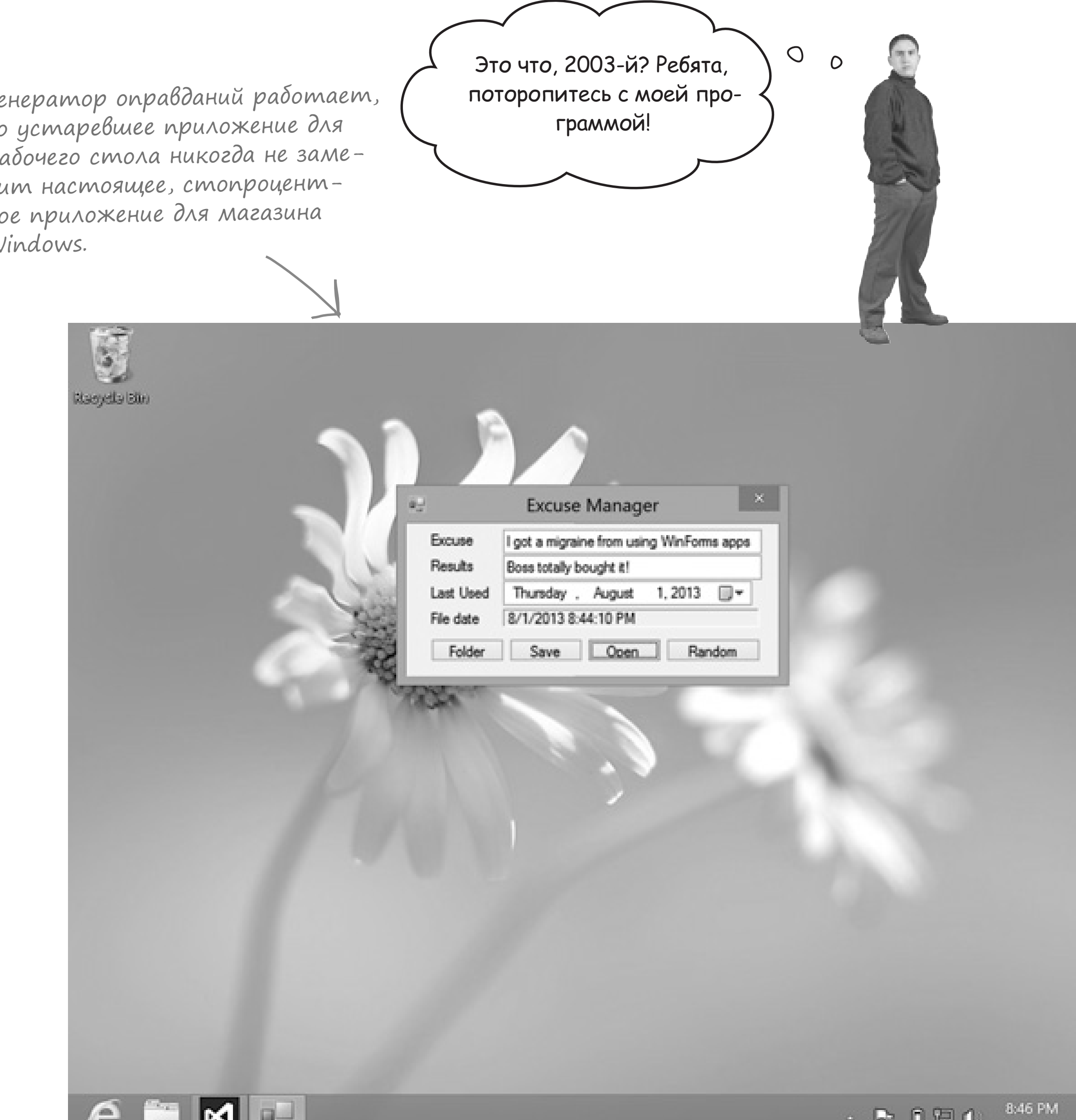
Приложения следующего уровня



Новые возможности разработки приложений. Работа с интерфейсом WinForms знакомит с важными понятиями C#, но не только. В этой главе вы создадите приложения для магазина Windows на языке XAML, научитесь конструировать страницы для различных устройств, встраивать данные путем привязки, а затем освоите тайны XAML-страниц, исследуя XAML-объекты при помощи Visual Studio.

Брайан запускает Windows 8

Приложение для рабочего стола, которым пользуется Брайан, выглядит так старомодно! Ему смертельно надоело расставлять эти маленькие флажки. Брайан хочет, чтобы генератор оправданий стал настоящей программой. Поможем ему осуществить мечту?



Хотите быстро усвоить эти концепции? Тогда сделайте это упражнение перед тем, как приступить к чтению главы!

Приложения из магазина Windows сложнее программ WinForms. Поэтому данный интерфейс является не более чем эффективным инструментом обучения. Имеет смысл вспомнить, как осуществляется обучение, чтобы повысить его результативность. Этим мы и займемся.

Вы уже обладаете базовыми знаниями о C#, объектах, коллекциях и других инструментах .NET. Теперь пора с помощью XAML приступить к приложениям для магазина Windows. Мы воспользуемся IDE для исследования создаваемых и управляемых интерфейсом WinForms объектов. Обратите внимание, чем они отличаются и чем схожи с объектами, входящими в состав XAML-приложений для магазина Windows.

Прежде чем продолжить чтение, сделайте следующее:

В главе 1 и большей части главы 2 мы знакомили вас с созданием приложений для магазина Windows при помощи XAML и .NET Framework для Windows Store. Эта информация пригодится при дальнейшем обучении. А пока остановимся на следующих моментах:

- Кто-то не видит проблемы в переходе от WinForms к XAML и обратно, а кому-то такой переход дается нелегко. **Но это поможет вам быстрее усвоить новый материал!**
- **Вернитесь к главе 1** и с нуля постройте игру *Save the Humans.* На этот раз весь код следует вводить вручную.
- Обязательно ***изучайте примеры кодов***! Там встречаются незнакомые вам вещи, но вы уже вполне способны угадать их назначение.
- Попробуйте собрать кусочки воедино и разобраться с тем, как игра работает, но не перенапрягайтесь — не все представленные концепции вам знакомы.
- Особое внимание обратите на шаблон Basic Page и его использование в качестве страницы по умолчанию *MainPage.xaml.* Мы будем проделывать это неоднократно.
- **Снова выполните** XAML-проект из главы 2. Даже упражнения. Теперь вы готовы!

И еще...

Вы еще не знакомы со всеми возможностями приложений WinForms. Более того, у этих приложений есть механизм для работы с графикой — GDI+, который позволяет создавать на удивление качественную картинку и обеспечивать взаимодействие с пользователем (хотя в XAML это сделать проще). Но наиболее важным способом изучения принципов программирования является реализация одной задачи разными методами. GDI+-графика рассматривалась в предыдущем издании Head First C#, и эти материалы доступны бесплатно. Рекомендуем с ними ознакомиться!

http://www.headfirstlabs.com/hfcsharp

Охота на код!

С момента построения программы *Save the Humans* вы познакомились с множеством концепций из C# и как следует попрактиковались. Теперь наденем шляпу детектива и проверим свои способности в сыскном деле. Найдите эти элементы C# в коде *Save the Humans*. Первый ответ мы дали. Теперь ваша очередь.

(В некоторых случаях можно дать несколько ответов.)

☐ **Использует строку как ключ поиска объекта в словаре.**

....

....

....

☐ **Использует инициализатор объекта.**

....

....

....

☐ **Добавляет два разных типа объекта в одну коллекцию.**

....

....

....

☐ **Вызывает статический метод.**

....

....

....

☐ **Применеяет метод обработчика события.**

☐ **Использует as для понижающего приведения объекта.**

☐ **Передает в метод ссылку на объект.**

☑ **Создает экземпляр объекта и одного из его методов.**

В методе AnimateEnemy() создается экземпляр объекта StoryBoard и вызывается его метод Begin().

☐ **Использует перечисление для присвоения значения.**

Охота на код!

Итак, мы надели шляпы детективов и принялись за поиск элементов C# в коде приложения *Save the Humans*. Вот что мы обнаружили. Надеемся, что вы можете похвастаться аналогичными успехами.

(В некоторых случаях можно дать несколько ответов.)

← Возможно, ваши ответы отличаются от наших!

☑ **Использует строку как ключ поиска объекта в словаре.**

Во второй строке метода AddEnemy() «EnemyTemplate» используется для поиска объекта ControlTemplate в словаре Resources.

☑ **Использует инициализатор объекта.**

Свойства From, To и Duration объекта DoubleAnimation получают начальные значения при помощи инициализатора в методе AnimateEnemy().

☑ **Добавляет два разных типа объекта в одну коллекцию.**

Объект StackPanel (человек) и Rectangle (портал) добавлены в коллекцию playArea.Children в методе StartGame().

☑ **Вызывает статический метод.**

Статические методы SetLeft() и SetTop() в классе Canvas вызываются в методе обработчика события target_PointerEntered().

☑ **Применение метода обработчика события.**

Окно Properties применяется для создания метода обработки события PointerPressed «человека» StackPanel.

☑ **Использует as для понижающего приведения объекта.**

Словарь Resources относится к типу <object, object>, поэтому возвращаемое значение Resources["EnemyTemplate"] приводится к типу ControlTemplate.

☑ **Передает в метод ссылку на объект.**

Ссылка на объект ContentControl передается как первый параметр метода AnimateEnemy().

☑ **Создает экземпляр объекта и одного из его методов.**

В методе AnimateEnemy() создается экземпляр объекта StoryBoard и вызывается его метод Begin().

☑ **Использует перечисление для присвоения значения.**

Перечисление Visibility в методе EndTheGame() присваивает свойству startButton.Visibility значение Visibility.Collapsed.

Windows Forms использует граф объекта, настроенный IDE

При создании программы Windows Desktop настраивались формы и генерировался файл *Form1.Designer.cs*. Но что реально содержит этот файл? Это не тайна. Как вы уже знаете, все элементы формы являются объектами, значит, ссылки на них можно сохранять в поля. Поэтому в файле находится объявление полей для всех объектов, код создания экземпляров и код их отображения на форме. Найдем все эти фрагменты, чтобы понять, что происходит с нашей формой.

1. Откройте построенный в главе 9 проект Simple Text Editor и файл *Form1.Designer.cs*. В его нижней части найдите объявления полей, по одному для каждого элемента управления:

```
Windows Form Designer generated code

private System.Windows.Forms.OpenFileDialog openFileDialog1;
private System.Windows.Forms.SaveFileDialog saveFileDialog1;
private System.Windows.Forms.TextBox textBox1;
private System.Windows.Forms.Button open;
private System.Windows.Forms.Button save;
private System.Windows.Forms.TableLayoutPanel tableLayoutPanel1;
private System.Windows.Forms.FlowLayoutPanel flowLayoutPanel1;
```

2. Раскройте раздел Windows Form Designer–generated code и найдите код, создающий экземпляры элементов управления:

```
private void InitializeComponent()
{
    this.openFileDialog1 = new System.Windows.Forms.OpenFileDialog();
    this.saveFileDialog1 = new System.Windows.Forms.SaveFileDialog();
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.open = new System.Windows.Forms.Button();
    this.save = new System.Windows.Forms.Button();
    this.tableLayoutPanel1 = new System.Windows.Forms.TableLayoutPanel();
    this.flowLayoutPanel1 = new System.Windows.Forms.FlowLayoutPanel();
```

Порядок строк у вас может отличаться, но для каждого элемента формы должна быть своя строка.

3. Элементы TableLayoutPanel и FlowLayoutPanel, содержащие другие элементы, обладают открытым свойством **`Controls`**. Это объект `ControlCollection`, во многом напоминающий объект `List<Control>`. Все элементы формы являются подклассом `Control`, и добавление к коллекции `Controls` заставляет их отображаться внутри панели. Найдите код добавления кнопок Open и Save к объекту FlowLayoutPanel:

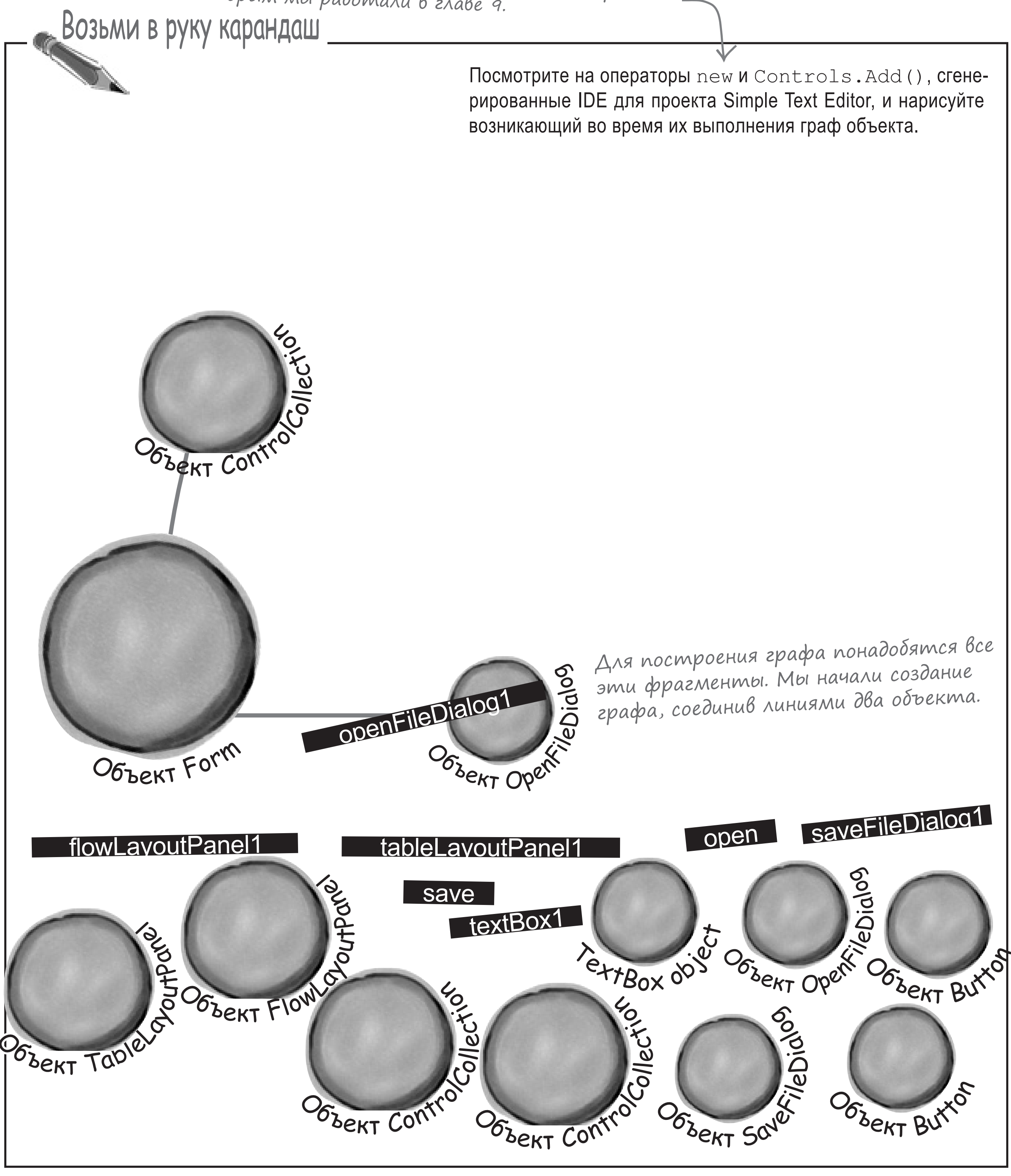
```
this.flowLayoutPanel1.Controls.Add(this.save);
this.flowLayoutPanel1.Controls.Add(this.open);
```

Объект FlowLayoutPanel представляет собой ячейку сетки TableLayoutPanel. Найдите, где он добавляется вместе с объектом TextBox:

```
this.tableLayoutPanel1.Controls.Add(this.textBox1, 0, 0);
this.tableLayoutPanel1.Controls.Add(this.flowLayoutPanel1, 0, 1]
```

Объект `Form` также представляет собой контейнер и включает TableLayoutPanel:

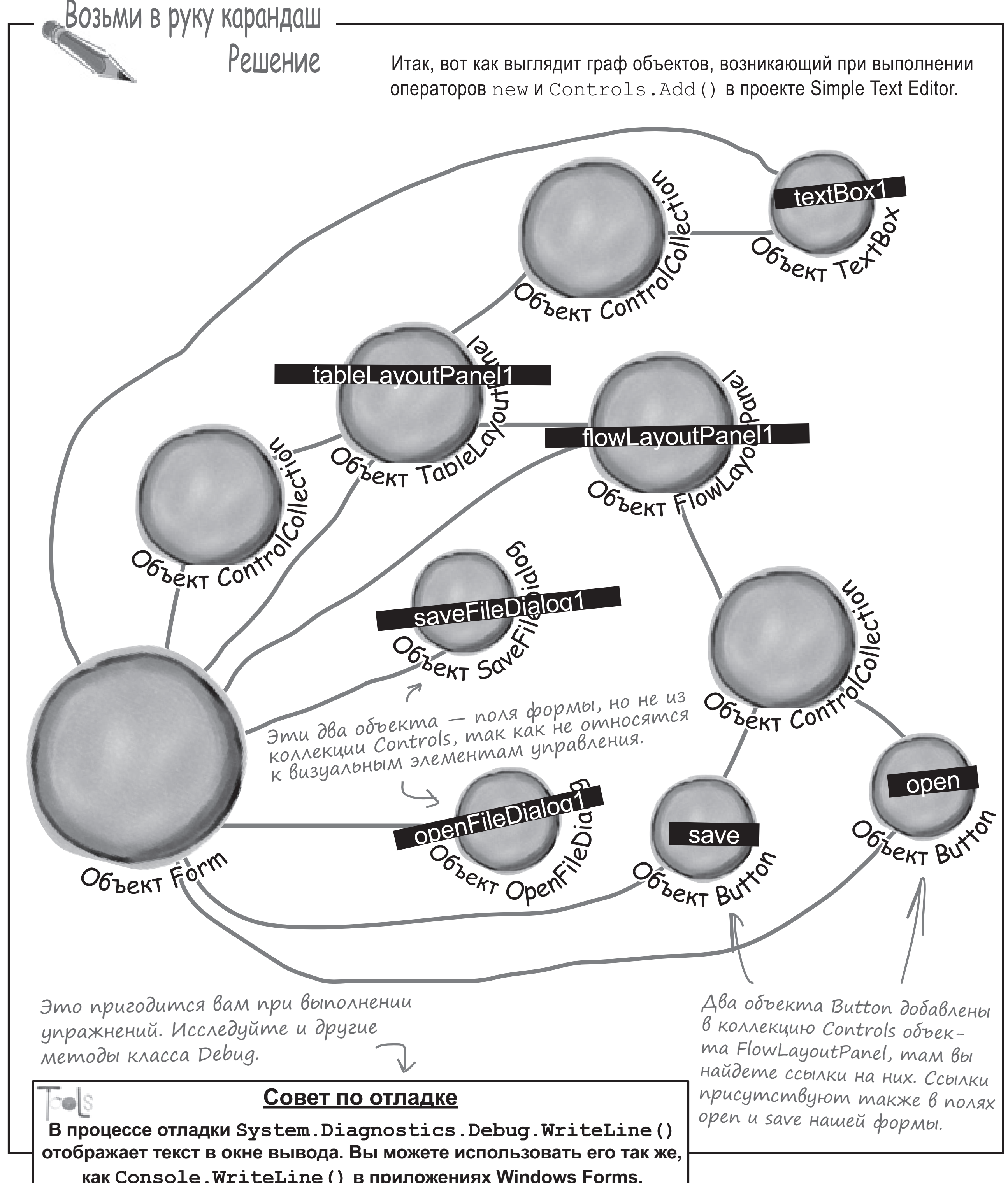
```
this.Controls.Add(this.tableLayoutPanel1);
```



Откройте файл Form1.Designer.cs в коде проекта Simple Text Editor, над которым мы работали в главе 9.
Возьми в руку карандаш
Посмотрите на операторы new и Controls.Add(), сгенерированные IDE для проекта Simple Text Editor, и нарисуйте возникающий во время их выполнения граф объекта.
Объект ControlCollection
Объект Form
openFileDialog1
Объект OpenFileDialog
Для построения графа понадобятся все эти фрагменты. Мы начали создание графа, соединив линиями два объекта.
flowLayoutPanel1
tableLayoutPanel1
open
saveFileDialog1
save
textBox1
Объект TableLayoutPanel
Объект FlowLayoutPanel
Объект ControlCollection
Объект ControlCollection
TextBox object
Объект OpenFileDialog
Объект Button
Объект SaveFileDialog
Объект Button

Возьми в руку карандаш
Решение

Итак, вот как выглядит граф объектов, возникающий при выполнении операторов `new` и `Controls.Add()` в проекте Simple Text Editor.

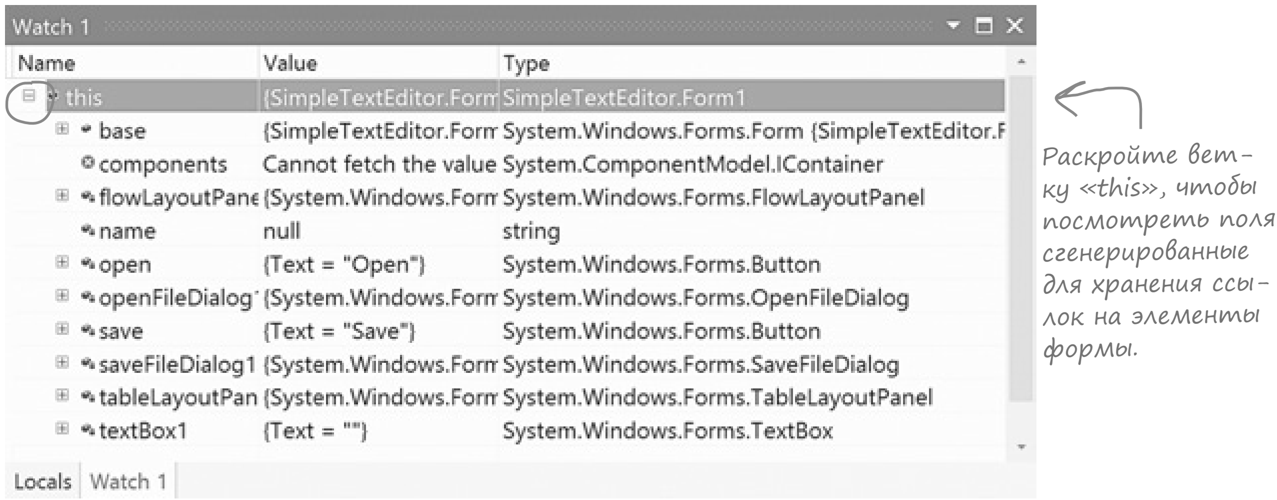


Совет по отладке

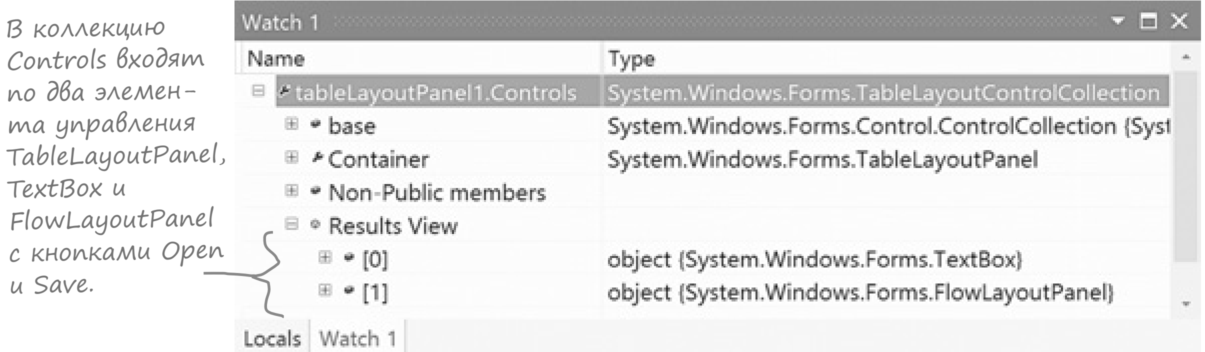
В процессе отладки `System.Diagnostics.Debug.WriteLine()` отображает текст в окне вывода. Вы можете использовать его так же, как `Console.WriteLine()` в приложениях Windows Forms.

Исследуем граф объекта в IDE

Вернемся к Simple Text Editor, поместим точку останова на вызов `InitializeComponent()` и приступим к отладке. После достижения точки останова нажмите F10 для пошагового прохода метода. Затем **перейдите к окну Watch и введите `this`**, чтобы исследовать граф объекта.

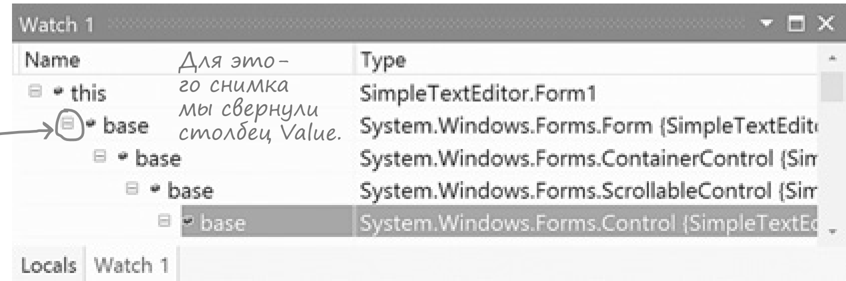


Добавьте окно просмотра для `tableLayoutPanel1.Controls` и раскройте ветку Results View:



Класс `System.Windows.Form` на самом деле не обладает свойством `Controls`, а наследует его от суперкласса `ContainerControl`, наследующего его от `ScrollableControl`, наследующего его от `Control`. Раскройте ⊞ base в окне Watch для просмотра иерархии `System.Windows.Forms.Control`. (`Form` наследует коллекцию Controls.) Раскройте представление **Results** коллекции `Controls`. Вы увидите объекты для всех элементов управления!

Раскрыв «base», можно увидеть свойства, наследуемые от базового класса:



На несколько глав мы прощаемся с приложениями WinForms! Но вернемся к ним еще пару раз, так как это эффективные инструменты изучения C#.

Применение XAML для создания UI объектов

Используя XAML для построения пользовательского интерфейса приложения Windows Store, вы создаете граф объекта. Как и в случае с WinForms, его можно исследовать в окне Watch. **Откроем программу «экспериментов с оператором if-else» из главы 2**. В файле *MainPage.xaml.cs* поместим точку останова на вызов метода `InitializeComponent()` и **исследуем UI-объекты приложения**.

1. Запустите отладку и нажмите F10 для пошагового прохода. Окно Visual Studio 2012 для Windows 8 слегка отличается от VS2012 для рабочего стола благодаря дополнительным функциям, например набору окон watch (которые позволяют следить одновременно за множеством параметров). Выберите **Debug→Windows→Watch→Watch 1** для отображения одного из окон и проследите за `this`:

Watch 1

Name	Type
⊟ this	Chapter2Program2.MainPage
⊞ base	Windows.UI.Xaml.Controls.Page {Chapter2Program2.MainPage}
_contentLoaded	bool
⊞ changeText	Windows.UI.Xaml.Controls.Button
⊞ enableCheckbox	Windows.UI.Xaml.Controls.CheckBox
⊞ labelToChange	Windows.UI.Xaml.Controls.TextBlock

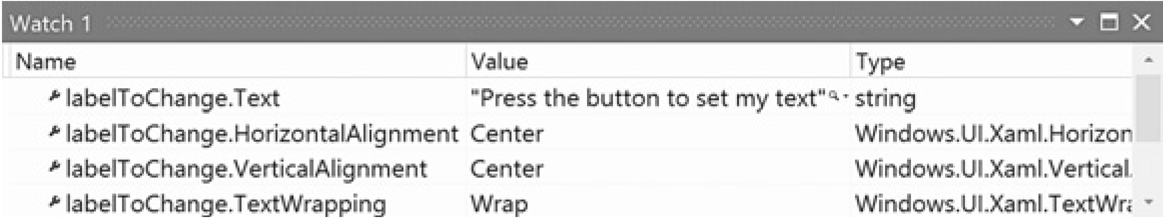
labelToChange — это экземпляр TextBlock

2. Еще раз посмотрим на определяющий страницу код XAML:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Button x:Name="changeText" Content="Change the label if checked"
            HorizontalAlignment="Center" Click="changeText_Click"/>

    <CheckBox x:Name="enableCheckbox" Content="Enable label changing"
              HorizontalAlignment="Center" IsChecked="true"
              Grid.Column="1"/>

    <TextBlock x:Name="labelToChange" Grid.Row="1" TextWrapping="Wrap"
               Text="Press the button to set my text"
               HorizontalAlignment="Center" VerticalAlignment="Center"
               Grid.ColumnSpan="2"/>
</Grid>
```

Определяющий элементы управления страницы код XAML превращен в объект Page с полями и свойствами, ссылающимися на элементы управления UI.

3 Добавим в окно Watch ряд свойств `labelToChange`:



Name	Value	Type
labelToChange.Text	"Press the button to set my text"	string
labelToChange.HorizontalAlignment	Center	Windows.UI.Xaml.Horizon
labelToChange.VerticalAlignment	Center	Windows.UI.Xaml.Vertical.
labelToChange.TextWrapping	Wrap	Windows.UI.Xaml.TextWr

Приложение автоматически определит эти свойства на основе вашего кода XAML:

```
<TextBlock x:Name="labelToChange" Grid.Row="1" TextWrapping="Wrap"
        Text="Press the button to set my text"
        HorizontalAlignment="Center" VerticalAlignment="Center"
        Grid.ColumnSpan="2"/>
```

Но поместите `labelToChange.Grid` или `labelToChange.ColumnSpan` в окно Watch. Элемент управления окажется объектом `Windows.UI.Controls.TextBlock`, не имеющим никаких свойств. Как вы думаете, куда делись свойства XAML?

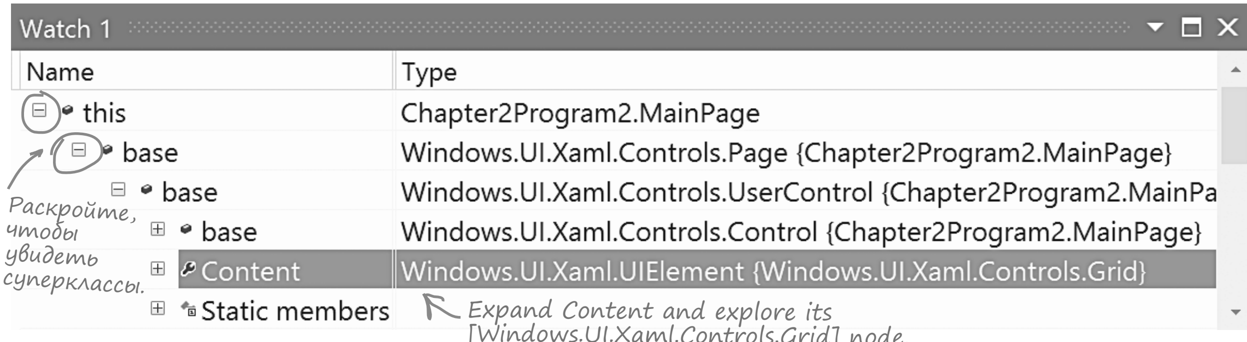
Наведите указатель мыши на Page.

4 Остановите программу, откройте файл *MainPage.xaml.cs* и найдите объявление класса для метода `MainPage`. Перед вами подкласс `Page`. Наведите указатель мыши на класс `Page`, чтобы IDE показала его полное имя:

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();
    }
```

class Windows.UI.Xaml.Controls.Page
Encapsulates a page of content that can be navigated to.

Запустите программу и нажмите F10, чтобы обойти вызов `InitializeComponent()`. Вернитесь в Watch и раскройте `this` >> `base` >> `base`, чтобы проследить за иерархией наследования.

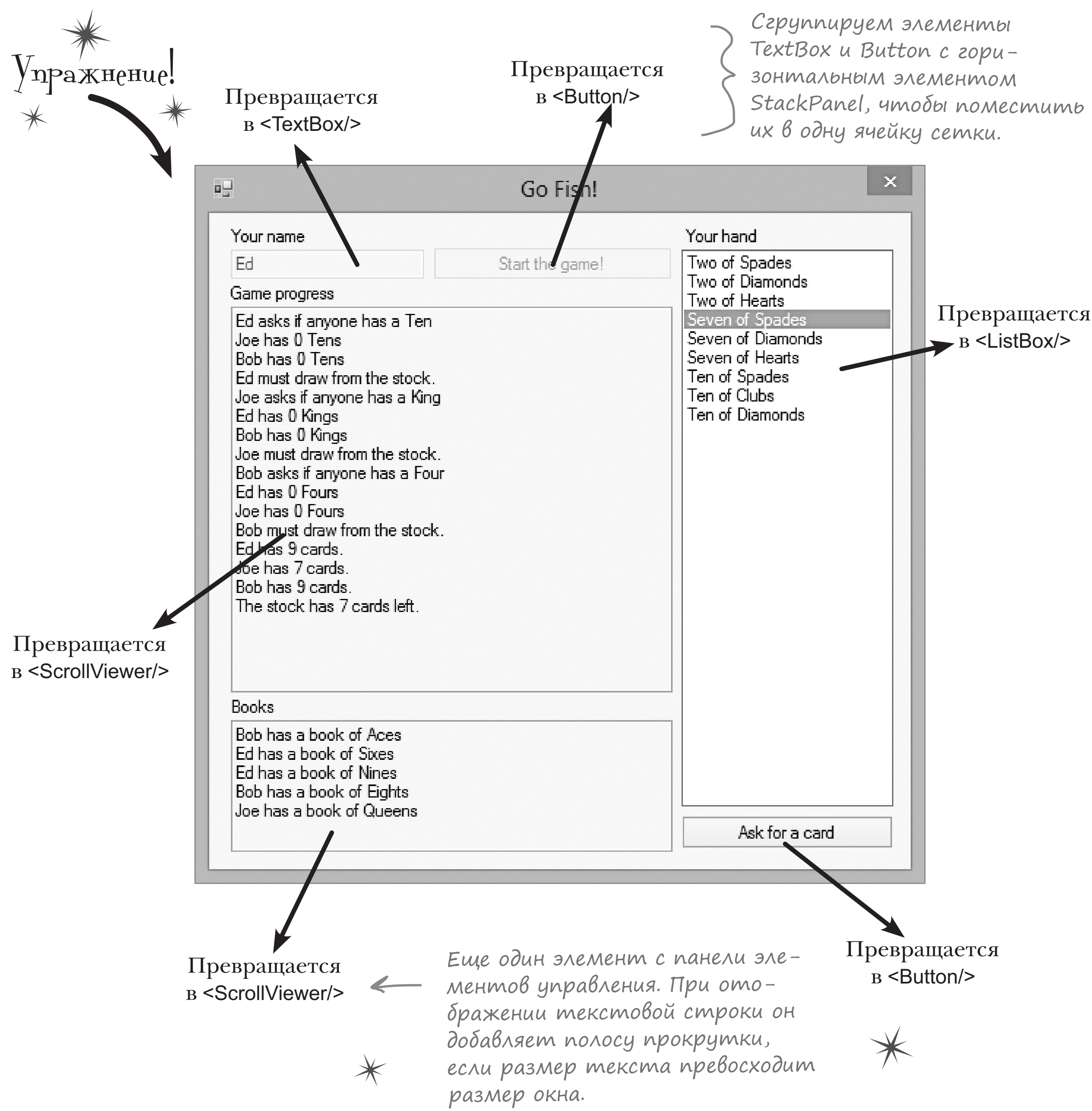


Name	Type
this	Chapter2Program2.MainPage
base	Windows.UI.Xaml.Controls.Page {Chapter2Program2.MainPage}
base	Windows.UI.Xaml.Controls.UserControl {Chapter2Program2.MainPa
base	Windows.UI.Xaml.Controls.Control {Chapter2Program2.MainPage}
Content	Windows.UI.Xaml.UIElement {Windows.UI.Xaml.Controls.Grid}
Static members	

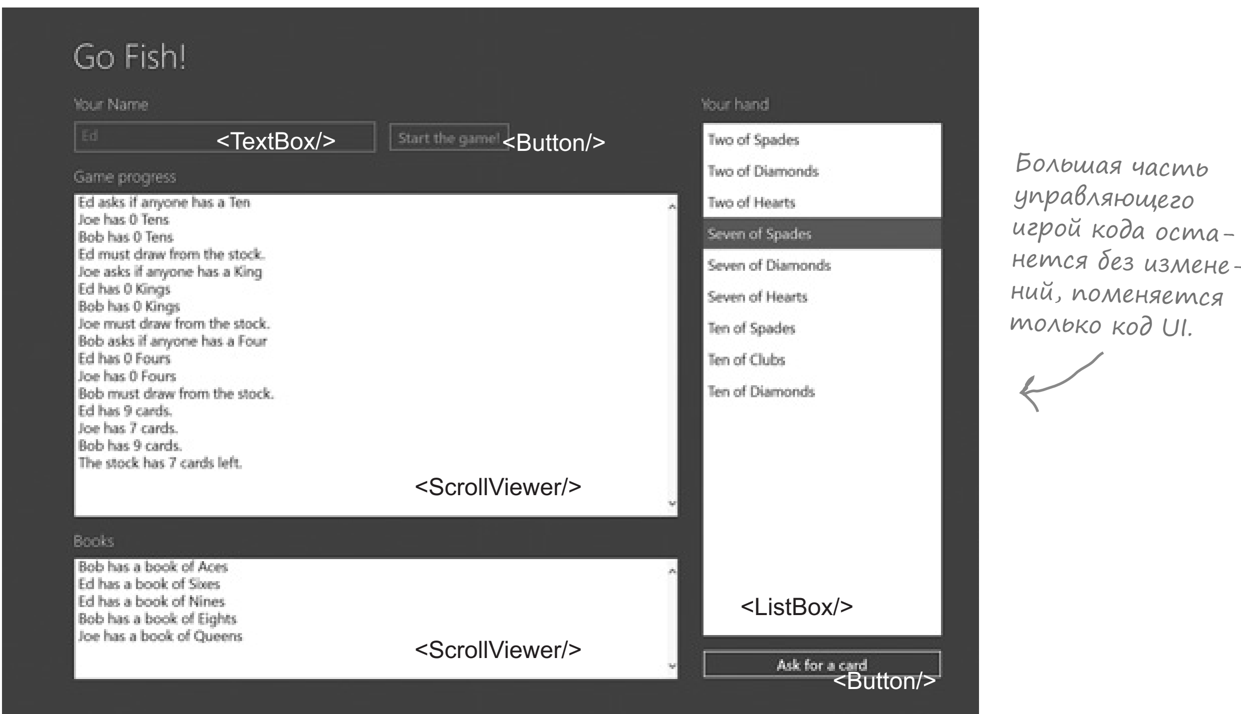
Исследуйте объекты, которые сгенерировал XAML. Чуть позже мы подробно рассмотрим некоторые из них. Пока вам достаточно ощутить, какое количество объектов скрывается в вашем приложении.

Переделаем форму Go Fish! в страницу Windows Store

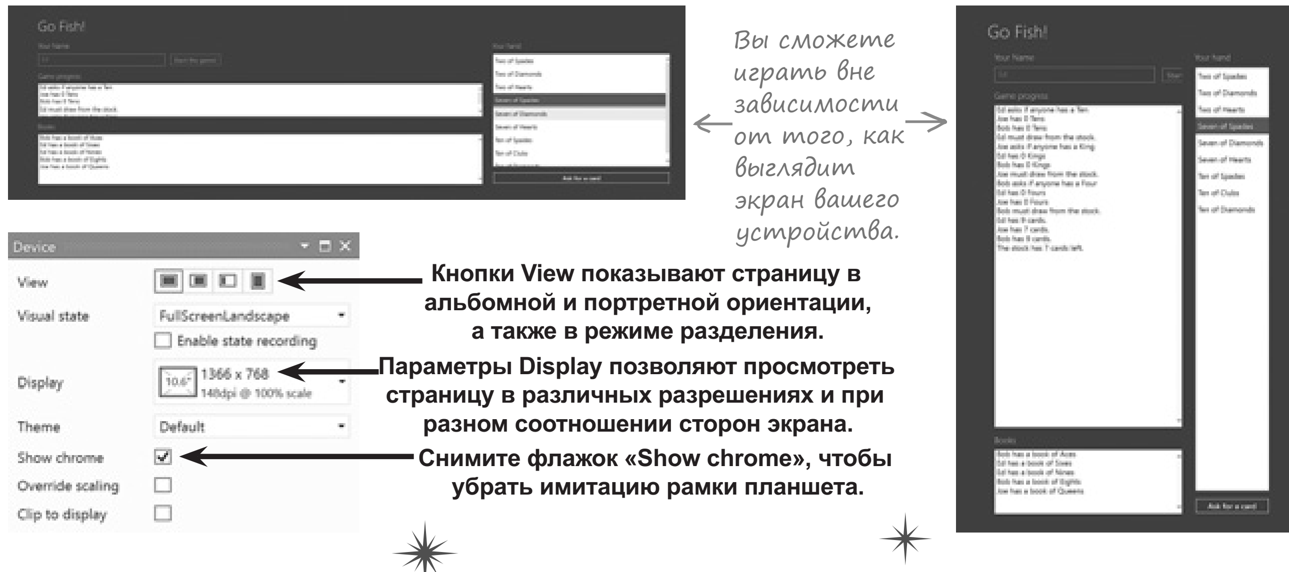
Созданную в главе 8 игру Go Fish! можно превратить в приложение для магазина Windows. Откройте Visual Studio 2012 для Windows 8 и **создайте новый проект Windows Store** (так же как *Save the Humans*). Сейчас мы воссоздадим на XAML игру, адаптирующуюся к устройствам разного размера. Вместо элементов управления Windows Desktop воспользуемся элементами управления Windows Store.



Вот как выглядят эти элементы управления на главной странице приложения:

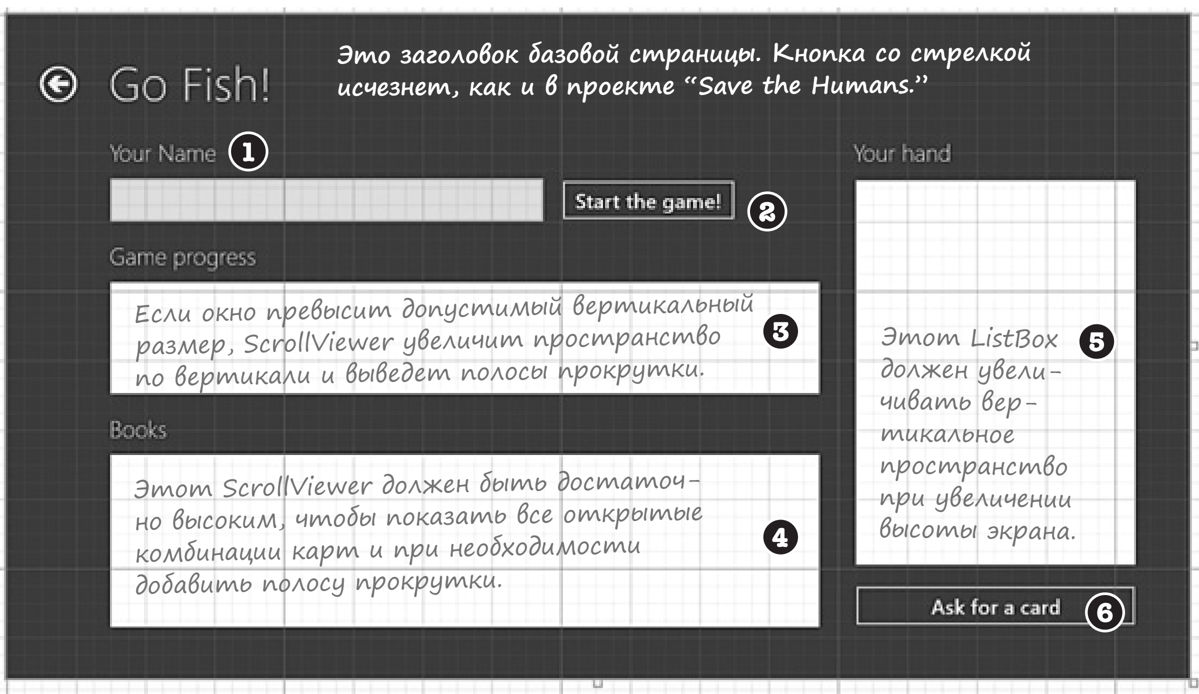


Элементы управления будут располагаться на сетке, подстраивающейся под размер устройства. Для тестирования различных конфигураций экрана можно воспользоваться **окном Device**.



Компоновка страницы начинается с элементов управления

XAML и WinForms при компоновке страницы полагаются на элементы управления. На странице Go Fish! две кнопки, ListBox для показа карт, элемент TextBox для ввода имени пользователя и четыре метки TextBlock. Кроме того, два элемента ScrollViewer с белым фоном показывают ход игры и взятки.



1 Шаблон Basic Page включает двустрочную сетку. В верхней строке расположен заголовок с названием приложения. Вторая содержит привязанный к сетке контент. Вся сетка вставляется в строку 1 шаблона Basic Page (столбец только один — #0).

```
<Grid Grid.Row="1" Margin="120,0,60,60">

    <TextBlock Text="Your Name" Margin="0,0,0,20"
    Style="{StaticResource SubheaderTextStyle}"/>
```

Поля позиционируют сетку на странице. Левое поле всегда равно 120 пикселям.

Открывающий тег сетки.

2 Элемент StackPanel поможет поместить в одну ячейку TextBox для имени игрока и кнопку Start:

```
    <StackPanel Orientation="Horizontal" Grid.Row="1">
        <TextBox x:Name="playerName" FontSize="24"
            Width="500" MinWidth="300" />

        <Button x:Name="startButton" Margin="20,0"
        Content="Start the game!"/>
    </StackPanel>
```

Это добавит 20 пикселей между элементами TextBox и Button. Первый параметр свойства Margin задает поля по горизонтали (левое/правое), а второй — по вертикали (верхнее/нижнее).

Все метки на странице («Your name», «Game progress» и т. п.) являются элементами TextBlock с небольшим полем сверху и свойством `SubHeaderTextStyle`:

```
<TextBlock Text="Game progress"
    Style="{StaticResource SubheaderTextStyle}"
    Margin="0,20,0,20" Grid.Row="2"/>
```

3 Элемент ScrollViewer показывает текущее состояние игры. При слишком большом объеме текста появляется полоса прокрутки:

```
<ScrollViewer Grid.Row="3" FontSize="24"
        Background="White" Foreground="Black" />
```

4 Элементы TextBlock и ScrollViewer нужны для отображения взяток. По умолчанию выравнивание элемента ScrollViewer по горизонтали и вертикали указано `Stretch`, что нам очень пригодится. Выберем строки и столбцы так, чтобы элементы управления ScrollViewer разворачивались на весь экран.

```
<TextBlock Text="Books" Style="{StaticResource SubheaderTextStyle}"
          Margin="0,20,0,20" Grid.Row="4"/>

<ScrollViewer FontSize="24" Background="White" Foreground="Black"
              Grid.Row="5" Grid.RowSpan="2" />
```

5 Маленький столбец в 40 пикселей добавил пространство, поэтому элементы ListBox и Button попали в третий столбец. Для ListBox объединим строки со второй по шестую, то есть запишем `Grid.Row="1"` и `Grid.RowSpan="5"` — это позволит элементу ListBox разворачиваться на всю страницу.

Помните, что нумерация строк и столбцов начинается с нуля, поэтому элементу в третьем столбце соответствует Grid.Column="2".

```
<TextBlock Text="Your hand" Style="{StaticResource SubheaderTextStyle}"
Grid.Row="0" Grid.Column="2" Margin="0,0,0,20"/>

<ListBox x:Name="cards" Background="White" FontSize="24" Height="Auto"
Margin="0,0,0,20" Grid.Row="1" Grid.RowSpan="5" Grid.Column="2"/>
```

6 Выравнивание кнопки «Ask for a card» по горизонтали и вертикали выбрано `Stretch`, что позволяет ей целиком заполнить ячейку. Небольшое пространство добавляет 20-пиксельное поле под элементом ListBox.

```
<Button x:Name="askForACard" Content="Ask for a card"
HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
Grid.Row="6" Grid.Column="2"/>
```

Мы закончим сетку на следующей странице →

Размеры столбцов и строк подстраиваются под страницу

Сетки являются эффективным инструментом компоновки, так как позволяют построить страницу, отображаемую на самых разных устройствах. Высота и ширина со знаком «*» после цифры **автоматически подстраивается** под геометрию экрана. Страница Go Fish! разделена на три столбца. Ширина первого и третьего столбцов 5* и 2*, соответственно, и они **растягиваются и сжимаются**, сохраняя соотношение 5:2. Ширина второго столбца фиксированная — 20 пикселей, он служит разделителем. Посмотрите на расположение строк и столбцов с элементами управления:

<ColumnDefinition Width="40"/>

<ColumnDefinition Width="5*"/>

<ColumnDefinition Width="2*"/>

<RowDefinition Height="Auto"/>
<TextBlock/>

<TextBlock Grid.Column= "1"/>

Row="1" соответствует второй строке, так как нумерация начинается с 0.

<RowDefinition Height="Auto"/>
<StackPanel Grid.Row="1">
<TextBlock/>
<Button/>
</StackPanel>

<ListBox Grid.Column="1" Grid.RowSpan="5"/>

Элемент ListBox объединяет пять строк, в том числе четвертую (с адаптивным размером). Благодаря этому ListBox может разворачиваться на всю правую половину страницы.

<RowDefinition Height="Auto"/>
<TextBlock Grid.Row="2"/>

<RowDefinition/>
<ScrollViewer Grid.Row="3"/>

Высота этой строки — 1, горизонтальное и вертикальное выравнивание элемента ScrollViewer — "Stretch", поэтому он подстраивается под размеры страницы.*

<RowDefinition Height="Auto"/>
<TextBlock Grid.Row="4"/>

<RowDefinition Height="Auto" MinHeight="150"/>
<ScrollViewer Grid.Row="5" Grid.RowSpan="2">

Элемент ScrollViewer объединяет эти две строки. Шестой строке присвоена высота 150, в результате элемент ScrollViewer физически не может стать меньше.

<RowDefinition Height="Auto"/>

<Button Grid.Row="6" Grid.Column="2" />

Нумерация строк и столбцов в XAML начинается с 0, поэтому Button находится в строке 6 и в столбце 2 (чтобы пропустить центральный столбец). Выравнивание этого элемента по вертикали и горизонтали — Stretch, поэтому кнопка заполняет ячейку. Высота строки — Auto, она зависит от контента (кнопки и ее полей).

Вот как определения строк и столбцов создают компоновку страницы:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="5*"/>
    <ColumnDefinition Width="40"/>
    <ColumnDefinition Width="2*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto" MinHeight="150" />
    <RowDefinition Height="Auto"/>
</Grid.RowDefinitions>
```

Первый столбец в 2,5 раза шире третьего (5:2), а разделяет их столбец в 40 пикселей. Отображающие данные элементы ScrollViewer и ListBox имеют HorizontalAlignment, равный «Stretch», что разворачивает их на всю ширину столбцов.

Высота четвертой строки — 1*, что позволяет ей занимать все свободное пространство. Элементы ListBox и первый ScrollViewer объединяют эту строку, а значит, тоже будут менять размер.

Высота практически всех строк имеет значение `Auto`. Существует всего одна строка с переменной высотой, и любой охватывающий ее элемент управления также будет менять свой размер.

Определения строк и столбцов в сетке можно расположить над или под элементами управления. Мы добавили их вниз.

```
</Grid>
```

Это закрывающий тег сетки. Мы объединим весь код в конце главы, когда будем отправлять игру Go Fish! в магазин приложений Windows.

Система сеток и компоновка страниц

Если в главе 1 вам не были нужны кнопки под конструктором, воспользуйтесь ими сейчас.

Вы обращали внимание на сходный внешний вид приложений для магазина Windows? Дело в том, что **система сеток** придает всем приложениям некоторую «общность». Сетка состоит из «кирпичиков», называемых *юнитами* и *субъюнитами*, — вы уже познакомились с ними в IDE.

Поля на странице добавляют зазор шириной 1 юнит между элементами и текстом, а столбец добавляет отступ шириной 2 юнита между элементами ScrollViewers и ListBox.

В приложении Go Fish! за пространство между элементами отвечает свойство `Margin` сетки `<Grid>`. Оно может иметь один параметр (ширина полей со всех сторон), два параметра (первый задает поля слева и справа, а второй — сверху и снизу) или четыре параметра, разделенных запятыми, — поля слева, сверху, справа и снизу.

На главной странице приложения Go Fish! ширина левого поля составляет 120 пикселей (6 юнитов), верхнее поле равно 0 пикселей, а правое и нижнее — 60 пикселям (3 юнитам):

```
<Grid Grid.Row="1" Margin="120,0,60,60">
```

Поле шириной в 1 юнит присутствует сверху и снизу от каждой метки:

```
        <TextBlock Text="Books"
                   Style="{StaticResource SubheaderTextStyle}"
                   Margin="0,20,0,20" Grid.Row="4"/>
```

Часто Задаваемые Вопросы

В: Как меняется ширина или высота при выборе значения «Auto»?

О: После присвоения свойству `Height` строки или `Width` столбца значения `Auto` их размер начинает подстраиваться под размер контента. Посмотрите, как это выглядит на практике. Создайте приложение Blank App, отредактируйте сетку в файле *MainPage.xaml* и добавьте строки и столбцы с высотой и шириной, равной `Auto`. Вы ничего не увидите в конструкторе, так как из-за отсутствия содержимого размер строк и столбцов уменьшается до нуля. Добавьте в ячейки элементы управления различной высоты и посмотрите, как при этом меняется размер ячеек.

В: И чем это отличается от присвоения значений 1*, 2* или 5*?

О: Знак `*` в высоте строки или в ширине столбца приводит к **пропорциональному** увеличению и заполнению всей сетки. При наличии трех столбцов шириной `3*`, `3*` и `4*` ширина первых двух составит 30% ширины сетки минус ширина фиксированных столбцов и столбцов со значением auto, а столбец `4*` займет 40% от получившегося значения.

Поэтому размер строк и столбцов по умолчанию равен 1*, тогда при изменении размеров сетки они будут распределяться равномерно.

В: «Пиксели»... Что вы имеете в виду, постоянно используя это слово?

О: Этим словом часто пользуются XAML-разработчики, и технически это не те пиксели, которые вы видите на экране. Это **независимый от устройства юнит** для параметров `Margin`, `Height`, `Width` и прочих свойств. Приложения для магазина Windows работают на устройствах с экранами разных размеров и пропорций, поэтому данная единица измерения во всех случаях будет равна 1/96 дюйма. Слово «юнит» используется для обозначения фрагментов компоновки страницы, — чтобы избежать путаницы, независимые от устройства единицы измерения стали называть пикселями.

При задании параметров XAML можно указывать in (дюймы), cm (сантиметры) или pt (точки, то есть 1/72 от дюйма). Скомпонуйте страницу в дюймах или сантиметрах. Затем измерьте экран линейкой и убедитесь, что Windows корректно меняет размеры приложения.

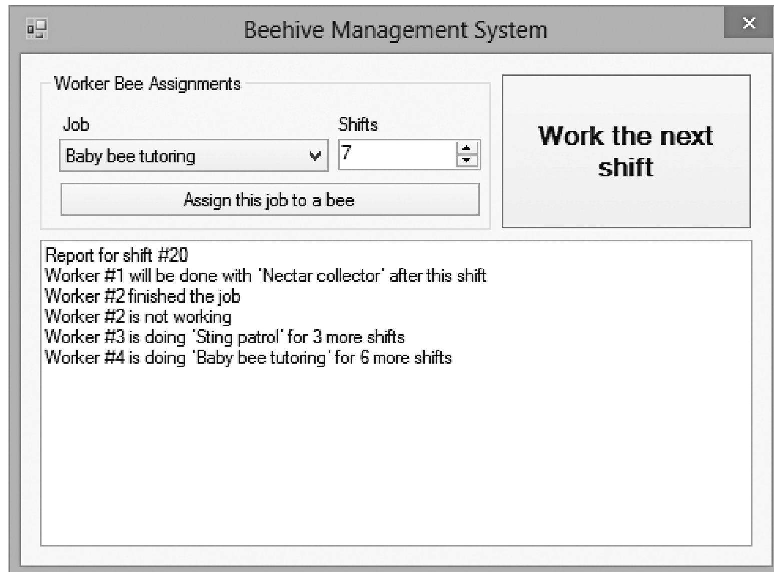
В: Можно ли проверить, насколько хорошо мое приложение будет выглядеть на разных мониторах?

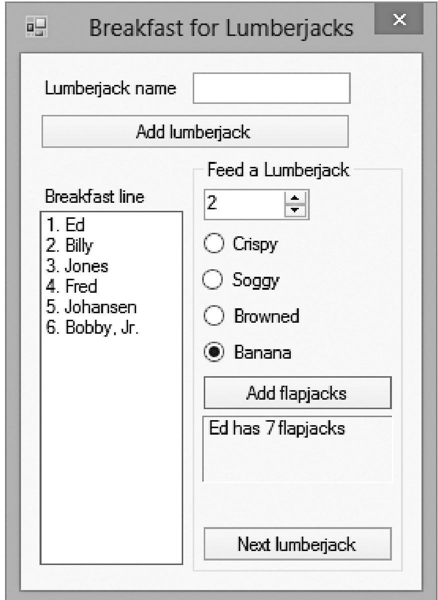
О: Да, конструктор XAML-страниц содержит необходимые для этого инструменты. Окно Device покажет страницу при разных разрешениях и режимах разбиения. Позднее мы покажем вам способ запуска приложения в симуляторе, позволяющем проверить его работу на устройствах с разными экранами.

Если высота строки или ширина столбца равна Auto, его размер будет подстраиваться под размер содержимого.

Дополнительно о компоновке страниц можно почитать здесь:
http://msdn.microsoft.com/ru-ru/library/windows/apps/hh872191.aspx

При помощи XAML переделайте формы из приложений Windows Desktop в приложения для магазина Windows. Создайте в каждом случае новый проект Windows Store Blank App, добавьте новый элемент Basic Page (как вы это сделали для проекта *Save the Humans*) и отредактируйте страницы, видоизменив сетку и добавив элементы управления. Не нужно делать их функциональными. Всего лишь создайте код XAML для следующих снимков экрана.





Найдем в каждом шаблоне Basic Page место для добавления нового элемента Grid или StackPanel, включающего в себя остальные элементы страницы. Поместим их во вторую строку (`Grid.Row="1"`) новой пустой страницы.

```
<Grid Style="{StaticResource LayoutRootStyle}">
    <Grid.RowDefinitions>
        <RowDefinition Height="140"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>

    <Grid Grid.Row="1" Margin="120,0"...>

    <!-- Back button and page title -->
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
```

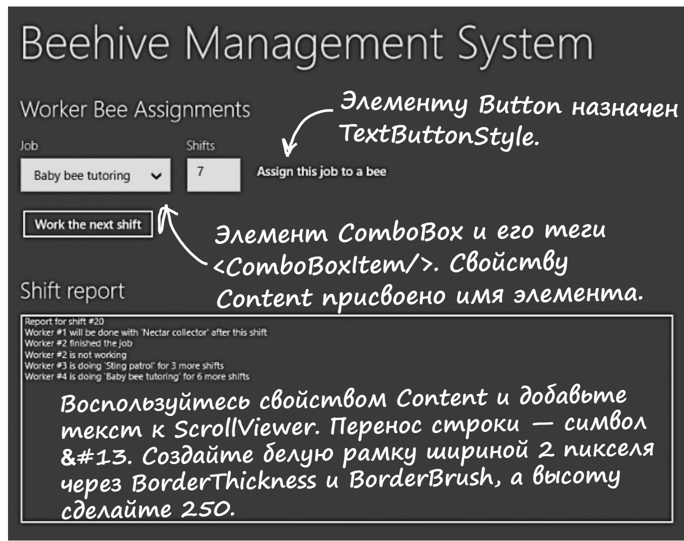
Вот тег <Grid> страницы. В IDE он свернут.

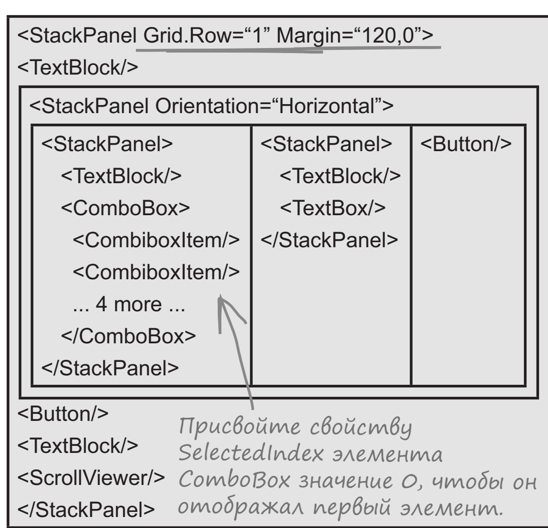
Добавьте новую сетку над этим комментарием.

В XAML не важен порядок тегов

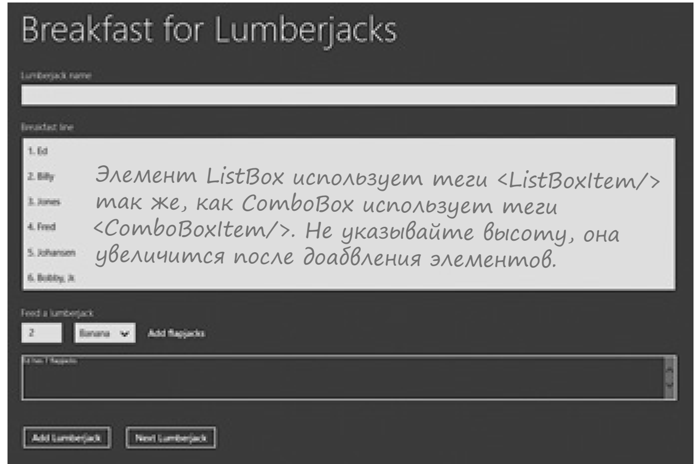
Предлагаем вам добавить код XAML с компоновкой страницы под определениями строк, так как там его будет проще всего найти. Некоторые разработчики XAML предпочитают располагать код в порядке его отображения на странице. Они могут поместить код под закрывающим тегом </Grid> сетки, содержащей кнопку back и заголовок страницы. Поэкспериментируйте, чтобы найти наиболее удобный способ работы.

Задающие форму элементы StackPanel делятся на две группы. К группе заголовков применен стиль GroupHeaderTextStyle, они разделены зазором в 40 пикселей, а после них идет зазор в 20 пикселей. К меткам применен стиль BodyTextStyle, зазор над элементом равен 10 пикселям, а между элементами — 20 пикселям.





Для конструирования формы воспользуйтесь сеткой из восьми строк с парамером `height`, равным `Auto`. Элементы StackPanel позволяют вставить несколько элементов управления в одну строку.




```
<Grid Grid.Row="1" Margin="120,0">
<TextBlock/>
<TextBox/>
<TextBlock/>
<ListBox>
   <ListBoxitem/>
   <ListBoxitem/>
   ... 4 more ...
</ListBox>
<TextBlock>
<StackPanel Orientation="Horizontal">
   <TextBox/>
   <ComboBox> ... 4 items ... </ComboBox>
   <Button/>
</StackPanel>
<ScrollViewer/>
<StackPanel Orientation="Horizontal">
   <Button/>
   <Button/>
</StackPanel>
```

Для придания формам сходства со скриншотами добавьте к элементам управления **фиктивные данные**. В нормальном режиме они добавляются через методы и свойства классов.

При добавлении шаблона Basic Page командой «New Item...» в конструкторе появляется сообщение об ошибке, так что перестройте решение.

Упражнение Решение

Вот как будут выглядеть формы Windows Desktop после превращения в приложения для магазина Windows.

Это поля. Так как правое и нижнее поля в явном виде не заданы, они определяются значениями верхнего и левого полей (120 и 0).

```
<StackPanel Grid.Row="1" Margin="120,0">
    <TextBlock Text="Worker Bee Assignments"
               Style="{StaticResource GroupHeaderTextStyle}"/>
    <StackPanel Orientation="Horizontal" Margin="0,20,0,0">
        <StackPanel Margin="0,0,20,0">
            <TextBlock Text="Job" Margin="0,0,0,10"
                       Style="{StaticResource BodyTextStyle}"/>
            <ComboBox SelectedIndex="0">
                <ComboBoxItem Content="Baby bee tutoring"/>
                <ComboBoxItem Content="Egg care"/>
                <ComboBoxItem Content="Hive maintenance"/>
                <ComboBoxItem Content="Honey manufacturing"/>
                <ComboBoxItem Content="Nectar collector"/>
                <ComboBoxItem Content="Sting patrol"/>
            </ComboBox>
        </StackPanel>
        <StackPanel>
            <TextBlock Text="Shifts" Margin="0,0,0,10"
                       Style="{StaticResource BodyTextStyle}"/>
            <TextBox/>
        </StackPanel>
        <Button Content="Assign this job to a bee" Margin="20,20,0,0"
                Style="{StaticResource TextButtonStyle}" />
    </StackPanel>
    <Button Content="Work the next shift" Margin="0,20,0,0" />

    <TextBlock Text="Shift report" Margin="0,40,0,20"
               Style="{StaticResource GroupHeaderTextStyle}"/>
    <ScrollViewer BorderThickness="2" BorderBrush="White" Height="250"
                  Content="
Report for shift #20&#13;
Worker #1 will be done with 'Nectar collector' after this shift&#13;
Worker #2 finished the job&#13;
Worker #2 is not working&#13;
Worker #3 is doing 'Sting patrol' for 3 more shifts&#13;
Worker #4 is doing 'Baby bee tutoring' for 6 more shifts
                        "/>
</StackPanel>
```

Ваш код XAML отличается от нашего? Существуют разные способы представления идентичных страниц на языке XAML.

Это заголовок второй группы с полями 40 пикселей сверху и 20 пикселей снизу.

Это фиктивные данные, при помощи которых мы заполнили отчет о сменах. Свойство Content игнорирует переносы строк, мы добавили их для лучшей читабельности кода.

```
<Grid Grid.Row="1" Margin="120,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/><RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/><RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/><RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/><RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>

    <TextBlock Text="Lumberjack name" Margin="0,0,0,10"
               Style="{StaticResource BodyTextStyle}"/>
    <TextBox Grid.Row="1"/>

    <TextBlock Grid.Row="2" Text="Breakfast line" Margin="0,20,0,10"
               Style="{StaticResource BodyTextStyle}"/>
    <ListBox Grid.Row="3">
        <ListBoxItem Content="1. Ed"/>
        <ListBoxItem Content="2. Billy"/>
        <ListBoxItem Content="3. Jones"/>
        <ListBoxItem Content="4. Fred"/>
        <ListBoxItem Content="5. Johansen"/>
        <ListBoxItem Content="6. Bobby, Jr."/>
    </ListBox>

    <TextBlock Grid.Row="4" Text="Feed a lumberjack" Margin="0,20,0,10"
               Style="{StaticResource BodyTextStyle}"/>
    <StackPanel Grid.Row="5" Orientation="Horizontal">
        <TextBox Text="2" Margin="0,0,20,0"/>
        <ComboBox SelectedIndex="0" Margin="0,0,20,0">
            <ComboBoxItem Content="Crispy"/>
            <ComboBoxItem Content="Soggy"/>
            <ComboBoxItem Content="Browned"/>
            <ComboBoxItem Content="Banana"/>
        </ComboBox>
        <Button Content="Add flapjacks" Style="{StaticResource TextButtonStyle}"/>
    </StackPanel>

    <ScrollViewer Grid.Row="6" Margin="0,20,0,0" Content="Ed has 7 flapjacks"
                  BorderThickness="2" BorderBrush="White"/>

    <StackPanel Grid.Row="7" Orientation="Horizontal" Margin="0,40,0,0">
        <Button Content="Add Lumberjack" Margin="0,0,20,0" />
        <Button Content="Next Lumberjack" />
    </StackPanel>

</Grid>
```

Мы удалили переносы строк, чтобы определения поместились на странице.

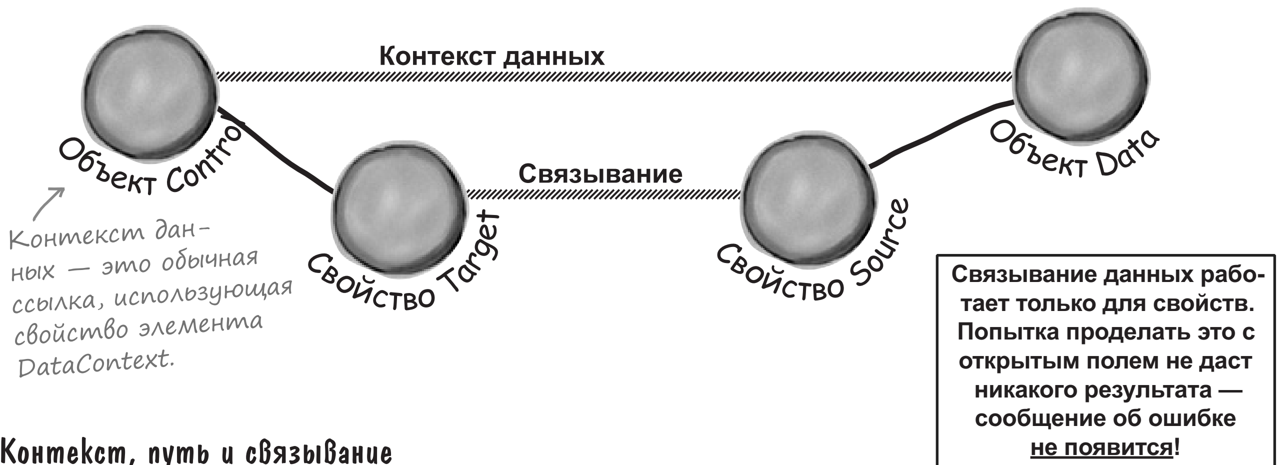
Мы попросили вас добавить фиктивные элементы, чтобы придать форме сходство с настоящей. Но скоро вы научитесь связывать элементы управления, например ListBox, со свойствами в классах.

Еще фиктивные данные...

Что вы думаете по поводу этой страницы? Может, стоило поместить кнопки Add и Next на стандартную панель приложения Windows 8?

Связывание страниц XAML с классами

Элементы TextBlock, ScrollViewer, TextBoxe и т. п. предназначены для отображения данных. При работе с WinForm отображение текста и добавление списков осуществлялось через свойства. С XAML дело обстоит по-другому: после **привязки (data binding)** данные появляются в элементах управления автоматически. Более того, через элементы управления можно обновлять свойства в классах.



Контекст, путь и связывание

Привязкой данных в XAML называется связь между ***свойством*** объекта, предоставляющего данные, и ***свойством*** элемента, который эти данные отображает. Для ее установки **контексту данных** элемента управления нужно присвоить ссылку на объект с данными. **Путь связывания —** это свойство объекта, к которому идет привязка. В настроенной системе свойства объекта-источника автоматически считываются и отображаются как содержимое объекта-приемника.

Элемент TextBlock будет связан через свойство Cash. В результате отображается значение Cash объекта, к которому сделана привязка.

Присвойте свойству, которое хотите связать **`{Путь связывания}`**:

```
<TextBlock x:Name="walletTextBlock" Text="{Binding Cash}"/>
```

В данном случае привязка осуществляется к объекту `Guy` с именем `joe`, свойству `Cash` которого присвоено значение 325.50. Присвоим свойству `DataContext` элемента ссылку на объект `Guy`.

```
Guy joe = new Guy("Joe", 47, 325.50M);

walletTextBlock.DataContext = joe;
```

Контекстом данных для этого элемента TextBlock будет ссылка на объект Guy. Элемент TextBlock считывает связанные свойства объекта Guy.

Есть контакт! Контексту данных присвоен экземпляр Guy, а в качестве пути связывания указано свойство `Cash`. TextBlock привязан к `Cash` и **ищет свойство `Cash`** в своем объекте данных.

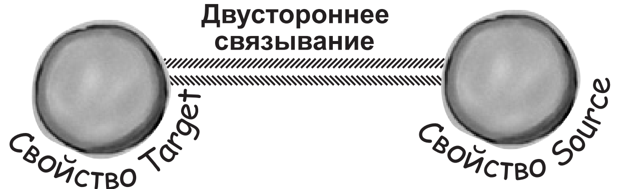
Путь можно не указывать, просто присвойте свойству значение `{Binding}`. В этом случае будет вызываться метод `ToString()` объекта `Guy`.

Двустороннее связывание: читаем или задаем свойство источника

Связывание бывает и **двусторонним**, тогда свойство можно не только читать, но и редактировать:

```
<TextBox x:Name="ageTextBox" Text="{Binding Age, Mode=TwoWay}"/>
```

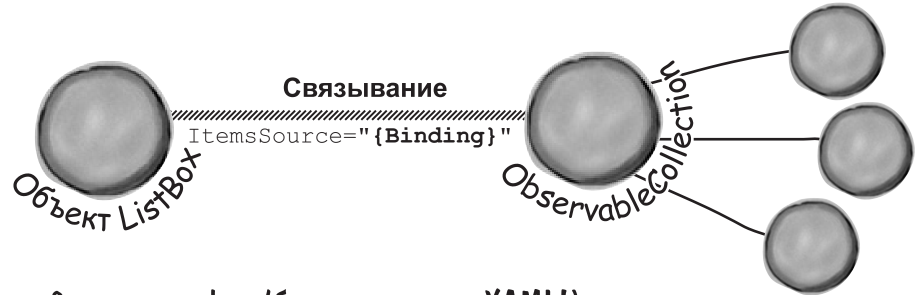
Путем связывания элемента TextBox в этом случае стало свойство `Age`. При показе страницы TextBox отобразит значение свойства `Age` объекта, к которому выполнена привязка. Если вы измените значение, элемент TextBox вызовет метод доступа свойства `Age` и выполнит обновление.



Связывание данных должно вызывать как можно меньше проблем. Если путь связывания указывает на свойство, находящееся вне контекста данных, данные не будут отображаться и редактироваться, но программа продолжит работу.

Связывание с коллекциями

Некоторые элементы управления, такие как TextBlock и TextBox, отображают строки, в то время как ScrollViewer показывает содержимое объекта, а ListBox и ComboBox отображают коллекции. Поэтому в .NET существует класс `ObservableCollection<T>`, предназначенный для связывания.



Связывание свойства ItemsSource элемента ListBox с коллекцией ObservableCollection позволяет элементу отобразить все содержимое коллекции.

Связывание через код (без применения XAML!)

Как таковое свойство `Binding` у элементов управления отсутствует. В C# нет способа напрямую сослаться на свойство объекта, ссылка возможна только на объект. При создании кода связывания в XAML применяется **экземпляр объекта `Binding`**, хранящий имя целевого свойства в виде строки. Вот код, создающий объект `Guy` и настраивающий связывание элемента TextBlock с именем `walletTextBlock`, соединяя его свойство `Text` со свойством `Cash` объекта `Guy`.

```
Guy joe = new Guy("Joe", 47, 325.50M);
Binding cashBinding = new Binding();
cashBinding.Path = new PropertyPath("Cash");
cashBinding.Source = joe;
walletTextBlock.SetBinding(TextBlock.TextProperty, cashBinding);
```

В классе TextBlock имеется целый набор статических свойств, являющихся экземплярами класса DependencyProperty. Одно из них называется TextProperty.

Элементы XAML могут содержать... не только текст

Поговорим немного о **разметке** XAML (именно на нее намекает буква «M» в аббревиатуре XAML) и про **код программной части** (он находится в файле *.cs*).

При использовании Grid и StackPanel все их элементы добавляются между открывающим и закрывающим тегами. Аналогично можно поступить с другими элементами управления. Свойство `Text` для TextBlock и TextBox можно задать, добавив текст и закрывающий тег:

```
<TextBlock>Текст, который нужно показать</TextBlock>
```

Аналогично использованию свойства Text.

Для разрывов строк вместо `
` нужно использовать тег `<LineBreak/>`. На самом деле вы определяете юникодный символ U+0013, который интерпретируется как перенос. В шестнадцатеричной форме перенос даст `
`, а `£` выведет £ (см. Charmap).

```
<TextBlock>First line<LineBreak/>Second line</TextBlock>
```

Добавьте этот TextBlock на страницу XAML, воспользуйтесь свойством Edit Text, чтобы отредактировать, и нажмите Shift-Enter для переноса строки. IDE добавит код:

```
<TextBlock>
    <Run Text="Первая строка"/>
    <LineBreak/>
    <Run Text="Вторая строка"/>
</TextBlock>
```

Результат похож, но при этом возникают разные графы объектов. Каждый тег `<Run>` превращается в строковый объект, и всем этим строкам можно присвоить имена:

```
<Run Text="Первая строка" x:Name="firstLine" />
```

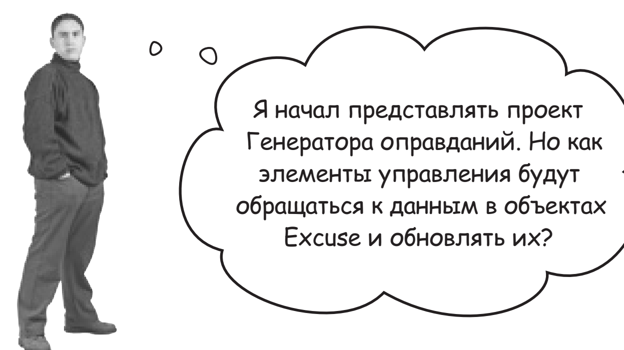
Вы можете модифицировать текст прямо в коде C#, определяющем форму XAML:

```
firstLine.Text = "Новый текст для первой строки";
```

Элементы управления контентом, например ScrollViewer, имеют свойство `Content` (вместо `Text`), которое может содержать любой элемент управления. Элементов управления контентом довольно много, например полезный элемент `Border`, добавляющий фон и рамку к другим элементам:

```
<Border Background="Blue"
        BorderBrush="Green" BorderThickness="3">
</Border>
```

Элемент ScrollViewer наследует от ContentControl, которым мы пользовались для создания врагов в приложении «Save the Humans». Тогда ContentControl содержал сетку, внутри которой располагались три эллипса.



Часто задаваемые вопросы

В: Внутри сетки на моей странице находится другая сетка, содержащая элемент StackPanel. Существует ли ограничение на уровень вложенности элементов?

О: Нет. Можно вкладывать элементы друг в друга, добавляя дополнительные элементы управления. Позднее вы научитесь создавать элементы управления, добавляя содержимое к контейнеру. Сетку можно поместить в *любой* элемент управления содержимым, вы уже поступали так при создании врагов в проекте *Save the Humans*. Именно в этом состоит преимущество XAML для проектирования приложений: вы получаете возможность создавать сложные страницы из простых элементов.

В: Если страницу можно скомпоновать как с помощью элемента Grid, так и через StackPanel, что выбрать?

О: Выбор зависит от ситуации. «Правильного» ответа нет: иногда имеет смысл прибегнуть к StackPanel, иногда лучше использовать Grid, а иногда выгодно их скомбинировать. Более того, существуют и дополнительные варианты. В проекте *Save the Humans* вы использовали контейнерный элемент Canvas, свойства которого `Canvas.Left` и `Canvas.Top` позволяют помещать элемент в определенную точку. Все три элемента являются подклассами `Panel`, и среди унаследованных от базового класса поведений присутствует свойство содержать в себе множество других элементов.

В: Означает ли это, что существуют элементы, служащие контейнером всего для одного элемента?

О: Да. Добавьте на страницу ScrollViewer и вставьте в него два элемента. Вот что вы увидите в результате:

```
<ScrollViewer>
    <TextBox/>
    <Button/>
</Sc
```

The property 'Content' is set more than once.

XAML задает принадлежащее типу `object` свойство `Content` объекта `ScrollViewer`. Если вместо тегов ScrollViewer поставить теги Grid:

```
<Grid>
    <TextBox/>
    <Button/>
</Grid>
```

Вложенные элементы управления добавляются в коллекцию Children. (Именно с ее помощью мы добавляли врагов в проекте «Save the Humans».)

В: Почему некоторые элементы управления, например TextBlock, имеют свойство `Text`, а не свойство `Content`?

О: В них можно вставлять только текст, поэтому свойство `Text` типа `string` используется вместо `Content` типа `object`. Это так называемые **свойства по умолчанию**. Для элементов Grid и StackPanel свойством по умолчанию является коллекция `Children`.

В: Следует ли мне вводить код XAML вручную или же лучше перетаскивать элементы с панели в конструкторе IDE?

О: Попробуйте оба варианта и выберите, что удобнее. Многие разработчики полагаются на конструктор, но есть и те, кто им практически не пользуется, так как быстрее вводить код вручную. Особенно в этом помогает функция IntelliSense.

В: Напомните, зачем нужно изучать WinForms? Почему не перейти непосредственно к XAML и приложениям для магазина Windows?

О: Дело в том, что эта информация упростит восприятие XAML. Например, возьмем коллекцию `Children`. Смогли бы вы разобраться с ответом на третий вопрос, не зная, что такое коллекции? Возможно. Но тем, кто знаком с данным понятием, ответ более очевиден. С другой стороны, перетаскивать элементы с панели на форму очень просто. WinForms проще конструирования страниц на XAML (что вполне разумно, ведь XAML более новая и гибкая технология). Рассмотрение WinForms упростит в дальнейшем разработку визуальных приложений и создание интересных проектов. А это, в свою очередь, позволит вам быстрее выучить XAML. Всегда полезно знать разные способы решения одной и той же задачи. Именно поэтому мы повторно рассматриваем проекты из предыдущих глав: вы лучше разберетесь в WinForms и в приложениях для магазина Windows, посмотрев на реализацию решения в обоих случаях.

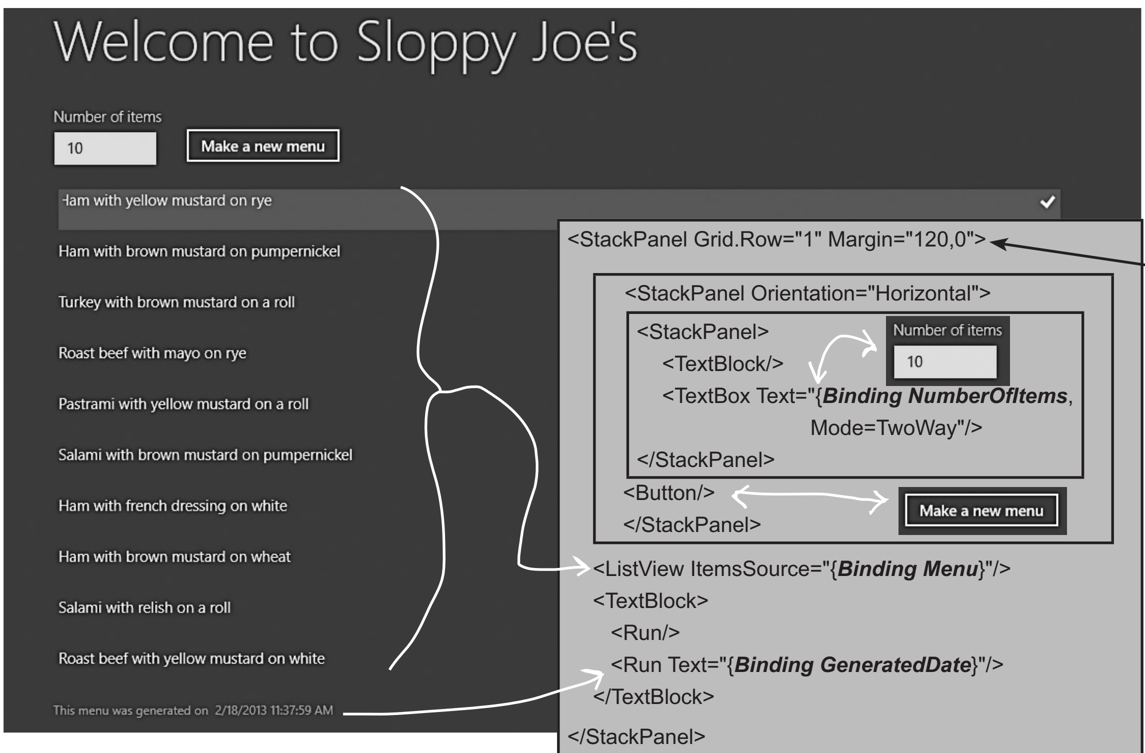
WinForms — замечательный инструмент для изучения C#, в то время как XAML больше подходит для создания гибких и эффективных приложений.

Обновим меню при помощи связывания данных

Помните Джо из главы 4? Он тоже установил Windows 8 и хочет оформить меню в виде приложения для магазина Window. Поможем ему в этом.

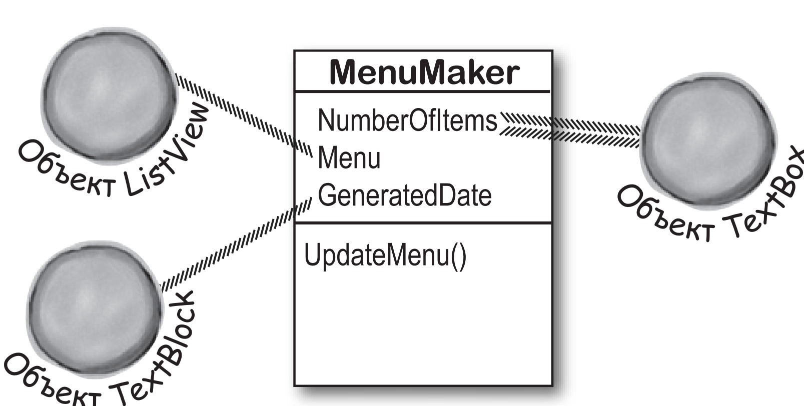
Вот страница, которую мы собираемся создать.

Одностороннее связывание заполняет элементы ListView и Run внутри элемента TextBlock, двустороннее связывание для TextBox использует один из тегов `<Run>` для формирования фактической связи.



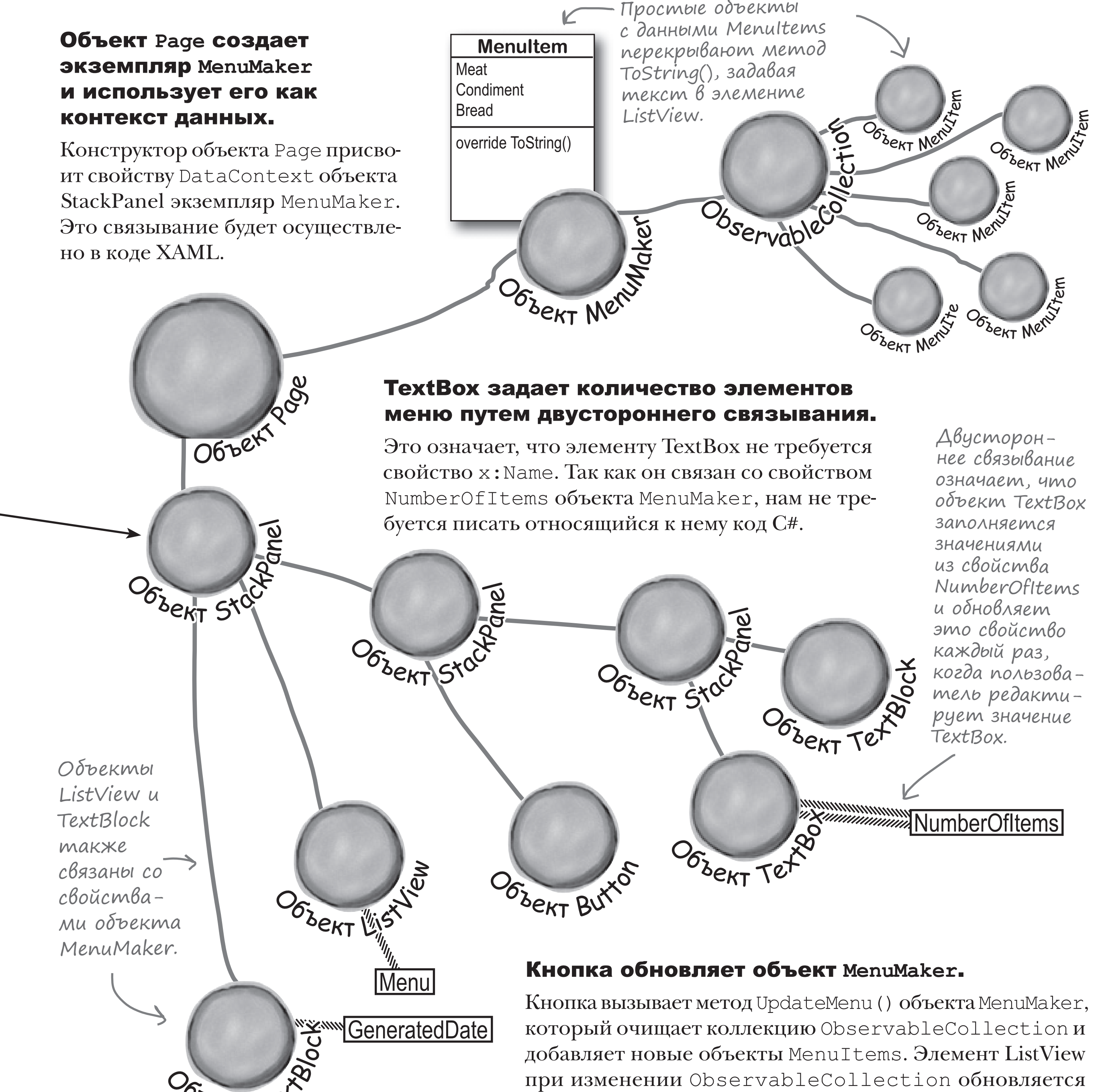
Нужен объект для связывания.

Объект `Page` обладает экземпляром класса `MenuMaker` с тремя открытыми свойствами: целочисленный `NumberOfItems`, коллекция `ObservableCollection` элементов меню с именем `Menu` и свойство типа `DateTime` с именем `GeneratedDate`.



Объект `Page` создает экземпляр `MenuMaker` и использует его как контекст данных.

Конструктор объекта `Page` присвоит свойству `DataContext` объекта StackPanel экземпляр `MenuMaker`. Это связывание будет осуществлено в коде XAML.



TextBox задает количество элементов меню путем двустороннего связывания.

Это означает, что элементу TextBox не требуется свойство `x:Name`. Так как он связан со свойством `NumberOfItems` объекта `MenuMaker`, нам не требуется писать относящийся к нему код C#.

Кнопка обновляет объект `MenuMaker`.

Кнопка вызывает метод `UpdateMenu()` объекта `MenuMaker`, который очищает коллекцию `ObservableCollection` и добавляет новые объекты `MenuItems`. Элемент ListView при изменении `ObservableCollection` обновляется автоматически.

Вот задача на написание кода. Основываясь на прочитанной информации, насколько вы сможете улучшить приложение для Джо до того, как перевернете страницу и увидите готовое решение?

Упражнение!

1 **Создадим новый проект и заменим MainPage.xaml на Basic Page.**

Создайте **новое приложение для магазина Windows**. Удалите *MainPage.xaml* и вместо него **добавьте шаблон Basic Page с именем *MainPage.xaml***. После этого потребуется перестроить проект, как вы это делали при работе над приложением *Save the Humans*.

2 **Добавим новый, улучшенный класс `MenuMaker`.**

Вы многое узнали после главы 4. Постройте хорошо инкапсулированный класс, позволяющий задавать количество элементов через свойство. В его конструкторе создайте коллекцию `ObservableCollection` объектов `MenuItem`, обновляемую при каждом вызове метода `UpdateMenu()`. Одновременно он обновляет свойство `GeneratedDate` типа `DateTime` временной меткой текущего меню. Добавьте к своему проекту новый класс `MenuMaker`:

```
using System.Collections.ObjectModel;

class MenuMaker {
    private Random random = new Random();
    private List<String> meats = new List<String>()
            { "Roast beef", "Salami", "Turkey", "Ham", "Pastrami" };
    private List<String> condiments = new List<String>() { "yellow mustard",
          "brown mustard", "honey mustard", "mayo", "relish", "french dressing" };
    private List<String> breads = new List<String>() { "rye", "white", "wheat",
          "pumpernickel", "italian bread", "a roll" };
    public ObservableCollection<MenuItem> Menu { get; private set; }
    public DateTime GeneratedDate { get; private set; }
    public int NumberOfItems { get; set; }
    public MenuMaker() {
        Menu = new ObservableCollection<MenuItem>();
        NumberOfItems = 10;
        UpdateMenu();
    }
    private MenuItem CreateMenuItem() {
        string randomMeat = meats[random.Next(meats.Count)];
        string randomCondiment = condiments[random.Next(condiments.Count)];
        string randomBread = breads[random.Next(breads.Count)];
        return new MenuItem(randomMeat, randomCondiment, randomBread);
    }
    public void UpdateMenu() {
        Menu.Clear();
        for (int i = 0; i < NumberOfItems; i++) {
            Menu.Add(CreateMenuItem());
        }
        GeneratedDate = DateTime.Now;
    }
}
```

Эта строка using требуется потому, что в данном пространстве находится коллекция ObservableCollection<T>.

Щелкните правой кнопкой мыши на имени проекта в окне Solution Explorer и добавьте новый класс.

Связывание позволит отобразить на странице данные из этих свойств. Двустороннее связывание будет обновлять значение NumberOfItems.

Новый метод CreateMenuItem() возвращает объекты MenuItem, а не строки. Это упрощает редактирование отображения элементов.

Внимательно посмотрите, как это работает. Здесь не создается новая коллекция MenuItem, а очищается старая, в которую добавляются новые элементы.

Что произойдет, если NumberOfItems присвоить отрицательное число?

Работа с датами

Дату позволяет хранить уже знакомый вам тип DateTime. Его можно использовать для редактирования даты и времени. Статическое свойство Now возвращает текущее время. DateTime обладает методами AddSeconds() для добавления и преобразования секунд, миллисекунд, дней и т. п. и свойствами Hour и DayOfWeek для представления даты. Как своевременно!

3 Добавляем класс `MenuItem`.

Вы уже видели, что применение для хранения данных классов вместо строк позволяет строить более гибкие программы. Вот простой класс для хранения элемента меню, добавьте его к вашему проекту:

```
class MenuItem {
    public string Meat { get; private set; }
    public string Condiment { get; private set; }
    public string Bread { get; private set; }

    public MenuItem(string meat, string condiment, string bread) {
        Meat = meat;
        Condiment = condiment;
        Bread = bread;
    }

    public override string ToString() {
        return Meat + " with " + Condiment + " on " + Bread;
    }
}
```

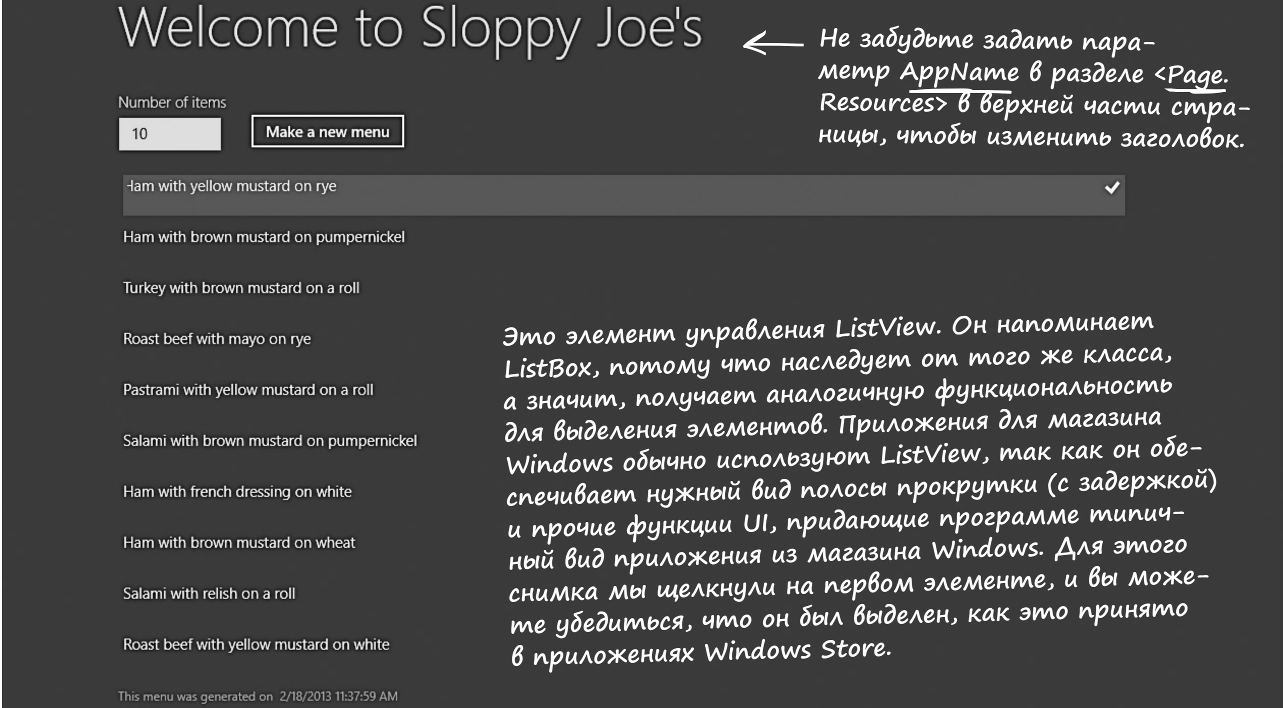
Три строки, составляющие элемент, передаются в конструктор и сохраняются в предназначенных только для чтения свойствах.

Перекроем метод ToString(), чтобы MenuItem знал, как он должен отображаться.

4 Построим страницу XAML.

Вот скриншот. Можете воссоздать его при помощи элементов StackPanels? Ширина элемента TextBox равна 100. Стиль нижнего элемента TextBlock — `BodyTextStyle`, и он обладает двумя тегами `<Run>` (второй используется для хранения даты).

На этот раз обойдемся без добавления фиктивных данных. Всю работу за нас сделает связывание.



Можете построить страницу самостоятельно только по виду скриншота?

5 **Добавим в XAML имена объектов и свяжем данные.**

Вот код XAML для файла *MainPage.xaml.* **Добавьте его к внешней сетке сразу над комментарием** `<!-- Back button and page title -->`, как вы делали с главной страницей приложения *Save the Humans.* Кнопке присвойте имя `newMenu`. Так как для элементов ListView, TextBlock и TextBox мы использовали связывание данных, присваивать имена не нужно. *(На самом деле нам не требовалось присваивать имя даже кнопке, мы это сделали, чтобы IDE автоматически добавила обработчик события newMenu_Click при двойном щелчке на кнопке в конструкторе.)*

```
<StackPanel Grid.Row="1" Margin="120,0" x:Name="pageLayoutStackPanel">
    <StackPanel Orientation="Horizontal" Margin="0,0,0,20">
        <StackPanel Margin="0,0,20,0">
            <TextBlock Style="{StaticResource BodyTextStyle}"
                Text="Number of items" Margin="0,0,0,10" />
            <TextBox Width="100" HorizontalAlignment="Left"
                        Text="{Binding NumberOfItems, Mode=TwoWay}" />
        </StackPanel>
        <Button x:Name="newMenu" VerticalAlignment="Bottom" Click="newMenu_Click"
        Content="Make a new menu" Margin="0,0,20,0"/>
    </StackPanel>
    <ListView ItemsSource="{Binding Menu}" Margin="0,0,20,0" />
    <TextBlock Style="{StaticResource CaptionTextStyle}">
    <Run Text="This menu was generated on " />
    <Run Text="{Binding GeneratedDate}"/>
    </TextBlock>
</StackPanel>
```

Двустороннее связывание позволяет считывать и задавать количество элементов внутри TextBox.

Это элемент ListView. Замените его на ListBox и посмотрите, как это повлияет на вид страницы.

А здесь нам пригодятся теги <Run>. Можно взять единый элемент TextBlock, но связать часть его текста.

6 **Добавим код страницы MainPage.xaml.cs.**

Конструктор страницы создает коллекцию меню и экземпляр `MenuMaker` и задает контекст для элементов управления, использующих связывание данных. Еще ему понадобится поле `MenuMaker` с именем `menuMaker`.

```
MenuMaker menuMaker = new MenuMaker();

public MainPage() {
    this.InitializeComponent();

    pageLayoutStackPanel.DataContext = menuMaker;
}
```

Класс главной страницы в файле MainPage.xaml.cs получит поле MenuMaker, которое используется в качестве контекста данных для StackPanel, содержащего все связанные элементы управления.

Вам остается задать контекст данных для внешнего элемента StackPanel. Он передаст этот контекст всем входящим в него элементам управления.

Напоследок дважды щелкните на кнопке, чтобы сгенерировать заглушку метода для обработки события `Click`. Вот код этого метода, он обновляет меню:

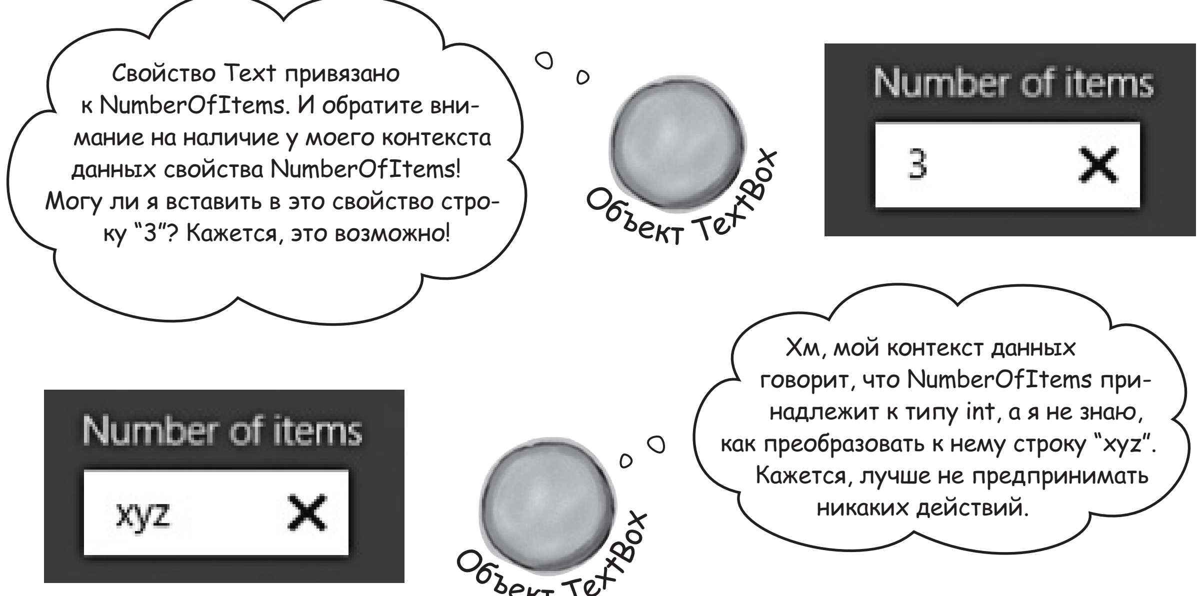
```
private void newMenu_Click(object sender, RoutedEventArgs e) {
    menuMaker.UpdateMenu();
}
```

Можно переименовать обработчик события таким образом, что он будет одновременно обновлять XAML и код C#.

Запустите программу! Попробуйте вводить в TextBox различные значения, например, 3, и вы получите меню из трех пунктов:

Теперь можно поиграть со связыванием, чтобы увидеть все его возможности. Попробуйте ввести в TextBox “xyz” или оставить это поле незаполненным. Ничего не произойдет! Вводя данные в TextBox, вы создаете строку. Элемент TextBox ничего не может с ней сделать. Он знает, что путем связывания является `NumberOfItems`, ищет в контексте данных этого объекта свойства с упомянутым именем и осуществляет приведение строки к типу, к которому принадлежит свойство.

Внимательно наблюдайте за генерируемыми данными —.пока они не обновляются вместе с меню. Возможно, мы что-то упустили.



Объявление объектов в XAML через статические ресурсы

При построении страницы с помощью XAML вы создаете граф объектов из таких элементов, как StackPanel, Grid, TextBlock и Button. И вы уже поняли, что ничего таинственного не происходит: вы добавляете к коду XAML тег `<TextBox>`, после чего объекты страницы получают поле TextBox со ссылкой на экземпляр объекта TextBox. И если вы присваиваете этому объекту имя через свойство `x:Name`, код C# будет использовать его для обращений к элементу TextBox.

Аналогичным способом создаются экземпляры *практически любого* класса, которые сохраняются в вашей странице в виде полей путем добавления к коду XAML **статического ресурса (static resource)**. Для таких ресурсов хорошо подходит связывание данных, особенно выполненное в конструкторе IDE. Вернемся к программе для Джо и превратим `MenuMaker` в статический ресурс.

1 **Удалите поле MenuMaker из кода.**
Мы собираемся настроить класс `MenuMaker` и контекст данных в XAML, поэтому удалите из кода C# эти строки:

```
MenuMaker menuMaker = new MenuMaker();

public MainPage() {
    this.InitializeComponent();

    pageLayoutStackPanel.DataContext = menuMaker;
}
```

В: Кнопка Close отсутствует! Как мне выйти из приложения?

О: У приложений для магазина Windows кнопка Close отсутствует, так как обычного выхода не существует. **Жизненный цикл** такого приложения имеет три состояния: неработающее, работающее и приостановленное. Последнее возникает при переходе пользователя к другому окну или при переходе ОС в режим пониженного энергопотребления. Windows закроет приложение, если потребуется восстановить память. Вы скоро узнаете, как заставить приложение работать в таком режиме.

2 **Помотрите на пространства имен вашей страницы.**
Посмотрите на верхнюю часть XAML-кода вашей страницы. В открывающем теге страницы вы увидите набор свойств `xmlns`. Каждое из них задает пространство имен. Найдите то, что начинается с `xmlns:local` и содержит пространство имен вашего проекта. Вот как оно должно выглядеть:

Это свойство XML, объявляющее пространство имен. Оно состоит из атрибута “xmlns:”, за которым следует идентификатор, в данном случае “local”.

Когда значение пространства имен начинается с “using:”, ссылка идет на одно из пространств имен в рамках проекта. Если же оно начинается с “http://”, то ссылка идет на стандартное пространство имен XAML.

```
xmlns:local="using:SloppyJoeChapter10"
```

Вы воспользуетесь этим идентификатором для создания объектов в пространстве имен вашего проекта.

Мы назвали проект SloppyJoeChapter10, и IDE создала соответствующее пространство имен. Найдите пространство имен вашего приложения, так как именно там находится `MenuMaker`.

3 Добавьте в код XAML статический ресурс и задайте контекст данных.

Найдите на странице раздел `<Page.Resources>` и введите **`<local:`** для вызова окна IntelliSense:

```
<Page.Resources>

    <local:
            App
            MenuItem
            MenuMaker

    <!-- TODO: Delete this line if the key AppName is declared in App.xaml -->
    <x:String x:Key="AppName">Welcome to Sloppy Joe's</x:String>
</Page.Resources>
```

Добавлять статические ресурсы можно только для классов, конструкторы которых не имеют параметров. Дело в том, что страница XAML не знает, какие аргументы следует передавать в конструктор с параметрами.

Окно покажет все доступные классы рассматриваемого пространства имен. Выберите вариант `MenuMaker` и присвойте ему имя `menuMaker`:

```
<local:MenuMaker x:Name="menuMaker"/>
```

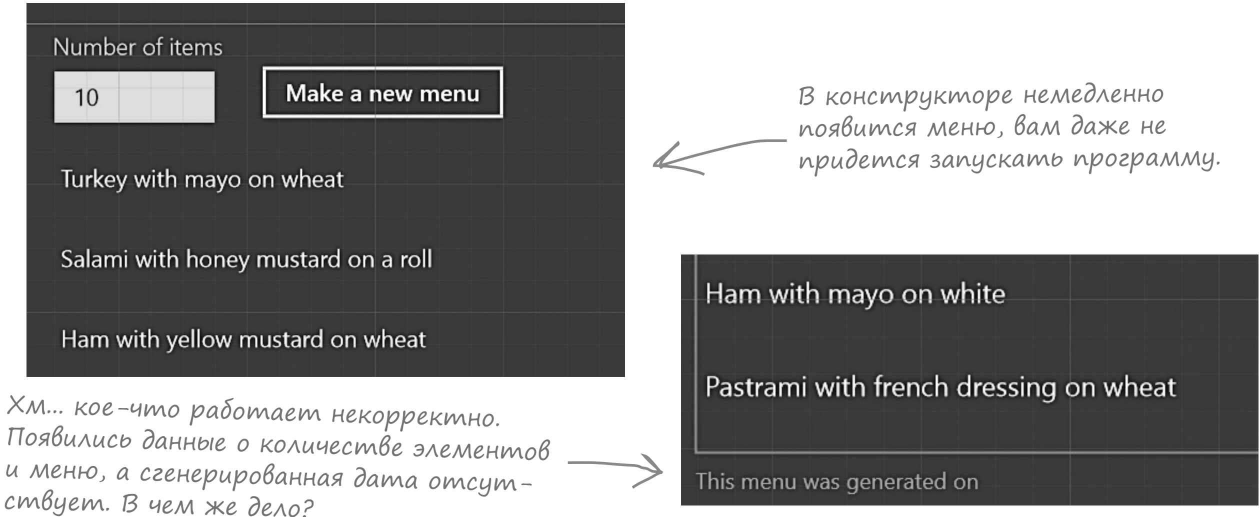
Теперь на странице имеется статический ресурс `MenuMaker` с именем `menuMaker`.

4 Задайте контекст данных для элемента StackPanel и всех его потомков.

Задайте свойство `DataContext` самого внешнего элемента StackPanel:

```
<StackPanel Grid.Row="1" Margin="120,0"
      DataContext="{StaticResource ResourceKey=menuMaker}">
```

Программа будет работать, как и раньше. Но вы заметили, что произошло в IDE после добавления к XAML-коду контекста данных? IDE сразу же создала экземпляр MenuMaker и воспользовалась его свойствами для передачи данных во все связанные с ним элементы управления. Прямо в конструкторе было сгенерировано меню, вы даже не успели запустить программу. Разве это не здорово?



Отображение объектов через шаблон данных

Отображая элементы списка, мы демонстрируем содержимое элементов ListViewItem, ListBoxItem или ComboBoxItem, связанных с объектами в `ObservableCollection`. Каждый ListViewItem в генераторе меню связан с объектом `MenuItem` в коллекции `Menu`. Объекты ListViewItem по умолчанию вызывают методы `ToString()` объектов `MenuMaker`, но вы можете воспользоваться **шаблоном данных**, который применяет связывание для информации из свойств связанных объектов.

Измените тег `<ListView>`, чтобы добавить основной шаблон данных. Используйте `{Binding}`, чтобы вызвать `ToString()`.

Оставьте тег ListView, заменив /> на > и добавив закрывающий тег </ListView>. Затем добавьте тег ListView.ItemTemplate, который будет содержать шаблон данных.

```
<ListView ItemsSource="{Binding Menu}" Margin="0,0,20,0">
    <ListView.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding}"/>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

Это основной шаблон данных, он похож на шаблон по умолчанию, отображавший ListViewItems.

Добавление {Binding} без пути вызывает метод ToString() связанного объекта.

Отредактируйте шаблон, добавив к меню цвет.

Можно связать отдельные теги Run. Редактируются цвет, шрифты и другие свойства.

```
<DataTemplate>
    <TextBlock>
        <Run Text="{Binding Meat}" Foreground="Blue"/><Run Text=" on "/>
        <Run Text="{Binding Bread}" FontWeight="Light"/><Run Text=" with "/>
        <Run Text="{Binding Condiment}" Foreground="Red" FontWeight="ExtraBold"/>
    </TextBlock>
</DataTemplate>
```

Замените <DataTemplate>, оставив остальную часть ListView без изменений.

Roast beef on rye with french dressing

Ham on pumpernickel with yellow mustard

Творите! Шаблон данных может содержать любые элементы.

```
<DataTemplate>
    <StackPanel Orientation="Horizontal">
        <StackPanel>
            <TextBlock Text="{Binding Bread}"/>
            <TextBlock Text="{Binding Bread}"/>
            <TextBlock Text="{Binding Bread}"/>
        </StackPanel>
        <Ellipse Fill="DarkSlateBlue" Height="Auto" Width="10" Margin="10,0"/>
        <Button Content="{Binding Condiment}" FontFamily="Segoe Script"/>
    </StackPanel>
</DataTemplate>
```

Свойство Content объекта DataTemplate может содержать один элемент, поэтому для набора элементов потребуется контейнер StackPanel.

white white white honey mustard

rye rye rye french dressing

Часто Задаваемые Вопросы

В: Итак, для компоновки страницы можно использовать StackPanel или Grid. Я могу пользоваться статическими ресурсами XAML или полями в коде. Я могу задавать свойства элементов или применять связывание. Зачем нужны разные способы выполнения одинаковых действий?

О: Так как C# и XAML гибкие инструменты. Эта гибкость позволяет проектировать качественные страницы, работающие на различных устройствах. Вам предоставляется инструментарий для *корректного* создания страниц. Воспринимайте его как набор вариантов, из которых можно выбрать наиболее подходящий под конкретную задачу.

В: Я все еще не понимаю, как работают статические ресурсы. Что происходит при добавлении тега внутрь `<Page.Resources>`?

О: При этом обновляется объект `Page`. Найдите ресурс `AppName`, через который вы задали заголовок страницы:

```
<x:String x:Key="AppName">Welcome to Sloppy
Joe's</x:String>
```

Теперь посмотрите на код, который IDE добавила как часть шаблона Basic Page, и найдите, где этот ресурс применяется:

```
<TextBlock x:Name="pageTitle" Grid.
Column="1"
      Text="{StaticResource AppName}"
      Style="{StaticResource
PageHeaderTextStyle}"/>
```

Ресурс задает текст на странице. Что же происходит внутри? Поместите точку останова в обработчик события кнопки, запустите код и щелкните на кнопке. Добавьте this.Resources["AppName"] в окно Watch, и вы увидите, что он содержит ссылку на строку. Так же работают все прочие ресурсы: создается объект, добавляемый к коллекции `Resources`.

В: Можно ли писать `{StaticResource}` в моем собственном коде или этот синтаксис допустим только в шаблонах, таких как Blank Page?

О: Вы можете без проблем создавать и использовать ресурсы где угодно. В шаблоне Blank Page, как и в остальных шаблонах, нет ничего особенного. Они используют обычный код XAML и C# и не делают ничего, что вы не можете сделать своими руками.

В: Я задал имя ресурса `MenuMaker` как `x:Name`, а ресурс `AppName` использует `x:Key`. В чем разница?

О: Свойство `x:Key` статического ресурса добавляет его в коллекцию `Resources` по указанному ключу, но поле при этом не создается (и вы не можете ввести `AppName` в код C#, к нему можно обращаться только через коллекцию `Resources`). Свойство `x:Name` добавляет ресурс в коллекцию `Resources`, создавая поле для объекта `Page`. Именно это позволило вызвать метод `UpdateMenu()` для статического ресурса `MenuMaker`.

В: Должен ли путь связывания относиться к типу string?

О: Нет, связать можно свойство любого типа при условии, что оно допускает конвертацию. В противном случае данные будут проигнорированы. И помните, что не все свойства ваших элементов являются текстом. Пусть у вас есть контекст данных `EnableMyObject` типа bool. Его можно связать с любым булевым свойством, например `IsEnabled`. Это управляет доступом к элементу на основе свойства `EnableMyObject`:

```
IsEnabled="{Binding EnableMyObject}"
```

Если привязать это к текстовому свойству, тогда будет выводиться True или False.

В: Почему IDE отображает данные формы после добавления статического ресурса и задания контекста данных в XAML, но не реагирует на действия с кодом C#?

О: Так как IDE понимает код XAML, снабженный информацией, необходимой для создания визуализирующих страницу объектов. После добавления к коду XAML ресурса `MenuMaker` IDE создает экземпляр `MenuMaker`. Но это невозможно сделать для оператора `new` в конструкторе, так как рядом может находиться множество других операторов. IDE запускает код C# только при выполнении программы. Но если добавить статический ресурс к странице, IDE его создаст, как создает экземпляры TextBlock, StackPanel и прочих элементов страницы. Она задает свойства элементов управления для отображения их в конструкторе. Поэтому когда вы задаете контекст данных и путь связывания, в конструкторе отображаются пункты меню.

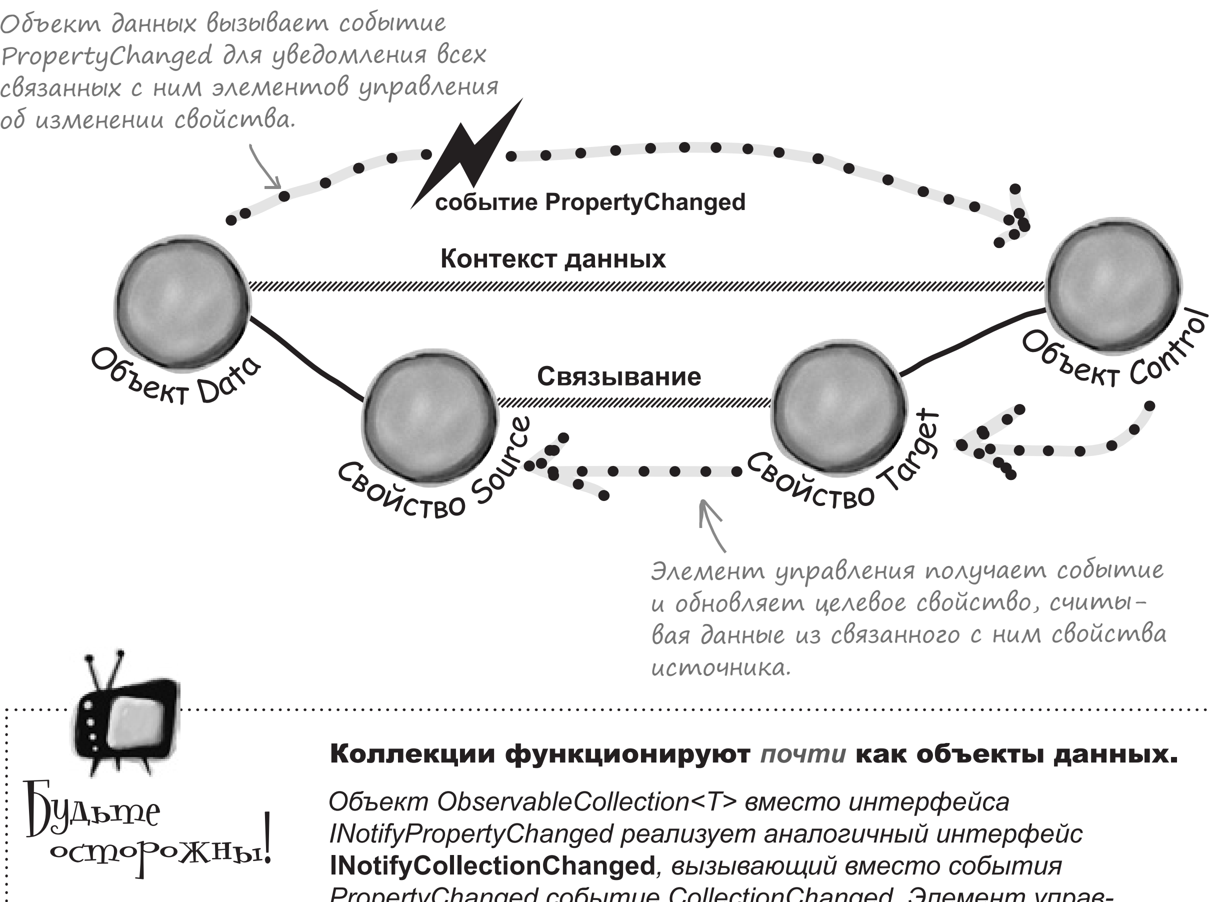
При первой загрузке страницы создаются экземпляры статических ресурсов, которыми могут в любой момент воспользоваться объекты приложения.

Слегка смущает название «статические ресурсы». Статические ресурсы создаются для каждого экземпляра; это не статические поля!

Интерфейс INotifyPropertyChanged

При обновлении меню классом `MenuMaker` обновляется связанный с ним класс ListView. Но одновременно `MenuMaker` обновляет свойство `GeneratedDate`. Почему же не обновляется связанный с ним элемент TextBlock? Дело в том, что при каждом изменении `ObservableCollection` **возникает событие**, рассказывающее всем связаным элементам управления об изменении их данных, — так же как `Button` при щелчке порождает событие `Click`, а `Timer` при завершении интервала — событие `Tick`. Любое добавление, перемещение и удаление элеметов `ObservableCollection` вызывает событие.

Объекты данных можно заставить уведомлять целевые свойства об изменениях. Достаточно **реализовать интерфейс `INotifyPropertyChanged`**, содержащий единственное событие `PropertyChanged`. Вызывайте это событие при любом изменении свойств, и связанные элементы управления начнут обновляться автоматически.



Коллекции функционируют *почти* как объекты данных.

Объект ObservableCollection<T> вместо интерфейса INotifyPropertyChanged реализует аналогичный интерфейс **INotifyCollectionChanged***, вызывающий вместо события PropertyChanged событие CollectionChanged. Элемент управления ищет это событие, так как ObservableCollection реализует интерфейс INotifyCollectionChanged. Присвоение свойству DataContext элемента ListView объекта INotifyCollectionChanged заставляет элемент реагировать на данные события.*

Предупредим MenuMaker об изменении свойства GeneratedDate

Интерфейс INotifyPropertyChanged находится в пространстве имен System.ComponentModel, поэтому добавьте в верхнюю часть файла класса MenuMaker оператор using:

```
using System.ComponentModel;
```

Заставим класс MenuMaker реализовывать INotifyPropertyChanged и автоматически реализуем его средствами IDE:

```
class MenuMaker : INotifyPropertyChanged
{
```

Implement interface 'INotifyPropertyChanged'
Explicitly implement interface 'INotifyPropertyChanged'

Первый вызов события

Обработчики событий мы пишем начиная с главы 1, но вызвать событие вам предстоит впервые. Механизм вызова и происходящие при этом процессы будут описаны в главе 15. А пока вам достаточно знать, что событие может входить в интерфейс и что метод OnPropertyChanged() следует стандартной процедуре C# по вызову событий для других объектов.

Это немного отличается от того, что вы видели в главах 7 и 8. Вместо методов и свойств добавьте событие:

```
public event PropertyChangedEventHandler PropertyChanged;
```

Затем добавьте метод OnPropertyChanged(), который будет вызывать событие PropertyChanged.

```
private void OnPropertyChanged(string propertyName) {
    PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
    if (propertyChangedEvent != null) {
        propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Стандартный шаблон .NET для вызова событий.

Теперь уведомим связанный элемент управления об изменении свойства, вызвав OnPropertyChanged() с именем этого свойства. Мы хотим, чтобы связанный с GeneratedDate элемент TextBlock обновлял данные при каждом обновлении меню. Для этого нужно добавить одну строку в конец метода UpdateMenu():

```
public void UpdateMenu() {
    Menu.Clear();
    for (int i = 0; i < NumberOfItems; i++) {
        Menu.Add(CreateMenuItem());
    }
    GeneratedDate = DateTime.Now;

    OnPropertyChanged("GeneratedDate");
}
```

Не забудьте реализовать INotifyPropertyChanged.

Связывание данных работает при условии реализации этого интерфейса. Без вставки `: INotifyPropertyChanged` в объявление класса связанные элементы управления не обновятся, несмотря на вызов событий PropertyChanged.

Теперь данные будут обновляться при генерации меню.

Завершим преобразование игры Go Fish! в приложение для магазина Windows. Вам потребуется добавить в код XAML связывание данных, скопировать все классы и перечисления из игры Go Fish! в главе 8 (или скачать их с нашего сайта) и обновить классы `Player` и `Game`.

1. **Добавим файлы классов и изменим их пространство имен.**
Добавьте эти файлы из главы 8 к вашему проекту: *Values.cs*, *Suits.cs*, *Card.cs*, *Deck.cs*, *CardComparer_bySuit.cs*, *CardComparer_byValue.cs*, *Game.cs* и *Player.cs*. Можно воспользоваться командой Add Existing Item в окне Solution Explorer, но нам нужно **изменить пространство имен** файлов в соответствии с новым проектом.

Постройте проект. Появятся сообщения об ошибках в файлах *Game.cs* и *Player.cs*:

1	The type or namespace name 'Forms' does not exist in the namespace 'System.Windows' (are you missing an assembly reference?)
2	The type or namespace name 'TextBox' could not be found (are you missing a using directive or an assembly reference?)
3	The type or namespace name 'TextBox' could not be found (are you missing a using directive or an assembly reference?)

2. **Удалим все ссылки на объекты и классы WinForms.**
Мы больше не работаем с формами, поэтому удалите строку `using System.Windows.Forms;` из верхней части файлов *Game.cs* и *Player.cs*. Кроме этого, нужно удалить все упоминания об элементе TextBox. Класс `Game` должен использовать `INotifyPropertyChanged` и `ObservableCollection<T>`, поэтому добавьте в верхнюю часть файла *Game.cs* строки:

```
using System.ComponentModel;
using System.Collections.ObjectModel;
```

3. **Добавим статический ресурс и зададим контекст данных.**
Добавьте в код XAML экземпляр `Game` (как статический ресурс) и используйте его в качестве контекста данных для содержащей страницу Go Fish! сетки. Вот код XAML этого статического ресурса: `<local:Game x:Name="game"/>`. Кроме того, вам потребуется новый конструктор, не имеющий параметров:

```
public Game() {
    PlayerName = "Ed";
    Hand = new ObservableCollection<string>();
    ResetGame();
}
```

4. **Добавим в класс `Game` открытые свойства для связывания данных.**
Эти свойства нужно привязать к свойствам элементов управления страницы:

```
public bool GameInProgress { get; private set; }
public bool GameNotStarted { get { return !GameInProgress; } }
public string PlayerName { get; set; }
public ObservableCollection<string> Hand { get; private set; }
public string Books { get { return DescribeBooks(); } }
public string GameProgress { get; private set; }
```

Воспользуемся связыванием для изменения доступа к элементам управления.
Элемент TextBox «Your Name» и кнопка Button «Start the game!» должны быть доступны до начала игры, а доступ к элементу ListBox «Your hand» и кнопке Button «Ask for a card» требуется только во время игры. Добавьте в класс `Game` свойство `GameInProgress`. Посмотрите на свойство `GameNotStarted`. Проанализируйте, как оно работает, и добавьте следующее связывание к элементам TextBox, ListBox и двум элементам Button:

Их должно быть два.

```
IsEnabled="{Binding GameInProgress}"   IsEnabled="{Binding GameNotStarted}"
IsEnabled="{Binding GameInProgress}"   IsEnabled="{Binding GameNotStarted}"
```

Отредактируем класс `Player`, заставив объект `Game` отображать прогресс игры.
В версии для WinForms класс `Player` использовал TextBox как параметр для своего конструктора. Теперь пусть он принимает ссылку на класс `Game`, сохраняя ее в закрытом поле. (Ниже приведен метод `StartGame()`, в котором этот новый конструктор добавляет игроков.) Найдите строки со ссылкой на TextBox и замените их вызовом метода `AddProgress()` объекта `Game` .

7 Отредактируем класс `Game`.
Пусть метод `PlayOneRound()` перестанет возвращать значение и показывает прогресс методом `AddProgress()` вместо TextBox. При выигрыше игрока отображается прогресс, игра сбрасывается и происходит возврат управления. При проигрыше обновляется коллекция `Hand` и описываются взятки.

Также нужно добавить/обновить эти четыре метода и понять, зачем они нужны и как работают.

```
public void StartGame() {
    ClearProgress();
    GameInProgress = true;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    Random random = new Random();
    players = new List<Player>();
    players.Add(new Player(PlayerName, random, this));
    players.Add(new Player("Bob", random, this));
    players.Add(new Player("Joe", random, this));
    Deal();
    players[0].SortHand();
    Hand.Clear();
    foreach (String cardName in GetPlayerCardNames())
        Hand.Add(cardName);
    if (!GameInProgress)
        AddProgress(DescribePlayerHands());
    OnPropertyChanged("Books");
}

public void ClearProgress() {
    GameProgress = String.Empty;
    OnPropertyChanged("GameProgress");
}
```

```
public void AddProgress(string progress)
{
    GameProgress = progress +
          Environment.NewLine +
          GameProgress;
    OnPropertyChanged("GameProgress");
}
```

Также нужно реализовать интерфейс `INotifyPropertyChanged` и добавить метод `OnPropertyChanged()`, использовавшийся в классе `MenuMaker`. Его применяют обновленные методы, и будет применять отредактированный метод `PullOutBooks()`.

```
public void ResetGame() {
    GameInProgress = false;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    books = new Dictionary<Values, Player>();
    stock = new Deck();
    Hand.Clear();
}
```

Упражнение Решение

Вот код, который вы должны были написать:

```
private void startButton_Click(object sender, RoutedEventArgs e) {
    game.StartGame();
}
private void askForACard_Click(object sender, RoutedEventArgs e) {
    if (cards.SelectedIndex >= 0)
        game.PlayOneRound(cards.SelectedIndex);
}
private void cards_DoubleTapped(object sender, DoubleTappedRoutedEventArgs e) {
    if (cards.SelectedIndex >= 0)
        game.PlayOneRound(cards.SelectedIndex);
}
```

Изменения, которые следовало внести в класс `Player`:

```
class Player {
    private string name;
    public string Name { get { return name; } }
    private Random random;
    private Deck cards;
    private Game game;
    public Player(String name, Random random, Game game) {
        this.name = name;
        this.random = random;
        this.game = game;
        this.cards = new Deck(new Card[] { });
        game.AddProgress(name + " has just joined the game");
    }
    public Deck DoYouHaveAny(Values value)
    {
        Deck cardsIHave = cards.PullOutValues(value);
        game.AddProgress(Name + " has " + cardsIHave.Count + " " + Card.Plural(value));
        return cardsIHave;
    }
    public void AskForACard(List<Player> players, int myIndex, Deck stock, Values value) {
        game.AddProgress(Name + " asks if anyone has a " + value);
        int totalCardsGiven = 0;
        for (int i = 0; i < players.Count; i++) {
            if (i != myIndex) {
                Player player = players[i];
                Deck CardsGiven = player.DoYouHaveAny(value);
                totalCardsGiven += CardsGiven.Count;
                while (CardsGiven.Count > 0)
                    cards.Add(CardsGiven.Deal());
            }
        }
        if (totalCardsGiven == 0) {
            game.AddProgress(Name + " must draw from the stock.");
            cards.Add(stock.Deal());
        }
    }

    // ... остальная часть класса Player осталась без изменений ...
```

Изменения, которые нужно внести в код XAML:

```
<Grid Grid.Row="1" Margin="120,0,60,60" DataContext="{StaticResource ResourceKey=game}" >

    <TextBlock Text="Your Name" Margin="0,0,0,20"
        Style="{StaticResource SubheaderTextStyle}"/>

    <StackPanel Orientation="Horizontal" Grid.Row="1">
        <TextBox x:Name="playerName" FontSize="24" Width="500" MinWidth="300"
         Text="{Binding PlayerName, Mode=TwoWay}" IsEnabled="{Binding GameNotStarted}" />

        <Button x:Name="startButton" Margin="20,0" IsEnabled="{Binding GameNotStarted}"
              Content="Start the game!" Click="startButton_Click" />
    </StackPanel>

    <TextBlock Text="Game progress"
        Style="{StaticResource SubheaderTextStyle}" Margin="0,20,0,20" Grid.Row="2" />

    <ScrollViewer Grid.Row="3" FontSize="24" Background="White" Foreground="Black"
            Content="{Binding GameProgress}" />

    <TextBlock Text="Books" Style="{StaticResource SubheaderTextStyle}"
            Margin="0,20,0,20" Grid.Row="4"/>

    <ScrollViewer FontSize="24" Background="White" Foreground="Black"
            Grid.Row="5" Grid.RowSpan="2" Content="{Binding Books}" />

    <TextBlock Text="Your hand" Style="{StaticResource SubheaderTextStyle}"
            Grid.Row="0" Grid.Column="2" Margin="0,0,0,20"/>

    <ListBox Background="White" FontSize="24" Height="Auto" Margin="0,0,0,20"
            x:Name="cards" Grid.Row="1" Grid.RowSpan="5" Grid.Column="2"
            ItemsSource="{Binding Hand}" IsEnabled="{Binding GameInProgress}"
            DoubleTapped="cards_DoubleTapped" />

    <Button x:Name="askForACard" Content="Ask for a card" HorizontalAlignment="Stretch"
            VerticalAlignment="Stretch" Grid.Row="6" Grid.Column="2"
            Click="askForACard_Click" IsEnabled="{Binding GameInProgress}" />

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="5*"/>
        <ColumnDefinition Width="40"/>
        <ColumnDefinition Width="2*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto" MinHeight="150" />
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
</Grid>
```

Контекстом данных для сетки является класс Game, так как все привязки выполнены к свойствам этого класса.

Элемент TextBox имеет двустороннее связывание с PlayerName.

Обработчик события Click для кнопки Start.

Прогресс игры и просмотр взяток привязаны к Progress и Books.

Свойство IsEnabled управляет доступом к элементу управления. Это булево свойство, поэтому его можно привязать к свойству, включающему и выключающему доступ к элементу управления.

Все изменения в классе `Game`, включая данный в приведенных выше инструкциях код.

```
using System.ComponentModel;
using System.Collections.ObjectModel;

class Game : INotifyPropertyChanged {
    private List<Player> players;
    private Dictionary<Values, Player> books;
    private Deck stock;
    public bool GameInProgress { get; private set; }
    public bool GameNotStarted { get { return !GameInProgress; } }
    public string PlayerName { get; set; }
    public ObservableCollection<string> Hand { get; private set; }
    public string Books { get { return DescribeBooks(); } }
    public string GameProgress { get; private set; }

    public Game() {
        PlayerName = "Ed";
        Hand = new ObservableCollection<string>();
        ResetGame();
    }

    public void AddProgress(string progress) {
        GameProgress = progress + Environment.NewLine + GameProgress;
        OnPropertyChanged("GameProgress");
    }

    public void ClearProgress() {
        GameProgress = String.Empty;
        OnPropertyChanged("GameProgress");
    }

    public void StartGame() {
        ClearProgress();
        GameInProgress = true;
        OnPropertyChanged("GameInProgress");
        OnPropertyChanged("GameNotStarted");
        Random random = new Random();
        players = new List<Player>();
        players.Add(new Player(PlayerName, random, this));
        players.Add(new Player("Bob", random, this));
        players.Add(new Player("Joe", random, this));
        Deal();
        players[0].SortHand();
        Hand.Clear();
        foreach (String cardName in GetPlayerCardNames())
            Hand.Add(cardName);
        if (!GameInProgress)
            AddProgress(DescribePlayerHands());
        OnPropertyChanged("Books");
    }
```

Эти строчки нужны для интерфейса INotifyPropertyChanged и коллекции ObservableCollection.

Эти свойства применяются при связывании данных в XAML.

Это новый конструктор Game. Мы создали одну коллекцию и очистили ее при сбросе игры. Если мы создадим новый объект, форма потеряет ссылку на него и обновления прекратятся.

Эти методы обеспечивают связывание данных, отвечающих за прогресс игры. Новые строки добавляются наверх, смещая вниз старые действия элемента ScrollViewer.

Все написанные раннее XAML-программы можно обновить до приложений магазина Windows Store. Но это можно делать разными способами! Именно поэтому в данном упражнении так много кода.

Это метод StartGame(), очищающий индикатор выполнения, создающий игроков, раздающий карты и обновляющий индикатор и взятки.

Раньше здесь возвращалось значение Boolean, давая форме возможность обновить индикатор выполнения. Теперь достаточно вызвать AddProgress, а обновление произойдет за счет связывания данных.

```
public void PlayOneRound(int selectedPlayerCard) {
    Values cardToAskFor = players[0].Peek(selectedPlayerCard).Value;
    for (int i = 0; i < players.Count; i++) {
        if (i == 0)
            players[0].AskForACard(players, 0, stock, cardToAskFor);
        else
            players[i].AskForACard(players, i, stock);
        if (PullOutBooks(players[i])) {
            AddProgress(players[i].Name + " drew a new hand");
            int card = 1;
            while (card <= 5 && stock.Count > 0) {
                players[i].TakeCard(stock.Deal());
                card++;
            }
        }
        OnPropertyChanged("Books");
        players[0].SortHand();
        if (stock.Count == 0) {
            AddProgress("The stock is out of cards. Game over!");
            AddProgress("The winner is... " + GetWinnerName());
            ResetGame();
            return;
        }
    }
    Hand.Clear();
    foreach (String cardName in GetPlayerCardNames())
        Hand.Add(cardName);
    if (!GameInProgress)
        AddProgress(DescribePlayerHands());
}
```

Взятки изменились, о чем нужно сообщить форме, чтобы она обновила элемент ScrollViewer.

Это модификация метода PlayOneRound(), которая обновляет индикатор прогресса при завершении игры, а также взятки и карты на руках у игрока, если игра продолжается.

```
public void ResetGame() {
    GameInProgress = false;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    books = new Dictionary<Values, Player>();
    stock = new Deck();
    Hand.Clear();
}
```

Это метод ResetGame(), о котором шла речь в инструкциях. Он очищает данные о взятках, колоде и картах игрока.

```
public event PropertyChangedEventHandler PropertyChanged;
private void OnPropertyChanged(string propertyName) {
    PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
    if (propertyChangedEvent != null) {
        propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
    }
}

// ... остальная часть класса Game осталась без изменений ...
```

Это стандартный шаблон события PropertyChanged, с которым вы уже сталкивались в этой главе.

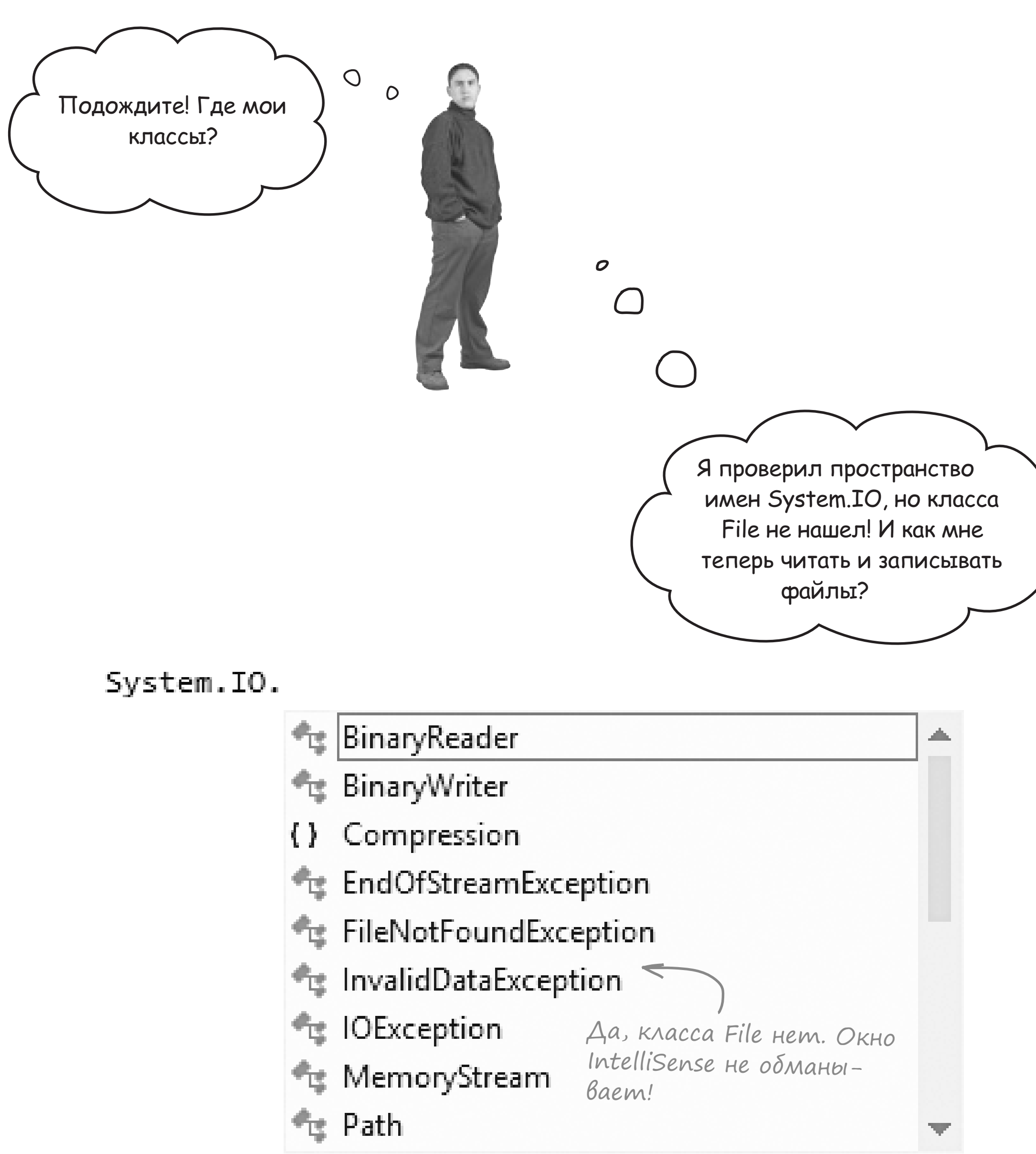
11 async, await и сериализация контрактов данных

Позвольте вас прервать

Никто не любит ждать... особенно пользователи.
Компьютеры умеют решать несколько задач одновременно, почему бы не добавить вашим приложениям аналогичную возможность? В этой главе вы увеличите «отзывчивость» приложений при помощи асинхронных методов. Вы научитесь использовать встроенные средства выбора файлов и диалоговые окна с сообщениями, а также асинхронный файловый ввод и вывод. Скомбинировав это с сериализацией, вы получите технологию создания современных приложений.

У Брайана проблемы с файлами

Брайан получил код XAML, выполнил связывание данных и приготовился портировать генератор оправданий в приложение Windows Store. И все бы хорошо, но...



И класса BinaryFormatter я тоже не нахожу. Как теперь сериализовать объекты?

```
System.Runtime.Serialization.
```

- CollectionDataContractAttribute
- ContractNamespaceAttribute
- DataContractAttribute
- DataContractResolver
- DataContractSerializer
- DataContractSerializerSettings
- DataMemberAttribute
- DateTimeFormat
- EmitTypeInformation

Впрочем, это выглядит многообещающе.

Приложения для магазина Windows улучшили мое приложение WinForms. Уверен, там есть хорошие инструменты... и веские причины отсутствия этих файлов.

Приложения для магазина Windows обладают потрясающими инструментами ввода/вывода.

Приложение для магазина Windows должно быть отзывчивым и интуитивно понятным. Вот почему .NET Framework для этих приложений содержит классы и методы, позволяющие отображать окна диалога и выполнять файловый ввод/вывод **асинхронно**. То есть при появлении окна диалога или в процессе записи файла приложение не блокируется. А благодаря **контрактам данных** для сериализации приложение записывает простые и понятные файлы.

Курсор в форме песочных часов означает, что программа заблокирована и не отвечает... все пользователи ненавидят эти часы! (Не так ли?)

Увеличение отзывчивости с оператором await

Что происходит при вызове метода MessageBox.Show() в программе WinForms? Все останавливается, и программа не продолжает работу, пока окно не закроется. Это пример максимальной неотзывчивости! Приложения же для магазина Windows должны быть отзывчивыми даже во время ожидания действий. Но некоторые вещи, например ожидание диалога, чтение и запись в файл, занимают изрядное время. Ситуацию, когда метод заставляет всю программу ждать завершения, называют **блокировкой**, и именно она делает программы максимально неотзывчивыми.

Избежать подобного поведения приложениям для магазина Windows помогают **оператор await и модификатор async**. Посмотрим на примере, как приложение может вызывать окно диалога, не блокируя свою работу. В этом нам поможет класс MessageDialog:

Объект MessageDialog создается аналогично тому, как вы создаете экземпляры любого другого класса.

Зададим конфигурацию объекта MessageDialog, дав ему сообщение и варианты ответов. Каждый ответ представляет собой объект UICommand.

```
MessageDialog dialog = new MessageDialog("Message");
dialog.Commands.Add(new UICommand("Response #1"));
dialog.Commands.Add(new UICommand("Response #2"));
dialog.Commands.Add(new UICommand("Response #3"));
dialog.DefaultCommandIndex = 1;
UICommand result = await dialog.ShowAsync() as UICommand;
```

Оператор **await** заставляет метод, запускающий этот код, остановиться и дождаться завершения метода ShowAsync(), в итоге метод блокируется, пока пользователь не выберет какую-либо команду. В это время остальная программа *продолжает отвечать на другие события*. Как только метод ShowAsync() вернет управление, вызывавший его метод начнет работу с прерванной точки (хотя он может и дожидаться завершения возникших в промежутке событий).

Метод, использующий оператор await, **должен объявляться с модификатором async**:

```
public async void ShowADialog() {
    // ... какой-то код ...
    UICommand result = await dialog.ShowAsync() as UICommand;
    // ... еще код:
}
```

Существуют варианты вызова метода, объявленного с модификатором async. Его можно вызвать как обычно. При этом, дойдя до оператора await, метод вернет управление, что помешает блокирующему вызову заморозить работу приложения.

Посмотрим, как это работает, **создав новое приложение Blank App** и добавив код XAML:

```
<StackPanel VerticalAlignment="Top" HorizontalAlignment="Center">
    <Button Click="Button_Click_1"  FontSize="36">Are you happy?</Button>
    <TextBlock x:Name="response" FontSize="36"/>
    <TextBlock x:Name="ticker"  FontSize="36"/>
</StackPanel>
```

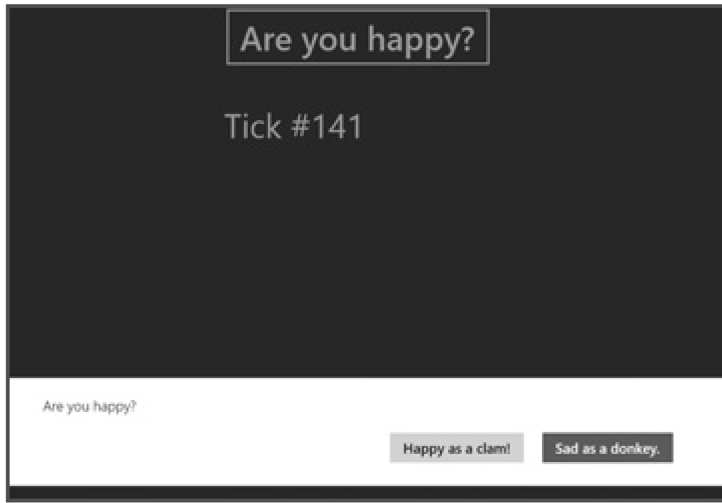
Упражнение!

Нужно добавить строку `using Windows.UI.Popups;`, так как объекты `MessageDialog` и `UICommand` находятся именно в этом пространстве имен.

```
DispatcherTimer timer = new DispatcherTimer();
private void Button_Click_1(object sender, RoutedEventArgs e) {
    timer.Tick += timer_Tick;
    timer.Interval = TimeSpan.FromMilliseconds(50);
    timer.Start();
    CheckHappiness();
}
int i = 0;
void timer_Tick(object sender, object e) {
    ticker.Text = "Tick #" + i++;
}
private async void CheckHappiness() {
    MessageDialog dialog = new MessageDialog("Are you happy?");
    dialog.Commands.Add(new UICommand("Happy as a clam!"));
    dialog.Commands.Add(new UICommand("Sad as a donkey."));
    dialog.DefaultCommandIndex = 1;
    UICommand result = await dialog.ShowAsync() as UICommand;
    if (result != null && result.Label == "Happy as a clam!")
        response.Text = "The user is happy";
    else
        response.Text = "The user is sad";
    timer.Stop();
}
```

Попробуйте вставить сюда строчку timer.Stop(). Таймер сразу же перестанет работать, так как помеченный модификатором async метод возвращает управление, достигнув оператора await.

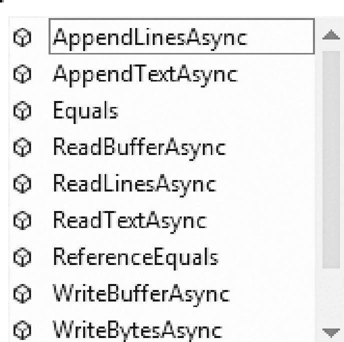
Запустив программу, вы увидите, что даже при открытом окне диалога таймер работает. Отзывчивость приложения сохранена! Таймер не прерывает отсчет, ожидая, пока вы выберете ответ в окне диалога и метод возобновит работу.



Чтение и запись файлов при помощи класса FileIO

Приложения WinForms пользуются классом `System.IO.File` для чтения и записи файлов, но в .NET Framework для приложений магазина Windows этот класс попросту отсутствует. И это правильно! Если воспользоваться методом `File.WriteAllText()` для записи гигантского файла, занимающего изрядную часть жесткого диска, программа перестанет отвечать.

Приложения для магазина Windows применяются **для записи и чтения из файла классы `Windows.Storage`**. В этом пространстве имен присутствует класс `FileIO`, включающий в себя ряд методов, которые уже встречались вам в окне IntelliSense.



Эти методы выглядят аналогично методам класса `File`. Класс `FileIO` class обладает методами `AppendLinesAsync()` и `ReadTextAsync()`, в то время как в классе `File` есть методы `AppendLines()` и `ReadText()`. Разница состоит в том, что эти методы объявляются с модификатором `async` и при чтении из файла используют оператор `await`. Это дает возможность написать код, осуществляющий чтение и запись без блокировки программы.

Средства выбора файлов

Окна MessageBoxes — отнюдь не единственный элемент, блокирующий нормальную работу программ WinForms. Тот же эффект вызывают окна для работы с файлами. В приложениях для магазина Windows существуют собственные **асинхронные** средства доступа к файлам и папкам, которые не вызывают блокировки. Вот как создается и используется класс `FileOpenPicker`, обеспечивающий поиск файла, который нужно открыть, и метод `ReadTextAsync()`, читающий текст из этого файла:

Свойства средства выбора настраиваются через инициализатор объекта. FileOpenPicker отображает файлы в виде списка и открывает папку пользователя documents library.

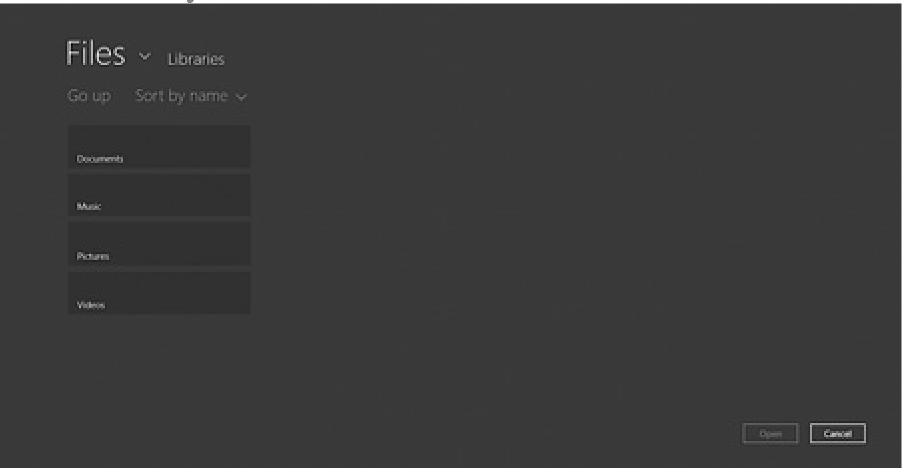
```
FileOpenPicker picker = new FileOpenPicker {
    ViewMode = PickerViewMode.List,
    SuggestedStartLocation = PickerLocationId.DocumentsLibrary
};
picker.FileTypeFilter.Add(".txt");
IStorageFile file = await picker.PickSingleFileAsync();
if (file != null) {
    string fileContents = await FileIO.ReadTextAsync(file);
}
```

У средства выбора есть коллекция FileTypeFilter, в которой перечислены типы доступных для загрузки файлов.

После выбора файла средство выбора возвращает интерфейс IStorageFile. Об этом пойдет речь через несколько страниц.

Для чтения содержимого файла можно передать ссылку IStorageFile непосредственно в метод FileIO.ReadTextAsync().

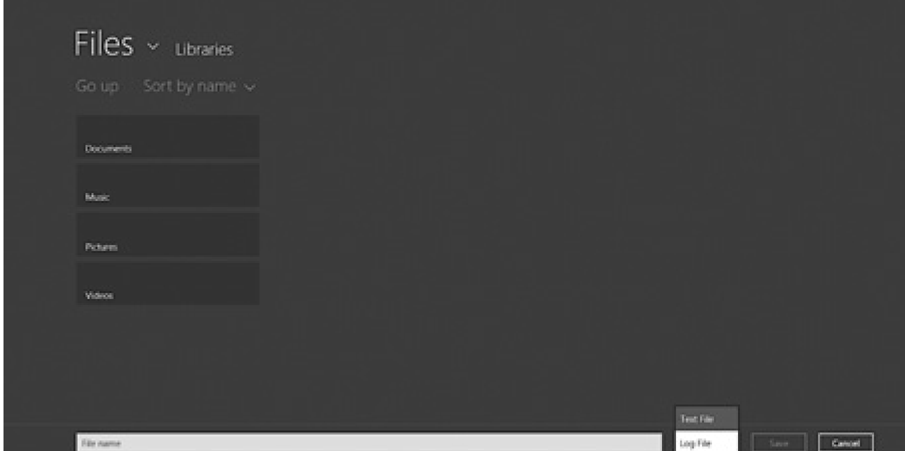
Вот как выглядит объект FileOpenPicker после его открытия.



Объект `FileSavePicker` позволяет пользователю выбрать сохраняемый файл. А вот как он в связке с методом `FileIO.WriteTextAsync()` применяется для записи текста в файл:

Объект FileSavePicker возвращает интерфейс IStorageFile. Он содержит всю информацию для чтения или записи в файл и передается непосредственно в метод WriteTextAsync().

```
FileSavePicker picker = new FileSavePicker {
    DefaultFileExtension = ".txt",
    SuggestedStartLocation = PickerLocationId.DocumentsLibrary
};
picker.FileTypeChoices.Add("Text File", new List<string>() { ".txt" });
picker.FileTypeChoices.Add("Log File",
                           new List<string>() { ".log", ".dat" });
IStorageFile saveFile = await picker.PickSaveFileAsync();
if (saveFile == null) return;
await FileIO.WriteTextAsync(saveFile, textToWrite);
```



Слегка усложненный текстовый редактор

Переделаем Simple Text Editor из главы 9 в приложение для магазина Windows. Для загрузки и сохранения файлов мы воспользуемся классами `FileIO`, `FileOpenPicker` и `FileSavePicker`. Но начнем мы с создания главной страницы. Так как приложение для магазина Windows Store должно открывать и сохранять файлы, **ему нужна панель с кнопками Open и Save.**

Элемент управления AppBar во многом напоминает ScrollViewer или Border, так как может содержать другие элементы. Он функционирует как любая другая панель приложения. Вам всего лишь нужно добавить код в раздел `<BottomAppBar>` или `<TopAppBar>` вашей страницы.

1. Создайте новый проект Blank App и замените страницу *MainPage.xaml* шаблоном Basic Page. Вот код XAML, который следует вставить в этот шаблон:

```
<Grid Grid.Row="1" Margin="120,0,60,60">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <TextBlock x:Name="filename" Margin="10" Style="{StaticResource TitleTextStyle}">
        Untitled
    </TextBlock>
    <Border Margin="10" Grid.Row="1">
        <TextBox x:Name="text" AcceptsReturn="True"
             ScrollViewer.VerticalScrollBarVisibility="Visible"
             ScrollViewer.HorizontalScrollBarVisibility="Visible"
             TextChanged="text_TextChanged" />
    </Border>
</Grid>
```

Свойство AcceptsReturn обеспечивает многострочный ввод в элемент TextBox.

Элемент TextBox может использовать горизонтальную и вертикальную прокрутку. Эти свойства делают их видимыми.

Щелкните правой кнопкой мыши на строке `text_TextChanged` и выберите в меню Navigate to Event Handler. IDE создаст обработчик события `TextChanged` для TextBox.

2. В окне Document Outline выделите Page (или любой элемент управления) и несколько раз нажмите клавишу Escape. В окне Properties раскройте раздел Common и найдите свойство `BottomAppBar`:

◢ Common
BottomAppBar [New] □

Щелкните на [New] и добавьте нижнюю панель приложения. IDE вставит на страницу код:

```
<common:LayoutAwarePage.BottomAppBar>
    <AppBar/>
</common:LayoutAwarePage.BottomAppBar>
```

3 В редакторе XAML замените <AppBar/>. Используйте <StackPanel> с кнопками Open и Save:

```
<common:LayoutAwarePage.BottomAppBar>
    <AppBar x:Name="bottomAppBar" Padding="10,0,10,0">
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Right">
            <Button x:Name="openButton" Click="openButton_Click"
                    Style="{StaticResource OpenFileAppBarButtonStyle}"/>
            <Button x:Name="saveButton" IsEnabled="false"
                    Click="saveButton_Click"
                    Style="{StaticResource SaveAppBarButtonStyle}"/>
        </StackPanel>
    </AppBar>
</common:LayoutAwarePage.BottomAppBar>
```

Эти стили будут подчеркнуты волнистой синей линией, пока вы не снимите комментарии в файле StandardStyles.xaml.

4 Кажется, у нас отсутствуют два статических ресурса, `OpenFileAppBarButtonStyle` и `SaveAppBarButtonStyle`! Это нормально. Шаблон Blank App ставится **файлом** *StandardStyles.xaml*, вы увидите его в разделе Common окна Solution Explorer. Большинство строк этого файла превращены в комментарии, но вы можете *снять эту пометку* с любого нужного вам стиля.

Выберите команды **Edit→Find и Replace→Quick Find** и найдите `OpenFileAppBarButtonStyle`:

Убедитесь, что вы ищете то, что нужно.

Нажимайте ➔, пока не дойдете до тега <Style> в файле *StandardStyles.xaml*:

```
<Style x:Key="OpenFileAppBarButtonStyle" TargetType="ButtonBase" BasedOn="{StaticResource AppBarButtonStyle}">
    <Setter Property="AutomationProperties.AutomationId" Value="OpenFileAppBarButton"/>
    <Setter Property="AutomationProperties.Name" Value="Open File"/>
    <Setter Property="Content" Value="&#xE1A5;"/>
</Style>
```

Добавьте `-->` и `<!--` для снятия комментария со стиля. Именно так заканчиваются и начинаются комментарии в XML:

```
-->
<Style x:Key="OpenFileAppBarButtonStyle" TargetType="ButtonBase" BasedOn="{StaticResource AppBarButtonStyle}">
    <Setter Property="AutomationProperties.AutomationId" Value="OpenFileAppBarButton"/>
    <Setter Property="AutomationProperties.Name" Value="Open File"/>
    <Setter Property="Content" Value="&#xE1A5;"/>
</Style>
<!--
```

Сделайте то же самое для `SaveAppBarButtonStyle`.

Выделите в окне XAML тег <AppBar>, чтобы отобразить панель приложения в конструкторе.

Отобразите в конструкторе панель приложения, выделив ее код XAML, и дважды щелкните на каждой из кнопок, чтобы добавить обработчик события Click.

Вот программный код приложения. Свойство `TextBox.Text` модифицирует текст в текстовом поле. Мы не пользуемся связыванием данных, а редактируем свойство объекта, чтобы сохранить сходство с кодом приложения Simple Text Editor в главе 9. Это обеспечит базу для сравнения приложений WinForms с приложениями для магазина Windows. В верхнюю часть файла нужно также добавить операторы `using`:

```
using Windows.System;
using Windows.Storage;
using Windows.Storage.Pickers;
using Windows.UI.Popups;
```

Переделка уже готовой программы с применением новых технологий позволяет лучше усвоить материал.

Вот остальная часть кода, который нужно поместить в класс `MainPage`.

*Нам потребуются эти три поля. Поля типа boolean будут добавлять * в конец имени файла. Интерфейс IStorageFile фиксирует сохраняемый файл, избавляя от необходимости отображать инструмент выбора сохраняемого файла.*

```
bool textChanged = false;
bool loading = false;
IStorageFile saveFile = null;

private async void openButton_Click(object sender, RoutedEventArgs e) {
    if (textChanged) {
        MessageDialog overwriteDialog = new MessageDialog(
            "Хотите загрузить новый файл, несмотря на несохраненные изменения?");
        overwriteDialog.Commands.Add(new UICommand("Yes"));
        overwriteDialog.Commands.Add(new UICommand("No"));
        overwriteDialog.DefaultCommandIndex = 1;
        UICommand result = await overwriteDialog.ShowAsync() as UICommand;
        if (result != null && result.Label == "No")
            return;
    }
    OpenFile();
}
```

Если в методе присутствует оператор `await`, в объявлении метода __должен быть__ модификатор `async`.

Кнопка Open при наличии несохраненных изменений вызывает окно диалога. Если пользователь подтверждает свое намерение, вызывается метод OpenFile(), чтобы отобразить средство выбора файла.

```
private void saveButton_Click(object sender, RoutedEventArgs e) {
    SaveFile();
}
```

Кнопка Save вызывает метод SaveFile().

```
private void text_TextChanged(object sender, TextChangedEventArgs e) {
    if (loading) {
        loading = false;
        return;
    }
    if (!textChanged) {
        filename.Text += "*";
        saveButton.IsEnabled = true;
        textChanged = true;
    }
}
```

*После изменения текста в конец имени файла добавляется *, но только один раз. За изменением текста следит поле textChanged.*

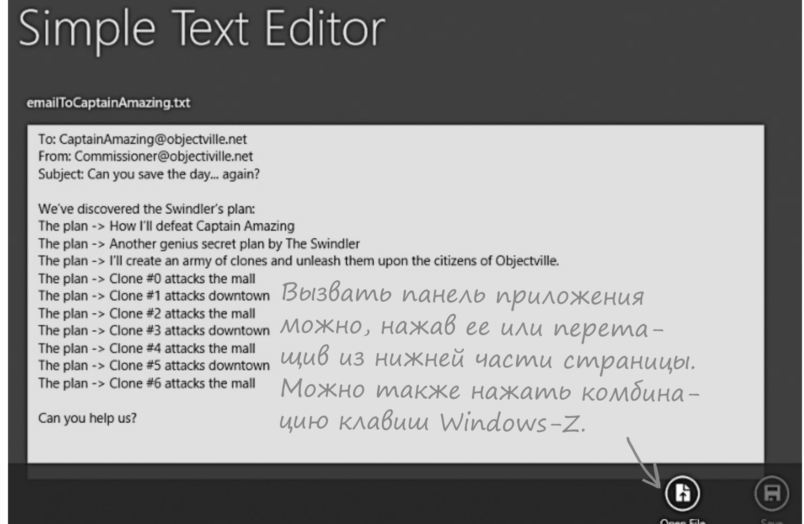
*Поле loading не дает добавить * сразу же после загрузки (ведь при этом меняется текст, что становится причиной вызова события). Сможете объяснить, как это работает?*

```
private async void OpenFile() {
    FileOpenPicker picker = new FileOpenPicker {
        ViewMode = PickerViewMode.List,
        SuggestedStartLocation = PickerLocationId.DocumentsLibrary
    };
    picker.FileTypeFilter.Add(".txt");
    picker.FileTypeFilter.Add(".xml");
    picker.FileTypeFilter.Add(".xaml");
    IStorageFile file = await picker.PickSingleFileAsync();
    if (file != null) {
        string fileContents = await FileIO.ReadTextAsync(file);
        loading = true;
        text.Text = fileContents;
        textChanged = false;
        filename.Text = file.Name;
        saveFile = file;
    }
}
private async void SaveFile() {
    if (saveFile == null) {
        FileSavePicker picker = new FileSavePicker {
            DefaultFileExtension = ".txt",
            SuggestedStartLocation = PickerLocationId.DocumentsLibrary
        };
        picker.FileTypeChoices.Add("Text File", new List<string>() { ".txt" });
        picker.FileTypeChoices.Add("XML File", new List<string>() { ".xml", ".xaml" });
        saveFile = await picker.PickSaveFileAsync();
        if (saveFile == null) return;
    }
    await FileIO.WriteTextAsync(saveFile, text.Text);
    await new MessageDialog("Wrote " + saveFile.Name).ShowAsync();
    textChanged = false;
    filename.Text = saveFile.Name;
}
```

Методы OpenFile() и SaveFile() напоминают код с предыдущей страницы. Они отображают средство выбора, а затем прибегают к методам FileIO для загрузки или сохранения файла.

Отобразить панель текущего приложения можно, нажав кнопки Windows и Z.

Вы закончили. Запустим программу!



Вызвать панель приложения можно, нажав ее или перетащив из нижней части страницы. Можно также нажать комбинацию клавиш Windows-Z.

Я понимаю, каким образом вставить в Генератор оправданий панель приложения, окна диалога с сообщениями и асинхронное программирование! Но я не вижу класса BinaryFormatter. Как же мне сериализовать мои объекты Excuse?

Разве не здорово было бы, если бы существовал способ сохранения объектов, не только использующий все удобства двоичной сериализации, но и создающий файлы, доступные для чтения и редактирования?

Такой способ есть! Это сериализация контрактов данных.

Записанный текстовый файл можно открыть в приложении Блокнот и посмотреть содержимое. Но в такой файл попадет весь код разметки данных.

Удобным механизмом является двоичная сериализация с классом BinaryFormatter. Но у нее есть недостатки! Двоичные файлы **уязвимы**. Небольшое изменение в одном классе — и вы уже не сможете загрузить файлы обратно! Кроме того, вы уже видели, как выглядят такие файлы в приложении Блокнот: они недоступны ни для чтения, ни для редактирования.

На помощь приходит сериализация контрактов данных. Это настоящая сериализация, так как объекты графов записываются автоматически. Но в результате вы получаете файлы XML, которые достаточно легко прочитать и даже отредактировать вручную (особенно если вы умеете работать с кодом XAML!).

При двоичной сериализации записываются «чистые» данные: байты склеиваются друг с другом и записываются в файл, дополняемые информацией для двоичного модуля форматирования о том, каким членам классов в графе объектов принадлежат те или иные байты. Достаточно внести изменение хотя бы в один класс, и последовательность байтов нарушится, что не даст провести десериализацию.

Контракт данных — абстрактное определение данных вашего объекта

Контрактом данных называется присоединенное к классу **формальное соглашение**. Данные для считывания и записи в процессе сериализации контракт указывает через атрибуты [DataContract] и [DataMember].

Контракт данных для сериализации экземпляров класса создается добавлением атрибута [DataContract] наверх класса и атрибутов [DataMember] ко всем сериализуемым членам класса. Вот простой класс Guy с контрактом данных:

```
using System.Runtime.Serialization;

[DataContract]
class Guy {

   [DataMember]
   public string Name { get; private set; }

   [DataMember]
   public int Age { get; private set; }

   [DataMember]
   public decimal Cash { get; private set; }

   public Guy(string name, int age, decimal cash) {
       Name = name; Age = age; Cash = cash;
   }
}
```

Атрибуты [DataContract] и [DataMember] находятся в пространстве имен System.Runtime.Serialization.

Атрибут [DataContract] устанавливает для этого класса контракт данных.

Все члены класса, которые во время сериализации нужно сохранить или восстановить, добавляются к контракту с атрибутом [DataMember].

В представленном ниже фрагменте XML класса <Guy> xmlns называется атрибутом. В файлах XAML вы можете найти теги с атрибутами Fill, Text и x:Name. В конструкторе IDE они называются свойствами, так как они задают свойства объектов.

Сериализация контрактов данных использует файлы XML

К счастью, вы уже много знаете о файлах XML, так как XML является основой языка XAML. Во всех файлах XML используются открывающие и закрывающие теги и определяющие данные атрибуты. У каждого члена есть имя, но имя требуется и самому контракту, точнее не имя, а **уникальное пространство имен**, так как сериализатор должен отличать файлы данных для контракта от остальных XML-файлов. Вот XML-файл, созданный при сериализации класса Guy, мы только добавили пробелы и переносы строк:

```
<Guy xmlns="http://schemas.datacontract.org/2004/07/XamlGuySerializer"
     xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
   <Age>37</Age>
   <Cash>164.38</Cash>
   <Name>Joe</Name>
</Guy>
```

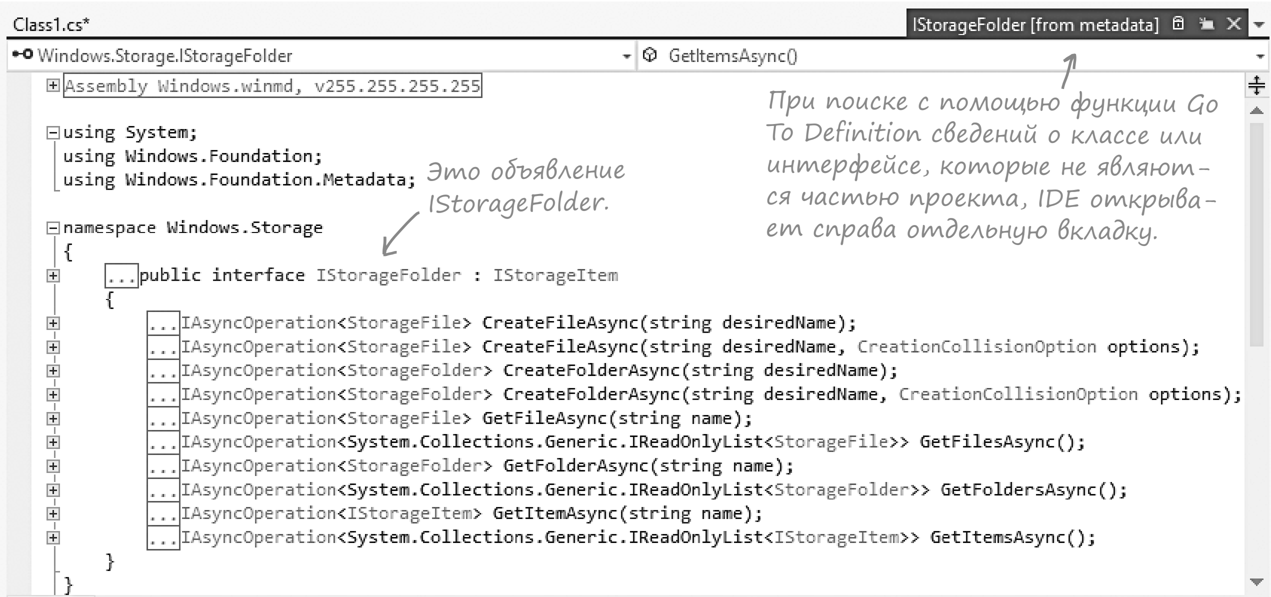
Каждый член данных получает собственный тег. В результате текст страновится читаемым!

Мы присвоили проекту имя XamlGuySerializer, образовав таким образом пространство имен для контракта.

Асинхронные методы для поиска и открытия файлов

Сериализация контрактов данных напоминает двоичную сериализацию. Нужно открыть файл, создать поток чтения или записи и вызвать методы для чтения или записи объектов. Но есть и отличия: приложения для магазина Windows открывают файлы асинхронными методами. Они построены на интерфейсах `IStorageFile` и `IStorageFolder`. Изучите интерфейсы и их членов средствами IDE.

Введите в любой строке любого метода **`Windows.Storage.IStorageFolder`**, щелкните правой кнопкой мыши на `IStorageFolder` и выберите Go To Definition (F12) для просмотра определения:



Каждый объект `IStorageFolder` представляет папку в файловой системе, с методами для работы с входящими в эту папку файлами:

- `CreateFileAsync()` асинхронный метод создания файла в папке.
- `CreateFolderAsync()` асинхронный метод создания подпапки.
- `GetFileAsync()` получает файл в папке и возвращает объект `IStorageFile`.
- `GetFolderAsync()` получает подпапку и возвращает еще один объект `IStorageFolder`.
- `GetItemAsync()` получает файл или папку и возвращает объект `IStorageItem`.
- `GetFilesAsync()`, `GetFoldersAsync()` и `GetItemsAsync()` возвращают коллекции элементов, это коллекции типа **`IReadOnlyList`**, позволяющие получить элемент по индексу, но без методов добавления, сортировки или сравнения.

Приложения для магазина Windows защищают файловую систему

Вернитесь к первому примеру кода в главе 9. Мы предупреждали, что не стоит записывать ничего в папку C:\, надеемся, что вы выбрали более безопасное место. Дело в том, что программы для рабочего стола Windows могут повредить системные файлы. Именно поэтому каждое приложение для магазина Windows имеет собственную папку.

В пространстве имен `Windows.Storage` находятся два интерфейса, помогающие в управлении элементами файловой системы. Интерфейс `IStorageFile` и объекты, которые его реализуют, перемещают, копируют и открывают файлы. Посмотрев на объявление `IStorageFolder`, вы увидите, что он расширяет интерфейс `IStorageItem`. Этот же интерфейс расширяет `IStorageFile`, что имеет смысл, если вспомнить про операции, применимые как к файлам, так и к папкам: удаление, переименование и получение имени, даты создания, пути и атрибутов.

Любое приложение для магазина Windows имеет локальную папку, доступ к которой осуществляется через `IStorageFolder` с именем `ApplicationData.Current.LocalFolder`. Затем можно **воспользоваться объектом `IStorageFile`, чтобы открыть файлы на чтение и запись,** вызвав их метод `OpenAsync()` (возвращающий `IRandomAccessStream`).

Имея контракт данных и поток, остается получить `DataContractSerializer`, чтобы читать и записывать объекты в файлы XML:

IStorageItem
Attributes
DateCreated
Name
Path

DeleteAsync()
GetBasicProperties-Async()
IsOfType()
RenameAsync()

IStorageFolder

CreateFileAsync()
CreateFolderAsync()
GetFileAsync()
GetFolderAsync()
GetItemAsync()
GetFilesAsync()
GetFoldersAsync()
GetItemsAsync()

IStorageFile
ContentType
FileType

CopyAndReplaceAsync()
CopyAsync()
MoveAndReplaceAsync()
MoveAsync()
OpenAsync()
OpenTransactedWrite-Async()

```
using Windows.Storage;
using Windows.Storage.Streams;
using System.Runtime.Serialization;
```

Вам потребуются эти операторы using.

```
Guy joe = new Guy("Joe", 37, 164.38M);
```

Класс Guy с контрактом данных с предыдущей страницы.

```
DataContractSerializer serializer =
       new DataContractSerializer(typeof(Guy));

IStorageFolder localFolder =
       ApplicationData.Current.LocalFolder;
```

Сериализатор контракта данных должен знать тип сериализуемого содержимого. И вы указваете, что нужно сериализовать объекты Guy и их графы.

```
IStorageFile guyFile = await localFolder.CreateFileAsync("Joe.xml",
                                CreationCollisionOption.ReplaceExisting);

using (IRandomAccessStream stream =
          await guyFile.OpenAsync(FileAccessMode.ReadWrite))
using (Stream outputStream = stream.AsStreamForWrite()) {

          serializer.WriteObject(outputStream, joe);

}
Guy copyOfJoe;

using (IRandomAccessStream stream =
       await guyFile.OpenAsync(FileAccessMode.ReadWrite))
using (Stream inputStream = stream.AsStreamForRead()) {

    copyOfJoe = serializer.ReadObject(inputStream) as Guy;

}
```

Можно передать в метод CreateFileAsync() имя файла и параметр для того, чтобы заменить, открыть, отменить операцию или сгенерировать уникальное имя, если файл существует.

Теперь есть входящий и исходящий потоки и можно сериализовать объекты.

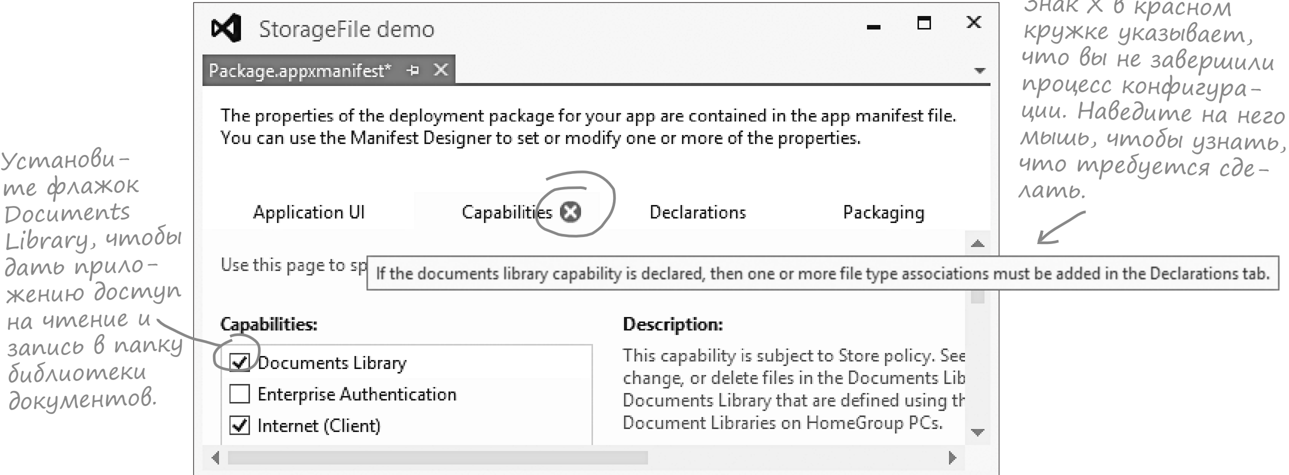
KnownFolders helps you access high-profile folders

Свойства класса `KnownFolders` из пространства имен `Windows.Storage` дают доступ к библиотекам документов, музыки и стандартным папкам учетной записи Windows. `KnownFolders.DocumentsLibrary` — это объект `StorageFolder` (реализующий `IStorageFolder`), предоставляющий доступ к библиотеке документов пользователя. Он обладает свойствами для доступа к библиотекам музыки, изображений и видео, съемным устройствам, серверам мультимедиа и домашним группам.

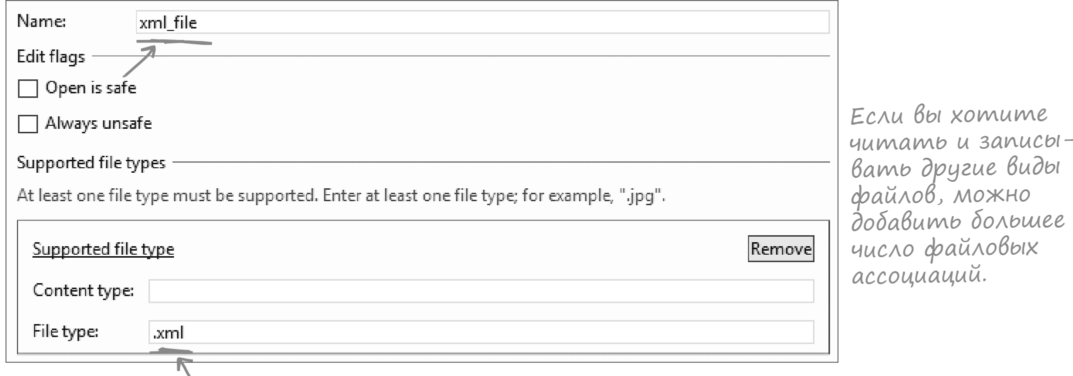
KnownFolders
DocumentsLibrary HomeGroup MediaServerDevices MusicLibrary PicturesLibrary RemovableDevices VideoLibrary

Но если вы хотите, чтобы приложение использовало для чтения и записи другую папку, нужно дать ему разрешение, **внеся поправки в манифест пакета**. Данная возможность будет видна всем, кто установит ваше приложение из магазина Windows.

Для добавления доступа к библиотеке документов **дважды щелкните на строке** *Package.appxmanifest* в окне Solution Explorer и в списке Capabilities установите флажок Documents Library.



Перейдите на вкладку Declarations, выберите File Type Associations и щелкните на Add. Откроется форма с полями, помеченными X в красном кружке. Присвойте **Name** значение `xml_file`, а **File** — значение *.xml*.



Сохраните и закройте манифест. Теперь приложение может читать и записывать файлы *.xml* в папку с библиотекой документов пользователя.

Граф объекта целиком сериализуется в XML

Когда сериализатор записывает контракт данных объекта, процесс проходит через весь граф. Каждый экземпляр класса с контрактом данных записывается в файл XML. Вид XML-файла можно настроить, выбрав пространство имен и указав имена членов в скобках рядом с `DataContract` и `DataMember`.

```
[DataContract(Namespace = "http://www.headfirstlabs.com/Chapter11")]
class Guy {
    public Guy(string name, int age, decimal cash){
        Name = name;
        Age = age;
        Cash = cash;
        TrumpCard = Card.RandomCard();
    }

    [DataMember]
    public string Name { get; private set; }

    [DataMember]
    public int Age { get; private set; }

    [DataMember]
    public decimal Cash { get; private set; }

    [DataMember(Name = "MyCard")]
    public Card TrumpCard { get; set; }

    public override string ToString() {
        return String.Format("My name is {0}, I'm {1}, I have {2} bucks, "
            + "and my trump card is {3}", Name, Age, Cash, TrumpCard);
    }
}
```

Это код XML для сериализованного Guy:

```
<Guy
   xmlns="http://www.headfirstlabs.com/Chapter11"
   xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
   <Age>37</Age>
   <MyCard>
      <Suit>Hearts</Suit>
      <Value>Three</Value>
   </MyCard>
   <Cash>176.22</Cash>
   <Name>Joe</Name>
</Guy>
```

Класс Guy содержит ссылку на объект Card с контрактом данных, поэтому он включается в XML в виде тега <Card>.

```
[DataContract(Namespace = "http://www.headfirstlabs.com/Chapter11")]
class Card {
    [DataMember]
    public Suits Suit { get; set; }

    [DataMember]
    public Values Value { get; set; }

    public Card(Suits suit, Values value) {
        this.Suit = suit;
        this.Value = value;
    }

    private static Random r = new Random();

    public static Card RandomCard() {
        return new Card((Suits)r.Next(4), (Values)r.Next(1, 14));
    }

    public string Name {
        get { return Value.ToString() + " of " + Suit.ToString(); }
    }

    public override string ToString() { return Name; }
}
```

Оба контракта находятся в одном пространстве имен, что превращает их в свойство xmlns тега <Guy> в сериализованном файле XML.

Имена членов контракта данных **не должны** совпадать с именами свойств. В классе `Guy` есть свойство `TrumpCard`, но параметр `Name` атрибута `DataMember` дает ему имя `MyCard`. Оно и будет показано в сериализованном XML.

Обратили внимание, что в сериализованном XML *нет типа* `Card`? Вы можете добавить атрибуты контракта данных к любому классу с совместимыми членами, например свойствами `Suit` и `Value` класса `Card`, которые сериализатор может задать, сопоставив их со значениями перечисления `Hearts` и `Three`.

Вот проект для экспериментов с сериализацией контрактов данных. **Создайте новое приложение для магазина Windows** и замените *MainPage.xaml* шаблоном Basic Bage. Затем **откройте** *Package.appxmanifest*, предоставьте доступ к библиотеке документов и добавьте тип файлов *.xml*. **Добавьте оба класса** с контрактами данных с предыдущей страницы (вам понадобится строка `using System.Runtime.Serialization`). Не забудьте добавить перечисления `Suits` и `Values` (для класса `Card`). Вот как выглядит страница, которую мы будем строить:

Guy Serializer

My name is Joe, I'm 37, I have 176.22 bucks, and my trump card is Seven of Clubs

My name is Bob, I'm 45, I have 4.68 bucks, and my trump card is Three of Clubs

My name is Ed, I'm 43, I have 37.51 bucks, and my trump card is Ten of Hearts

Write Joe | Write Bob | Write Ed

Last filename written

C:\Users\Public\Documents\Guys\Joe.xml

Read a new Guy

New Guy:

Date created

1/15/2013 5:24:12 PM -05:00

Content type

text/xml

1 Добавьте на страницу статический ресурс `GuyManager` (и задайте имя приложения). Класс `GuyManager` мы добавим на следующей странице.

```
<Page.Resources>
    <local:GuyManager x:Name="guyManager"/>
    <x:String x:Key="AppName">Guy Serializer</x:String>
</Page.Resources>
```

Чтобы избавиться от ошибки IDE для этого тега, можно добавить пустой класс GuyManager. Вы заполните его на следующей странице. Не забудьте после этого перестроить приложение, иначе в конструкторе появятся сообщения об ошибке.

2 Вот код XAML для страницы.

```
<Grid Grid.Row="1" DataContext="{StaticResource guyManager}" Margin="120,0">
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

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

    <StackPanel>
        <TextBlock Text="{Binding Joe}" Style="{StaticResource ItemTextStyle}"
            Margin="0,0,0,20"/>
        <Button x:Name="WriteJoe" Content="Write Joe" Click="WriteJoe_Click"/>
    </StackPanel>

    <StackPanel Grid.Column="1">
        <TextBlock Text="{Binding Bob}" Style="{StaticResource ItemTextStyle}"
            Margin="0,0,0,20"/>
        <Button x:Name="WriteBob" Content="Write Bob" Click="WriteBob_Click"/>
    </StackPanel>

    <StackPanel Grid.Column="2">
        <TextBlock Text="{Binding Ed}" Style="{StaticResource ItemTextStyle}"
            Margin="0,0,0,20"/>
        <Button x:Name="WriteEd" Content="Write Ed" Click="WriteEd_Click"/>
    </StackPanel>

    <StackPanel Grid.Row="1" Grid.ColumnSpan="2" Margin="0,0,20,0">
        <TextBlock>Last filename written</TextBlock>
        <TextBox Text="{Binding Path, Mode=TwoWay}" Margin="0,0,0,20"/>
        <TextBlock>Date created</TextBlock>
        <TextBlock Text="{Binding LatestGuyFile.DateCreated}" Margin="0,0,0,20"
            Style="{StaticResource SubheaderTextStyle}"/>
        <TextBlock>Content type</TextBlock>
        <TextBlock Text="{Binding LatestGuyFile.ContentType}"
                   Style="{StaticResource SubheaderTextStyle}"/>
    </StackPanel>

    <StackPanel Grid.Row="1" Grid.Column="2">
        <Button x:Name="ReadNewGuy" Content="Read a new Guy" Click="ReadNewGuy_Click"
        Margin="0,10,0,0"/>
        <TextBlock Style="{StaticResource ItemTextStyle}" Margin="0,0,0,20">
    <Run>New Guy: </Run>
    <Run Text="{Binding NewGuy}"/>
        </TextBlock>
    </StackPanel>
</Grid>
```

Страница разделена на три столбца и две строки.

Контекстом данных для сетки является статический ресурс GuyManager.

В каждый столбец верхнего ряда вставлен элемент StackPanel, содержащий элемент TextBlock и кнопку Button.

Этот элемент TextBlock связан со свойством Ed класса GuyManager.

Первая ячейка нижнего ряда объединяет два столбца. В нее помещен ряд связанных со свойствами элементов. Почему для пути используется TextBox?

Объект LatestGuyFile является интерфейсом IStorageFile, и элементы управления TextBlock связаны с его свойствами.

Это еще не все — переверните страницу!

Эти операторы понадобятся для класса GuyManager.

```
using System.ComponentModel;

using Windows.Storage;

using Windows.Storage.Streams;

using System.IO;

using System.Runtime.Serialization;
```

3 Добавим класс GuyManager.

```
class GuyManager : INotifyPropertyChanged
{
    private IStorageFile latestGuyFile;
    public IStorageFile LatestGuyFile { get { return latestGuyFile; } }

    private Guy joe = new Guy("Joe", 37, 176.22M);
    public Guy Joe
    {
        get { return joe; }
    }

    private Guy bob = new Guy("Bob", 45, 4.68M);
    public Guy Bob
    {
        get { return bob; }
    }

    private Guy ed = new Guy("Ed", 43, 37.51M);
    public Guy Ed
    {
        get { return ed; }
    }

    public Guy NewGuy { get; private set; }

    public string Path { get; set; }


    public async void ReadGuyAsync()
    {
        if (String.IsNullOrEmpty(Path))
            return;
        latestGuyFile = await StorageFile.GetFileFromPathAsync(Path);

        using (IRandomAccessStream stream =
                  await latestGuyFile.OpenAsync(FileAccessMode.Read))
        using (Stream inputStream = stream.AsStreamForRead())
        {
            DataContractSerializer serializer = new DataContractSerializer(typeof(Guy));
            NewGuy = serializer.ReadObject(inputStream) as Guy;
        }
        OnPropertyChanged("NewGuy");
        OnPropertyChanged("LatestGuyFile");
    }
```

Вспомогательное поле этого свойства задается методом ReadGuyAsync(), а элементы TextBlock связаны с его свойствами DateCreated и ContentType.

Это три предназначенных только для чтения свойства Guy с закрытыми вспомогательными полями. В коде XAML присутствует TextBlock, связанный с каждым из них.

Четвертый TextBlock связан с этим свойством Guy, которое задается методом ReadGuyAsync().

Этот статический метод StorageFile.GetFileFromPathAsync() позволяет создать из пути IStorageFile.

Метод ReadGuyAsync() использует путь в элементе TextBox для задания поля latestGuyFile объекта IStorgeFile. Он читает объекты из файла XML через сериализатор, затем вызывает события PropertyChanged для свойств, использующих атрибуты IStorageFile.

```
    public async void WriteGuyAsync(Guy guyToWrite)
    {
        IStorageFolder guysFolder =
            await KnownFolders.DocumentsLibrary.CreateFolderAsync("Guys",
                                                 CreationCollisionOption.OpenIfExists);
        latestGuyFile =
                await guysFolder.CreateFileAsync(guyToWrite.Name + ".xml",
                                          CreationCollisionOption.ReplaceExisting);

        using (IRandomAccessStream stream =
                            await latestGuyFile.OpenAsync(FileAccessMode.ReadWrite))
        using (Stream outputStream = stream.AsStreamForWrite())
        {
            DataContractSerializer serializer = new DataContractSerializer(typeof(Guy));
            serializer.WriteObject(outputStream, guyToWrite);
        }

        Path = latestGuyFile.Path;

        OnPropertyChanged("Path");
        OnPropertyChanged("LatestGuyFile");
    }

    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
        if (propertyChangedEvent != null)
        {
            propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Мы создаем в библиотеке документов папку Guys для хранения XML. Если она существует, то просто открываем ее.

Этот код создает файл XML, открывает поток и записывает в файл граф объекта Guy.

Метод WriteGuyAsync() записывает объект в файл XML в папку Guys внутри библиотеки документов. Он присваивает полю latestGuyFile объекта IStorageFile ссылку на готовый файл и вызывает события, связанные с изменением использующих данное поле свойств.

Этот код вы уже использовали для реализации INotifyPropertyChanged и вызова событий PropertyChanged.

4 Это методы обработки событий для файла *MainPage.xaml.cs*:

```
private void WriteJoe_Click(object sender, RoutedEventArgs e) {
    guyManager.WriteGuyAsync(guyManager.Joe);
}
private void WriteBob_Click(object sender, RoutedEventArgs e) {
    guyManager.WriteGuyAsync(guyManager.Bob);
}
private void WriteEd_Click(object sender, RoutedEventArgs e) {
    guyManager.WriteGuyAsync(guyManager.Ed);
}
private void ReadNewGuy_Click(object sender, RoutedEventArgs e) {
    guyManager.ReadGuyAsync();
}
```

Тестирование нашего сериализатора

Воспользуемся нашим сериализатором объекта Guy для экспериментов:

★ Запишите каждый объект `Guy` в папку Document Library. Щелкните на кнопке ReadGuy для чтения только что записанного объекта. Попробуйте присвоить элементу TextBox путь к другому объекту. Что произойдет, если попытаться прочитать файл?

★ В построенном приложении Simple Text Editor файлы XML служат для открытия и сохранения средств выбора файлов, и мы можем с их помощью отредактировать файлы Guy. Откройте один из файлов Guy, внесите изменения, сохраните их и попробуйте его прочитать. Что произойдет при добавлении некорректного XML? Что случится, если новая масть или достоинство карты перестанут совпадать с корректным значением перечисления?

★ В Simple Text Editor нет сбрасывающей результаты кнопки New. Сможете ее добавить? (Вы можете его просто перезагрузить.) Скопируйте файл Guy и вставьте его в новый файл XML в папке *Guys*. Что произойдет при попытке прочитать его сериализатором?

★ Добавьте или удалите имена `DataMember` (`[DataMember(Name="...")]`). Как это повлияет на XML? Что произойдет при попытке загрузить после обновления контракта ранее сохраненный файл XML? Сможете ли вы исправить файл XML?

★ Измените пространство имен контракта данных `Card`. Что произойдет с XML?

Часто задаваемые вопросы

В: Я не настраивал возможности моей программы в Simple Text Editor. Почему ей доступна библиотека документов?

О: Если в приложении используется File Picker, пользователь получает доступ к файлам и папкам **без** редактирования манифеста пакета, так как функция File Picker *по умолчанию поддерживает безопасность файловой системы*: вас не пустят в папки установки, локальные папки, временные папки и прочие места системы, которым ваше приложение может нечаянно навредить. Редактировать возможности следует при написании кода, обращающегося к папкам напрямую.

В: Почему иногда при редактировании XAML или кода конструктор просит перестроить приложение?

О: Как вы уже знаете, когда в XAML используются статические ресурсы, в класс `Page` добавляются ссылки на объекты. И для отображения в конструкторе сначала следует создать экземпляры этих объектов. При редактировании класса, используемого для статических ресурсов, конструктор не будет обновляться, пока вы не перестроите этот класс. Так как IDE перестраивает проект только по запросу, до этого момента в памяти отсутствует скомпилированный код для создания экземпляров статических ресурсов.

Например, откройте сериализатор Guy и отредактируйте метод Guy.ToString(), добавив к возвращаемому значению несколько слов. Вернитесь к главной странице конструктора. Там будет демонстрироваться старая версия. Выберите в меню Build команду Rebuild. Конструктор обновится сразу же после завершения команды. Добавьте еще один TextBlock, связанный с объектом Guy. И снова IDE будет использовать старую версию объекта, пока вы не перестроите приложение.

В: Чем отличаются пространства имен в программе и в файле XML?

О: C#, файлы XML, файловая система Windows и веб-страницы пользуются разными (часто связанными) системами присвоения уникальных имен классам, XML-документам, файлам и веб-страницам. В главе 9 вы создали класс `KnownFolders`, для слежения за папками с оправданиями. Но в .NET Framework класс есть с таким именем. Он находится в пространстве имен Windows.Storage, поэтому вы можете без проблем пользоваться классом с таким же именем. Это называется **снятием многозадачности**.

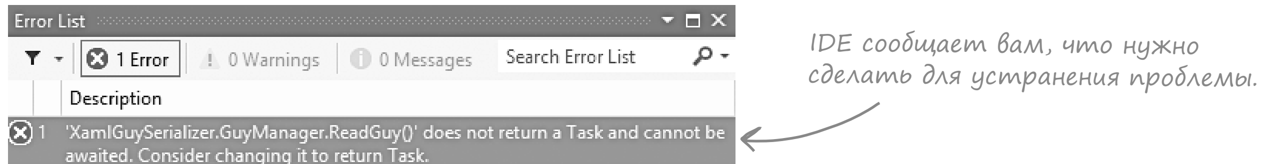
Разрешать многозадачность требуется и для контрактов данных. В книге вы видели несколько версий класса Guy. Представьте, что нам требуются два контракта для сериализации двух версий класса Guy. Их можно поместить в разные пространства имен, и проблема будет решена. И эти пространства имен будут отделены от пространств имен ваших классов, чтобы не создавать путаницы между классами и контрактами.

Класс Task

Помеченный модификатором `async` метод может использоваться другими асинхронными методами. Но для этого нужно внести в него одно изменение. Попробуйте добавить этот метод в файл *GuyManager.cs*:

```
private async void MethodThatReadsGuys()
{
    await ReadGuy();
}
```

Вы получите подчеркнутую волнистой линией ошибку и сообщение в окне Error List:

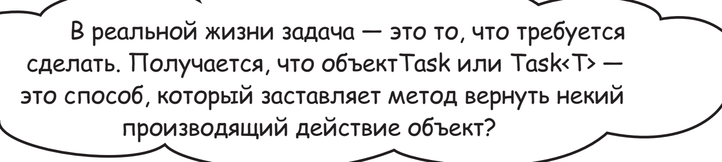


Один асинхронный метод может вызывать другой, если тип возвращаемого значения этого второго метода принадлежит к классу `Task` (или, если метод должен вернуть значение, к подклассу `Task<T>`). Так как `ReadGuy()` не возвращает значений, просто **заменете в объявлении void на Task**:

```
public async Task ReadGuyAsync()
{
    // Как и на предыдущей странице
}
```

В соответствии с соглашением об именовании к именам асинхронных методов, которые вызываются с оператором `await`, добавляется `Async`. Поэтому мы поменяем имя метода с `ReadGuy()` на `ReadGuyAsync()`.

Теперь при вызове метода с оператором `await` он будет функционировать как любой другой асинхронный метод и вернет управление, дойдя до асинхронной операции. Если бы нам требовался метод, возвращающий значение, это был бы метод типа `Task<T>`. К примеру, чтобы заставить метод `ReadGuyAsync()` вернуть прочитанный объект `Guy`, нужно указать тип `Task<Guy>`.



Да! Класс **Task** представляет асинхронную операцию.

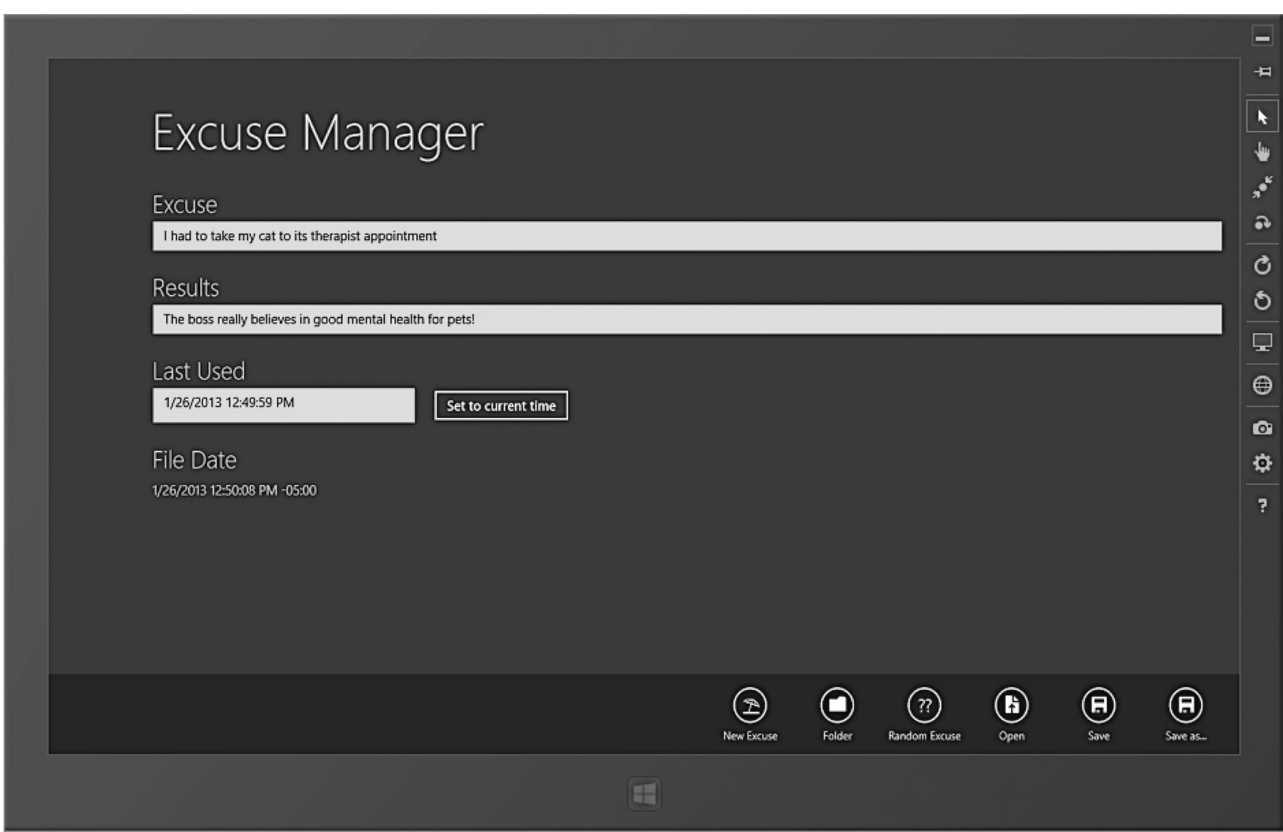
Модификатор `async`, ключевое слово `await` и класс `Task` упрощают асинхронный код путем инкапсуляции работы возвращаемого элемента управления в классе `Task`. Посмотрите на свойства и методы этого класса при помощи функции «Go to Definition». Он обладает методами `Run()`, `Continue()` и `Wait()`, а также свойствами `IsCompleted` и `IsFaulted`. Это должно подсказать вам, что делается автоматически, упрощая написание асинхронных методов.

Более подробно об асинхронном программировании можно почитать здесь: http://msdn.microsoft.com/ru-ru/library/vstudio/hh191443.aspx

Новый генератор оправданий для Брайана

Вы уже умеете создавать страницы XAML, читать и записывать файлы и сериализовать объекты. А значит, сможете превратить Генератор оправданий Брайана в приложение для магазина Windows.

Вот его главная страница:



Запуск приложений для магазина Windows в симуляторе

Снимок экрана на этой странице был сделан в симуляторе. Это встроенное в Visual Studio приложение для рабочего стола, которое позволяет запускать программы на полноэкранном имитаторе устройства. Оно крайне полезно, если вы хотите проверить, как программа отвечает на прикосновения и аппаратные события. (Это не эмулятор, а симулятор.)

Для запуска симулятора при старте программы щелкните на стрелке рядом с ▶ Local Machine ▾ и выберите ▶ Simulator.

Подробнее об управлении симулятором можно почитать на странице: http://msdn.microsoft.com/ru-ru/library/windows/apps/hh441475.aspx

Разделим страницу, оправдания и Excuse Manager

Старый объект Excuse умел читать и записывать самого себя, что было не так уж и плохо. Но существуют и другие варианты проектирования. В приложении Guy Serializer данные об объекте Guy были в одном классе, а методы чтения и записи объектов Guy — в классе GuyManager. Этот же шаблон мы используем для нового приложения Excuse Manager.

Это еще один пример **разделения ответственности**, о котором мы говорили в главах 5 и 6. Класс Guy должен всего лишь демонстрировать контракт данных; определять, что с этим делать, будет другой класс — GuyManager. И ни в одном из этих классов не будет кода обновления пользовательского интерфейса, ведь он не связан с отображением извинений, — это работа объекта MainPage.

ExcuseManager
CurrentExcuse FileDate
NewExcuseAsync() SetToCurrentTime() ChooseNewFolderAsync() OpenExcuseAsync() OpenRandomExcuseAsync() SaveCurrentExcuseAsync() UpdateFileDateAsync() SaveCurrentExcuseAsAsync() WriteExcuseAsync() ReadExcuseAsync()

Элементы управления страницы связывают объект ExcuseManager с объектами чтения и записи Excuse.

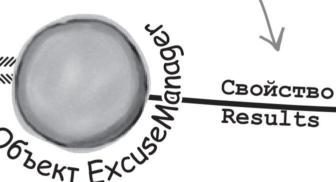
Excuse
Description Results LastUsed DateWarning

Объект MainPage

На главной странице отсутствует код выбора папки, чтения и записи файлов. Весь этот код инкапсулирован в класс ExcuseManager.

Кнопка Folder

Обработчик события Click вызывает ChooseNewFolder()



Элемент управления отображает данные благодаря связыванию. Текстовые поля были привязаны непосредственно к свойствам текущего объекта Excuse, отображаемого объектом ExcuseManager.

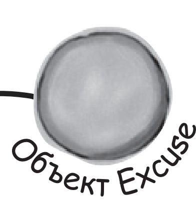
Results TextBox

Двустороннее связывание

Объект ExcuseManager

Свойство Results

Объект Excuse



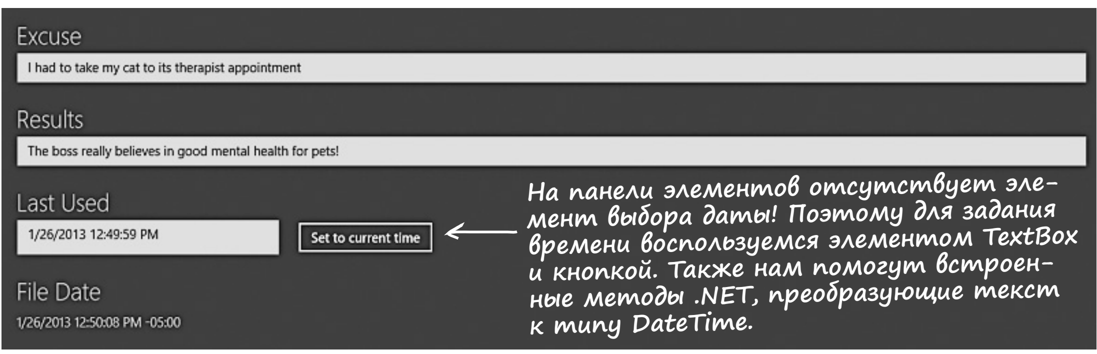
В классах Excuse и ExcuseManager отсутствует код обновления пользовательского интерфейса. А вы *умеете пользоваться сериализацией контрактов данных или асинхронным программированием в программах WinForms*? Можете ли вы с их помощью заставить Windows Forms-версию программы Брайана читать и записывать те же самые файлы оправданий, что и новый Excuse Manager для магазина Windows?

Главная страница генератора извинений

Создайте проект для магазина Windows и **замените** *MainPage.xaml* **шаблоном Basic Page**. Вам потребуется статический ресурс `ExcuseManager`. Добавьте пустой класс `ExcuseManager`, чтобы код начал компилироваться, а затем добавьте его как статический ресурс в `<Page.Resources>`:

```
<Page.Resources>
    <local:ExcuseManager x:Name="excuseManager"/>
    <x:String x:Key="AppName">Excuse Manager</x:String>
</Page.Resources>
```

Код XAML этой страницы — простая компоновка на основе элемента StackPanel. В качестве контекста данных этого элемента укажите ресурс `ExcuseManager`.




```
<StackPanel Grid.Row="1" Margin="120,0,0,0"
            DataContext="{StaticResource ResourceKey=excuseManager}">
    <TextBlock Style="{StaticResource SubheaderTextStyle}" Text="Excuse" Margin="0,0,0,10"/>
    <TextBox Text="{Binding CurrentExcuse.Description, Mode=TwoWay}" Margin="0,0,20,20"/>
    <TextBlock Style="{StaticResource SubheaderTextStyle}" Text="Results" Margin="0,0,0,10"/>
    <TextBox Text="{Binding CurrentExcuse.Results, Mode=TwoWay}" Margin="0,0,20,20"/>
    <TextBlock Style="{StaticResource SubheaderTextStyle}" Text="Last Used" Margin="0,0,0,10"/>
    <StackPanel Orientation="Horizontal"  Margin="0,0,0,20">
        <TextBox Text="{Binding CurrentExcuse.LastUsed, Mode=TwoWay}"
                 MinWidth="300" Margin="0,0,20,0"/>
        <Button Content="Set to current time" Click="SetToCurrentTimeClick" Margin="0,0,20,0"/>
        <TextBlock Foreground="Red" Text="{Binding CurrentExcuse.DateWarning}"
                 Style="{StaticResource SubtitleTextStyle}"/>
    </StackPanel>

    <TextBlock Style="{StaticResource SubheaderTextStyle}" Text="File Date" Margin="0,0,0,10"/>
    <TextBlock Text="{Binding FileDate}" Style="{StaticResource ItemTextStyle}"/>
</StackPanel>
```

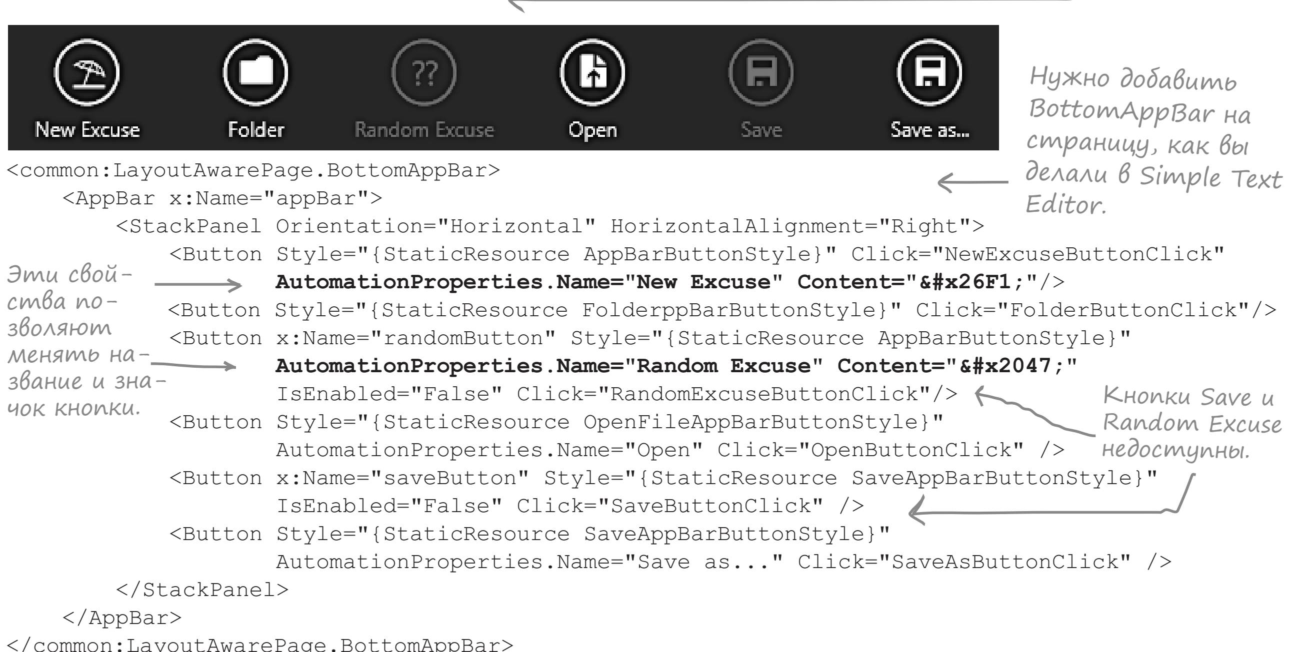
При вводе некорректной даты в DateWarning появляется предупреждение, отображемое элементом TextBlock.

Элементы управления TextBox двусторонне связаны со свойствами объекта CurrentExcuse в классе ExcuseManager. Элемент TextBlock с датой файла связан со свойством FileDate класса ExcuseManager.

Добавим панель приложения

В файле StandardStyles.xaml из поставки Visual Studio 2012 есть опечатка. В слове FolderAppBarButtonStyle отсутствует буква "А".

Добавим нижнюю панель приложения. Раскомментируйте `OpenFileAppBarButtonStyle`, `SaveAppBarButtonStyle` и `FolderppBarButtonStyle` для кнопок Open, Save и Folder.



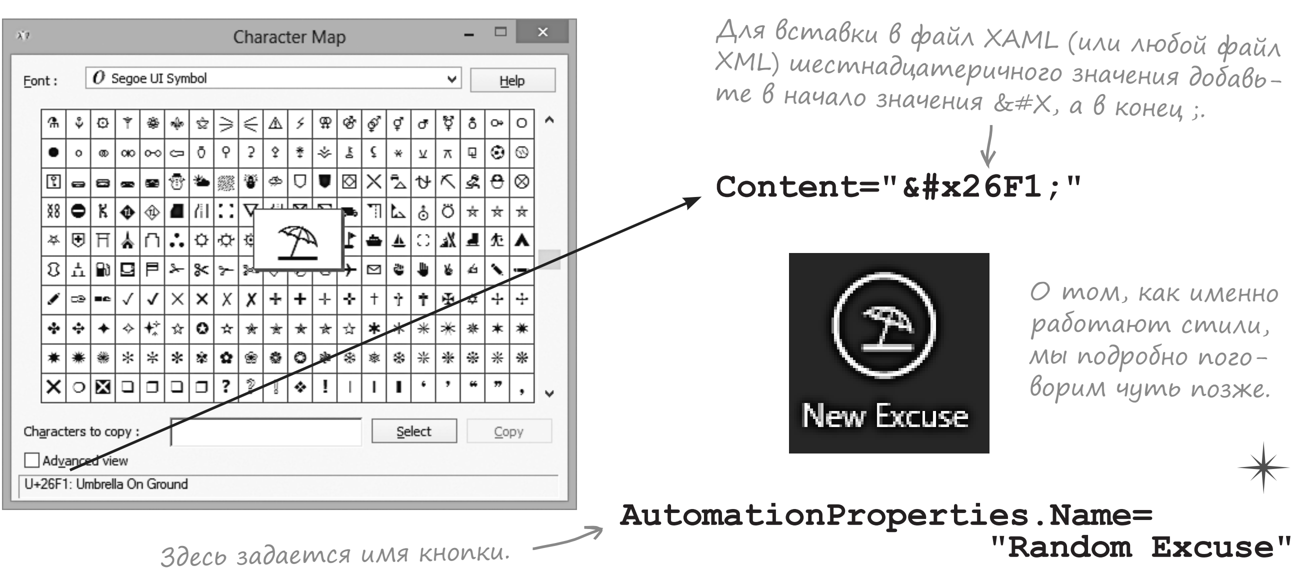
Нужно добавить BottomAppBar на страницу, как вы делали в Simple Text Editor.

```
<common:LayoutAwarePage.BottomAppBar>
    <AppBar x:Name="appBar">
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Right">
            <Button Style="{StaticResource AppBarButtonStyle}" Click="NewExcuseButtonClick"
                    AutomationProperties.Name="New Excuse" Content="&#x26F1;"/>
            <Button Style="{StaticResource FolderppBarButtonStyle}" Click="FolderButtonClick"/>
            <Button x:Name="randomButton" Style="{StaticResource AppBarButtonStyle}"
                    AutomationProperties.Name="Random Excuse" Content="&#x2047;"
                    IsEnabled="False" Click="RandomExcuseButtonClick"/>
            <Button Style="{StaticResource OpenFileAppBarButtonStyle}"
                    AutomationProperties.Name="Open" Click="OpenButtonClick" />
            <Button x:Name="saveButton" Style="{StaticResource SaveAppBarButtonStyle}"
                    IsEnabled="False" Click="SaveButtonClick" />
            <Button Style="{StaticResource SaveAppBarButtonStyle}"
                    AutomationProperties.Name="Save as..." Click="SaveAsButtonClick" />
        </StackPanel>
    </AppBar>
</common:LayoutAwarePage.BottomAppBar>
```

Эти свойства позволяют менять название и значок кнопки.

Кнопки Save и Random Excuse недоступны.

Как код XAML повлиял на вид кнопки? Посмотрите на любой раскомментированный стиль из файла *StandardStyles.xaml*, и вы увидите шестнадцатеричные значения  для папки и  для кнопки Save. Текст использует шрифт Segoe UI Symbol, а значок — символ Unicode bp этого шрифта.



Для вставки в файл XAML (или любой файл XML) шестнадцатеричного значения добавьте в начало значения &#X, а в конец ;.

```
Content="&#x26F1;"
```

О том, как именно работают стили, мы подробно поговорим чуть позже.

```
AutomationProperties.Name=
                    "Random Excuse"
```

Здесь задается имя кнопки.

Теперь перейдем к классу ExcuseManager →

Создание класса ExcuseManager

Не забудьте, что класс ExcuseManager должен реализовывать INotifyPropertyChanged.

ExcuseManager
CurrentExcuse FileDate
NewExcuseAsync() SetToCurrentTime() ChooseNewFolderAsync() OpenExcuseAsync() OpenRandomExcuseAsync() SaveCurrentExcuseAsync() UpdateFileDateAsync() SaveCurrentExcuseAsAsync() WriteExcuseAsync() ReadExcuseAsync()

Вот *большая часть* кода класса `ExcuseManager`, остальную часть, как и создание класса `Excuse`, вы должны выполнить самостоятельно. Для связывания у нас есть два открытых свойства: `CurrentExcuse` — загруженный в настоящее время объект `Excuse` — и строка `FileDate`, показывающая или дату, или строку «`(файл не загружен)`» (если текущее оправдание не сохранено или не загружено).

Метод `ChooseNewFolderAsync()` отображает средство выбора папки и возвращает значение true после выбора папки пользователем. Это асинхронный метод, возвращающий булево значение `Task<bool>`.

Используйте эти операторы `using`:

```
using System.ComponentModel;
using System.IO;
using System.Runtime.Serialization;
using Windows.Storage;
using Windows.Storage.Streams;
using Windows.Storage.FileProperties;
using Windows.Storage.Pickers;
using Windows.UI.Popups;
```

Task находится в пространстве имен System.Threading.Tasks, но IDE добавила нужный оператор using.

```
public Excuse CurrentExcuse { get; set; }

public string FileDate { get; private set; }

private Random random = new Random();

private IStorageFolder excuseFolder = null;

private IStorageFile excuseFile;

public ExcuseManager() {
    NewExcuseAsync();
}

async public void NewExcuseAsync() {
    CurrentExcuse = new Excuse();
    excuseFile = null;
    OnPropertyChanged("CurrentExcuse");
    await UpdateFileDateAsync();
}

public void SetToCurrentTime() {
    CurrentExcuse.LastUsed = DateTimeOffset.Now.ToString();
    OnPropertyChanged("CurrentExcuse");
}

public async Task<bool> ChooseNewFolderAsync() {
    FolderPicker folderPicker = new FolderPicker() {
        SuggestedStartLocation = PickerLocationId.DocumentsLibrary
    };
    folderPicker.FileTypeFilter.Add(".xml");
    IStorageFolder folder = await folderPicker.PickSingleFolderAsync();
    if (folder != null) {
        excuseFolder = folder;
        return true;
    }
    MessageDialog warningDialog = new MessageDialog("Папка не выбрана");
    await warningDialog.ShowAsync();
    return false;
}
```

excuseFile IStorageFile следит за текущим файлом оправданий. Если файл не был загружен или сохранен, оно получает значение null.

При щелчке на кнопке New Excuse ExcuseManager сбрасывает текущее оправдание и вызывает UpdateFileDateAsync() для обновления FileDate.

Асинхронный метод `NewExcuseAsync()` можно вызвать из обычного метода. Уберите ключевое слово `await`, и метод будет заблокирован. IDE отобразит окно с вопросом, действительно ли вы хотите это сделать.

Этот метод присваивает строке LastUsed текущее время и вызывает PropertyChanged.

Этот асинхронный метод возвращает булево значение, поэтому тип возвращаемого им значения Task<bool>.

При выборе папки метод возвращает значение true. Асинхронный метод, возвращающий значение Task<bool>, вернет обычное булево значение.

Объект `FolderPicker` является средством выбора файла. Он функционирует так же, как и остальные средства выбора из пространства имен `Windows.Storage.Pickers`: http://msdn.microsoft.com/library/windows/apps/BR207928

```
public async void OpenExcuseAsync() {
    FileOpenPicker picker = new FileOpenPicker {
        SuggestedStartLocation = PickerLocationId.DocumentsLibrary,
        CommitButtonText = "Open Excuse File"
    };
    picker.FileTypeFilter.Add(".xml");
    excuseFile = await picker.PickSingleFileAsync();
    if (excuseFile != null)
        await ReadExcuseAsync();
}
```

Метод OpenExcuseAsync() аналогичен методу ReadGuyAsync() в программе Guy Serializer.

Ой! Где-то ошибка. Нашли? Мы исправим ее в следующей главе.

```
public async void OpenRandomExcuseAsync() {
    IReadOnlyList<IStorageFile> files = await excuseFolder.GetFilesAsync();
    excuseFile = files[random.Next(0, files.Count())];
    await ReadExcuseAsync();
}
```

Метод UpdateFileDateAsync() присваивает свойству FileDate дату последнего изменения текущего файла с оправданием. Если файл не загружен, присваивается строка. Это асинхронный метод, вызываемый другим асинхронным методом, поэтому он возвращает значение типа Task.

```
public async Task UpdateFileDateAsync() {
    if (excuseFile != null) {
        BasicProperties basicProperties = await excuseFile.GetBasicPropertiesAsync();
        FileDate = basicProperties.DateModified.ToString();
    }
    else
        FileDate = "(файл не загружен)";
    OnPropertyChanged("FileDate");
}
```

Метод `IStorageFile.GetBasicPropertiesAsync()` возвращает объект `BasicProperties` с предназначенными только для чтения свойствами `DateModified` и `Size`, содержащими отредактированную дату и размер файла.

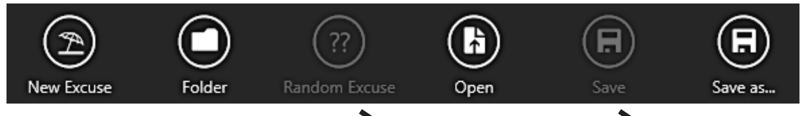
```
public async void SaveCurrentExcuseAsync() {
    if (CurrentExcuse == null) {
        await new MessageDialog("Нет загруженных оправданий").ShowAsync();
        return;
    }
    if (String.IsNullOrEmpty(CurrentExcuse.Description)) {
        await new MessageDialog("У текущего оправдания отсутствует описание").ShowAsync();
        return;
    }
    if (excuseFile == null)
        excuseFile = await excuseFolder.CreateFileAsync(CurrentExcuse.Description + ".xml",
                        CreationCollisionOption.ReplaceExisting);

    await WriteExcuseAsync();
}

public async Task ReadExcuseAsync() {
    // Этот метод напишете вы
}

public async Task WriteExcuseAsync() {
    // Этот метод напишете вы
}

public async void SaveCurrentExcuseAsAsync() {
    // Этот метод напишете вы
}
```

Метод SaveCurrentExcuseAsync() сначала проверяет, не равно ли текущее оправдание значению null и присутствует ли у него описание, и отображает предупреждение. При наличии корректного оправдания он вызывает метод WriteExcuseAsync() для записи этого оправдания. Если файл с оправданием пока отсутствует, вызывается метод CreateFileAsync() папки, чтобы его создать.

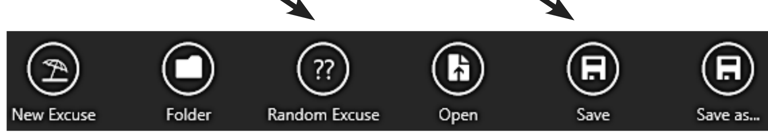
Завершим преобразование приложения → *дальше*

Добавление программного кода

Это весь нужный вам программный код. Обработчики событий кнопок просто вызывают методы класса `ExcuseManager`. Это преимущество отделения управления оправданиями от отображения пользовательского интерфейса. Код пользовательского интерфейса получается простым, так как большую часть работы делают другие классы.

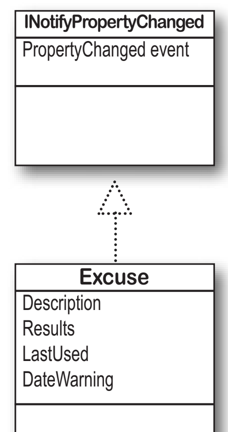


Кнопки Random Excuse и Save доступны только после выбора папки, поэтому метод обработки события кнопки Folder использует возвращаемое значение метода `ChooseNewFolderAsync()`. Если это значение `true`, появляется доступ к кнопкам Random Excuse и Save.




```
private void OpenButtonClick(object sender, RoutedEventArgs e) {
    excuseManager.OpenExcuseAsync();
}

private void SaveButtonClick(object sender, RoutedEventArgs e) {
    excuseManager.SaveCurrentExcuseAsync();
}

private void NewExcuseButtonClick(object sender, RoutedEventArgs e) {
    excuseManager.NewExcuseAsync();
}

private void SaveAsButtonClick(object sender, RoutedEventArgs e) {
    excuseManager.SaveCurrentExcuseAsAsync();
}

private void SetToCurrentTimeClick(object sender, RoutedEventArgs e) {
    excuseManager.SetToCurrentTime();
}

private void RandomExcuseButtonClick(object sender, RoutedEventArgs e) {
    excuseManager.OpenRandomExcuseAsync();
}

private async void FolderButtonClick(object sender, RoutedEventArgs e) {
    bool folderChosen = await excuseManager.ChooseNewFolderAsync();
    if (folderChosen) {
        saveButton.IsEnabled = true;
        randomButton.IsEnabled = true;
    }
}
```

Завершение классов `Excuse` и `ExcuseManager` для нового Генератора оправданий.



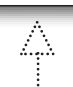
Построим класс `Excuse`.
Нам нужен контракт данных с пространством имен `http://www.headfirstlabs.com/ExcuseManager` и три члена данных. Первые два `Description` и `Results` являются автоматическими строковыми свойствами. Третьим будет поле `DateTime` с именем **`lastUsed`**, которое представляет собой вспомогательное поле для строкового свойства `LastUsed` (оно редактируется в методе `ExcuseManager.SetToCurrentTime()`).

В качестве значения по умолчанию для поля `lastUsed` класс `Excuse` использует особое значение `DateTime.MinValue`. Это самая ранняя дата, которую можно сохранить в переменной `DateTime`, и класс `Excuse` использует ее для оправданий без даты. Метод доступа `LastUsed` возвращает `lastUsed.ToString()` при заданной дате и `String.Empty`, если ей присвоено `MinValue`.

Метод записи `LastUsed` использует этот код для преобразования строки в дату:

```
DateTime d;
bool dateIsValid = DateTime.TryParse(value, out d);
lastUsed = d;
```

Метод `TryParse()` возвращает `true` для корректной даты и `false` в противном случае. При вводе некорректной даты метод присваивает предназначенному только для чтения свойству `DateWarning` значение «`Некорректная дата:`», за которым следует сама дата. В результате пользователь видит красный элемент TextBlock. Не забудьте вызвать событие `PropertyChanged`, чтобы сообщить странице об обновлении свойства `DateWarning`.



ExcuseManager
CurrentExcuse
FileDate
NewExcuseAsync()
SetToCurrentTime()
ChooseNewFolderAsync()
OpenExcuseAsync()
OpenRandomExcuseAsync()
SaveCurrentExcuseAsync()
UpdateFileDateAsync()
SaveCurrentExcuseAsAsync()
WriteExcuseAsync()
ReadExcuseAsync()

Реализуем метод `ExcuseManager.ReadExcuseAsync()`.
Этот метод открывает поток и сериализует текущее оправдание в файл, управляемый `IStorageFile`, который в настоящее время хранится в поле `excuseFile`. Затем метод отображает сообщение о корректной записи оправдания и вызывает `UpdateFileDateAsync()` для обновления свойства `FileDate`.

Реализуем метод `ExcuseManager.WriteExcuseAsync()`.
Этот метод открывает поток и десериализует новый объект `Excuse` из файла оправданий, управляемого `excuseFile`. Он вызывает событие `PropertyChanged`, информируя страницу об обновлении `CurrentExcuse`, а затем вызывает метод `UpdateFileDateAsync()`. Также вам нужно реализовать `INotifyPropertyChanged` и добавить метод `OnPropertyChanged()`.

Реализуем метод `ExcuseManager.SaveCurrentExcuseAsAsync()`.
Этот метод отображает объект `FileSavePicker`, позволяющий пользователю выбрать сохраняемый файл XML. Как только пользователь укажет файл, будет вызван метод `WriteExcuseAsync()` для его сохранения.

Вот методы, которые следует добавить в класс `ExcuseManager`.
Не забудьте, что класс должен расширять `INotifyPropertyChanged`.

```
public async Task ReadExcuseAsync() {
    using (IRandomAccessStream stream =
            await excuseFile.OpenAsync(FileAccessMode.Read))
    using (Stream inputStream = stream.AsStreamForRead())     {
        DataContractSerializer serializer = new DataContractSerializer(typeof(Excuse));
        CurrentExcuse = serializer.ReadObject(inputStream) as Excuse;
    }

    await new MessageDialog("Извинение из файла " + excuseFile.Name).ShowAsync();
    OnPropertyChanged("CurrentExcuse");
    await UpdateFileDateAsync();
}

public async Task WriteExcuseAsync() {
    using (IRandomAccessStream stream =
            await excuseFile.OpenAsync(FileAccessMode.ReadWrite))
    using (Stream outputStream = stream.AsStreamForWrite()) {
        DataContractSerializer serializer = new DataContractSerializer(typeof(Excuse));
        serializer.WriteObject(outputStream, CurrentExcuse);
    }
    await new MessageDialog("Оправдание записано в файл " + excuseFile.Name).ShowAsync();
    await UpdateFileDateAsync();
}
```

Методы чтения и записи объектов Excuse очень похожи на аналогичные методы из приложения Guy Serializer.

```
public async void SaveCurrentExcuseAsAsync() {
    FileSavePicker picker = new FileSavePicker {
        SuggestedStartLocation = PickerLocationId.DocumentsLibrary,
        SuggestedFileName = CurrentExcuse.Description,
        CommitButtonText = "Save Excuse File"
    };
    picker.FileTypeChoices.Add("XML File", new List<string>() { ".xml" });
    IStorageFile newExcuseFile = await picker.PickSaveFileAsync();
    if (newExcuseFile != null) {
        excuseFile = newExcuseFile;
        await WriteExcuseAsync();
    }
}
```

Метод SaveCurrentExcuseAsAsync() отображает средство выбора, а затем сохраняет оправдание в указанный файл. Он обновляет поле excuseFile, фиксируя новый сохраненный файл (поэтому кнопка Save сохраняет в этот новый файл).

```
public event PropertyChangedEventHandler PropertyChanged;

private void OnPropertyChanged(string propertyName) {
    PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
    if (propertyChangedEvent != null) {
        propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Это обычный код вызова события PropertyChanged.

Это новый класс `Excuse`. Он содержит контракт данных, включающий в себя свойства `Description` и `Results` и вспомогательное поле `lastUsed` для свойства `LastUsed`.

```
using System.ComponentModel;
using System.Runtime.Serialization;

[DataContract(Namespace="http://www.headfirstlabs.com/ExcuseManager")]
class Excuse : INotifyPropertyChanged {
    public string DateWarning { get; set; }

    [DataMember]
    public string Description { get; set; }

    [DataMember]
    public string Results { get; set; }

    [DataMember]
    private DateTime lastUsed = DateTime.MinValue;
    public string LastUsed {
        get {
            if (lastUsed != DateTime.MinValue)
                return lastUsed.ToString();
            else
                return String.Empty;
        }
        set {
            DateTime d = DateTime.MinValue;
            bool dateIsValid = DateTime.TryParse(value, out d);
            lastUsed = d;

            if (!String.IsNullOrEmpty(value) && !dateIsValid) {
                DateWarning = "Некорректная дата: " + value;
            }
            else
                DateWarning = String.Empty;
            OnPropertyChanged("DateWarning");
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string propertyName) {
        PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
        if (propertyChangedEvent != null) {
            propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Применение атрибута [DataMember] к вспомогательному полю lastUsed приводит к тому, что это поле начинает записываться и читаться во время сериализации и десериализации.

При вводе корректной даты метод DateTime.TryParse() преобразует ее к типу DateTime и возвращает значение true. В противном случае он оставляет переменной d значение DateTime.MinValue.

Если поле lastUsed имеет значение DateTime.MinValue, полю DateWarning присваивается предостережение, которое следует показать пользователю.

Код вызова события `PropertyChanged` уже встречался в этой главе. Но если, скопировав его в свой класс `Excuse` или `ExcuseManager`, вы забыли добавить к объявлению класса `: INotifyPropertyChanged`, элементы управления страницы не смогут выполнить связывание данных. В результате элементы управления страницы не смогут среагировать на вызываемые объектами события `PropertyChanged`.

Борьба с огнем надоедает

Программисты не должны уподобляться пожарным.
Вы усердно работали, штудировали справочники и руководства и, наконец, достигли вершины: теперь вы **главный программист**. Но вам до сих пор продолжают звонить с работы по ночам, потому что **программа упала** или **работает неправильно**. 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**.

Брайану нужны мобильные оправдания

Недавно Брайан перешел в международный отдел и теперь он путешествует по всему миру. Но ему по-прежнему приходится оправдываться, поэтому он установил написанную нами программу на ноутбук.

Но программа не работает!

После щелчка на кнопке Random Excuse (Случайное оправдание) появляется сообщение об ошибке. Оправдания не найдены? Как же так?

Возьми в руку карандаш

Вот пример неработающего кода. Справа вы видите сообщения о пяти исключениях. Укажите, какие строки кода стали причиной появления этих сообщений.

```
public static void BeeProcessor() {
    object myBee = new HoneyBee(36.5, "Zippo");
    float howMuchHoney = (float)myBee;

    HoneyBee anotherBee = new HoneyBee(12.5, "Buzzy");
    double beeName = double.Parse(anotherBee.MyName);

    double totalHoney = 36.5 + 12.5;
    string beesWeCanFeed = "";
    for (int i = 1; i < (int) totalHoney; i++) {
        beesWeCanFeed += i.ToString();
    }
    float f =
      float.Parse(beesWeCanFeed);

    int drones = 4;
    int queens = 0;
    int dronesPerQueen = drones / queens;

    anotherBee = null;
    if (dronesPerQueen < 10) {
        anotherBee.DoMyJob();
    }
}
```

Метод double.Parse("32") преобразует строку в переменную типа double со значением 32.

An unhandled exception of type 'System.OverflowException' occurred in mscorlib.dll

Additional information: Value was either too large or too small for a Single.

An unhandled exception of type 'System.NullReferenceException' occurred in BeeProcessingSystem.exe

Additional information: Object reference not set to an instance of an object.

An unhandled exception of type 'System.InvalidCastException' occurred in BeeProcessingSystem.exe

Additional information: Specified cast is not valid.

An unhandled exception of type 'System.DivideByZeroException' occurred in BeeProcessingSystem.exe

Additional information: Attempted to divide by zero.

An unhandled exception of type 'System.FormatException' occurred in mscorlib.dll

Additional information: Input string was not in a correct format.

Возьми в руку карандаш
Решение

Вот какие строки кода стали причиной появления сообщений об ошибках.

```
object myBee = new HoneyBee(36.5, "Zippo");
float howMuchHoney = (float)myBee;
```

Переменную myBee можно привести к типу float, но присвоить значение этого типа объекту HoneyBee невозможно. При запуске кода исполняющая среда не понимает, как осуществить такое приведение, поэтому появляется сообщение о необработанном исключении InvalidCastException.

An unhandled exception of type 'System.InvalidCastException' occurred in BeeProcessingSystem.exe

Additional information: Specified cast is not valid.

```
HoneyBee anotherBee = new HoneyBee(12.5, "Buzzy");
double beeName = double.Parse(anotherBee.MyName);
```

Методу Parse() нужно передать строку в определенном формате. При этом метод не знает, как преобразовать строку Buzzy в число. Это и становится причиной появления сообщения о необработанном исключении FormatException.

An unhandled exception of type 'System.FormatException' occurred in mscorlib.dll

Additional information: Input string was not in a correct format.

```
double totalHoney = 36.5 + 12.5;
string beesWeCanFeed = "";
for (int i = 1; i < (int) totalHoney; i++) {
    beesWeCanFeed += i.ToString();
}
float f = float.Parse(beesWeCanFeed);
```

Цикл for создает строку beesWeCanFeed, содержащую число, которое состоит более чем из 60 цифр. Переменной типа float невозможно присвоить столь большое число, именно поэтому мы видим исключение OverflowException.

An unhandled exception of type 'System.OverflowException' occurred in mscorlib.dll

Additional information: Value was either too large or too small for a Single.

Разумеется, все эти исключения появляются не сразу, программа выводит первое и останавливается. Второе исключение вы сможете увидеть только после того, как исправите ошибку, приведшую к первому.

```
int drones = 4;
int queens = 0;
int dronesPerQueen = drones / queens;
```

Увидеть такое сообщение об ошибке очень легко. Достаточно поделить любую переменную на ноль.

An unhandled exception of type 'System.DivideByZeroException' occurred in BeeProcessingSystem.exe

Additional information: Attempted to divide by zero.

Чтобы предотвратить появление ошибки, достаточно проверить значение параметра queens, ***перед тем*** *как делить на него переменную drones.*

```
anotherBee = null;
if (dronesPerQueen < 10) {
    anotherBee.DoMyJob();
}
```

Присвоив ссылке anotherBee значение null, вы дали понять компилятору, что она никуда не ведет. Исключение NullReferenceException — это способ, которым C# сообщает вам об отсутствии объекта, метод DoMyJob() которого вы вызываете.

An unhandled exception of type 'System.NullReferenceException' occurred in BeeProcessingSystem.exe

Additional information: Object reference not set to an instance of an object.

Ошибка `DivideByZero` вообще не должна появляться. Ведь ее можно заметить, просто посмотрев на код. Впрочем, то же самое можно сказать и про остальные исключения. Все эти ошибки предотвращаемы. Чем больше вы узнаете об исключениях, тем меньше ошибок будет в ваших программах.

Объект Exception

Итак, вы узнали, каким образом .NET сообщает вам, что с программой что-то не так, — это **исключения (exception)**. При их появлении создается объект, который называется, как несложно догадаться, `Exception`.

Представьте массив из четырех элементов. А теперь попытайтесь получить доступ к элементу номер 16 (с индексом 15, так как отсчет ведется с нуля):

> **ис-клю-че-ние, сущ.** человек или вещь, выдающиеся из общего ряда, не подчиняющиеся правилам. *Джим ненавидит арахисовое масло, но делает **исключение** для конфет от Кена.*

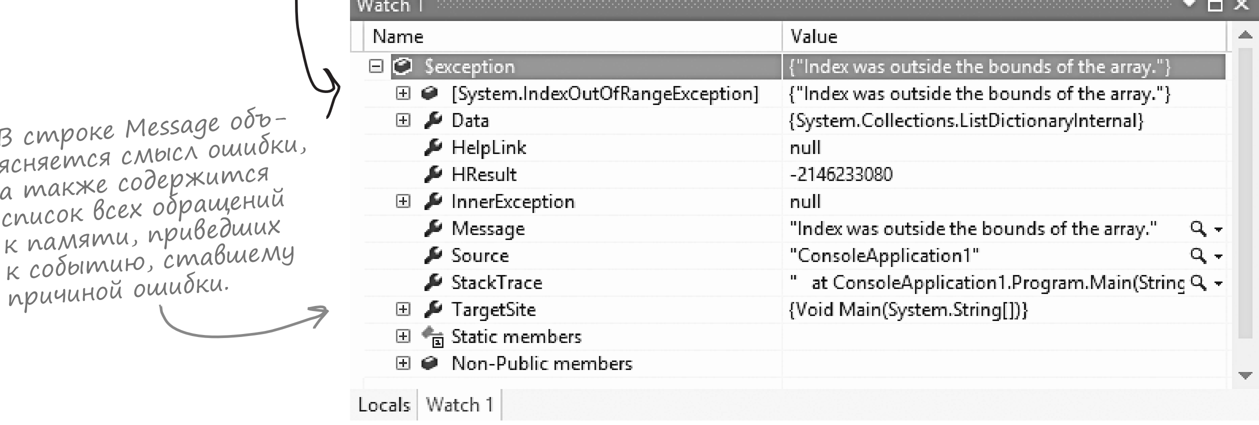
```
int[] anArray = {3, 4, 1, 11};
int aValue = anArray[15];
```

Очевидно, что это ошибочный код.

При возникновении исключения создается объект, содержащий сведения об ошибке.

Объект Exception

Когда IDE прекращает работу из-за исключения, для просмотра информации следует **добавить `$exception` в окно Watch**. Кроме того, информацию можно увидеть в окне Locals, которое отличается от окна Watch тем, что показывает текущие локальные переменные.



.NET создает объект, чтобы предоставить вам информацию об ошибке, ставшей причиной исключения. Может потребоваться редактирование кода или другие изменения.

В данном случае исключение **`IndexOutOfRangeException`** указывает на попытку обратиться к элементу массива с несуществующим индексом. Кроме того, вы знаете, в каком месте программы возникла проблема. То есть вы можете легко локализовать ее даже в случае очень длинного кода.

Часто Задаваемые Вопросы

В: Почему исключений так много?

О: Существует много способов сделать ошибку. В случае общей формулировки («Проблема в строке 37») сложно понять смысл проблемы. Ошибку проще исправить, когда точно знаешь, в чем она заключается.

В: Так что же такое исключение?

О: Это объект, который .NET создает в случае возникновения проблем. Впрочем, вы и сами можете создавать такие объекты (об этом мы поговорим чуть позже).

В: Что? Это *объекты*?

О: Да, исключение — это **объект**. Свойства объекта сообщают вам информацию об исключении. Например, свойство `Message` — это строка вида «Указанное присвоение неосуществимо» или «Слишком большое (слишком маленькое) значение для переменных данного типа», которая появляется во всплывающем окне. Вам дается максимум возможной информации о том, что именно происходит при выполнении оператора, который стал причиной исключения.

В: К сожалению, я все равно не понял: зачем нужно такое количество исключений?

О: Потому что способов некорректной работы кода великое множество. Существуют ситуации, в которых код просто перестает работать. Не зная, что стало причиной, устранить проблему крайне сложно. Создавая разные исключения, .NET дает вам информацию, позволяющую отследить ошибку и исправить ее.

В: То есть исключения придуманы, чтобы помочь пользователям?

О: Да! Большинство пользователей расстраиваются при виде сообщения об исключении. Но эти сообщения нужно воспринимать как помощь в отслеживании ошибок.

В: Правда ли, что появление исключения вовсе не означает, что я сделал что-то не так?

О: Правда. Иногда данные ведут себя не так, как вы ожидаете: например, можно написать метод, который будет работать с массивом иной длины, чем изначально предполагалось. Следует помнить, что пользователи действуют непредсказуемым образом. Благодаря исключениям программы не останавливаются в нетипичных ситуациях, а продолжают работу.

В: После сообщений об ошибках стало ясно, что код на предыдущей странице работать не будет. Всегда ли исключения позволяют понять, что происходит?

О: К сожалению, иногда локализовать проблему, просто посмотрев на код, невозможно. Поэтому в IDE предусмотрен режим **отладки**. Программа выполняется строчка за строкой, оператор за оператором, показывая мгновенное значение каждой переменной и каждого поля. Это позволяет обнаружить, где именно код работает не так, как вы предполагали.

Исключения помогают обнаружить и исправить код, который работает не так, как вы предполагаете.

Код Брайана работает не так, как предполагалось

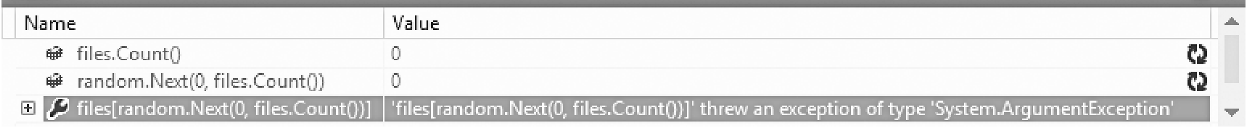
Когда мы писали программу для поиска оправданий, никто не предполагал, что пользователь начнет искать оправдания в пустой папке.

1. После перенесения программы Excuse Manager на ноутбук она стала ссылаться на пустую папку. В итоге щелчок на кнопке Random привел к появлению окна с сообщением о необработанном исключении:

2. Информации достаточно. Нам сообщают, что некое значение вышло за границы диапазона. Щелчок на кнопке Break вернет вас в отладчик, при этом выполнение программы будет остановлено на определенной строке:

```
public async void OpenRandomExcuseAsync()
{
    IReadOnlyList<IStorageFile> files = await excuseFolder.GetFilesAsync();
    excuseFile = files[random.Next(0, files.Count())];
    await ReadExcuseAsync();
}
```

3. Исследуем проблему в окне Watch. Добавим туда метод `files.Count()`. Кажется, он возвращает значение 0. Добавим контрольное значение для `random.Next(0, files.Count())`. Снова возвращается 0. Проследим за `files[random.Next(0, files.Count())]`.



В окне Watch можно вызывать методы и индексаторы. Порождаемые ими исключения также отображаются в окне Watch.

4 Что же произошло? Кажется, вызов метода `GetFilesAsync()` объекта `IStorageFolder` возвращает коллекцию `IReadOnlyList<IStorageFile>`. И, как это всегда бывает с коллекциями, попытка доступа к несуществующему элементу приводит к исключению. Попробуйте прочитать нулевой элемент пустой коллекции, и программа выдаст `System.ArgumentException` с сообщением «Value does not fall within the expected range».

К счастью, ситуацию легко исправить. Достаточно перед обращение к файлу проверить наличие нужного элемента в коллекции:

```
public async void OpenRandomExcuseAsync()
{
    IReadOnlyList<IStorageFile> files = await excuseFolder.GetFilesAsync();
    if (files.Count() == 0) {
        await new MessageDialog("The current excuse folder is empty.").ShowAsync();
        return;
    }
    excuseFile = files[random.Next(0, files.Count())];
    await ReadExcuseAsync();
}
```

Проверив наличие файлов с оправданиями в папке **до** создания объекта Excuse, мы предотвращаем появление сообщения об исключении — и вызываем окно со вспомогательной информацией.

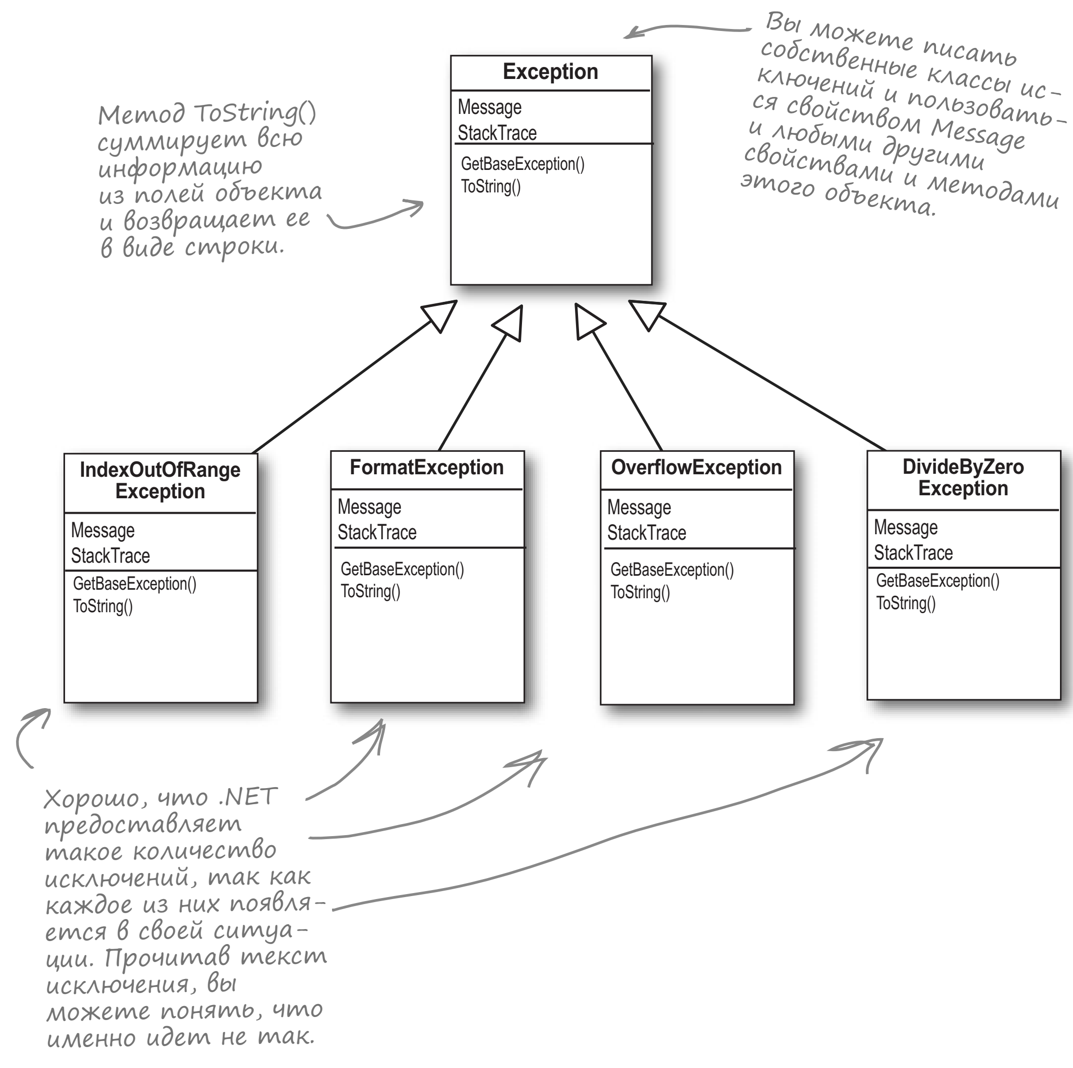
Именно так. Исключения являются полезным инструментом, который находит код, работающий неожиданным для нас способом.

Многие программисты расстраиваются, когда впервые сталкиваются с исключением. Но исключения можно превратить в преимущество. Ведь они дают ключ к пониманию причин неверной работы кода. И вы можете разработать новый, более удачный сценарий программы.

Исключения наследуют от объекта Exception

В .NET существует множество исключений. Так как многие из них схожи, имеет смысл говорить о наследовании. .NET определяет базовый класс `Exception`, от которого и наследуют все типы исключений.

Свойство `Message` этого класса содержит сообщение об ошибке. А свойство `StackTrace` указывает, какой код выполнялся в момент появления исключения и что именно привело к его появлению.



Работа с отладчиком

Панель инструментов Debug появляется только в режиме отладки программы.

Перед тем как добавлять обработку исключения, нужно понять, какие именно операторы стали причиной его появления. В этом вам поможет встроенный в IDE **отладчик**. Вам уже приходилось им пользоваться, но давайте теперь рассмотрим его подробно. После его запуска появляется панель инструментов с кнопками.

Щелкните на кнопке панели инструментов Debug и выберите команду «Add or Remove Buttons», чтобы изучить доступные команды отладки.

Вы уже пользовались кнопками Continue, Break All и Stop Debuging для приостановки, возобновления и завершения работы программ.

Кнопка "Refresh Windows app" применяется при работе с приложениями JavaScript. Для приложений C# она недоступна.

Add or Remove Buttons

Команда	Сочетание клавиш
Continue	F5
Break All	Ctrl+Alt+Break
Stop Debugging	Shift+F5
Restart	Ctrl+Shift+F5
Refresh Windows app	F4
Show Next Statement	Alt+Num *
Step Into	F11
Step Over	F10
Step Out	Shift+F11
Hex	
Show Threads in Source	
Windows	
Customize...	
Reset Toolbar	

Кнопка Show Next Statement перемещает редактор IDE к следующему оператору.

Вы уже выполняли программу пошагово. Кнопка Step Over позволяет пропустить метод. Кнопка Step Into перемещает вас к первому оператору вызываемого метода, а кнопка Step Out завершает текущий метод и приостанавливает программу после вызвавшего метод оператора.

В этой книге мы не будем говорить о потоках.

Если включить режим Hex появится возможность просматривать целочисленные значения int, long и byte в шестнадцатеричной системе.

value 0x3afb83d9 ← Одно и то же число слева показано слева в шестнадцатеричной системе, а справа — в десятичной. → value 989561817

Поиск ошибки в приложении Excuse Manager с помощью отладчика

Воспользуемся отладчиком для поиска ошибки в приложении Excuse Manager. В предыдущих главах вам уже приходилось работать с этим инструментом, тем не менее рассмотрим процедуру пошагово, чтобы не упустить никаких деталей.

1 **Добавим точку останова к обработчику события кнопки Random.**

Вам известно, с чего начинать. Исключение возникло при щелчке на кнопке Random Excuse после того, как была выбрана пустая папка. Поэтому откройте код кнопки и командой Debug→Toggle Breakpoint (F9) добавьте в метод точку останова. Начните отладку приложения, выберите пустую папку и щелкните на кнопке Random:

```
private void RandomExcuseButtonClick(object sender, RoutedEventArgs e)
{
    excuseManager.OpenRandomExcuseAsync();
}
```

2 **Войдем в метод OpenRandomExcuseAsync().**

Для отладки воспользуйтесь командой **Step Into** (F11). Затем при помощи команды Step Over (F10) пошагово пройдите метод. Из-за пустой папки программа покажет окно MessageDialog() и выйдет из метода.

Отладчик ждет окно MessageDialog, хотя оно вызывается с ключевым словом await.

```
IReadOnlyList<IStorageFile> files = await excuseFolder.GetFilesAsync();
if (files.Count() == 0)
{
    await new MessageDialog("The current excuse folder is empty.").ShowAsync();
    return;
}
excuseFile = files[random.Next(0, files.Count())];
```

Теперь выберите папку с извинениями, снова щелкните на кнопке Random и пошагово пройдите метод. Код пропустит оператор if и перейдет на следующую строку.

```
public async void OpenRandomExcuseAsync()
{
    IReadOnlyList<IStorageFile> files = await excuseFolder.GetFilesAsync();
    if (files.Count() == 0)
    {
        await new MessageDialog("The current excuse folder is empty.").ShowAsync();
        return;
    }
    excuseFile = files[random.Next(0, files.Count())];
    await ReadExcuseAsync();
}
```

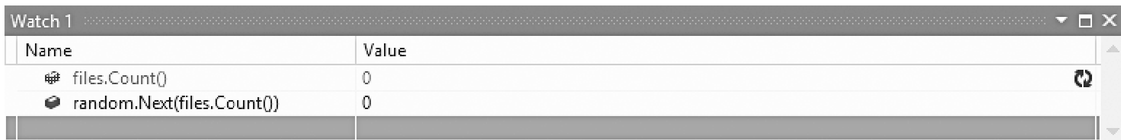
Воссоздадим проблема в окне Watch.

Остановите программу, удалите старую точку останова и поставьте **новую на второй строке метода `OpenRandomExcuseAsync()`**. Запустите программу, выберите пустую папку и щелкните на кнопке Random Excuse. После входа отладчика в метод выделите `files.Count()`, щелкните на строке правой кнопкой мыши и выберите Add Watch, чтобы добавить контрольное значение в окно Watch:

Нужно остановиться на второй строке, так как именно там происходит обращение к файловому объекту.

Добавим еще одну контрольную точку, чтобы начать отслеживание проблемы.

Отладка немного напоминает *судебно-медицинскую экспертизу*. Далеко не всегда известно, что именно мы ищем, поэтому нужно воспользоваться отладчиком, чтобы обнаружить улики и найти виновника. Так как метод `files.Count()` оказался ни при чем, перейдем к следующему подозреваемому. Выделите метод `random.Next(files.Count())` и добавьте его в окно Watch:



Еще одна полезная «фича». Вы можете **менять значения** отображаемых переменных и полей и даже **выполнять методы и создавать объекты**. При этом появляется значок, щелчок на котором вызывает метод повторно. Иногда повторный запуск дает другой результат (как в случае с методом `Random`).

Ловушка для виновника исключения.

Добавим в отладчик еще одну строку — оператор, который породил исключение: `files[random.Next(0, files.Count())]`. Как только вы его введете, окно Watch произведет оценку... и вызовет исключение!

Watch 1

Name	Value
files.Count()	0
random.Next(files.Count())	0
files[random.Next(0, files.Count())]	'files[random.Next(0, files.Count())]' threw an exception of type 'System.ArgumentException'

Даже после устранения проблемы путем добавления проверяющего наличие файлов в папке кода, исключение может появиться в окне Watch.

Щелкните на значке +, чтобы раскрыть исключение, и вы увидите, что свойство `Message` имеет значение «Value does not fall within the expected range». Теперь вы знаете, что стало причиной проблемы. Метод `GetFilesAsync()` возвращает коллекцию `IReadOnlyList<IStorageFile>`, у которой счетчик пустой папки (count) равен 0. Если воспользоваться индексатором (`files[0]`), появится `ArgumentException`.

Любое исключение можно сымитировать с помощью отладчика и воспользоваться объектом `Exception` для исправления ошибок в коде.

Часто Задаваемые Вопросы

В: При запуске программы в IDE сведения об исключении можно посмотреть в окне Watch. А как быть, если программа была запущена без IDE?

О: Напишите комментарий по поводу изменений, внесенных в метод OpenRandomExcuseAsync() для решения проблемы, и запустите приложение, выбрав в меню Debug команду Start Without Debugging. Это аналог запуска приложения на экране Start. Выберите пустую папку, щелкните на кнопке Random Excuse, и... приложение просто исчезнет.

Именно так происходит при наличии в приложении необработанных исключений. (Способы их обработки будут рассмотрены чуть позднее.) Большинство пользователей не хочет видеть окна со списком методов и информацией об исключении. Но не волнуйтесь, ваше исключение не пропало. Откройте панель управления Windows (введите «Control Panel» в поле поиска на экране Start), поищите слово «event» и просмотрите журнал событий. Раскройте раздел Windows Logs и щелчком выделите Application. Одно из событий Error в журнале Application будет содержать ваше исключение вместе с трассировкой стека, показывающей строку, которая стала причиной исключения, строку, которая ее вызвала, и т. п. (стек вызовов). В режиме отладки результат трассировки стека содержится в свойстве StackTrace объекта Exception.

В: То есть если программа работает вне IDE, с появлением сообщения об исключении ее работа просто останавливается и пользователь ничего не может сделать?

О: Да, программа останавливается, столкнувшись с ***необработанным*** исключением. Но ведь нигде не сказано, что все исключения будут необработанными! О способах обработки исключений и о том, каким образом создать программу без необработанных исключений, мы поговорим чуть позже.

В: Как выбрать место для точки останова?

О: Это хороший вопрос, который, к сожалению, не имеет однозначного ответа. При появлении исключений имеет смысл начинать с операторов, которые стали их причиной. Но обычно проблема гнездится в предыдущих строках кода, а исключение является не более чем следствием. К примеру, оператор, ставший причиной появления исключения «Деление на ноль», может использовать значения, вычисленные десятком строк ранее. Поэтому ответ на вопрос о месте точки останова в каждом конкретном случае будет отличаться. Но если вы представляете принцип работы своего кода, проблемы с выбором места не будет.

В: Любой метод можно запускать в окне Watch?

О: Да, там будут работать любые корректные операторы, даже те, которые не имеют смысла. Например, запустите программу, прервите ее выполнение и вставьте в окно Watch строку: System.Threading.Thread.Sleep(2000). Этот метод вызывает двухсекундную задержку программы. При реальной отладке мы не стали бы этого делать, но сейчас просто посмотрим, что произойдет: на две секунды, которые занимает выполнение метода, курсор перейдет в режим ожидания. Так как метод Sleep() не возвращает значения, в окне Watch появится сообщение «Expression has been evaluated and has no value», информирующее об отсутствии возвращаемого значения. Но значение было рассчитано, более того, появилось окно IntelliSense, помогающее вводу кода. Оно отображает доступные свойства и методы для находящихся сейчас в памяти объектов.

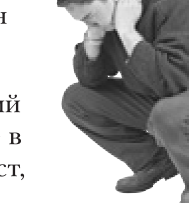
В: То есть я могу запустить в окне Watch что-то, меняющее способ работы программы?

О: Да! Пусть не всегда, но здесь вы можете влиять на конечный результат работы программы. Более того, даже ***наведение указателя мыши*** на поля внутри отладчика может изменить поведение программы, так как при этом ***выполняется метод чтения свойства***. И если этот метод задает какое-то значение, оно перейдет в программу. Такое поведение делает результаты отладки непредсказуемыми. Программисты в шутку называют такие результаты **гейзенбергскими** (эта шутка понятна только физикам и котам в ящике).

Запущенная в IDE программа при появлении необработанных исключений прерывается, как при достижении точки останова.

А код все равно не работает...

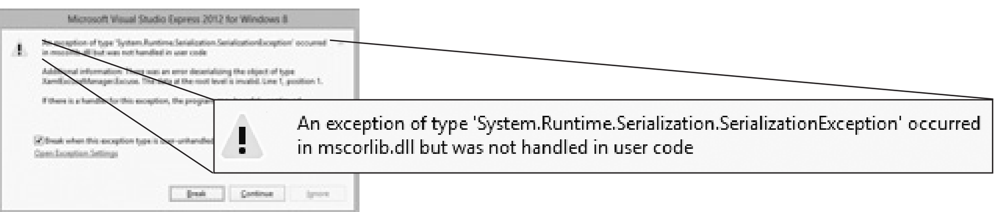
Только не это!



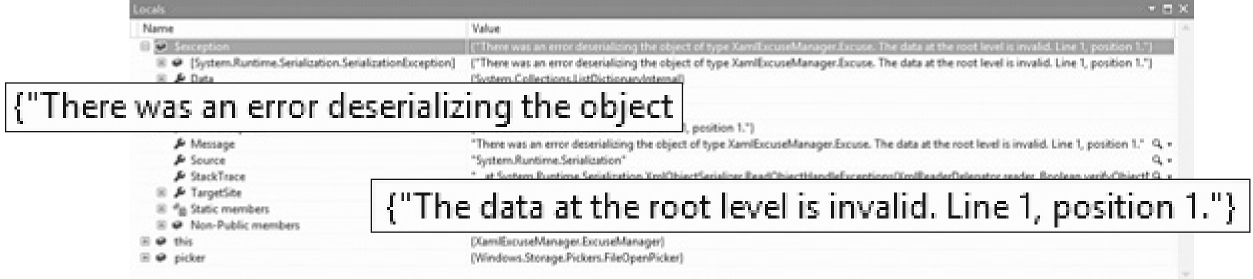
Брайан был счастлив, пока его Excuse Manager не открыл папку, полную чужих XML-файлов. Попробуем разобраться, что произошло, когда он попытался загрузить один из таких файлов...

1. Мы можем воссоздать проблему Брайана. Найдите файл XML, содержащий сериализованный объект Excuse. Откройте его в Блокноте и добавьте в самое начало — сразу за открывающим символом < — некорректный текст, не являющийся кодом XML.

2. Вызовите Excuse Manager в IDE и откройте оправдания. Появится исключение! Прочитайте сообщение и щелкните на кнопке Break для перехода к расследованию.



3. Откройте окно Locals и откройте строку `$exception` (ее можно ввести и в окне Watch). Внимательно посмотрите на члены этого объекта, может быть, вы поймете, что именно пошло не так.



Поняли, почему программа сгенерировала исключение? Нужно ли программе аварийно прекращать работу, обнаружив некорректный XML-файл Excuse?

Есть идеи, что делать в этой ситуации?

Разумеется, программа должна была прекратить работу, ведь я дала ей не тот файл. Пользователи все время делают что-то не то, и с этим невозможно бороться.

На самом деле бороться с этим можно.

Разумеется, пользователи постоянно делают что-то не так. Такова жизнь. Но существуют программы, способные работать с неверными данными, ошибками при их вводе и другими неожиданными ситуациями: их называют **робастными (robust)**. Инструменты обработки исключений C# дают вам возможность создавать именно такие программы. Разумеется, вы *не можете* контролировать действия пользователей, но вы *можете* гарантировать, что программа не прекратит работу, что бы они ни делали.

ро-баст-ный, прил.
прочный; способный противостоять неблагоприятным условиям.

Будьте осторожны!

Сериализаторы генерируют исключение, если что-то не так с сериализованным файлом.

Получить исключение **`SerializationException`** *для программы Excuse Manager очень просто. Дайте ей файл, который не является сериализованным объектом Excuse. В процессе десериализации* **`DataContractSerializer`** *ищет объект, совпадающий с контрактом читаемого класса. Если же файл содержит что-то другое, метод* **`ReadObject()`** *вызывает исключение* **`SerializationException`**.

Класс BinaryFormatter генерирует исключение SeralizationException, если ему предоставить файл, не содержащий корректно сериализованного объекта. Разборчивостью он даже превосходит класс DataContractSerializer!

Ключевые слова try и catch

Происходящее в C# можно описать фразой «**Протестируйте (try)** этот код и при появлении исключения **превите (catch)** его *другим* кодом». Тестируемая часть кода называется **блоком `try`**, а часть, обрабатывающая исключения, — **блоком `catch`**. В блоке `catch` можно добавить сообщение об ошибке, не давая программе аварийно завершить работу.

Поместите в блок try код, который может стать причиной исключения. Если исключение не появится, операторы в блоке catch block будут проигнорированы. Если же в блоке try возникнет исключение, остаток этого блока выполняться не будет.

```
public async Task ReadExcuseAsync() {
   try
   {
     using (IRandomAccessStream stream =
             await excuseFile.OpenAsync(FileAccessMode.Read))
     using (Stream inputStream = stream.AsStreamForRead()) {
        DataContractSerializer serializer
                  = new DataContractSerializer(typeof(Excuse));
        CurrentExcuse = serializer.ReadObject(inputStream) as Excuse;
     }

      await new MessageDialog("Excuse read from "
                               + excuseFile.Name).ShowAsync();
      OnPropertyChanged("CurrentExcuse");
      await UpdateFileDateAsync();
    }
    catch (SerializationException)
    {
       new MessageDialog("Unable to read " + excuseFile.Name).ShowAsync();
    }
}
```

Это блок try. С него начинается обработка исключения. Мы поместим в него существующий код.

Этот код вы знаете, так как мы поместили целый метод в блок try.

Ключевое слово catch означает, что следующий блок операторов содержит обработчик исключения.

После появления исключения программа немедленно переходит к операторам блока catch.

Самый простой способ обработки исключений — остановить программу, показать сообщение об исключении и продолжить работу. Обратили внимание на отсутствие ключевого слова `await` при показе `MessageDialog`? Дело в том, что **в теле блока catch ожидание невозможно**. Можно вызвать метод `ShowAsync()`, но он блокируется до момента, пока пользователь не закроет окно диалога.

Если исключение немедленно передает управление операторам блока `catch`, что происходит с объектами и данными, с которыми вы работали до этого?

Вызов сомнительного метода

Пользователи непредсказуемы. Они вводят в программу странные данные, щелкают на кнопках. И это прекрасно, ведь вы можете справиться с последствиями ввода таких данных, обрабатывая исключения.

1 **Предположим, пользователь вводит не те данные.**

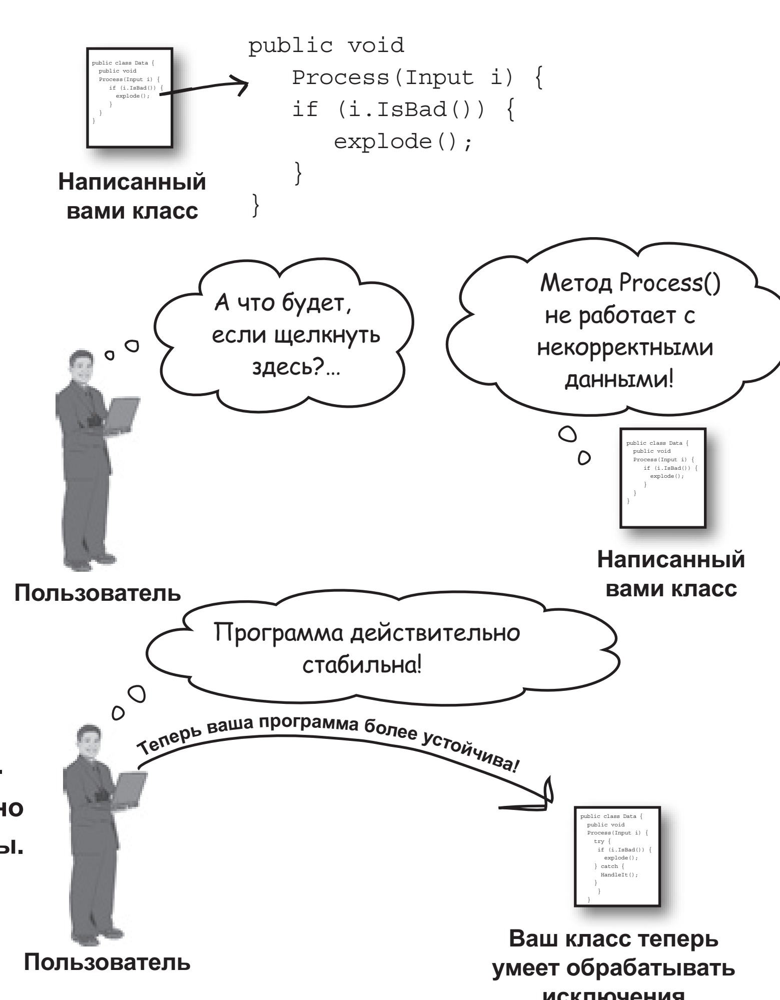
2 **Метод делает что-то странное, что может и не сработать во время прогона.**

Временем прогона (Runtime) называется время работы вашей программы. Исключения иногда еще называют «ошибками при исполнении» (runtime errors).

3 **Вы должны *знать,* что вызываемый метод сомнителен.**

Лучше всего, если вы предусмотрите обходной путь на случай возникновения исключений! Впрочем, полностью устранить риск не получится, поэтому следует поступать так.

4 **В этом случае вы сможете написать код, обрабатывающий исключение. Если оно появится, вы будете готовы.**



Часто задаваемые вопросы

В: Так когда используются ключевые слова `try` и `catch`?

О: В случаях, когда вы пишете сомнительный код, который может привести к исключениям. Сложнее всего понять, что код является сомнительным.

Вы уже видели, что сомнительную работу кода может инициировать ввод неверных данных. Пользователи могут выбирать не те файлы, вводить буквы вместо цифр и имена вместо дат, а еще они нажимают все кнопки, до которых могут дотянуться. Хорошая программа должна работать предсказуемо вне зависимости от вводимых данных. Пользователь может не получить нужного результата, но по крайней мере он узнает о наличии проблемы и ознакомится с предложенным решением.

В: А как можно предложить решение проблемы, о существовании которой заранее неизвестно?

О: Для этого вам потребуется блок `catch`, который выполняется только после того, как блок `try` выдаст исключение. Вы даете пользователю сигнал: что-то идет не так и ситуацию можно и нужно исправить. Аварийное завершение работы программы при вводе некорректных данных бесполезно. Нет также никакой пользы от попыток читать все вводимые данные. А вот появление окна с сообщением о невозможности прочитать файл позволяет принять верное решение.

В: То есть отладчик нужен для поиска причины исключений?

О: Нет. Вы уже не раз видели, что отладчик работает с любым кодом. Иногда имеет смысл пошагово просмотреть значения определенных полей и переменных — именно так можно гарантировать корректную работу сложных методов.

Но основным назначением отладчика является поиск и устранение дефектов программы. Исключения также относятся к дефектам. Но отладчик решает проблемы и другого рода, например поиск кода, дающего непредсказуемый результат.

В: Я не совсем понимаю принцип работы с окном Watch.

О: В процессе отладки обращается внимание на значения определенных полей и переменных. Именно для этого нужно окно Watch. Переменные, к которым были добавлены контрольные значения, обновляют свои значения в окне Watch с каждым следующим шагом. Это позволяет отслеживать, что именно с ними происходит после выполнения каждого оператора.

В окно Watch можно ввести любой оператор и увидеть его значение. Если оператор влияет на значения полей и переменных, значит, он будет выполнять еще и эту функцию. Это позволяет менять параметры, не прерывая работу программы, что дает вам еще один инструмент для отслеживания дефектов и исключений.

Редактирование данных в окне Watch влияет на их состояние в памяти на время работы программы. Для возвращения переменным исходных значений достаточно перезагрузить программу.

Блок catch выполняется только при обнаружении исключения в блоке try. Это дает возможность снабдить пользователя информацией о путях решения проблемы.

Результаты применения ключевых слов try/catch

Следует помнить, что при обнаружении исключения в блоке try остальная часть кода игнорируется. Программа немедленно переходит на первую строчку блока catch. ***Впрочем, вы можете не верить нам на слово...***

1. Добавьте блок try/catch, который рассматривался несколько страниц назад, в метод ReadExcuseAsync() приложения Excuse Manager. Затем поместите точку останова на открывающуюся скобку { в блоке try.

> **SerializationException находится в пространстве имен System.Runtime.Serialization. К счастью, в верхней части файла *ExcuseManager.cs* уже есть строка using System.Runtime.Serialization.**

2. Начните отладку приложения и откройте **некорректный файл оправданий** (тем не менее имеющий расширение *.xml*). После точки остановка щелкните пять раз на кнопке Step Over (или нажмите F10) для перехода к оператору, вызывающему метод ReadObject(), чтобы десериализовать объект Excuse. Вот как должен выглядеть экран отладчика:

```
public async Task ReadExcuseAsync()
{
    try
    {
        using (IRandomAccessStream stream =
                await excuseFile.OpenAsync(FileAccessMode.Read))
        using (Stream inputStream = stream.AsStreamForRead())
        {
            DataContractSerializer serializer
                    = new DataContractSerializer(typeof(Excuse));
            CurrentExcuse = serializer.ReadObject(inputStream) as Excuse;
        }

        await new MessageDialog("Excuse read from to "
                              + excuseFile.Name).ShowAsync();
        OnPropertyChanged("CurrentExcuse");
        await UpdateFileDateAsync();
    }
    catch (SerializationException)
    {
        new MessageDialog("Unable to read " + excuseFile.Name).ShowAsync();
    }
}
```

Поместите точку останова на открывающуюся скобку блока try.

Выполните пошаговый проход, пока желтая полоса не покажет, что следующий выполняемый оператор начнет читать из потока объект Excuse.

3 Продолжайте пошаговый проход кода. Сразу после выполнения оператора `ReadObject()` появится исключение, и программа, **проигнорировав остаток метода, перейдет к блоку catch**.

Отладчик выделит желтым строчку с ключевым словом catch, в то время как остальной код блока будет помечен серым, то есть готовым к выполнению.

```
public async Task ReadExcuseAsync()
{
    try
    {
        using (IRandomAccessStream stream =
                await excuseFile.OpenAsync(FileAccessMode.Read))
        using (Stream inputStream = stream.AsStreamForRead())
        {
            DataContractSerializer serializer
                    = new DataContractSerializer(typeof(Excuse));
            CurrentExcuse = serializer.ReadObject(inputStream) as Excuse;
        }

        await new MessageDialog("Excuse read from to "
                        + excuseFile.Name).ShowAsync();
        OnPropertyChanged("CurrentExcuse");
        await UpdateFileDateAsync();
    }
    catch (SerializationException)
    {
        new MessageDialog("Unable to read " + excuseFile.Name).ShowAsync();
    }
}
```

4 Снова запустите программу, щелкнув на кнопке Continue (или нажав F5). Выполнение начнется с подсвеченного желтым блока, в данном случае с блока `catch`. Программа отобразит окно диалога и будет работать дальше, как будто ничего не произошло. Мы обработали исключение, приводящее к аварийному завершению программы.

Совет на будущее: интервью с соискателями на должность программиста часто включают в себя вопрос о том, как следует поступать с исключениями в конструкторе.

Поосторожнее с исключениями в конструкторе!

*Думаю, вы уже заметили, что у конструктора отсутствует возвращаемое значение. Дело в том, что его назначением является инициализация объекта, и именно поэтому так сложно обрабатывать исключения, возникшие внутри конструктора. Наличие исключения означает, что оператор, создающий экземпляр класса, **не смог его получить**. Блок `try/catch` в этом случае имеет смысл поместить в обработчик событий кнопки, чтобы код не ожидал найти в классе `CurrentExcuse` корректный объект `Excuse`.*

Ключевое слово finally

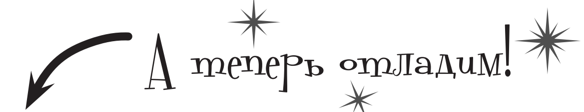
В случае исключения возможны два варианта развития событий. ***Не***обработанное исключение ведет к аварийному завершению программы. В противном случае управление переходит к блоку catch. Но что происходит с остальным кодом блока try? Представьте, что в этой части закрывался поток. Тогда код необходимо выполнить, несмотря на исключение. На помощь в этой ситуации приходит блок **finally**. Он **запускается всегда**, вне зависимости от появления или отсутствия исключения. Вот, как он позволяет гарантировать, что метод ReadExcuseAsync() всегда будет вызывать событие PropertyChanged:

```
public async Task ReadExcuseAsync() {
    try
    {
        using (IRandomAccessStream stream =
                await excuseFile.OpenAsync(FileAccessMode.Read))
        using (Stream inputStream = stream.AsStreamForRead()) {
            DataContractSerializer serializer
                        = new DataContractSerializer(typeof(Excuse));
            CurrentExcuse = serializer.ReadObject(inputStream) as Excuse;
        }

        await new MessageDialog("Excuse read from to "
                                    + excuseFile.Name).ShowAsync();
        await UpdateFileDateAsync();
    }
    catch (SerializationException)
    {
        new MessageDialog("Unable to read " + excuseFile.Name).ShowAsync();
        NewExcuse();
    }
    finally
    {
        OnPropertyChanged("CurrentExcuse");
    }
}
```

Вызов метода NewExcuse() сбрасывает объект Excuse, но без события PropertyChanged страница не будет считывать свойство CurrentExcuse. Блок finally гарантирует возникновение события PropertyChanged вне зависимости от появления исключения.

Добавление в блок catch метода NewExcuse() очищает форму, если конструктор Excuse вызывает исключение.

Рядом с ключевым словом catch указано, что отслеживается исключение SerializationException. Это принятый в C# код **catch (Exception)**, хотя тип исключения можно и опустить, оставив только слово **catch**. В этом случае **будут отслеживаться все возможные исключения**. Хотя это ***не очень хорошо***. Лучше указывать, какое исключение вы хотите отследить, и указывать как можно более подробно.



1. Обновите метод `ReadExcuseAsync()`, добавив код с предыдущей страницы. Поместите точку останова на открывающуюся скобку блока `try` и запустите отладку.

2. Запустите программу в обычном режиме и убедитесь, что кнопка Open работает при загрузке корректного файла с оправданиями. В точке останова отладка прервется:

```
public async Task ReadExcuseAsync()
{
    try
    {
        using (IRandomAccessStream stream =
                await excuseFile.OpenAsync(FileAccessMode.Read))
        using (Stream inputStream = stream.AsStreamForRead())
        {
            DataContractSerializer serializer
                    = new DataContractSerializer(typeof(Excuse));
            CurrentExcuse = serializer.ReadObject(inputStream) as Excuse;
        }

        await new MessageDialog("Excuse read from to "
                              + excuseFile.Name).ShowAsync();
        await UpdateFileDateAsync();
    }
    catch (SerializationException)
    {
        new MessageDialog("Unable to read " + excuseFile.Name).ShowAsync();
        NewExcuse();
    }
    finally
    {
        OnPropertyChanged("CurrentExcuse");
    }
}
```

Когда желтая строка попадает на точку останова, на красной точке на полях появляется желтая стрелка.

Особое внимание обращайте на окна диалога. Иногда окно не отображается до завершения метода. Добро пожаловать в мир асинхронных методов!

3. Пошагово просмотрите обработчик событий кнопки Random Excuse и убедитесь, что она работает корректно. После завершения блока `try` должен произойти переход к блоку `finally`, так как исключений не обнаружено.

4. Теперь попробуем открыть поврежденный файл с оправданиями. Из блока `try` при обнаружении исключения управление переходит к блоку `catch`. После выполнения всех операторов блока `catch` начинается выполнение блока `finally`.

Часто Задаваемые Вопросы

В: При отсутствии блока catch моя программа, столкнувшись с исключением, будет просто остановлена. Что же в этом хорошего?

О: Основным преимуществом исключений является то, что они не дают пройти мимо проблемы. В больших приложениях сложно следить за всеми объектами, с которыми ведется работа. Исключения привлекают внимание к проблемам и помогают понять причины их возникновения.

Появление исключения в вашей программе предупреждает: что что-то идет не так, как было запланировано. 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. Подобные вещи меняют результат работы вашей программы.

А теперь представьте, что вы даже не знаете обо всех этих ошибках. Пользователь же вводит некорректные данные и начинает жаловаться, что приложение нестабильно. Нарушающие работу вашей программы исключения позволяют узнать о проблеме на том этапе, когда ее решение еще можно провести относительно легко и безболезненно.

В: Чем обработанное исключение отличается от необработанного?

О: При появлении исключений программа начинает искать блок `catch`, занимающийся обработкой. При наличии такого блока будут выполнены все полагающиеся в случае конкретного исключения действия. Фактически написанием блока `catch` вы проводите предварительную работу над ошибками. Если же такой блок отсутствует, программа просто прекращает работу, выбрасывая окно с сообщением. В этом случае речь идет о необработанном исключении.

В: Зачем указывать в блоке catch тип обрабатываемого исключения? Разве не безопаснее использовать универсальный код?

О: Безопаснее всего **вообще избегать ситуаций, в которых возникают объекты `Exception`**. Лучше провести профилактику, чем употреблять лекарство. Это правило работает и с исключениями. Попытка обработать все исключения сразу указывает на плохое программирование. Например, лучше воспользоваться методом `File.Exists()` для проверки наличия файла, а не обрабатывать исключение `FileNotFoundException`. Хотя бывают исключения, которых просто не избежать, но большинство из них не имеет приоритета.

Иногда имеет смысл оставлять исключения необработанными. Логика реальных программ зачастую крайне сложна, и корректно обойти ошибку не всегда удается, особенно если проблема возникает где-то в нижней части кода. Обрабатывая конкретные исключения, избегая слишком общих подходов и попыток решить все проблемы сразу, позволяя исключениям всплывать, чтобы обработать их на верхнем уровне, вы сделаете свой код намного более стабильным.

Систему, немедленно сообщающую о неполадках (вместо того чтобы медленно терять стабильность), иногда называют «системой с быстрым прекращением».

В: Что произойдет, если не указать тип исключения рядом с ключевым словом `catch`?

О: Блок `catch` будет работать с любым исключением из блока `try`.

В: Но если блок `catch` может обрабатывать любые исключения, зачем мне указывать конкретный тип?

О: Не существует универсального способа обработки всех исключений. Для устранения проблемы с делением на ноль в блоке `catch` нужно изменить значения некоторых переменных и сохранить некоторые данные. А чтобы убрать ссылку на значение null, может потребоваться создание нового экземпляра.

В: Обработка ошибок происходит только в последовательности `try`/`catch`/`finally`?

О: Нет, воспользуйтесь **набором блоков `catch`**, если вы хотите учесть различные виды ошибок. Если этого блока нет, исключения не будут обрабатываться, а код блока `finally` начнет выполняться даже при остановке из-за исключения в блоке `try`.

Необработанное исключение означает непредсказуемую работу. Поэтому программа перестает работать при их появлении.

Ребус в бассейне

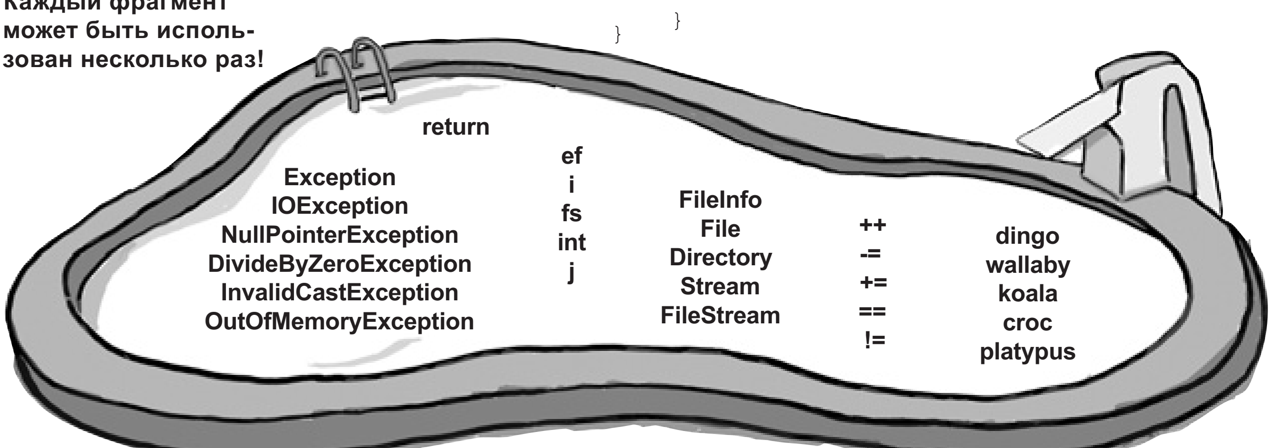
Поместите фрагменты кода из бассейна на пустые строчки программы. Каждый фрагмент может использоваться несколько раз. Имеются и лишние фрагменты. Нужно получить следующую строку.

Результат: ⟶ **G'day Mate!**

```
using System.IO;
public static void Main() {
    Kangaroo joey = new Kangaroo();
    int koala = joey.Wombat(
           joey.Wombat(joey.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) {
            __ = _____.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 может быть использован несколько раз!



Ребусы становятся все более сложными, а имена переменных все менее очевидными. Над решением приходится много думать! Но ребусы решать не обязательно, вы можете просто перевернуть страницу и продолжить чтение… это для любителей головоломок!

Решение ребуса в бассейне

Метод Joey.Wombat() вызывался три раза и на третий раз вернул ноль. Поэтому метод WriteLine() вызвал исключение DivideByZeroException.

```
public static void Main() {
    Kangaroo joey = new Kangaroo();
    int koala = joey.Wombat(joey.Wombat(joey.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;
    }
}
```

Этот catch работает только с исключениями, возникшими в результате деления на ноль.

Объект FileStream имеет метод OpenRead() и становится причиной появления исключения IOException.

Этот код открывает файл wobbiegong. Затем он открывает его повторно. Но файл не закрывается, что приводит к исключению IOException.

Помните, что желательно обрабатывать исключения конкретного типа. Впрочем, в ребусах мы делаем и другие вещи, которых нужно избегать при написании реальных программ. Например, мы выбираем для переменных незначимые имена.

Вы уже знаете, что после завершения работы с файлом его требуется закрыть, иначе файл окажется заблокированным. Попытка снова его открыть станет причиной исключения IOException.

Получение сведений о проблеме

Мы уже несколько раз повторили, что при появлении исключения .NET создает объект Exception. Доступ к нему предоставляется через код блока catch. Вот как это работает:

Если оператор метода DoSomethingRisky() генерирует необрабатываемое внутри метода исключение, оно перехватывается обработчиком для кода вызвавшего метода. При отсутствии обработчика исключение будет перемещаться выше. Достигнув верхнего уровня, оно превратится в необработанное исключение, заставляющее программу аварийно прервать работу.

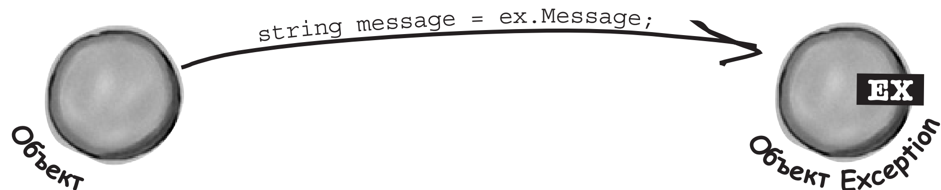
1. Некий объект выполняет некие функции, и вдруг нештатная ситуация приводит к появлению исключения.

2. К счастью, срабатывает блок try/catch. Внутри блока catch исключению присваивается имя **ex**.

```
try {
    DoSomethingRisky();
}
catch (RiskyThingException ex) {
    string message = ex.Message;
    new MessageDialog("I took too many risks! "
                                + message).ShowAsync();
}
```

Если рядом с объявлением типа исключения в блоке catch указать имя переменной, эту переменную можно будет использовать для доступа к объекту Exception.

3. После завершения работы блока catch ссылка **ex** исчезает, и объект отправляется в мусорную корзину.



Обработка исключений разных типов

Вы уже знаете, как работать с исключением определенного типа... но что делать, если код имеет несколько проблемных мест? Вам может потребоваться обработать набор различных исключений. В этом случае не обойтись без набора блоков `catch`. Вот пример кода для завода по переработке нектара, в котором обрабатываются исключения различных типов. В некоторых случаях используются свойства объекта `Exception`, например свойство `Message`, содержащее информацию об исключении. Можно также прибегнуть к оператору `throw` для сообщений об аномальных ситуациях.

Метод `ToString()` позволяет вывести в окно MessageBox относящуюся к делу информацию.

```
public void ProcessNectar(NectarVat vat, Bee worker, HiveLog log) {
    try {
        NectarUnit[] units = worker.EmptyVat(vat);
        for (int count = 0; count < worker.UnitsExpected, count++) {
            stream hiveLogFile = log.OpenLogFile();
            worker.AddLogEntry(hiveLogFile);
        }
    }
    catch (VatEmptyException) {
        vat.Emptied = true;
    }
    catch (HiveLogException ex) {
        throw;
    }
    catch (IOException ex) {
        worker.AlertQueen("An unspecified file error happened: "
            + "Message: " + ex.Message + "\r\n"
            + "Stack trace: " + ex.StackTrace + "\r\n"
            + "Data: " + ex.Data + "\r\n");
    }
    finally {
        vat.Seal();
        worker.FinishedJob();
    }
}
```

*Иногда требуется **повторно вызвать** перехваченное исключение. Для этого используется оператор throw.*

Если вы не собираетесь использовать объект Exception, объявлять его не нужно.

Набор блоков catch обрабатывается по очереди. В нашем случае сначала рассматривается VatEmptyException, а потом HiveLogException. Последний блок catch обрабатывает IOException. Это базовый класс для различных исключений, в том числе для FileNotFoundException (Файл не найден) и EndOfStreamException (Конец потока).

*В блоке catch объекту exception сопоставляют переменную **ex**, которая затем может использоваться для получения информации об объекте.*

Для объекта Exception в различных блоках можно использовать одно и то же имя (ex).

Этот оператор использует три свойства объекта Exception: Message, описывающее текущее исключение («Попытка деления на ноль»); StackTrace, возвращающее строковое представление стека вызова; и Data, иногда содержащее дополнительную информацию об исключении.

Один класс создает исключение, другой его обрабатывает

Разумеется, бывает и такое, что один метод может формировать исключение, которое ликвидируется другим методом этого же класса.

В момент создания класса не известно, как с ним будут работать. Иногда действия пользователей становятся источником проблемы. Именно в этих ситуациях возникают исключения.

Вам нужно заранее понять, что может пойти не так, и предусмотреть план перехвата. Вы обычно не видите, какой метод создает исключение, а какой устраняет его. Это, как правило, различные методы, принадлежащие различным объектам.

Вместо того чтобы...

Без обработки исключений программа перестает работать. Вот что происходит в программе управления профилями пчел.

Конструктор объекта BeeProfile ожидает имя файла, содержащего профиль пчелы, чтобы открыть его методом File.Open(). Если файл не открывается, программа перестает работать.

```
stream = File.Open(profile);
```

new BeeProfile("prof.dat")

Объект Hive

Объект BeeProfile

An unhandled exception of type 'System.IO.FileNotFoundException' occurred in mscorlib.dll

FileNotFoundException

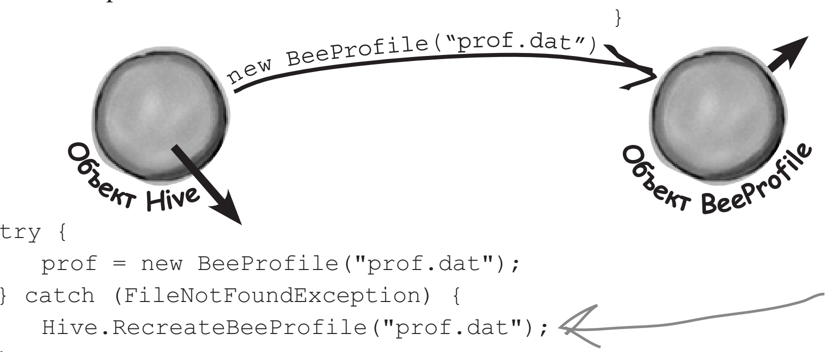
Объект BeeProfile пытается читать отсутствующий файл, поэтому метод File.OpenRead() информирует об исключении. В программе отсутствует блок catch, поэтому исключение остается необработанным.

*Обратите внимание, как объект BeeProfile **перехватывает** исключение, информирует о нем при помощи метода WriteLogEntry() и вызывает его повторно для передачи в улей.*

```
try {
   stream = File.OpenRead(profile);
} catch (FileNotFoundException ex) {
   WriteLogEntry("unable to find " +
      profile + ": " + ex.Message();
   throw;
}
```

...мы можем поступить так.

Объект BeeProfile может перехватить исключение и добавить запись об этом в журнал. Затем исключение вызывается повторно и передается в улей для обработки.




```
try {
   prof = new BeeProfile("prof.dat");
} catch (FileNotFoundException) {
   Hive.RecreateBeeProfile("prof.dat");
}
```

Теперь при попытке улья создать объект BeeProfile, передав неверное имя файла, будет сделана запись об ошибке и появится сообщение об исключении. Улей сможет обработать исключение, например, путем повторного создания файла с профилем пчелы.

Исключение OutOfHoney для пчел

Методы показывают такое исключение при некорректных значениях параметров.

Ваши классы могут формировать собственные исключения. Например, при получении внутри метода параметра null вместо ожидаемого значения имеет смысл использовать исключение, которое вызывает в такой ситуации .NET:

```
throw new ArgumentException();
```

Но иногда исключение требуется из-за особых обстоятельств, возникающих в процессе работы программы. Например, количество меда, потребляемого пчелой, зависит от ее веса. Но для ситуации, когда меда в улье не осталось, имеет смысл создать собственное исключение. Для этого потребуется класс, наследующий от класса `Exception`.

Exception
Message StackTrace
GetBaseException() ToString()

your Exception
Message StackTrace
GetBaseException() ToString()

```
class OutOfHoneyException : System.Exception {
    public OutOfHoneyException(string message) : base(message) { }
}
class HoneyDeliverySystem {
    ...
    public void FeedHoneyToEggs() {
       if (honeyLevel == 0) {
            throw new OutOfHoneyException("The hive is out of honey.");
       } else {
            foreach (Egg egg in Eggs) {
               ...
}
public partial class Form1 : Form {
...
   private void consumeHoney_Click(object sender, EventArgs e) {
      HoneyDeliverySystem delivery = new HoneyDeliverySystem();
      try {
           delivery.FeedHoneyToEggs()
      }
      catch (OutOfHoneyException ex){
          MessageBox.Show(ex.Message, "Warning: Resetting Hive");
          Hive.Reset();
      }
   }
}
```

Для вашего исключения потребуется класс, который должен быть производным по отношению к классу System.Exception. Обратите внимание, каким образом перегружается конструктор, чтобы передать сообщение об исключении.

Вызывается новый экземпляр объекта exception.

При наличии в улье меда исключение не появляется и управление передается этому коду.

Имя пользовательского исключения указывается в блоке catch, а дальше, как обычно, выполняются все операции для его обработки.

Если в улье отсутствует мед, деятельность пчел останавливается и симулятор прекращает работать. Для возвращения к нормальному функционированию требуется перезагрузка. Именно этот код помещен в блок catch.

The hive is out of honey

OK

```
public static void Main() {
    Console.Write("when it ");
    ExTestDrive.Zero("yes");
    Console.Write(" it ");
    ExTestDrive.Zero("no");
    Console.WriteLine(".");
}

class MyException : Exception { }
```

Исключительные магниты

Расположите магниты с кодом таким образом, чтобы на консоль был выведен следующий результат.

Результат:

when it thaws it throws.

```
}
```

```
}
```

```
}
```

```
if (t == "yes") {
```

```
Console.Write("a");
```

```
Console.Write("o");
```

```
Console.Write("t");
```

```
Console.Write("w");
```

```
Console.Write("s");
```

```
try {
```

```
} catch (MyException) {
```

```
throw new MyException();
```

```
} finally {
```

```
DoRisky(test);
```

```
        Console.Write("r");
    }
}
```

```
class ExTestDrive {
    public static void Zero(string test) {
```

```
static void DoRisky(String t) {
    Console.Write("h");
```

Решение задачи с магнитами

Расположите магниты с кодом таким образом, чтобы на консоль был выведен следующий результат.

Результат:

when it thaws it throws.

```
public static void Main() {
    Console.Write("when it ");
    ExTestDrive.Zero("yes");
    Console.Write(" it ");
    ExTestDrive.Zero("no");
    Console.WriteLine(".");
}

class MyException : Exception { }
```

```
class ExTestDrive {
   public static void Zero(string test) {
    try {
       Console.Write("t");
       DoRisky(test);
       Console.Write("o");
    } catch (MyException) {
       Console.Write("a");
    } finally {
       Console.Write("w");
    }
    Console.Write("s");
   }

   static void DoRisky(String t) {
     Console.Write("h");
     if (t == "yes") {
       throw new MyException();
     }
     Console.Write("r");
   }
}
```

Эта строчка определяет пользовательское исключение MyException, которое обрабатывается в блоке catch.

В зависимости от того, какой параметр test был передан методу Zero() (строка yes или что-то другое), выводится строка thaws или throws.

Блок finally гарантирует, что метод всегда выводит символ w. А так как символ s выводится вне обработчика исключений, он также всегда попадает в список вывода.

Эта строчка выполняется только в случае, когда метод doRisky() не становится причиной появления исключения.

Метод doRisky() вызывает исключение при передаче ему строки yes.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Причиной исключения может стать любой оператор.
- Для обработки исключений пользуйтесь блоком `try/catch`. Необработанные исключения приводят к прекращению работы программы.
- Обнаружение исключения в блоке `try` приводит к немедленной передаче управления первому оператору блока `catch`.
- Объект `Exception` содержит информацию об исключении. Объявив переменную `Exception` в операторе `catch`, вы получаете доступ к информации об исключении, появившемся в блоке `try`:

```
try {
  // операторы, которые могут
  // вызвать исключение
} catch (IOException ex) {
  // информация об исключении
  // содержится в переменной ex
}
```

- Существуют различные типы исключений. Каждому соответствует объект, унаследованный от класса `Exception`. Старайтесь избегать обнаружения исключений «вообще», работайте с исключениями определенного типа.
- Каждому оператору `try` может соответствовать несколько операторов `catch`:

```
try { ... }
catch (NullReferenceException ex) {
   // эти операторы срабатывают при
   // NullReferenceException
}
catch (OverflowException ex) { ... }
catch (FileNotFoundException) { ... }
catch (ArgumentException) { ... }
```

- Для сообщения об аномальных ситуациях используется оператор `throw`:

```
throw new Exception("Сообщение");
```

- Оператор `throw` позволяет повторно вызвать перехваченное исключение, но только внутри блока `catch`.
- Наследованием от класса `Exception` можно создать пользовательское исключение.

```
class CustomException : Exception;
```

- В большинстве случаев достаточно встроенных исключений .NET. Прибегая к различным типам исключений, вы **предоставляете пользователю больше информации**.

Оператор using как комбинация операторов try и finally

Объявляя ссылку внутри оператора using, вы автоматически вызываете в конце блока операторов метод Dispose().

Вы уже знаете, что оператор `using` гарантирует закрытие ваших файлов. Иногда он может использоваться и как **быстрый вызов** для операторов `try` и `finally`!

```
using (YourClass c
        = new YourClass() ) {

   // код

}
```

аналогично →

```
YourClass c = new YourClass();

try {

    // код

} finally {

    c.Dispose();
}
```

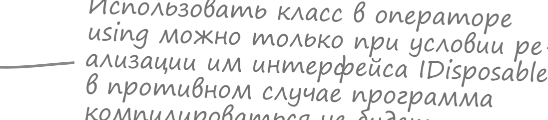
При вызове метода Dispose() в блоке finally можно использовать сокращенную запись с оператором using.

Интерфейс IDisposable позволяет избежать распространенных исключений. Используйте оператор using при работе с реализующими этот интерфейс классами.

Избегаем исключений при помощи интерфейса IDisposable

Потоки снабжены кодом для их закрытия после удаления объекта. Но что делать с пользовательским объектом, после удаления которого требуется произвести некие действия? Имеет смысл написать свой код для случая, когда объект используется внутри оператора `using`.

В C# это можно сделать при помощи интерфейса `IDisposable`. Реализуйте его и вставьте освобождающий ресурсы код в метод `Dispose()`, как показано ниже:




```
class Nectar : IDisposable {
  private double amount;
  private BeeHive hive;
  private Stream hiveLog;
  public Nectar(double amount, BeeHive hive, Stream hiveLog) {
    this.amount = amount;
    this.hive = hive;
    this.hiveLog = hiveLog;
  }
  public void Dispose() {
    if (amount > 0) {
       hive.Add(amount);
       hive.WriteLog(hiveLog, amount + " mg nectar added to the hive");
       amount = 0;
    }
  }
}
```

Объект, который предполагается использовать вместе с оператором using, должен реализовывать интерфейс IDisposable.

Единственным членом интерфейса IDisposable является метод Dispose(). Все помещенное внутрь метода будет выполнено в конце оператора using.

Метод Dispose() уже написан, так что он может быть вызван произвольное количество раз.

Этот код выливает в улей весь оставшийся нектар и пишет сообщение. Так как это действие непременно должно быть осуществлено, мы поместили его в метод Dispose().

В руководстве по интерфейсу `IDispose` сказано, что метод `Dispose()` можно вызывать много раз. Вы понимаете важность этого обстоятельства?

Теперь можно несколько раз воспользоваться оператором `using`. Возьмем встроенный объект `Stream`, реализующий `IDisposable`, как и наш обновленный объект `Nectar`:

Вложенные операторы using используются при необходимости объявить две ссылки на интерфейс IDisposable в одном блоке кода.

```
using (Stream log = new File.Write("log.txt"))
using (Nectar nectar = new Nectar(16.3, hive, log)) {
      Bee.FlyTo(flower);
      Bee.Harvest(nectar);
      Bee.FlyTo(hive);
}
```

Объект Nectar использует поток log, автоматически закрывающийся в конце внешнего оператора using.

Затем объект Bee использует объект Nectar, автоматически добавляющий нектар в улей в конце внутреннего оператора using.

Часто Задаваемые Вопросы

В: Можно ли использовать объект с оператором using, если он не реализует интерфейс IDisposable?

О: Нет, с операторами using можно создавать только объекты, реализующие IDisposable, так как они подогнаны друг под друга. Добавление оператора using эквивалентно созданию экземпляра класса, просто в этом случае в конце блока всегда вызывается метод Dispose(). Именно поэтому класс **обязан реализовывать** интерфейс IDisposable.

В: Любой ли оператор может быть помещен в блок using?

О: Разумеется. Оператор using всего лишь гарантирует уничтожение любого созданного вами объекта. Но использовать эти объекты вы можете на свое усмотрение. Можно даже создать объект при помощи оператора using и не упоминать его внутри блока. Впрочем, это не имеет практического смысла.

В: Может ли метод быть вызван Dispose() вне оператора using?

О: Конечно. На самом деле в этом случае оператор using вообще не нужен. Вызывайте метод Dispose(), завершив работу с объектом. Он высвободит указанные ресурсы, как и при вызове вручную метода Close() для потока. Оператор using всего лишь облегчает чтение и понимание кода и предотвращает проблемы, которые могут возникнуть, если не удалить объект.

В: Вы упомянули блок try/finally. Означает ли это, что операторы try и finally могут фигурировать без оператора catch?

О: Да! Вы можете скомбинировать блок try непосредственно с блоком finally, как показано здесь:

```
try {
     DoSomethingRisky();
     SomethingElseRisky();
}
finally {
     AlwaysExecuteThis();
}
```

При обнаружении исключения в методе DoSomethingRisky() немедленно будет запущен блок finally.

В: Правда ли, что метод Dispose() работает только с файлами и потоками?

О: Нет, многие классы реализуют интерфейс IDisposable, и при работе с ними всегда нужно использовать оператор using. (С некоторыми из этих классов вы познакомитесь в следующей главе.) Если вы пишете класс, который нужно утилизировать определенным способом, также реализуйте интерфейс IDisposable.

Если блоки try/catch — это такой полезный инструмент, почему он не используется по умолчанию, а нам приходится вводить код вручную?

Сначала нужно определить тип появляющегося исключения, чтобы правильно его обработать.

Ведь обработка исключений не сводится к выводу стандартного сообщения об ошибке. Скажем, в программе для поиска оправданий, зная о появлении исключения FileNotFoundException, можно вывести сообщение с советом, где искать нужные файлы. А в случае исключений, связанных с базами данных, можно по электронной почте отправить сообщение администратору. Так что все ваши действия зависят от типа обнаруженного исключения.

Именно поэтому так много классов наследуют от класса Exception. Порой вам даже приходится писать такие классы самостоятельно.

Наихудший вариант блока catch

Блок catch позволяет программе продолжить работу. Появившееся исключение обрабатывается, и вместо аварийной остановки и сообщения об ошибке вы двигаетесь дальше. Но это не всегда хорошо.

Рассмотрим странно работающий класс Calculator. Что же происходит?

```
class Calculator {
...
  public void Divide(int dividend, int divisor) {
    try {
      this.quotient = dividend / divisor;
    } catch {
      // Примечание Джима: нужно понять, как предотвратить ввод
      // пользователями нулевого значения в делитель.
    }
  }
}
```

Если делитель равен нулю, появляется исключение DivdeByZeroException.

Почему, несмотря на наличие блока catch, мы получаем сообщение об ошибке?

Программист подумал, что сможет **скрыть** исключения при помощи пустого блока catch, но всего лишь создал проблему пользователям программы.

Исключения должны обрабатываться, а не скрываться

Тот факт, что программа продолжает работу, не означает *обработки* исключений. Написанный 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? При равенстве делителя нулю метод, скорее всего, вернет неправильное (и неожиданное) значение.

Нужно не просто добавить комментарий, а **обработать исключение**. Даже если вы не знаете, что делать, ***не оставляйте блок catch пустым или закомментированным!*** Это лишь усложняет пользование программой. Лучше пусть появится сообщение об исключении, — это хотя бы позволяет понять, что именно работает не так.

Помните, что если исключение не обрабатывается, оно поднимается вверх в стеке вызовов. Это тоже своего рода обработка. Предоставить исключению возможность всплывать наверх — более разумное в некоторых случаях действие, чем использование блока try/catch.

Временные решения

...к сожалению, в реальной жизни «временные» решения зачастую становятся постоянными.

Иногда, столкнувшись с проблемой, вы не знаете, что делать. В этом случае имеет смысл внести запись в журнал, снабдив ее примечанием. Это не так хорошо, как обработка исключения, но лучше, чем ничего.

Вот временное решение для калькулятора:

Остановите 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?

```
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());
      };
    }
  }
}
```

Проблема никуда не исчезла, но, по крайней мере, стало ясно, где она возникла. Лучше всего разобраться, почему ваш метод Divide вызывается при нулевом знаменателе, и устранить эту возможность.

Я понял! Мы используем обработку исключений, чтобы пометить проблемную область.

Обработка исключения далеко не всегда означает УСТРАНЕНИЕ исключения.

Возможность аварийной остановки программы — это плохо. Но непонимание причин такого поведения намного хуже. Поэтому всегда нужно обрабатывать ошибки, которые вы можете предсказать, и записывать в журнал информацию об ошибках, с которыми вы не умеете бороться.

Краткие принципы обработки исключений

Красиво оформляйте код обработки ошибок.

Предоставляйте ИНФОРМАТИВНЫЕ сообщения об ошибках.

Старайтесь прибегать к встроенным исключениям .NET, а не создавать собственные.

Думайте о том, как можно сократить код в блоке try.

...и самое главное...

Избегайте ошибок, связанных с файлами... всегда пользуйтесь блоком USING, работая с потоками! ВСЕГДА, ВСЕГДА, ВСЕГДА!

Или **другими элементами**, реализующими интерфейс IDisposable.

Наконец Брайан получил свой отдых...

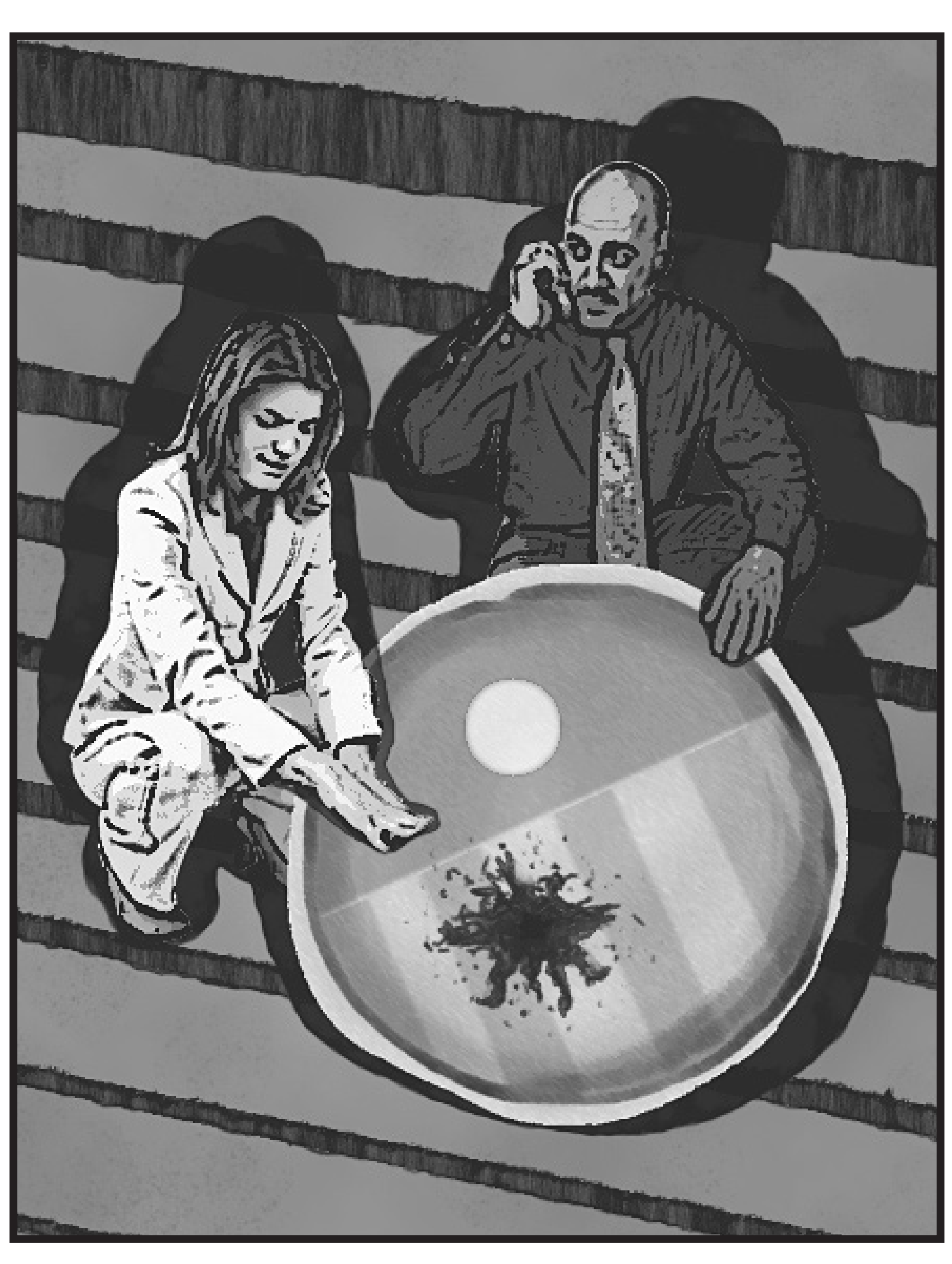
Теперь, после того как Брайан позаботился об исключениях, он получил свой заслуженный (и одобренный шефом!) выходной.

Ваше умение работать с исключениями не просто предотвратило проблему. Оно прежде всего гарантировало, что шеф Брайана не узнает о его прогулах!

Правильная обработка исключений не заметна пользователям. Программа не останавливает работу, а проблемы обрабатываются аккуратно, без сообщений об ошибках.

КАПИТАН ВЕЛИКОЛЕПНЫЙ
СМЕРТЬ ОБЪЕКТА

Head First	
$4	глава 13



Самый великолепный капитан Объектвиля преследует своего врага...

ВОТ ТЫ ГДЕ, ЖУЛИК!

СЛИШКОМ ПОЗДНО! МОИ КЛОНЫ ЗАХВАТИЛИ ФАБРИКУ...

...ЧТОБЫ НАНЕСТИ УЩЕРБ ОБЪЕКТВИЛЮ!

Welcome to
Objectville
Home of Polymorphism

POW!!

Я УНИЧТОЖУ ССЫЛКИ НА КЛОНЫ ОДНУ ЗА ОДНОЙ.

SELF DESTRUCT

4:00

3:59

3:58

BOOM!!

Погиб ли Капитан Великолепный?..

Возьми в руку карандаш

Вот код, описывающий битву между Великолепным и Жуликом (а также с армией клонов). Нарисуйте, что происходит в куче при создании экземпляров класса `FinalBattle`.

```
class FinalBattle {
  public CloneFactory Factory = new CloneFactory();
  public List<Clone> Clones = new List<Clone>() { ... };
  public SwindlersEscapePlane escapePlane;

  public FinalBattle() {
    Villain swindler = new Villain(this);
    using (Superhero captainAmazing = new Superhero()) {
      Factory.PeopleInFactory.Add(captainAmazing);
      Factory.PeopleInFactory.Add(swindler); ❶
      captainAmazing.Think("Я уничтожу ссылки на клоны,
                              одну за одной");
      captainAmazing.IdentifyTheClones(Clones);
      captainAmazing.RemoveTheClones(Clones);
      swindler.Think("Через несколько минут моя армия станет мусором");
      swindler.Think("(будет собрана!)");
      escapePlane = new SwindlersEscapePlane(swindler); ❷
      swindler.TrapCaptainAmazing(Factory);
      new MessageDialog("Жулик убежал.").ShowAsync();
    }
  }
} ❸
[Serializable]
class Superhero : IDisposable {
      private List<Clone> clonesToRemove = new List<Clone>();
      public void IdentifyTheClones(List<Clone> clones) {
          foreach (Clone clone in clones)
              clonesToRemove.Add(clone);
      }
      public void RemoveTheClones(List<Clone> clones) {
          foreach (Clone clone in clonesToRemove)
              clones.Remove(clone);
       ...
      }
      ...
}
class Villain {
  private FinalBattle finalBattle;
  public Villain(FinalBattle finalBattle) {
    this.finalBattle = finalBattle;
  }
  public void TrapCaptainAmazing(CloneFactory factory) {
    factory.SelfDestruct.Tick += new EventHandler(SelfDestruct_Tick);
    factory.SelfDestruct.Interval = TimeSpan.FromSeconds(60);
    factory.SelfDestruct.Start();
  }
  private void SelfDestruct_Tick(object sender, EventArgs e) {
    finalBattle.Factory = null;
  }
}
```

Можно предположить, что клоны были созданы при помощи инициализатора коллекции.

Первый рисунок сделайте для этой точки. Покажите, что же происходит на фабрике.

Нарисуйте, что происходит в момент создания экземпляра объекта SwindlersEscapePlane (План побега Жулика).

Как будет выглядеть куча после запуска конструктора FinalBattle (Финальная битва)?

Это класс Clone, который мы вам не показываем, так как для ответа на вопросы он не нужен.

Здесь должен быть дополнительный код (включающий метод Dispose(), реализующий интерфейс IDisposable). Но он скрыт, так как не имеет значения для решения задачи.

```
class SwindlersEscapePlane {
  public Villain PilotsSeat;
  public SwindlersEscapePlane(Villain escapee) {
    PilotsSeat = escapee;
  }
}

class CloneFactory {
  public DispatcherTimer SelfDestruct = new DispatcherTimer();
  public List<object> PeopleInFactory = new List<object>();
  ...
}
```

Класс Clone также не показывается как несущественный для решения данной задачи.

Не забывайте про метки объектов, указывающих на ссылочные переменные.

Мы начали отвечать на первый вопрос. Линии демонстрируют архитектуру. Скажем, мы провели линию между фабрикой клонов и объектом Villain, так как фабрика ссылается на этот объект (через свое поле PeopleInFactory).

1

factory

CloneFactory

swindler

Объект Villain

Здесь нарисовано не все. Остальные объекты вставьте самостоятельно.

Рисовать объекты Clone и List не нужно. Добавьте только объекты, представляющие Капитана, Жулика, фабрику клонов и план побега Жулика.

2

3

Нарисуйте, что происходит в куче в остальных двух точках.

В каком месте кода умирает Капитан Великолепный?

....

....

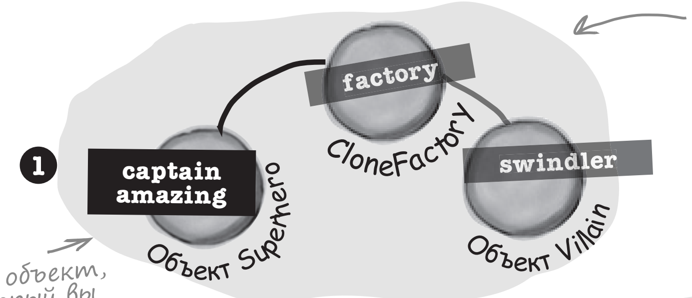
....

....

Оставьте соответствующий комментарий на диаграмме.

Возьми в руку карандаш
Решение

Вот как выглядит куча по мере выполнения программы `FinalBattle`.



Ссылка captainAmazing указывает на объект Superhero, а ссылка swindler — на объект Villain. Список же PeopleInFactory нашей фабрики клонов содержит ссылки на оба этих объекта.

Это объект, который вы должны были добавить к диаграмме.

2

factory

CloneFactory

captain amazing

Объект Superhero

swindler

Объект Villain

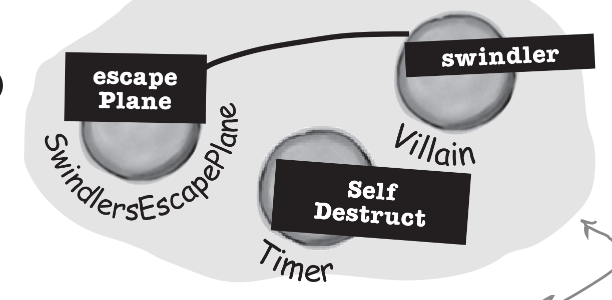
escape Plane

SwindlersEscapePlane

Теперь ссылка escapePlane указывает на новый экземпляр объекта SwindlersEscapePlane, а поле PilotSeat — на объект Villain.

Пока существует ссылка escapePlane на объект swindler, этот объект не будет отправлен в мусор.

При появлении события selfDestruct ссылка factory начинает указывать на значение null, а значит, отправляется в мусор и исчезает с нашей диаграммы.



Вслед за ссылкой factory в мусор отправляется объект CloneFactory, это приводит к исчезновению объекта List... а это было последним, что сохраняло наш объект SuperHero. Он исчезнет при следующем проходе сборщика мусора.

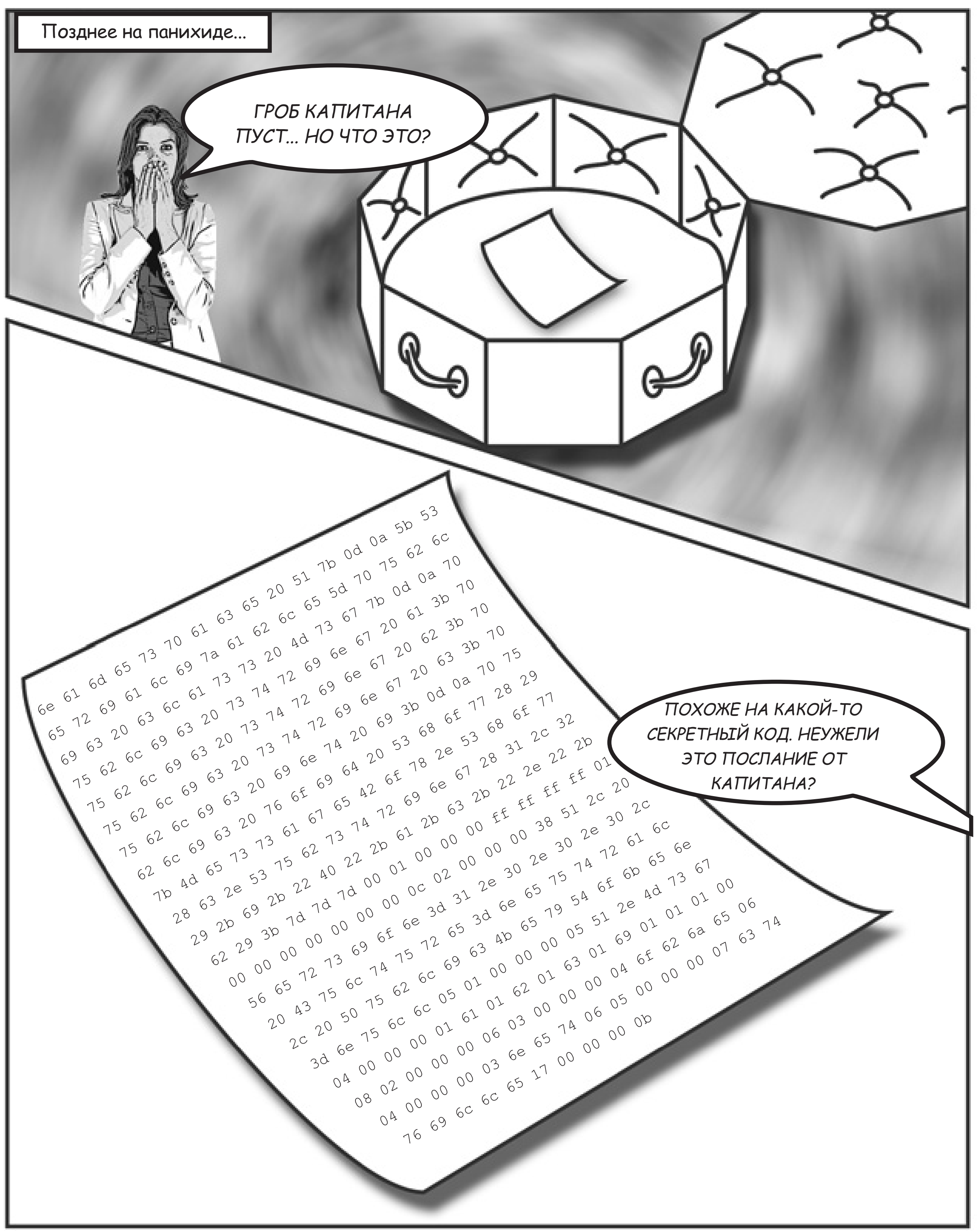
Вот момент исчезновения нашего супергероя:

```
void SelfDestruct_Tick(object sender, EventArgs e) {
  finalBattle.factory = null;
}
```

Герой исчезнет после запуска конструктора FinalBattle.

Ссылка `finalBattleFactory` стала указывать на null, и это привело к исчезновению последней ссылки на Капитана!

Так как экземпляр SuperHero не имеет клонов, на которые мог бы сослаться объект factory, он предназначается для сборщика мусора.


Позднее на панихиде...
ГРОБ КАПИТАНА ПУСТ... НО ЧТО ЭТО?
ПОХОЖЕ НА КАКОЙ-ТО СЕКРЕТНЫЙ КОД. НЕУЖЕЛИ ЭТО ПОСЛАНИЕ ОТ КАПИТАНА?

Метод завершения объекта

Вам не придется писать метод завершения для объектов, обладающих управляемыми ресурсами. Все, с чем вы сталкивались в этой книге, было управляемым — оно управлялось CLR. Но бывают случаи, когда программистам требуется доступ к базовым ресурсам Windows, которые не являются частью .NET Framework. Скажем, код с атрибутом `[DllImport]` может использовать неуправляемые ресурсы. Если их вовремя не удалить (например, соответствующим методом), эти ресурсы могут повлиять на стабильность системы. Именно для такой цели требуется метод завершения объекта.

Иногда нужно, чтобы некие действия, например **высвобождение неуправляемых ресурсов,** произошли ***до того,*** как объект отправится в мусор

Так называемый **метод завершения объекта (finalizer)** позволяет написать код, который будет выполняться перед уничтожением объекта. Его можно представить как персональный блок `finally`: он всегда выполняется последним.

Вот пример такого метода для класса `Clone`:

```
[Serializable]
class Clone {
  string Location;
  int CloneID;

  public Clone (int cloneID, string location){
    this.CloneID = cloneID;
    this.Location = location;
  }

  public void TellLocation(string location, int cloneID){
     Console.WriteLine("Мой номер {0} и " +
                       "ты найдешь меня тут: {1}.", cloneID, location);
  }

  public void WreakHavoc(){...}

  ~Clone() {
    TellLocation(this.Location, this.CloneID);
    Console.WriteLine ("{0} has been destroyed", CloneID);
  }
}
```

Это конструктор. Он заполняет поля CloneID и Location при каждом создании объекта Clone.

Знак ~ указывает, что код в этом блоке выполняется, когда объект отправляется в мусор.

Это метод завершения объекта. Он отправляет негодяю сообщение с ID и координатами несчастного клона. Но только после того, как объект будет помечен для сборки мусора.

Метод завершения объекта отличается от конструктора тем, что вместо модификатора доступа перед именем класса помещается знак ~. И .NET выполнит этот код прямо перед отправкой объекта в мусор.

Этот метод не имеет параметров, так как .NET должна уничтожить объект.

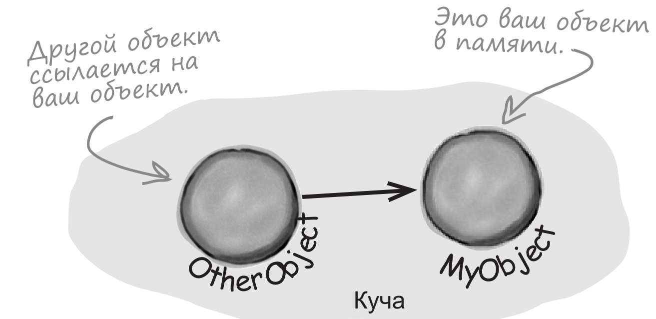
Будьте осторожны!

Часть кода в книге предназначена только для учебных целей.

*Пока вы только слышали, что объект отправляется в мусор, как только исчезает ссылка на него. Мы готовы показать вам код, автоматически запускающий сборку мусора **при помощи метода `GC.Collect()`** и **вызывающий `MessageBox` в методе завершения объектов**. Эти вещи затрагивают «сердце» CLR. Мы показываем их, чтобы объяснить принцип сборки мусора. **Никогда не используйте их в рабочих программах.***

Когда запускается метод завершения объекта

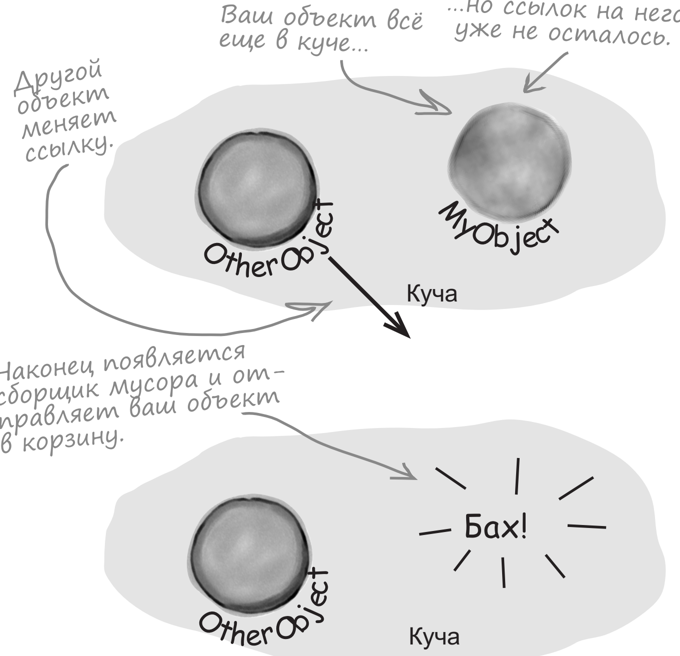
Метод завершения объекта запускается **после** исчезновения всех ссылок, но **до** отправки объекта в мусор. Ведь сборка мусора не всегда запускается *сразу после* исчезновения ссылок.



Предположим, у нас есть объект и ссылка на него. Запущенный .NET сборщик мусора проверяет его состояние. Обнаружив ссылку, сборщик игнорирует этот объект, и он остается в памяти.

Затем последний объект, ссылавшийся на *ваш* объект, удаляется. Доступ к нему пропадает. По сути, объект **умер**.

Но дело в том, что ***сборкой мусора управляет .NET,*** а не объекты. Поэтому до запуска процедуры сборки объект живет в памяти. Его невозможно использовать, но он есть. **И его метод завершения объекта еще не начал свою работу.**



И вот .NET в очередной раз запускает сборщик мусора. Запускается и метод завершения объекта... возможно, через несколько минут после исчезновения последней ссылки. И только после этого объект окончательно исчезает.

Принудительная сборка мусора

.NET позволяет вам ***предложить*** запуск сборки мусора. **По большому счету, пользоваться этой возможностью вряд ли стоит, так как сборка мусора отвечает множеству условий CLR и принудительный ее вызов — *не очень хорошая идея.*** Но чтобы посмотреть на работу метода завершения объектов, вы можете воспользоваться методом `GC.Collect()`.

Будьте внимательны. Этот метод ***не заставляет*** .NET немедленно приступить к сборке мусора. Он только говорит: «Выполните эту процедуру как можно быстрее».

```
public void RemoveTheClones(
                List<Clone> clones) {
  foreach (Clone clone in clonesToRemove)
     clones.Remove(clone);
  GC.Collect();
}
```

Мы не можем не подчеркнуть еще раз, насколько плохой идеей является вызов метода GC.Collect() в серьезных программах, так как вы можете сбить настроенную процедуру сборки мусора. Но для **обучения** нет ничего лучше, поэтому мы создадим игровую программу, в которой воспользуемся этим методом.

Явное и неявное высвобождение ресурсов

Как вы уже видели, метод Dispose() работает и без оператора using. Его многократное применение не имеет побочных эффектов.

Метод `Dispose()` запускается, когда объект, созданный при помощи оператора `using`, получает значение null. Если оператор `using` не применялся, ссылка на значение null не станет причиной вызова метода `Dispose()` — вам придется вызывать его вручную. Метод завершения объекта работает со сборщиком мусора. Посмотрим на практике, чем эти методы отличаются друг от друга. Запустите **Visual Studio** для рабочего стола Windows и создайте приложение **Windows Forms**.

1 Создайте класс `Clone`, реализующий интерфейс `IDisposable`.
Класс должен иметь автоматическое свойство `Id` типа `int`, а также конструктор, метод `Dispose()` и метод завершения объекта:

> **Вот хороший пример изучения C# и .NET на основе приложения для рабочего стола. Вы снова создадите проект Windows Forms и исследуете работу сборщика мусора при помощи окна `MessageBox`.**

Памятка: пользователи Visual Studio Professional, Premium и Ultimate могут создавать приложения для магазина Windows и для рабочего стола.

```
using System.Windows.Forms;

class Clone : IDisposable {
    public int Id { get; private set; }

    public Clone(int Id) {
        this.Id = Id;
    }

    public void Dispose() {
        MessageBox.Show("I've been disposed!",
                     "Clone #" + Id + " says...");
    }

    ~Clone() {
        MessageBox.Show("Aaargh! You got me!",
                     "Clone #" + Id + " says...");
    }
}
```

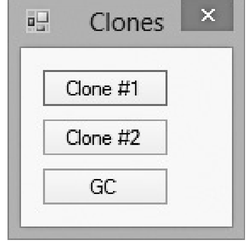
Так как класс реализует интерфейс IDisposable, в нем должен присутствовать метод Dispose().

Напоминаем: вызов окна MessageBox в методе завершения объекта может внести путаницу в работу CLR. Применять его в целях, отличных от учебных, не следует.

Это метод завершения объекта. Он запускается перед отправкой объекта в сборку мусора.

Создайте форму с тремя кнопками
При помощи оператора `using` создайте экземпляр `Clone` в обработчике события `Click`. Это первая часть кода кнопки:

Вот так выглядит ваша форма.




```
private void clone1_Click(object sender, EventArgs e) {
    using (Clone clone1 = new Clone(1)) {
        // Ничего не делайте!
    }
}
```

Метод создает экземпляр Clone и немедленно убивает его, убирая все ссылки.

Так как clone1 был объявлен при помощи оператора using, запускается его метод Dispose().

После завершения работы блока using вызывается метод Dispose() объекта Clone. Ссылок на объект не остается, и он помечается для сборки мусора.

3 **Подключите две другие кнопки**

Создайте еще один экземпляр `Clone` в обработчике события `Click` второй кнопки и присвойте ему значение `null`:

```
private void clone2_Click(object sender, EventArgs e) {
    Clone clone2 = new Clone(2);
    clone2 = null;
}
```

Так как в данном случае отсутствует оператор using, метод Dispose() запущен не будет. Но сработает метод завершения объекта.

Для третьей кнопки вызовите метод `GC.Collect()`, чтобы запустить процедуру сборки мусора.

```
private void gc_Click(object sender, EventArgs e) {
    GC.Collect();
}
```

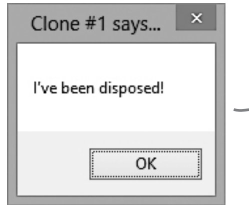
Эта строка принудительно запускает сбор мусора.

Помните, что подобного делать не следует. В данном случае мы делаем это исключительно для учебных и демонстрационных целей.

4 **Запустите программу**

Щелкните на первой кнопке, чтобы вызвать метод `Dispose()`.

Не забудьте добавить в верхнюю часть класса Clone строчку «using System.Windows.Forms;».



Хотя объекту Clone1 было присвоено значение null и был вызван метод Dispose, он все равно присутствует в куче и ожидает сборщика мусора.

clone1
Объект Clone
Куча

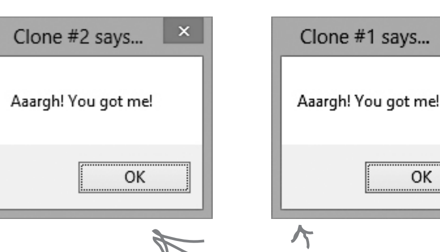
Мусор ***в итоге*** собран. В большинстве случаев вы ***не*** увидите окна с сообщением об этом, так как между присвоением объекту значения null и сборкой мусора проходит некоторое время.

Щелкните на второй кнопке. Ничего не произойдет, так как мы не использовали оператор `using`. Окно диалога с сообщением от метода завершения объекта появится только после сборки мусора.

Щелкните на третьей кнопке, чтобы принудительно запустить эту процедуру. Появятся два окна диалога: для объектов `Clone1` и `Clone2`.

clone1
Объект Clone
clone2
Объект Clone
Куча

Объект Clone2 также пока остается в куче, но ссылок на него уже нет.



Вызов метода GC.Collect() запускает метод окончания для обоих объектов, и они исчезают из кучи.

Поиграйте с программой. Несколько раз по очереди пощелкайте на кнопках. Иногда первым будет исчезать клон #1, в других случаях — клон #2. А иногда сборка мусора будет запускаться еще до вызова метода `GC.Collect()`.

Возможные проблемы

Вы не можете зависеть от запуска метода завершения объекта в произвольный момент. Даже вызов метода `GC.Collect()`, которым не стоит пользоваться, если у вас нет веских на то причин, только ***предлагает*** запустить сборщик мусора. И нет гарантии, что это случится немедленно. Более того, узнать, в каком порядке будут удаляться объекты, невозможно.

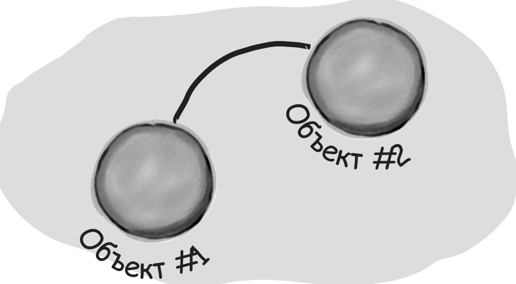
Что это означает на практике? Представим, что у вас есть ссылающиеся друг на друга объекты. Если объект #1 будет удален первым, ссылка объекта #2 начнет указывать в никуда. И наоборот. Другими словами, ***вы не можете полагаться на ссылки в методе завершения объекта***. То есть помещать в метод завершения объекта операции, зависящие от ссылок, явно не стоит.

Хорошим примером процедуры, которая **ни в коем случае не должна оказаться внутри метода завершения объекта**, является сериализация. Если объект ссылается на множество других объектов, сериализации подвергается ***вся*** цепочка. Но если сборка мусора уже произошла, вы можете **недосчитаться** важных частей программы, так как некоторые объекты могут быть отправлены в мусор ***до*** запуска метода их завершения.

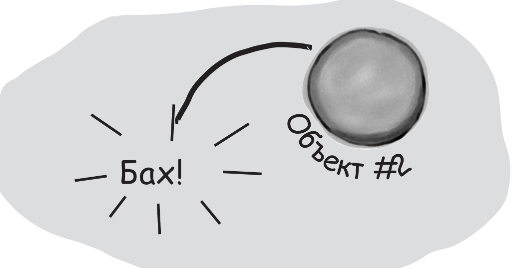
К счастью, C# предлагает удачное решение данной проблемы: интерфейс `IDisposable`. Все редактирования ключевых данных или данных, зависящих от находящихся в памяти объектов, нужно делать частью метода `Dispose()`.

Иногда пользователи считают метод завершения объекта более надежным вариантом метода `Dispose()`, и не без оснований: на примере объектов `Clone` вы уже видели, что реализация интерфейса `IDisposable` не означает вызова метода `Dispose()`. При этом если метод `Dispose()` зависит от объектов, находящихся в куче, его вызов в методе завершения объекта может привести к проблемам. Лучше всего **всегда использовать оператор `using`** для создания объектов, реализующих интерфейс `IDisposable`.

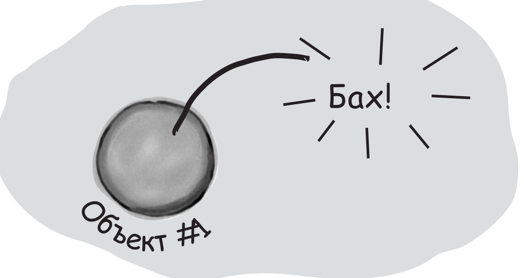
Предположим, у вас есть два объекта, ссылающиеся друг на друга...



...они оба помечены для сборки мусора, но объект #1 удаляется первым...



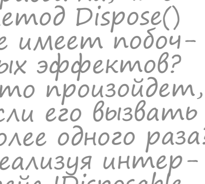
...хотя первым мог быть удален и объект #2. Порядок выполнения этой процедуры узнать невозможно...



...именно поэтому метод завершения одного объекта не может полагаться на объекты, до сих пор присутствующие в куче.

Сериализуем объект в методе Dispose()

Теперь, когда вы поняли разницу между методом Dispose() и методом завершения объекта, напишем объект, автоматически сериализующий себя перед удалением.



Сделаем сериализуемым класс Clone (со с. 610)

Просто добавьте в верхнюю часть класса атрибут Serializeable.

```
[Serializable]
class Clone : IDisposable
```

Отредактируем метод Dispose()

Воспользуемся классом BinaryFormatter для записи объекта Clone в файл внутри метода Dispose():

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

// существующий код

public void Dispose() {
  string filename = @"C:\Temp\Clone.dat";
  string dirname = @"C:\Temp\";
  if (File.Exists(filename) == false) {
    Directory.CreateDirectory(dirname);
  }
  BinaryFormatter bf = new BinaryFormatter();
  using (Stream output = File.OpenWrite(filename)) {
    bf.Serialize(output, this);
  }
  MessageBox.Show("Должен... сериализовать... объект!",
                  "Clone #" + Id + " говорит...");
}
```

Для доступа к используемым нами классам I/O вам потребуются несколько директив using.

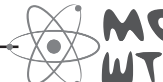
Мы вернулись к двоичной сериализации и встроенным папкам, так как это очень просто, и так как мы не хотим, чтобы вы пользовались подобными приемами в рабочем коде! Это приемлемо только в учебных программах.

Объект Clone создаст папку C:\Temp и сериализуется в файл Clone.dat.

Имя файла было включено в код в виде строковой константы. Для учебной программы это нормально, но может сопровождаться проблемами. Понимаете ли вы, какие это проблемы и видите ли пути их решения?

Запустите приложение

Вы увидите ровно то же самое, что и несколькими страницами ранее... но теперь перед удалением объект clone1 будет сохранен в файл. Откройте этот файл, и вы увидите двоичное представление объекта.

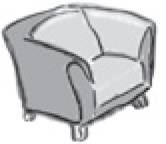
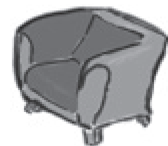


Как выглядит полный код объекта SuperHero? Часть его показана на с. 604. Вы можете написать остальное? Нужно ли писать этот код?

Можно заставить объект сериализовать себя после отправки в мусорную корзину. Но стоит ли? Не нарушает ли это принцип разделения ответственности? Не приведет ли это к сложностям в управлении кодом? Какие еще проблемы могут возникнуть?

Правда ли, что метод Dispose() не имеет побочных эффектов? Что произойдет, если его вызвать более одного раза? Реализуя интерфейс IDisposable, имеет смысл заранее думать о таких вещах.

Беседа у камина

Метод Dispose() и метод завершения объекта спорят о том, кто из них ценнее.

Dispose() | **Метод завершения объекта**

Dispose(): Честно говоря, приглашение сюда меня удивило. Я думал, программисты уже пришли к соглашению о том, что я более ценный. Ты выглядишь жалко. Ты не в состоянии сериализовать себя или отредактировать ключевые данные. Ты же нестабилен, разве не так?

Метод завершения объекта: Потрясающе! Я жалок... хорошо. Я не хотел переходить на личности, но после такого выпада... мне, по крайней мере, не требуется интерфейс для работы. А ты без интерфейса `IDisposable` — не более чем еще один бесполезный метод.

Dispose(): Интерфейс существует **именно потому**, что я очень важен. Более того, я единственный метод этого интерфейса!

Метод завершения объекта: Конечно, конечно... продолжай утешать себя. А что произойдет, если пользователь при создании экземпляра забудет оператор `using`? Тебя даже никто не найдет!

Dispose(): Да, если программисты не используют оператор `using`, они должны вызвать меня вручную. Но они всегда знают, когда я запускаюсь, и могут вызывать меня, когда необходимо удалить объект. Я мощный, надежный и легкий в применении. Я универсален. А ты? Никто не знает, когда ты появишься, и в каком состоянии в этот момент будет приложение.

Обработчики используются программами для непосредственного взаимодействия с Windows. Так как .NET о них не знает, она не может удалить их за вас.

Метод завершения объекта: Хорошо. Но если нужно сделать что-то в самый последний момент перед отправкой объекта на удаление, без меня не обойтись. Я освобождаю сетевые ресурсы, а также обработчики и потоки Windows и все, что может стать причиной проблем, если его вовремя не удалить. Я гарантирую аккуратное удаление объектов, и на это ты ничего возразить не сможешь.

Dispose(): По сути ты не делаешь ничего, чего не мог бы сделать я. Но ты считаешь себя важным только потому, что запускаешься при сборке мусора.

Метод завершения объекта: Именно так, приятель. Я запускаюсь всегда, а тебе для запуска требуется импульс со стороны. Мне же никто и ничто не нужно!

Часто задаваемые вопросы

В: Может ли метод завершения пользоваться полями и методами объектов?

О: Конечно, вы не можете передавать ему параметры, но можете пользоваться полями объектов как напрямую, так и при помощи ключевого слова `this`. Но будьте аккуратны в случаях, когда поля ссылаются на другие объекты. Можно также вызывать другие методы для утилизируемых объектов.

В: Что происходит с исключениями, оказавшимися в методе завершения объекта?

О: Ничто не мешает поместить в этот метод блок `try/catch`. Создайте исключение «деление на ноль» в блоке `try` написанной нами программы для клонов. Пусть окно с сообщением «I just caught an exception» появляется перед сообщением «…I've been destroyed». Запустите программу и щелкните на первой и третьей кнопках. По очереди появятся оба окна диалога. (Разумеется, вызывать окна диалога в методе завершения объектов ни в коем случае не нужно.)

В: Как часто происходит автоматическая сборка мусора?

О: На этот вопрос нет ответа. Не существует постоянного цикла, и вы никак не можете контролировать этот процесс. Он гарантированно запускается при выходе из программы или после вызова метода `GC.Collect()`.

В: Как быстро начинается сбор мусора после вызова метода `GC.Collect()`?

О: Метод `GC.Collect()` просит .NET осуществить сбор мусора как можно быстрее. ***Обычно*** .NET приступает к этой процедуре после завершения текущих заданий. То есть мусор убирается довольно оперативно, но вы не можете контролировать этот процесс.

В: Если мне обязательно нужно что-то запустить, имеет ли смысл поместить это в метод завершения объекта?

О: Код этого метода может и не быть запущен. Но в общем случае да, метод завершения объекта обязательно запускается. Если вы не освобождаете неуправляемые ресурсы, лучше воспользоваться интерфейсом `IDisposable` и операторами `using`.

Тем временем на улицах Объектвиля...

ВЕЛИКОЛЕПНЫЙ... ОН ВЕРНУЛСЯ!

НО ЧТО-ТО НЕ ТАК. ОН НА СЕБЯ НЕ ПОХОЖ... И ЧТО С ЕГО СИЛОЙ?

ВЕЛИКОЛЕПНЫЙ ДОБИРАЛСЯ ТАК ДОЛГО, ЧТО ПУШИСТОГО УСПЕЛИ СНЯТЬ С ДЕРЕВА...

МЯУ!

Позднее...

Еще позднее...

Captain Amazing's Hideout Collection TOP SECRET

БАХ... БАХ... ОХ! Я ИСТОЩЕН!

Что произошло? Куда исчезла сила Капитана? Неужели это конец?

Структура напоминает объект...

Одним из типов .NET, о которых мы еще не упоминали, являются **структуры (structure)**. Они, как и объекты, обладают полями и свойствами. Их можно даже передать методу, принимающему параметры типа `object`:

```
public struct AlmostSuperhero : IDisposable {
    public int SuperStrength;
    public int SuperSpeed { get; private set; }

    public void RemoveVillain(Villain villain)
    {
        Console.WriteLine("OK, " + villain.Name +
             " сдавайся и прекращай безумие!");
        if (villain.Surrendered)
            villain.GoToJail();
        else
            villain.Kill();
    }

    public void Dispose() { ... }
}
```

Структуры могут реализовывать интерфейсы, но не могут быть унаследованы.

Структуры могут обладать свойствами и полями...

...и определять методы.

...но объектом не является

Структуры могут иметь поля и методы, но для них невозможен метод завершения объекта. Наследования для них также невозможно.

Все структуры являются производными от класса System.ValueType, который в свою очередь наследует от класса System.Object. Поэтому каждая структура обладает методом ToString(). Но этим способность к наследованию у структур и исчерпывается.

struct

Структуры не могут наследовать от объектов.

Вместо отдельных объектов можно использовать структуры, но они не в состоянии участвовать в построении иерархий.

Именно поэтому вы чаще пользуетесь классами, а не структурами. Но это не значит, что структуры не имеют применения.

Достоинством объектов является их умение имитировать поведение реальных сущностей при помощи наследования и полиморфизма.

Структуры же лучше всего использовать для хранения данных, несмотря на их ограниченность.

Но основным отличием структур от классов является тот факт, что ***структуры являются типами значений, в то время как классы относятся к ссылочным типам.*** Рассмотрим на примере, что это означает...

Значения копируются, а ссылки присваиваются

Вы уже знаете, чем один тип отличается от другого. С одной стороны, имеются **значимые типы**, такие как `int`, `bool` и `decimal`, с другой —**объекты**, такие как `List`, `Stream` и `Exception`. И они работают по-разному.

Вспомним, чем отличаются значимые типы от объектов.

В случае значимых типов оператор присваивания **копирует значение**. Переменные при этом никак не связаны друг с другом. В случае же ссылок оператор присваивания **нацеливает обе ссылки на один и тот же объект**.

Объявление переменных и операция присваивания одинаковы для обоих типов:

Помните, мы говорили, что методы и операторы ВСЕГДА находятся внутри классов? Это не совсем так, ведь они могут находиться и внутри структур.

int и bool — это значимые типы, в то время как List и Exception принадлежат типу object.

```
int howMany = 25;
bool Scary = true;
List<double> temperatures = new List<double>();
Exception ex = new Exception("Does not compute");
```

Присвоение начальных значений осуществляется стандартным способом.

Различия начинаются при присвоении значений. Для значимых типов эта операция осуществляется методом копирования:

Значение переменной fifteenMore копируется в переменную howMany, затем к нему прибавляется 15.

Изменение значения переменной fifteenMore никак не влияет на переменную howMany.

```
int fifteenMore = howMany;
fifteenMore += 15;
Console.WriteLine("howMany has {0}, fifteenMore has {1}",
                  howMany, fifteenMore);
```

Результат демонстрирует, что переменные `fifteenMore` и `howMany` ***никак не*** связаны:

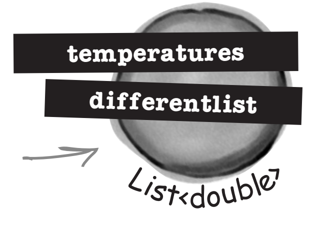
```
howMany has 25, fifteenMore has 40
```

В случае объектов присваиваются ссылки, а не значения:

```
temperatures.Add(56.5D);
temperatures.Add(27.4D);
List<double> differentList = temperatures;
differentList.Add(62.9D);
```

В этой строчке ссылка differentList начинает указывать на тот же объект, что и ссылка temperatures.

Обе ссылки указывают на один объект.



Изменение объекта `List` меняет значения обеих ссылок:

```
Console.WriteLine("temperatures has {0}, differentlist has {1}",
                  temperatures.Count(), differentList.Count());
```

Результат демонстрирует, что обе ссылки нацелены на ***один и тот же*** объект:

```
temperatures has 3, differentList has 3
```

Метод differentList.Add() добавляет новую температуру к объекту, на который указывают ссылки differentList и temperatures.

Структуры — это значимые, а объекты — ссылочные типы

Создавая структуру, вы создаете **значимый тип**. Это означает, что операция присваивания представляет собой *копирование* структуры в новую переменную. Так что хотя структура *выглядит* как объект, таковым она не является.

1 Создайте структуру Dog

Вот простая структура для хранения данных о собаке. Добавьте ее в **новое консольное приложение**.

```
public struct Dog {
  public string Name;
  public string Breed;

  public Dog(string name, string breed) {
    this.Name = name;
    this.Breed = breed;
  }

  public void Speak() {
      Console.WriteLine("Меня зовут {0} и я {1}.", Name, Breed);
  }
}
```

Да, класс не инкапсулирован.

2 Создайте класс Canine

Скопируйте структуру `Dog`, **заменив `struct` на `class`**, а **`Dog`, заменив на `Canine`**. (Не забудьте переименовать и конструктор.) Теперь у вас есть класс `Canine`, практически идентичный структуре `Dog`.

3 Добавьте метод Main(), делающий копии Dog и Canine

Вот код этого метода:

```
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();

Console.ReadKey();
```

Вы уже работали со структурами. Помните **DateTime** из предыдущих глав? Это были структуры!

Возьми в руку карандаш

4 Перед запуском программы...

Запишите результат, который, с вашей точки зрения, будет выведен на консоль:

....

....

Возьми в руку карандаш
Решение

На консоли будет написано:

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 указывали на один и тот же объект, соответственно меняли одни и те же поля и обе имели доступ к методу Speak(). Но структуры работают не так. Создав структуру betty, вы скопировали в нее данные из структуры jake. При этом друг от друга эти структуры не зависят.

Создан объект Canine, на который теперь указывает ссылка spot.

1 spot Spot pug Объект Canine

```
Canine spot = new Canine("Spot", "pug"); 1
Canine bob = spot; 2
bob.Name = "Spike";
bob.Breed = "beagle";
spot.Speak(); 3
```

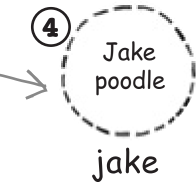
Была создана новая ссылочная переменная bob, но новый объект в куче не появился — переменные bob и spot указывают на один и тот же объект.

2 spot bob Spot pug Объект Canine

Так как переменные spot и bob указывают на один объект, записи spot.Speak() и bob.Speak() вызывают один и тот же метод с одним и тем же результатом — Spike и beagle.

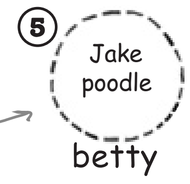
```
Dog jake = new Dog("Jake", "poodle"); 4
Dog betty = jake; 5
betty.Name = "Betty";
betty.Breed = "pit bull";
jake.Speak(); 6
```

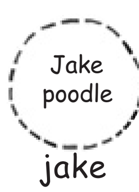
Создание структуры напоминает создание объекта — вы получаете переменную для доступа к полям и методам.



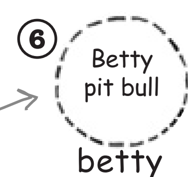
Операция присваивания в случае структур приводит к появлению независимой копии данных. Ведь структура — это ЗНАЧИМЫЙ ТИП.

А вот и отличие. Добавив переменную betty, вы получили новое значение.





Благодаря копированию данных изменение полей структуры betty не влияет на состояние полей структуры jake.



Сравнение стека и кучи

Понять, чем отличается структура от объекта, несложно. Но как операция копирования выглядит изнутри?

.NET CLR помещает данные в разные области памяти. Вы уже знаете, что объекты живут в **куче**. Другая часть памяти — **стек** — хранит все объявляемые вами в методах локальные переменные и передаваемые этим методам параметры. Стек можно представить в виде набора мест, в которые вы помещаете значения. При вызове метода CLR добавляет в стек дополнительные места. После завершения работы они удаляются.

Помните, что в процессе работы вашей программы CLR управляет памятью, выделяя место в куче и собирая мусор.

Несмотря на возможность ***назначить структуру переменной типа object****, структуры и объекты отличаются.*

Код

Это код, который вы можете обнаружить в программе.

Стек

А здесь находятся структуры и локальные переменные.

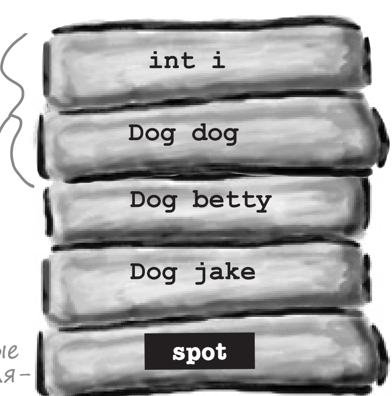
Вот как выглядит стек после выполнения этих двух строк кода.

```
Canine spot = new Canine("Spot", "pug");

Dog jake = new Dog("Jake", "poodle");
```

```
Canine spot = new Canine("Spot", "pug");

Dog jake = new Dog("Jake", "poodle");

Dog betty = jake;
```

При создании новой структуры — или другой переменной значимого типа — в стеке появляется новое место. Сюда копируется указанное вами значение.

```
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 верх стека и удаляет их после завершения работы метода.

А зачем мне все это знать? Ведь я же не могу никак повлиять на эти процессы, разве не так?

И все-таки желательно понимать, как копируемая по значению структура отличается от копируемого по ссылке объекта.

Иногда бывает необходимо написать метод, который работает **или** со значимым типом, **или** со ссылочным типом. Например, метод, работающий со структурой `Dog` или с объектом `Canine`. В этом случае применяется ключевое слово `object`:

```
public void WalkDogOrCanine(object getsWalked) { ... }
```

Переданная этому методу структура **упаковывается** в специальную «оболочку», позволяющую ей находиться в куче. В это время работать со структурой невозможно. Ее требуется сначала «распаковать». К счастью, это происходит *автоматически* при передаче методу вместо объекта значимого типа.

Чтобы определить, структура перед вами или другой значимый тип, упакованный в «оболочку» и помещенный в кучу, используйте ключевое слово is.

1 Вот как стек и куча выглядят после создания переменной типа `object` и присвоения ей структуры `Dog`.

```
Dog sid = new Dog("Sid", "husky");
WalkDogOrCanine(sid);
```

obj

Sid
husky

Dog sid (упакован)

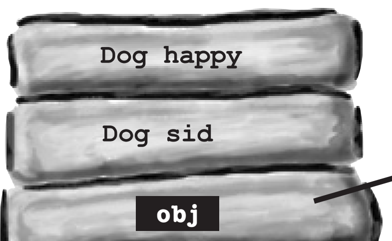
После упаковки структуры появляются две копии данных: в стеке и в куче.

Метод WalkDogOrCanine() принимает ссылку на объект, поэтому структура Dog упаковывается перед передачей. После обратного приведения к структуре Dog происходит распаковка.

2 Объект достаточно привести к правильному типу, и он будет распакован автоматически. **Со значимыми типами использовать ключевое слово as нельзя**, поэтому приведем его к `Dog`.

```
Dog happy = (Dog) getsWalked;
```

Эти структуры помещаются в кучу только после упаковки.

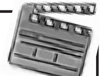


После этой строчки вы получаете третью копию данных в структуре happy, которая получает <u>свое собственное</u> место в стеке.

obj

Sid
husky

Dog sid (упакован)



Когда вызывается метод, он ищет свои аргументы в стеке.

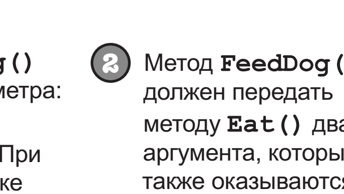
Стек играет важную роль в способе, которым CLR запускает ваши программы. Мы принимаем как должное тот факт, что один метод может вызывать другой и далее по цепочке. Метод может даже вызывать сам себя (это называется *рекурсия*). Эту возможность мы имеем благодаря стеку.

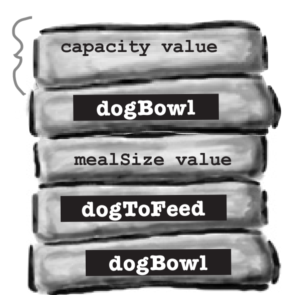
Вот пара методов из симулятора собачьего питомника. Они просты: метод `FeedDog()` вызывает метод `Eat()`, который в свою очередь вызывает метод `CheckBowl()`.

```
public void FeedDog(Canine dogToFeed, Bowl dogBowl) {
    double eaten = Eat(dogToFeed.MealSize, dogBowl);
    return eaten + .05d; // Немного всегда разливается
}

public void Eat(double mealSize, Bowl dogBowl) {
    dogBowl.Capacity -= mealSize;
    CheckBowl(dogBowl.Capacity);
}

public void CheckBowl(double capacity) {
    if (capacity < 12.5d) {
        string message = "Моя миска почти пуста!";
        Console.WriteLine(message);
    }
}
```

Напомним терминологию: параметр — это часть объявления метода, определяющая, какие значения вам требуются; аргумент — это реальное значение или ссылка, передаваемые вызываемому методу.

Вот как выглядит стек, когда метод `FeedDog()` вызывает метод `Eat()`, который вызывает метод `CheckBowl()`, вызывающий метод `Console.WriteLine()`:





1. Метод **`FeedDog()`** имеет два параметра: ссылку **`Canine`** и ссылку **`Bowl`**. При его вызове в стеке оказываются два переданных ему аргумента.

2. Метод **`FeedDog()`** должен передать методу **`Eat()`** два аргумента, которые также оказываются в стеке.

3. По мере вызова методов размер стека увеличивается.

4. После завершения метода **`Console.WriteLine()`** его аргументы удаляются из стека. Метод **`Eat()`** продолжает работу, как будто ничего не случилось. Вот как полезен стек!

Модификатор out

Существуют различные способы получения значений от программы. Они реализуются при помощи добавленных к объявлению метода **модификаторов**, в частности модификатора **out**. Вот как он работает. Создайте новое приложение Windows Forms и добавьте к форме пустое объявление метода. Оба параметра пометьте ключевым словом out:

```
public int ReturnThreeValues(out double half, out int twice)
{
       return 1;
}
```

Параметр out дает методу возможность вернуть более одного значения.

Попытавшись построить код, вы получите два сообщения об ошибке: **До передачи управления из текущего метода параметру, помеченному ключевым словом out, «half» должно быть присвоено значение** (аналогично для параметра «twice»). Работая с ключевым словом out, вы ***всегда*** должны задавать параметр до возвращения методом значения. Вот как выглядит метод целиком:

```
Random random = new Random();
public int ReturnThreeValues(out double half, out int twice) {
    int value = random.Next(1000);
    half = ((double)value) / 2;
    twice = value * 2;
    return value;
}
```

Параметрам, помеченным ключевым словом out, нужно заранее присвоить значения, иначе код компилироваться не будет.

Напоминание: когда программа Windows Forms вызывает метод Console.WriteLine(), он обновляет окно Output (View→Output).

Воспользуемся заданными параметрами. Добавьте кнопку со следующим обработчиком событий:

```
private void button1_Click(object sender, EventArgs e) {
    int a;
    double b;
    int c;
    a = ReturnThreeValues(b, c);
    Console.WriteLine("value = {0}, half = {1}, double = {2}", a, b, c);
}
```

Вы заметили, что присваивать начальные значения переменным b и c не требуется? Это не нужно делать до момента, пока вы не начнете использовать их в качестве аргументов для помеченного словом out параметра.

Ой! Снова ошибки построения: **Аргумент 1 должен быть передан с ключевым словом out**. В процессе вызова метода с параметром out нужно использовать это ключевое слово при передаче ему аргумента. Вот как это должно выглядеть:

```
    a = ReturnThreeValues(out b, out c);
```

Теперь программу можно построить и запустить. Метод ReturnThreeValues() задает и возвращает три значения: a получает возвращаемое значение метода, b — значение, возвращаемое параметром half, а c — значение, возвращаемое параметром twice.

Для этого проекта мы выбрали приложение Windows Forms как самый простой вариант. Нет ничего сложного в том, чтобы нажимать кнопки и смотреть на результат в окне Output.

Модификатор ref

Вам снова и снова придется сталкиваться с тем, что при передаче методу значений типа `int`, `double`, `struct` или другого значимого типа вы фактически передаете копию этого значения. Процедура так и называется: **передача по значению**.

Но аргументы можно передавать и методом, который называется **передача по ссылке**. Это реализуется при помощи модификатора **`ref`**. Как и модификатор `out`, он используется при объявлении и при вызове метода. Значимому или ссылочному типу принадлежит переменная, в данном случае не имеет значения, поскольку любая переменная с модификатором `ref` будет редактироваться непосредственно методом.

В отличие от аргумента, помеченного модификатором ref, аргумент, помеченный модификатором out, не требует инициализации перед его передачей.

Добавьте к программе метод, чтобы посмотреть, как это работает:

```
public void ModifyAnIntAndButton(ref int value, ref Button button) {
    int i = value;
    i *= 5;
    value = i - 3;
    button = button1;
}
```

Задавая значение и параметры кнопки, метод меняет переменные q и b в вызвавшем его методе button2_Click().

Добавим кнопку с обработчиком события для вызова метода:

```
private void button2_Click(object sender, EventArgs e) {
    int q = 100;
    Button b = button1;
    ModifyAnIntAndButton(ref q, ref b);
    Console.WriteLine("q = {0}, b.Text = {1}", q, b.Text);
}
```

Здесь выводится q = 497, b.Text = button1, так как метод поменял значения переменных q и b.

При вызове обработчиком `button2_Click()` метода `ModifyAnIntAndButton()` переменные `q` и `b` передаются по ссылке. Метод `ModifyAnIntAndButton()` работает с ними как с любыми другими переменными. Но благодаря передаче по ссылке он все время обновляет эти переменные, а не просто копирует их. После завершения метода `q` и `b` будут иметь отредактированные значения.

Запустите режим отладки и добавьте к переменным `q` и `b` контрольные значения, чтобы понять, как все работает.

Рассмотрим параметр out, встроенный в значимый тип.

Иногда строку вида "35.67" нужно преобразовать в значение типа double. Это можно сделать при помощи метода double.Parse("35.67"). Но запись double.Parse("xyz") приведет к исключению FormatException. Иногда требуется именно такой результат, а иногда требуется проверить возможность преобразования строки в значение. Здесь вам пригодится метод TryParse(): запись double.TryParse("xyz", out d) вернет значение false и присвоит параметру i значение 0, в то время как запись double.TryParse("35.67", out d) вернет значение true и присвоит переменной d значение 35.67.

Помните, как в главе 9 при помощи оператора switch мы преобразовывали Spades в Suits. Spades? Так вот, существуют статические методы Enum.Parse() и Enum.TryParse(), которые делают то же самое!

Необязательные параметры

Бывает так, что метод снова и снова вызывается с одними и теми же аргументами, но иногда ему требуется дополнительный параметр. Например, при наличии значений по умолчанию аргументы указываются, если при вызове метода что-то изменилось.

В такой ситуации вам на помощь придут необязательные параметры. В объявлении метода за такими параметрами следует знак равенства и значение по умолчанию. Количество необязательных параметров может быть произвольным.

Вот пример метода, в котором с помощью необязательных параметров проверяется, нет ли у человека высокой температуры:

```
void CheckTemperature(double temperature, double tooHigh = 99.5, double tooLow = 96.5)
{
    if (temperature < tooHigh && temperature > tooLow)
        Console.WriteLine("Чувствую себя хорошо!");
    else
        Console.WriteLine("Ой-ой! Вызовите врача!");
}
```

Значения по умолчанию для необязательных параметров указываются при их объявлении.

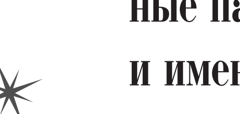
Необязательный параметр `tooHigh` имеет значение по умолчанию 99.5, а необязательный параметр `tooLow` — значение по умолчанию 96.5. При вызове метода `CheckTemperature()` с одним аргументом для параметров `tooHigh` и `tooLow` используются именно эти значения. Если же в вызове метода указать два аргумента, второй аргумент будет присвоен переменной `tooHigh`, в то время как переменная `tooLow` останется с заданным по умолчанию значением. Если же указать три аргумента, их значения будут переданы всем трем параметрам.

Для передачи значения определенному параметру существует и такая функция, как **именованные аргументы**, — достаточно присвоить параметру имя, указав через двоеточие его значения.

Добавим к форме метод `CheckTemperature()` и кнопку со следующим обработчиком события. Воспользуйтесь процедурой отладки, чтобы понять, как все это работает:

```
private void button3_Click(object sender, EventArgs e)
{
    // Такова температура среднестатистического человека
    CheckTemperature(101.3);

    // Температура собаки должна быть между 100.5 и 102.5 по Фаренгейту
    CheckTemperature(101.3, 102.5, 100.5);

    // У Боба всегда слишком низкая температура, поэтому присвоим
    // переменной tooLow значение 95.5
    CheckTemperature(96.2, tooLow: 95.5);
}
```

Для методов со значениями по умолчанию используйте необязательные параметры и именованные аргументы.



Типы, допускающие значение null

В ранних проектах `null` указывало на отсутствие значения. Это обычная ситуация: значение `null` указывает на пустоту переменной, поля или свойства, а проверка на равенство `null` означает **проверку на наличие значения**. Но структуры (а также int, boolean, enum и прочие значимые типы) не допускают значения null. Операторы:

```
bool myBool = null;
DateTime myDate = null;
```

на стадии компиляции станут причиной сообщения об ошибке!

В приложении Excuse Manager из главы 11 вы показывали, что дата не была задана через DateTime.MinValue. Структура Nullable<DateTime> упростит чтение как вашего кода, так и сериализованных XML-файлов.

Предположим, вам нужны данные о дате и времени. Для этого используется переменная `DateTime`. Но что делать, если ей не во всех случаях требуется присваивать значение? Воспользуйтесь типами, допускающими значение null. Достаточно поставить знак (?) после указания типа:

```
bool? myNulableInt = null;
DateTime? myNullableDate = null;
```

Свойство `Value` указывает на тип проверяемых значений. К примеру, для `DateTime?` свойство `Value` равно `DateTime`, для `int?` — соответственно int и т. п. Свойство `HasValue` возвращает значение true, если параметр не равен null.

Значимый тип всегда можно преобразовать к типу, допускающему значение null:

```
DateTime myDate = DateTime.Now;
DateTime? myNullableDate = myDate;
```

Но обратное преобразование сопровождается операцией приведения:

```
myDate = (DateTime) myNullableDate;
```

Кроме того, существует полезное свойство `Value`, возвращающее значение:

```
myDate = myNullableDate.Value;
```

Если свойство `HasValue` имеет значение false, свойство `Value` вызывает исключение `InvalidOperationException`, как и операция приведения (ведь она производится с использованием свойства `Value`).

Nullable<DateTime>
Value: DateTime HasValue: bool ...
GetValueOrDefault(): DateTime ...

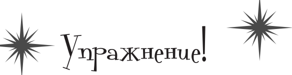
Структура Nullable<T> позволяет хранить как значимые типы, так и значение null. На диаграмме вы видите методы и свойства структуры Nullable<DateTime>.

После добавления к любому значимому типу знака вопроса (например, int? или decimal?) компилятор начнет воспринимать полученный результат как структуру Nullable<T> (Nullable<int> или Nullable<decimal>). Добавьте к программе переменную Nullable<DateTime>, поместите на нее точку останова и создайте контрольное значение. В окне watch отобразится System.DateTime?. Это пример псевдонима (alias). Наведите курсор на любое значение типа int. Оно будет преобразовано в структуру System.Int32:

```
int value;
struct System.Int32
Represents a 32-bit signed integer.
```

int.Parse() и int.TryParse() — члены этой структуры

Проделайте эту операцию для всех типов из начала главы 4. Обратите внимание, что все они, кроме типа string, который является классом System.String (то есть ссылочным, а не значимым типом), имеют псевдонимы для структур.



Типы, допускающие значение null, увеличивают робастность программы

Пользователи порой делают сумасшедшие вещи. Они могут щелкать на кнопках в неверном порядке, вводить по 256 пробелов в текстовое поле или при помощи Диспетчера задач прерывать программу на середине процесса записи в файл. В главе 10 мы говорили о том, что программы, умеющие обрабатывать некорректные данные, называются **робастными (robust)**. Увеличить робастность программы позволяют и типы, допускающие значение null. Убедитесь в этом сами: **создайте консольное приложение** и добавьте в него класс RobustGuy:

Добавив метод RobustGuy.ToString() и введя Birthday.Value, обратите внимание на окно IntelliSense. Вы увидите перечень членов для типа DateTime.

Метод ToLongDateString() преобразует это в читаемую строку.

Ticks
TimeOfDay
ToBinary
ToFileTime
ToFileTimeUtc
ToLocalTime
ToLongDateString
ToLongTimeString
ToOADate
ToShortDateString
ToShortTimeString
ToString
ToUniversalTime
Year

```
class RobustGuy {
    public DateTime? Birthday { get; private set; }
    public int? Height { get; private set; }

    public RobustGuy(string birthday, string height) {
        DateTime tempDate;
        if (DateTime.TryParse(birthday, out tempDate))
            Birthday = tempDate;
        else
            Birthday = null;

        int tempInt;
        if (int.TryParse(height, out tempInt))
            Height = tempInt;
        else
            Height = null;
    }

    public override string ToString() {
        string description;
        if (Birthday.HasValue)
            description = "Я родился " + Birthday.Value.ToLongDateString();
        else
            description = "Я не знаю дату своего рождения";
        if (Height.HasValue)
            description += ", во мне " + Height + " дюймов роста";
        else
            description += ", и я не знаю свой рост";
        return description;
    }
}
```

Воспользуйтесь типами DateTime и int в методе TryParse() для преобразования введенных пользователем данных в значения.

При вводе некорректных данных метод HasValue() вернет значение false.

Поэкспериментируйте с другими методами, связанными с типом DateTime, которые начинаются с To, чтобы понять, как они влияют на конечный результат.

А вот код метода Main(). Он использует метод **Console.ReadLine()** для ввода данных:

```
static void Main(string[] args) {
    Console.Write("Укажите дату рождения: ");
    string birthday = Console.ReadLine();
    Console.Write("Укажите рост в дюймах: ");
    string height = Console.ReadLine();
    RobustGuy guy = new RobustGuy(birthday, height);
    Console.WriteLine(guy.ToString());
    Console.ReadKey();
}
```

Метод Console.ReadLine() позволяет пользователю вводить информацию в консольное окно. После нажатия клавиши Enter метод возвращает строку.

Запустите программу и попробуйте вводить различные данные. Многие из них будут распознаны методом `DateTime.TryParse()`. Если сделать это не удастся, у свойства `Birthday` класса `RobustGuy` не будет значения.

Ребус в бассейне

Возьмите фрагменты кода из бассейна и заполните пробелы. Один и тот же фрагмент может использоваться несколько раз. В бассейне есть и лишние фрагменты. Получите код, выводящий на консоль запись «Вернусь через 20 минут» **при создании экземпляра класса 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);
    }
}
```

При создании объекта Faucet появляется строка:

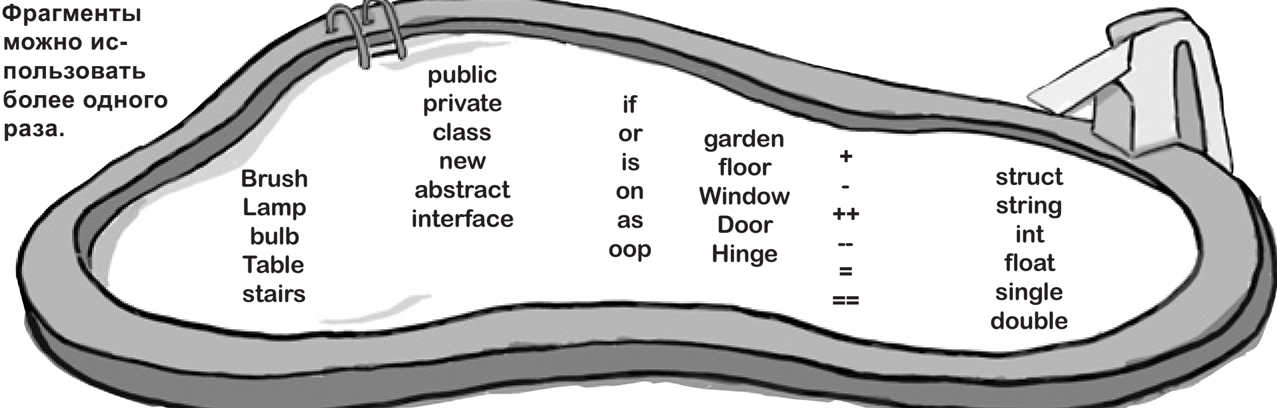
back in 20 minutes

Этот результат вам и требуется получить.

```
public ________ Table {
    public string stairs;
    public Hinge floor;
    public void Set(Hinge b) {
        floor = b;
    }
    public void Lamp(object oil) {
        if (oil _____ int)
            ________.bulb = (int)oil;
        else if (oil _____ string)
            stairs = (string)oil;
        else if (oil _____ Hinge) {
            ________ vine = oil _____ ________;
            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;
    }
}
```

Дополнительное задание: обведите строчки, в которых происходит упаковка.

Фрагменты можно использовать более одного раза.



Ответ на с. 638.

Часто задаваемые вопросы

В: Какая мне разница, что происходит в стеке?

О: Так как понимание различий между стеком и кучей позволяет корректно пользоваться ссылочными и значимыми типами. Легко забыть, что структуры и объекты функционируют по-разному, ведь операция присваивания для них выглядит одинаково. Представление о том, какие процессы происходят в .NET и CLR, позволяет понять, ***чем именно*** отличаются ссылочные и значимые типы.

В: А зачем нужна информация по поводу упаковки?

О: Потому что нужно понимать, когда действие переходит в стек и когда данные начинают копироваться туда и обратно. Упаковка требует места в памяти и занимает время. Разумеется, вы не заметите особой разницы, если эта процедура выполняется редко. Но представьте программу, выполняющую однотипные действия много раз в секунду. Работа такой программы будет требовать все больше ресурсов, программа будет замедляться, поэтому, наверное, имеет смысл избегать упаковки в часто повторяющейся части кода.

В: Я понял, что при операции присваивания одна структурная переменная копируется в другую. Но как я могу использовать эти знания?

О: Это поможет вам, к примеру, при **инкапсуляции**. Посмотрите на уже знакомый вам код класса, определяющего местоположение:

```
protected Point location;
public Point Location {
  get { return location;}
}
```

Если бы `Point` был классом, инкапсуляция не сработала бы. Закрытость поля `location` не имела бы значения, ведь вы создали открытое, предназначенное только для чтения свойство, возвращающее ссылку на это поле, то есть дали доступ другим объектам.

К счастью для нас, `Point` — это структура. И открытое свойство `Location` возвращает копию переменной. Работающий с ней объект может делать что хочет — это никак не скажется на состоянии закрытого поля `location`.

Вернитесь к проекту обмена метками из главы 4. Там вы неявно использовали структуру point и свойство location, то есть код присваивал значения структурам (несмотря на то, что в явном виде вы их не объявляли).

В: Как определить, что мне нужно в текущий момент — класс или структура?

О: В большинстве случаев программисты работают с классами, потому что структуры имеют слишком много ограничений. Они не поддерживают наследование, абстракции и полиморфизм, а вы уже знаете, насколько важны эти вещи при создании больших приложений.

Структуры же полезны при повторяющейся работе с ограниченными типами данных. Хорошим примером служат прямоугольники и точки — они применяются только в определенных ситуациях, зато с удивительной регулярностью. При наличии у вас небольших групп разнородных данных, которые требуется сохранить в поле или передать методу в качестве параметра, скорее всего, имеет смысл воспользоваться структурой.

Структура позволяет улучшить инкапсуляцию класса, так как возвращающее ее, предназначенное только для чтения свойство всегда создает новую копию.

Возьми в руку карандаш

Предполагалось, что этот метод убьет объект `Clone`, но он не работает. Почему?

```
private void SetCloneToNull(Clone clone) {
    clone = null;
}
```

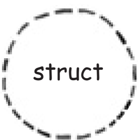
....

....

Ответ на с. 632.

Что осталось от Великолепного

После разговора об упаковке вы должны сообразить, почему Капитан Великолепный потерял свою силу. Все дело в том, что это уже не он, а упакованная структура



vs.

Объект

Возможность легко получать копии является большим преимуществом структур и других значимых типов.

1. **Структуры не наследуют от классов**
   Именно поэтому Капитан так ослабел. Ведь он больше не наследует нужного поведения.

2. **Структуры копируются по значению**
   Это одно из самых больших их преимуществ, незаменимых для инкапсуляции.

Важное замечание: ключевое слово "is" позволяет проверить, реализует ли структура интерфейсы, что является существенным аспектом поддерживаемого структурами полиморфизма.

1. **Вы не можете создать копию объекта**
   В процессе операции присваивания вы копируете ***ссылку*** на ***ту же самую*** переменную.

2. **Вы можете пользоваться ключевым словом as**
   Объекты могут функционировать как их родители, то есть для них допустим полиморфизм.

Вернемся в лабораторию

КАЖЕТСЯ, Я НАШЕЛ СПОСОБ ПЕРЕДАТЬ ЕГО СИЛУ ОБЫЧНОМУ ЧЕЛОВЕКУ!

Сущность Великолепного

Методы расширения

Помните модификатор доступа sealed из главы 7? С его помощью создаются классы, не допускающие расширения.

Иногда требуется расширить класс, от которого невозможно наследование (к примеру, многие классы .NET помечены модификатором sealed). На этот случай в C# имеются **методы расширения (extension methods)**. Они позволяют **добавить методы в существующие классы**. Вам нужно только создать статический класс и добавить туда статистический метод, в качестве первого параметра принимающий экземпляр этого класса.

Предположим, у вас есть помеченный модификатором `sealed` класс `OrdinaryHuman`:

```
sealed class OrdinaryHuman {
    private int age;
    int weight;

    public OrdinaryHuman(int weight){
        this.weight = weight;
    }

    public void GoToWork() { /* код похода на работу */ }
    public void PayBills() { /* код оплаты счетов */ }
}
```

От класса OrdinaryHuman (Обычный человек) невозможно наследовать. Но что, если добавить в него метод?

Вы используете метод расширения, указывая первый параметр при помощи ключевого слова this.

Добавим метод расширения `SuperSoldierSerum` (Суперсолдат):

```
static class SuperSoldierSerum {
    public static string BreakWalls(this OrdinaryHuman h, double wallDensity) {
        return ("Я сломал стену плотностью" + wallDensity + ".");
    }
}
```

Чтобы расширить класс OrdinaryHuman, указываем его в качестве первого параметра с ключевым словом this.

Методы расширения всегда являются статическими и должны находиться в **статических классах**.

Созданный экземпляр класса OrdinaryHuman имеет непосредственный доступ к методу BreakWalls() — до тех пор, пока у него есть доступ к классу SuperSoldierSerum.

При добавлении класса `SuperSoldierSerum` класс `OrdinaryHuman` получает метод `BreakWalls`, который может использоваться формой:

```
static void Main(string[] args){
    OrdinaryHuman steve = new OrdinaryHuman(185);
    Console.WriteLine(steve.BreakWalls(89.2));
}
```

Попробуйте это сделать! **Создайте консольное приложение** и добавьте туда два класса и метод Main(). Запустите отладку и посмотрите, что происходит в методе BreakWalls().

Возьми в руку карандаш
Решение

Почему этот метод не уничтожил объект `Clone`?

```
private void SetCloneToNull(Clone clone) {
    clone = null;
}
```

Этот метод присваивает своему параметру значение null, но данный параметр является всего лишь ссылкой на объект Clone.

Так как параметр Clone находится в стеке, присвоение ему значения null никак не скажется на состоянии кучи.

Часто Задаваемые Вопросы

В: Почему бы вместо методов расширения не добавить код нужного метода непосредственно в класс?

О: Именно так и нужно поступать, если речь идет о добавлении метода в один класс. Методы расширения следует использовать аккуратно и только в случаях, когда по каким-то причинам вы не можете отредактировать нужный вам класс (например, потому, что он является частью .NET Framework). Методы расширения также применяются для редактирования поведения сущностей, к ***которым отсутствует доступ***, например типа или объекта из .NET Framework или другой библиотеки.

В: А зачем нужны методы расширения, если класс можно расширить при помощи наследования?

О: Если вы можете расширить класс, это нужно сделать — методы расширения не являются заменой наследованию. Но существуют и классы, для которых это невозможно. Методы расширения позволяют менять поведение групп объектов и даже добавлять функциональность к базовым классам .NET Framework. При этом, чтобы воспользоваться новым поведением, нужно работать с новым производным классом.

В: Методы расширения влияют на все экземпляры класса или только на некоторые?

О: Они влияют на все экземпляры расширенного вами класса. Более того, созданный метод расширения будет показываться ИСР вместе с обычными для рассматриваемого класса методами.

Нужно помнить, что метод расширения не дает доступа к внутреннему коду класса.

Я понял! Методы расширения позволяют отредактировать поведение встроенных классов .NET Framework.

Именно так! Есть классы, от которых вы не можете наследовать.

Откройте любой проект и введите в любой класс вот такой код:

```
class x : string { }
```

При компиляции появится сообщение об ошибке, потому что некоторые классы .NET помечены модификатором **sealed**, запрещающим наследование. Методы расширения дают возможность поменять поведение такого класса.

Но этим их функциональность не исчерпывается. Они позволяют расширять **интерфейсы**. Достаточно после ключевого слова `this` подставить имя интерфейса вместо имени класса. В результате метод расширения добавляется **во все классы, реализующие данный интерфейс**. LINQ будет подробно рассматриваться в следующей главе. В процессе изучения этого материала следует помнить, что система целиком построена на методах, расширяющих интерфейс `IEnumerable<T>`.

Объединение интерфейса с методами расширения крайне полезно, так как позволяет добавить поведение в любой реализующий интерфейс класс.

Расширяем фундаментальный тип: string

Необходимость менять поведение такого фундаментального типа, как strings, возникает нечасто. Но вы вполне можете это сделать! Создайте новый проект и добавьте к нему файл *HumanExtensions.cs.* Тип создаваемого проекта не имеет значения, для изучения принципов работы методов расширения мы будем пользоваться IDE.

1 Поместим методы расширения в отдельное пространство имен

Сохранить расширения отдельного от основного кода — хорошая идея. Это позволит легко обнаружить их при необходимости:

```
namespace MyExtensions {
  public static class HumanExtensions {
```

Отдельное пространство имен — это хороший организационный инструмент.

Методы расширения должны находиться в статическом классе.

2 Создайте статический метод расширения, определите его первый параметр ключевым словом this и укажите, что вы расширяете

Объявляя метод расширения, укажите расширяемый им класс в качестве первого параметра.

Метод расширения должен быть статическим.

```
public static bool IsDistressCall (this string s){
```

this string указывает, что мы расширяем класс string.

Метод расширения также должен быть статическим.

3 Поместите в метод код, оценивающий строку

```
public static class HumanExtensions {
  public static bool IsDistressCall(this string s){
    if (s.Contains("Помогите!"))
      return true;
    else
      return false;
  }
}
```

Нам нужен доступ к этому классу из других пространств имен, поэтому используйте модификатор public!

Здесь проверяется, не содержит ли строка определенного значения... которое отсутствует в исходном классе string.

4 Используйте новый метод расширения IsDistressCall().

Откройте любой другой класс и добавьте сверху `using MyExtensions;`. Теперь при работе со строками вы можете пользоваться методами расширения. Чтобы убедиться в этом, введите имя строковой переменной и точку:

```
static void Main(string[] args)
{
    string message = "Clones are wreaking havoc at the factory. Help!";
    message.
}
```

```
IndexOf
IndexOfAny
Insert
Intersect<>
IsDistressCall
IsNormalized
Join<>
Last<>
LastIndexOf
```

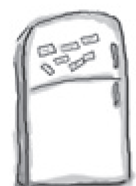
(extension) bool string.IsDistressCall()

Сразу после ввода точки появится окно с перечнем методов класса string... в их число включен и ваш метод расширения.

Превратите строку using в комментарий, и метод расширения исчезнет из окна IntelliSense.

Окно IntelliSense скажет, что это расширение.

Этот пример продемонстрировал вам синтаксис методов расширения. В следующей главе вы получите намного больше информации о них. Ведь именно при помощи этих методов реализован LINQ.



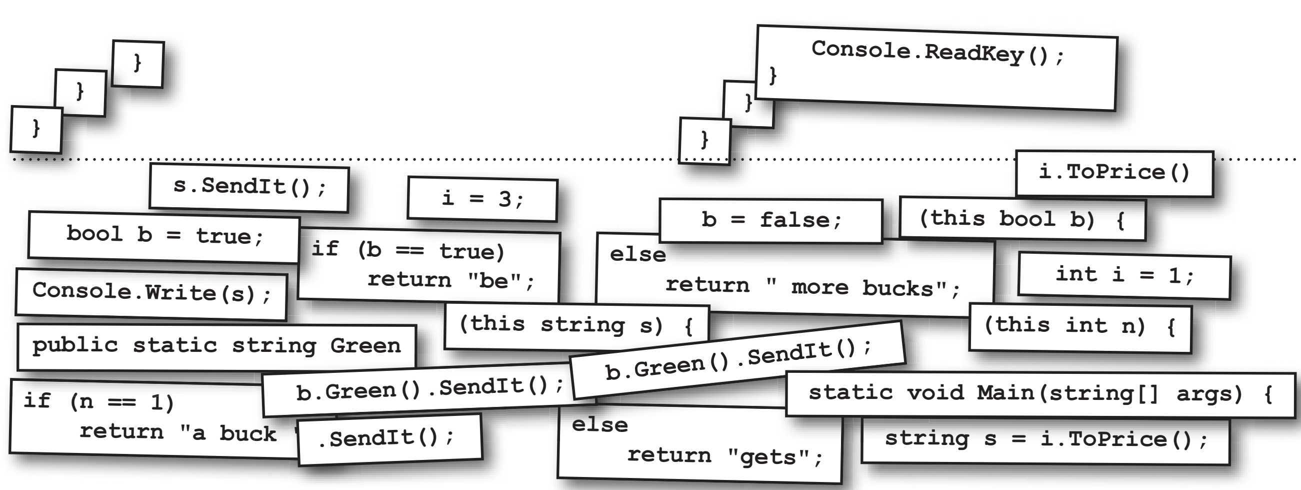
Магниты расширения

Расположите магниты таким образом, чтобы на выходе получилась строка:

`a buck begets more bucks` (*деньги к деньгам*)

```
namespace Upside {
  public static class Margin {
    public static void SendIt

    }
    public static string ToPrice

    }
```

```
using Upside;
namespace Sideways {
class Program {
```



Магниты расширения

Вот как нужно было расположить магниты, чтобы получить на выходе поговорку:

a buck begets more bucks

Расширения содержатся в пространстве имен Upside. Точка входа находится в пространстве имен Sideways.

Класс Margin расширяет строку путем добавления метода SendIt(), который выводит содержимое строки на консоль. Тип int он расширяет при помощи метода ToPrice(), возвращающего значение a buck при равенстве целой переменной 1 и more bucks в остальных случаях.

```
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 Green расширяет класс bool — он возвращает строку be, если булева переменная имеет значение true, и get — в случае значения false.

Точка входа использует расширения, добавленные в класс Margin.

```
using Upside;
namespace Sideways {
  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();
      Console.ReadKey();
    }
  }
}
```

ВСЕЛЕННАЯ

КАПИТАН ВЕЛИКОЛЕПНЫЙ ВЕРНУЛСЯ

Смерть — это не конец!

Статья Баки Барнса
корреспондента ВСЕЛЕННОЙ

Объектвиль

Десериализация и **триумфальное возвращение Великолепного**

Капитан Великолепный чудесным образом вернулся в Объектвиль. В прошлом месяце было обнаружено, что его гроб пуст. Вместо тела там оказалась странная записка. Ее анализ предоставил нам ДНК объекта Captain Amazing — его последние поля и значения, записанные в двоичном формате.

Сегодня Капитан был благополучно десериализован. На вопрос об источнике подобной идеи Капитан отвечает ссылкой на главу 10. Его окружение отказывает комментировать этот загадочный ответ, но рассказывают, что перед неудачным покушением на Жулика Капитан много читал о методе Dispose и выживании. Мы ожидаем, что Капитан Великолепный...

Великолепный вернулся!

Метод Lamp() задает различные строки и переменные типа int. При вызове его с целочисленной переменной в качестве параметра он поместит в поле Bulb ссылку, на которую указывает объект Hinge.

Результат после создания объекта Faucet:
`back in 20 minutes`

```
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);
    }
}
```

Вот почему Table — это структура. Если бы это был класс, переменные wine и book.Garden указывали бы на один и тот же объект, и строка back in оказалась бы переписанной.

Обведены строчки, в которых происходит упаковка.

Метод Lamp() использует в качестве параметра объект. Значит, упаковка будет автоматически осуществляться при передаче ему значений типа int или string.

```
public struct Table {
    public string stairs;
    public Hinge floor;
    public void Set(Hinge b) {
        floor = b;
    }
    public void Lamp(object oil) {
        if (oil is int)
            floor.bulb = (int)oil;
        else if (oil is string)
            stairs = (string)oil;
        else if (oil is Hinge) {
            Hinge vine = oil as Hinge;
            Console.WriteLine(vine.Table()
             + " " + floor.bulb + " " + stairs);
        }
    }
}
```

Метод Lamp помещает переданную ему строку в поле Stairs.

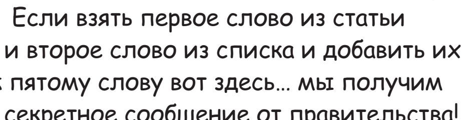
Помните, что ключевое слово as работает только с классами?

```
public class Hinge {
    public int bulb;
    public Table garden;
    public void Set(Table a) {
        garden = a;
    }
    public string Table() {
        return garden.stairs;
    }
}
```

Класс Hinge и структура Table обладают методом Set(). В классе Hinge этот метод задает поле Garden структуры Table. А у структуры Table метод задает поле Floor в классе Hinge.

14 LINQ

Управляем данными



Этот мир управляется данными... вам лучше знать, как в нем жить. Времена, когда можно было программировать днями и даже неделями, **не касаясь множества данных**, давно позади. В наши дни ***с данными связано все***. Часто приходится работать с данными из разных источников и даже разных форматов. Базы данных, XML, коллекции из других программ... все это давно стало частью работы программиста на C#. В этом ему помогает **LINQ**. Эта функция не только упрощает запросы, но и умеет **разбивать данные на группы** и, наоборот, **соединять данные из различных источников**.

Джимми — фанат комиксов...

Познакомьтесь, это Джимми, один из самых успешных коллекционеров комиксов про супергероя Капитана Великолепного и всего, что с этими комиксами связано. Он знает о Капитане всё, приобрел все коллекционные фан-арты ко всем фильмам и собрал коллекцию, к которой больше всего подходит эпитет «великолепная».

...но его коллекция валяется по всему дому

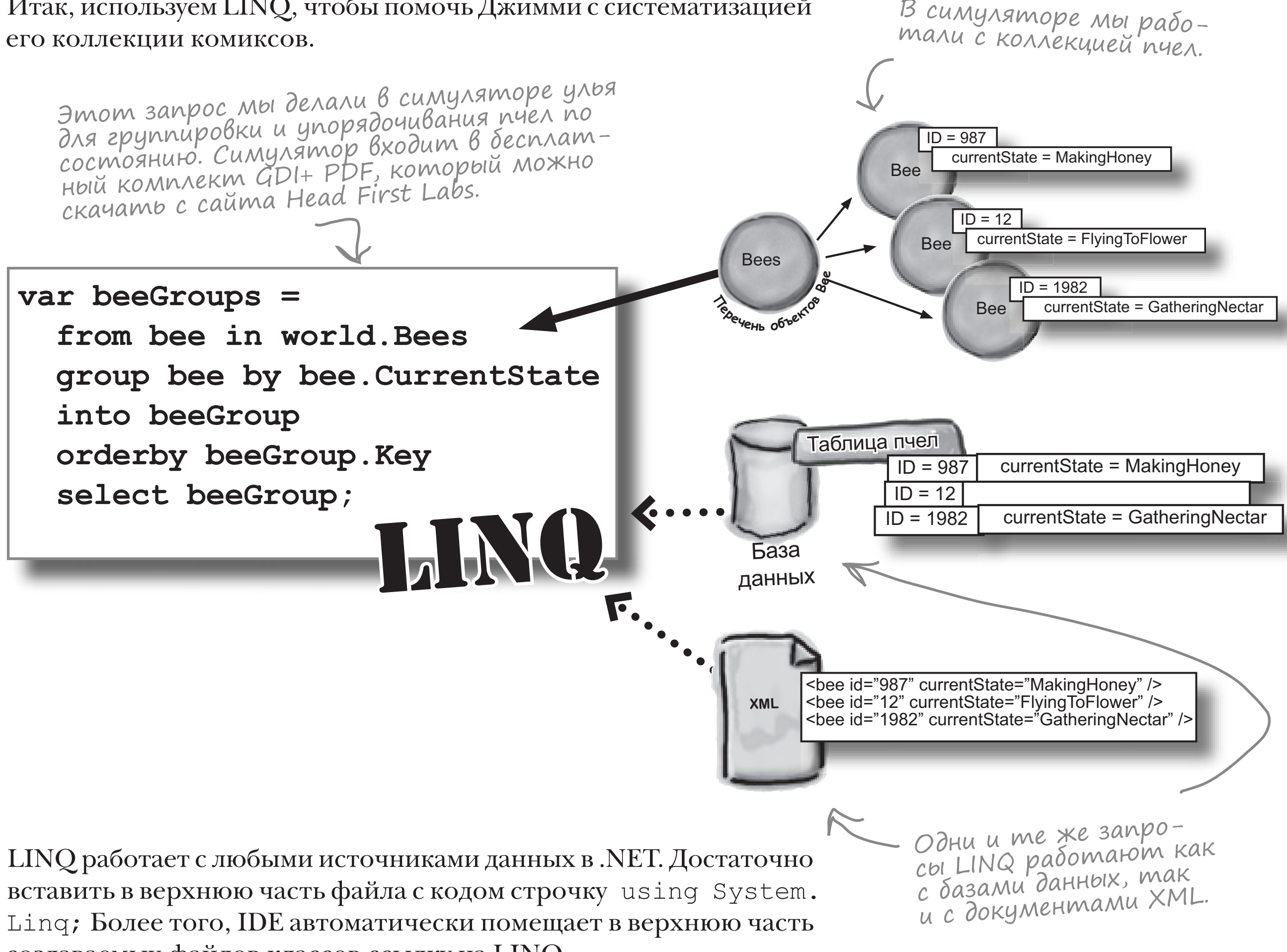
Джимми увлеченный, но не очень дисциплинированный коллекционер. Он пытается отсыкать комиксы, которые считает «жемчужинами» своей коллекции, но ему нужна ваша помощь. Поможем Джимми написать приложение для систематизации его сокровищ?

В рамку помещена обложка легендарного выпуска «Death of the Object», подписанная авторами.

Сбор данных из разных источников

Вас спасет LINQ! Эта аббревиатура расшифровывается как Language Integrated Query (Встроенный язык запросов) и позволяет писать запросы для извлечения данных из коллекций. Впрочем, коллекциями его применение не ограничивается, вы можете писать запросы к *любому* объекту, реализующему интерфейс IEnumerable<T> .

Итак, используем LINQ, чтобы помочь Джимми с систематизацией его коллекции комиксов.



LINQ работает с любыми источниками данных в .NET. Достаточно вставить в верхнюю часть файла с кодом строчку `using System.Linq;` Более того, IDE автоматически помещает в верхнюю часть создаваемых файлов классов ссылку на LINQ.

Коллекции .NET уже настроены под LINQ

Все коллекции .NET реализуют интерфейс `IEnumerable<T>`, с которым вы познакомились в главе 8. Вспомним, как он функционирует: введите в код строчку `System.Collections.Generic.IEnumerable<int>`, щелкните на ней правой кнопкой мыши и выберите команду Go to Definition (или нажмите клавишу F12). Вы увидите, что интерфейс `IEnumerable` определяет метод `GetEnumerator()`:

Обратили внимание, как интерфейс IEnumerable<T> расширяет интерфейс IEnumerable? Для получения детальной информации снова воспользуйтесь командой Go to Definition.

```
namespace System.Collections.Generic {
  interface IEnumerable<T> : IEnumerable {
    // Резюме:
    //     Осуществляет простой перебор элементов коллекции.
    //
    // Возвращает:
    //     System.Collections.Generic.IEnumerator<T>, который
    //       и перебирает элементы коллекции.
    IEnumerator<T> GetEnumerator();
  }
}
```

Это единственный метод интерфейса. Его реализует каждая коллекция. Вы можете создавать и собственные объекты, реализующие интерфейс IEnumerable<T>... а затем работать с ними при помощи LINQ.

Метод требует указать способ перемещения объекта от одного элемента коллекции к другому. Это условие любого запроса LINQ. Если вы можете просматривать коллекцию элемент за элементом, значит, можете и реализовывать интерфейс `IEnumerable<T>`. Соответственно LINQ в состоянии посылать коллекции запросы.

За сценой

Для запросов, сортировки и обновления данных LINQ использует **методы расширения**. Убедитесь в этом сами. Создайте массив типа `int` с именем `linqtest`, поместите в него числа и введите эту строку:

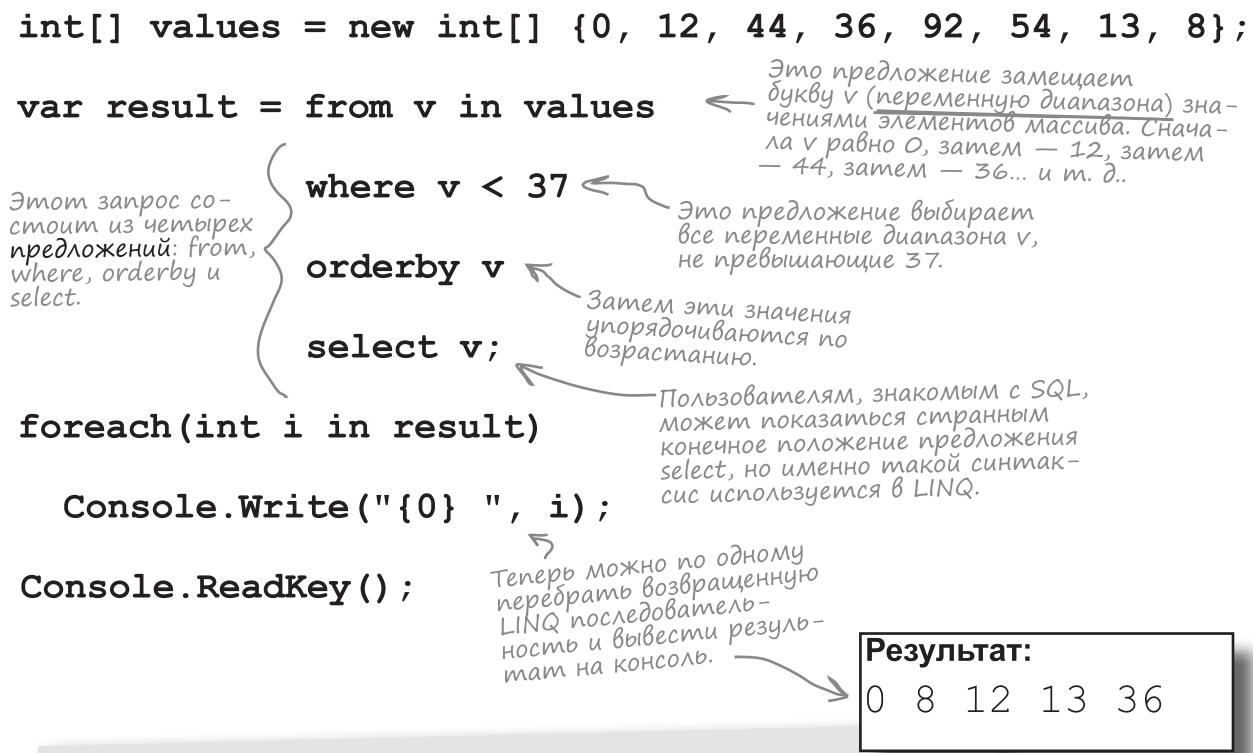
```
IEnumerable<int> result = from i in linqtest where i < 3 select i;
```

А теперь превратите в комментарий строку `using System.Linq;` в заголовке файла. Теперь решение построить не удастся. Ведь именно те методы, которые вы вызываете, работая с LINQ, использовались для расширения массива.

Теперь вы видите, почему методы расширения, с которыми вы познакомились в главе 14, так важны... они позволят .NET (а заодно и вам) менять поведение существующих типов.

Простой способ сделать запрос

Перед вами пример синтаксиса LINQ. Он выбирает из массива типа `int` числа меньше 37 и располагает их в возрастающем порядке. Для этого используются четыре **предложения (clauses)**, указывающие порядок запросов, критерии выбора, критерии сортировки и способ возвращения результата.



Ключевое слово var заставляет компилятор определять тип переменной. .NET диагностирует его по типу локальной переменной, которую вы использовали для запроса LINQ.

В рассматриваемом примере при компиляции этой строки:

```
var result = from v in values
```

ключевое слово var заменяется на:

```
IEnumerable<int>
```

И раз уж мы коснулись интерфейсов, работающих с коллекциями, вспомним о том, что интерфейс IEnumerable<T> позволяет просматривать элементы один за другим. Большинство запросов LINQ реализованы при помощи методов, расширяющих этот интерфейс, так что сталкиваться с ним вам придется довольно часто.

Вернитесь к главе 8 для повторения материала о работе интерфейса **`IEnumerable<T>`**.

Сложные запросы

Джимми продал свою недавно созданную фирму крупному инвестору и хочет потратить часть прибыли на покупку самых дорогих комиксов про Капитана Великолепного, которые только сможет найти. Каким образом LINQ может помочь в этом поиске?

1. С сайта фанатов Великолепного Джимми скачал список всех выпусков и поместил их в коллекцию `List<T>` объекта `Comic`. Этот объект имеет два поля `Name` (Название) и `Issue` (Выпуск).

```
class Comic {
    public string Name { get; set; }
    public int Issue { get; set; }
}
```

Для построения каталога Джимми воспользовался инициалами:

```
private static IEnumerable<Comic> BuildCatalog()
{
    return new List<Comic> {
        new Comic { Name = "Johnny America vs. the Pinko", Issue = 6 },
        new Comic { Name = "Rock and Roll (ограниченный выпуск)", Issue = 19 },
        new Comic { Name = "Woman's Work", Issue = 36 },
        new Comic { Name = "Hippie Madness (с опечатками)", Issue = 57 },
        new Comic { Name = "Revenge of the New Wave Freak (поврежден)", Issue = 68 },
        new Comic { Name = "Black Monday", Issue = 74 },
        new Comic { Name = "Tribal Tattoo Madness", Issue = 83 },
        new Comic { Name = "The Death of an Object", Issue = 97 },
    };
}
```

Этот метод был указан как статический для того, чтобы его можно было легко вызвать из метода точки входа консольного приложения.

Выпуск #74 комиксов про Капитана Великолепного называется «Black Monday».

2. К счастью, существует замечательный каталог Грега. Джимми узнал, что выпуск #57 «Hippie Madness» из-за опечаток был практически полностью уничтожен издателем. Он обнаружил редкую копию, недавно проданную через каталог Грега за $13525. После долгих поисков нужной информации Джимми смог создать словарь, сопоставляющий номер выпуска и его цену.

```
private static Dictionary<int, decimal> GetPrices()
{
    return new Dictionary<int, decimal> {
        { 6, 3600M },
        { 19, 500M },
        { 36, 650M },
        { 57, 13525M },
        { 68, 250M },
        { 74, 75M },
        { 83, 25.75M },
        { 97, 35.25M },
    };
}
```

Этот синтаксис для инициализатора словарей мы изучали в главе 8.

Выпуск #57 стоит $13,525.

Внимательно исследуйте запрос на с. 644. Как именно должен сформировать свой запрос Джимми, чтобы найти самый дорогой выпуск комиксов?

Анатомия запроса

Проанализировать данные, которые собрал Джимми, можно путем единственного запроса LINQ. Предложение `where` указывает, какие элементы коллекции нужно включить в конечный результат. При этом можно не ограничиваться простой операцией сравнения, а включить любые выражения из C#. Например, воспользоваться словарным полем `values` для включения в результат комиксов, стоящих дороже $500. Затем полученная последовательность будет упорядочена при помощи предложения `orderby`.

```
IEnumerable<Comic> comics = BuildCatalog();
Dictionary<int, decimal> values = GetPrices();

var mostExpensive =
    from comic in comics
    where values[comic.Issue] > 500
    orderby values[comic.Issue] descending
    select comic;

foreach (Comic comic in mostExpensive)
    Console.WriteLine("{0} стоит {1:c}",
                      comic.Name, values[comic.Issue]);
```

Запрос LINQ извлекает объекты Comic из предложенного списка на основе данных из словарного поля values.

Первым в запросе идет предложение from. В качестве источника данных оно указывает на коллекцию **comics**, а переменной диапазона присваивает имя **comic**.

В предложении from можно использовать любое имя. Мы взяли имя comic.

В предложения where и orderby можно включить ЛЮБЫЕ операторы C#, поэтому мы используем словарные значения для выбора комиксов, цена которых превышает $500. Затем мы сортируем результат по убыванию.

Появившееся в предложении from имя **comic** затем используется в предложениях where и orderby.

Запись «{1:c}» в параметрах метода WriteLine означает, что второй параметр нужно вывести в формате локальной валюты.

Результат запроса возвращается в виде перечислителя IEnumerable<T>, который называется mostExpensive. Что именно вошло в результат, определило предложение select — запрос вернул набор объектов Comic.

Результат:

```
Hippie Madness (с опечатками) стоит $13,525.00
Johnny America vs. the Pinko стоит $3,600.00
Woman's Work стоит $650.00
```

Все LINQ-запросы в этой главе рассматриваются дважды: сначала в консольном приложении для демонстрации принципа их работы, затем в приложении для магазина Windows для демонстрации работы в контексте, — ведь мозг лучше усваивает понятия, показанные в контексте!

Даже если вы ничего не знаете об SQL, поводов для беспокойства нет — для работы с LINQ вам не потребуются никакие сопутствующие сведения.

Я не буду это покупать. Я знаю SQL — именно на него похожи запросы LINQ, не так ли?

LINQ, может быть, и *выглядит* как SQL, но *работает* он по-другому.

У знатоков SQL может возникнуть соблазн пропустить главу про LINQ в предположении, что тут все интуитивно понятно и очевидно. В LINQ и в самом деле используются позаимствованные из SQL ключевые слова `select`, `from`, `where`, `descending` и `join`. Но LINQ при этом сильно отличается от SQL, поэтому код, написанный в привычной вам манере, скорее всего, **не будет работать ожидаемым образом**.

SQL работает с ***таблицами***, а не с ***перечислимыми объектами***. Таблицы не имеют порядка. Выполняя SQL-запрос `select`, можно быть уверенным, что таблица обновляться не будет. SQL снабжен различными средствами обеспечения безопасности данных, которым вполне можно доверять.

Если рассмотреть SQL более детально, то его запросы являются операциями над множествами. Это означает, что обращение к столбцам таблицы неупорядочено. Коллекции же, при своей способности сохранять *что угодно* — значения, структуры, объекты и т. п., имеют определенный порядок. LINQ позволяет осуществлять любые операции, которые поддерживают происходящие в коллекции процессы, — вы можете даже вызывать методы для содержащихся внутри объектов. Просмотр элементов осуществляется циклично, то есть в строго определенном порядке. Может показаться, что все это не имеет особого значения, но если вы привыкли работать с SQL, написанные в аналогичной манере запросы LINQ дадут вам результат, далекий от ожидаемого.

Существуют и другие отличия LINQ от SQL, но мы не будем вдаваться в подробности! Достаточно, чтобы вы поняли, что не нужно ожидать от LINQ-запросов знакомого вам поведения.

Идем на помощь Джимми

Сделаем приложение для магазина Windows, чтобы помочь Джимми разобраться с коллекцией комиксов и показать, насколько полезен LINQ при работе с данными.

Навигация в приложениях для магазина Windows

Найдите на панели инструментов XAML-эквивалент элемента TabControl, применяемого при работе с WinForms. Не видите? Это не случайно. Вкладки являются неотъемлемым атрибутом приложений для рабочего стола, но они перегружают экран. Приложения для магазина Windows используют **навигационную систему на основе страниц**, которая уменьшает количество объектов на экране и позволяет создать интуитивно понятный интерфейс.

При переходе приложения на следующую страницу становится видимой кнопка back, и Джимми может вернуться назад.

Jimmy's Comics

Available Queries

Expensive comics

Когда Джимми щелкает на элементе списка запросов на основной странице, приложение открывает страницу с подробными сведениями по этому запросу.

Подробно схемы навигации приложений для магазин Windows описаны на странице: http://msdn.microsoft.com/ru-ru/library/windows/apps/hh761500.aspx

Навигационная система приложения

Вот еще одна обучающая возможность IDE. Возьмите любое из построенных приложений для магазина Windows и откройте файл *App.xaml.cs.* Это основной файл приложения, являющийся подклассом класса `Application` в пространстве имен `Windows.UI.Xaml`. Он всегда называется *App.xaml.* Объект `Application` инициализирует приложение и управляет его жизненным циклом: загрузкой, приостановкой и возобновлением работы. Он создает объект **`Frame`** (из пространства имен `Windows.UI.Xaml.Controls`), при помощи которого приложение поддерживает навигацию по XAML-страницам.

Найдите в `App` метод `OnLaunched()`, он запускается при загрузке приложения и задает оформление:

```
/// <summary>
/// Invoked when the application is launched normally by the end user.  Other entry points
/// will be used when the application is launched to open a specific file, to display
/// search results, and so forth.
/// </summary>
/// <param name="args">Details about the launch request and process.</param>
protected override void OnLaunched(LaunchActivatedEventArgs args)
{
    Frame rootFrame = Window.Current.Content as Frame;

    // Do not repeat app initialization when the Window already has content,
    // just ensure that the window is active
    if (rootFrame == null)
    {
        // Create a Frame to act as the navigation context and navigate to the first page
        rootFrame = new Frame();

        if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
        {
            //TODO: Load state from previously suspended application
        }

        // Place the frame in the current Window
        Window.Current.Content = rootFrame;
    }

    if (rootFrame.Content == null)
    {
        // When the navigation stack isn't restored navigate to the first page,
        // configuring the new page by passing required information as a navigation
        // parameter
        if (!rootFrame.Navigate(typeof(MainPage), args.Arguments))
        {
            throw new Exception("Failed to create initial page");
        }
    }
    // Ensure the current window is active
    Window.Current.Activate();
}
```

Воспользуйтесь командой меню «Go To Definition» для исследования классов Window и Frame, представляющих главное окно приложения и его навигационную систему.

Именно здесь создается новая навигационная система, которая будет содержать все страницы приложения.

При удалении страницы MainPage.xaml и замещении ее одноименным шаблоном Basic Page вместо удаленного класса MainPage добавляется новый, чтобы метод Navigate() смог создать экземпляр новой страницы вместо страницы по умолчанию.

Приложение загружает основную страницу. Метод `Frame.Navigate()` создает экземпляр страницы и отображает ее содержимое. Ключевое слово `typeof` заставляет вернуть тип класса, чтобы метод узнал нужный тип экземпляра страницы.

Для постраничной навигации приложение используется `Frame.Navigate()`. Любая страница XAML обладает **свойством `Frame`**. Если добавить страницу AnotherPage, до нее можно добраться, используя переданный методу `Navigate()` **аргумент query**. Это **параметр, переданный** только что созданной странице.

```
if (this.Frame != null)
    this.Frame.Navigate(typeof(AnotherPage), query);
```

При создании AnotherPage IDE добавит к проекту класс AnotherPage, и код будет перемещать вас к новому экземпляру AnotherPage, передавая это «query» в качестве аргумента.

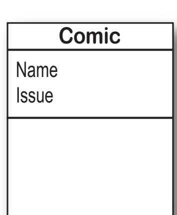
Начало работы над приложением для Джимми

Мы построим приложение, использующее навигационную систему для выполнения различных запросов LINQ, начиная с пары уже известных вам запросов.

1. **Создадим новый проект приложения для магазина Windows.**

   Воспользуйтесь шаблоном Blank App, удалите файл *MainPage.xaml* и добавьте новый шаблон Basic Page с именем *MainPage.xaml*. Затем **добавьте еще один шаблон Basic Page с именем *QueryDetail.xaml***. Перед выполнением следующего шага не забудьте выбрать в меню Build команду **Rebuild Solution**.

2. **Добавим класс `Comic`.**



   Несколько страниц назад вы уже видели класс `Comic`. Теперь добавьте его к проекту.

```
class Comic {
    public string Name { get; set; }
    public int Issue { get; set; }
}
```

3. **Добавим класс `ComicQuery`.**

   Этот класс представляет запрос, и после завершения работы приложения у вас будет по одному экземпляру `ComicQuery` для каждого запроса LINQ. Посмотрите на снимок экрана, приведенный двумя страницами ранее. Каждому запросу соответствует значок, и это следует отразить в вашем классе. Воспользуемся объектом `BitmapImage` из пространства имен `Windows.UI.Xaml.Media.Imaging`, для чего в верхнюю часть файла класса следует добавить строку `using`.

ComicQuery
Title
Subtitle
Description
Image

```
using Windows.UI.Xaml.Media.Imaging;

class ComicQuery {
    public string Title { get; private set; }
    public string Subtitle { get; private set; }
    public string Description { get; private set; }
    public BitmapImage Image { get; private set; }

    public ComicQuery(string title, string subtitle,
                      string description, BitmapImage image) {
        Title = title;
        Subtitle = subtitle;
        Description = description;
        Image = image;
    }
}
```

Если хотите, можете поместить оператор using в объявление пространства имен файла .cs.

4 **Добавим класс менеджера запросов так, чтобы ваши элементы управления имели возможность связываться.**

ComicQueryManager
AvailableQueries CurrentQueryResults Title
UpdateAvailableQueries() UpdateQueryResults() static BuildCatalog() static GetPrices() private LinqMakesQueriesEasy() private ExpensiveComics() private CreateImageFromAssets()

Для приложения Джимми воспользуемся знакомым шаблоном. Всю работу по запуску запросов и отображению свойств с результатами выполнит класс `ComicQueryManager`. Каждая страница XAML имеет статический ресурс с экземпляром `ComicQueryManager`, который вызывает методы, запускающие запросы, и связывает результаты с элементами управления.

```
using System.Collections.ObjectModel;
using Windows.UI.Xaml.Media.Imaging;

class ComicQueryManager {

    public ObservableCollection<ComicQuery> AvailableQueries { get; private set; }

    public ObservableCollection<object> CurrentQueryResults { get; private set; }

    public string Title { get; set; }

    public ComicQueryManager() {
        UpdateAvailableQueries();
        CurrentQueryResults = new ObservableCollection<object>();
    }

    private void UpdateAvailableQueries() {
        AvailableQueries = new ObservableCollection<ComicQuery> {
            new ComicQuery("LINQ упрощает запросы", "Пример запроса",
                "Продемонстрируем Джимми гибкость LINQ",
                CreateImageFromAssets("purple_250x250.jpg")),

            new ComicQuery("Дорогие комиксы", "Комиксы дороже $500",
                "Комиксы, цена которых превышает 500 долларов."
                 + " Эта штука позволит Джимми понять, какие комиксы он хочет больше.",
                CreateImageFromAssets("captain_amazing_250x250.jpg")),
        };
    }

    private static BitmapImage CreateImageFromAssets(string imageFilename) {
        return new BitmapImage(new Uri("ms-appx:///Assets/" + imageFilename));
    }

    public void UpdateQueryResults(ComicQuery query) {
        Title = query.Title;

        switch (query.Title) {
            case "LINQ упрощает запросы": LinqMakesQueriesEasy(); break;
            case "Дорогие комиксы": ExpensiveComics(); break;
        }
    }
```

Элемент запросов ListView на главной странице связан со свойством AvailableQueries.

Свойства CurrentQueryResults и Title используются для отображения результатов запросов на странице QueryDetail.

Инициализатор коллекции создает объекты ComicQuery, которые будут отображаться на главной странице.

Этим методом страница QueryDetail запускает запрос LINQ.

Обратите внимание на метод `CreateImageFromAssets()`. Зачем он нужен?

Перед тем как перевернуть страницу и увидеть остальной код класса, подумайте, на что похожи методы `LinqMakesQueriesEasy()` и `ExpensiveComics()`? Приложение вызывает их для запуска запросов LINQ.

```
    public static IEnumerable<Comic> BuildCatalog() {
        return new List<Comic> {
            new Comic { Name = "Johnny America vs. the Pinko", Issue = 6 },
            new Comic { Name = "Rock and Roll (ограниченный выпуск)", Issue = 19 },
            new Comic { Name = "Woman's Work", Issue = 36 },
            new Comic { Name = "Hippie Madness (с опечатками)", Issue = 57 },
            new Comic { Name = "Revenge of the New Wave Freak (поврежден)", Issue = 68 },
            new Comic { Name = "Black Monday", Issue = 74 },
            new Comic { Name = "Tribal Tattoo Madness", Issue = 83 },
            new Comic { Name = "The Death of an Object", Issue = 97 },
        };
    }

    private static Dictionary<int, decimal> GetPrices() {
        return new Dictionary<int, decimal> {
            { 6, 3600M }, { 19, 500M }, { 36, 650M }, { 57, 13525M },
            { 68, 250M }, { 74, 75M }, { 83, 25.75M }, { 97, 35.25M },
        };
    }
```

Это методы BuildCatalog() и GetPrices(), которые вы уже видели несколько страниц назад.

```
    private void LinqMakesQueriesEasy() {
        int[] values = new int[] { 0, 12, 44, 36, 92, 54, 13, 8 };
        var result = from v in values
                     where v < 37
                     orderby v
                     select v;

        CurrentQueryResults.Clear();
        foreach (int i in result)
            CurrentQueryResults.Add(
                  new {
                          Title = i.ToString(),
                          Image = CreateImageFromAssets("purple_250x250.jpg"),
                      }
                );
    }
```

Мы до сих пор не до конца понимаем принцип работы метода CreateImageFromAssets(), но все прояснится на следующей странице.

Каждый из методов выполняет один из представленных ранее запросов LINQ. Результаты не выводятся на консоль, а добавляются в свойство `CurrentQueryResults` коллекции `ObservableCollection<object>`. Но посмотрите на оператор `new { }`. Мы используем ключевое слово с инициализатором объекта. Обычно после него указывается тип, но не в этом случае, поэтому мы получаем экземпляры анонимных типов.

```
    private void ExpensiveComics() {
        IEnumerable<Comic> comics = BuildCatalog();
        Dictionary<int, decimal> values = GetPrices();

        var mostExpensive = from comic in comics
            where values[comic.Issue] > 500
            orderby values[comic.Issue] descending
            select comic;

        CurrentQueryResults.Clear();
        foreach (Comic comic in mostExpensive)
            CurrentQueryResults.Add(
                    new {
                        Title = String.Format("{0} is worth {1:c}",
                                              comic.Name, values[comic.Issue]),
                        Image = CreateImageFromAssets("captain_amazing_250x250.jpg"),
                    }
                );
    }
}
```

Создание анонимных типов

Начиная с главы 3 мы пользовались ключевым словом `new` для создания экземпляров объектов. И каждый раз указывали тип (так `new Guy()` создавал экземпляр типа `Guy`). Но можно обойтись и без этого, получив **анонимный тип**, — корректный тип со свойствами, предназначенными только для чтения, и не имеющий имени. Добавление к нему свойств выполняется через инициализатор объекта.

Вот оператор из запроса `ExpensiveComics` с предыдущей страницы, который создает экземпляр анонимного типа, добавляемый в свойство `CurrentQueryResults` коллекции:

```
new {
    Title = String.Format("{0} is worth {1:c}",
    comic.Name, values[comic.Issue]),
    Image = CreateImageFromAssets("captain_amazing_250x250.jpg"),
}
```

При запуске программы можно увидеть создаваемые им объекты, которые ничем не отличаются от всех прочих. Вот как выглядит экземпляр анонимного типа в окне Watch:

Image	{Windows.UI.Xaml.Media.Imaging.BitmapImage}
Title	"Hippie Madness (misprinted) is worth $13,525.00"

Все работает как и с любым другим инициализатором объектов. Для заполнения свойств объекта можно пользоваться методами инициализатора `CreateImageFromAssets()` и `String.Format()`. (Разумеется, им можно просто присваивать значения.)

Единственное, чего вы *не можете* делать — это ссылаться на тип по имени, так как имя попросту отсутствует! Тут вам пригодится ключевое слово `var`, так как оно позволяет сделать ссылку на анонимный тип:

```
var myAnonymousObject = new {
    Name = "Bob",
    Cash = 186.3M,
    Age = 37,
};
Console.WriteLine(myAnonymousObject.Name);
```

ано-ним-ный, прил. — не идентифицируемый по имени. *Агент Даш Мартин работает под псевдонимом, чтобы остаться* ***анонимным*** *и не дать агентам КГБ себя раскрыть.*

Этот код создает экземпляр анонимного типа, сохраняет ссылку на новый объект в переменной `myAnonymousObject` и использует ее для записи свойства `Name` в результат работы программы.

Переверните страницу, чтобы завершить приложение →

5 **Добавим файлы изображений в папку *Assets* вашего проекта.**

Найдите файлы с изображениями *purple_250x250.jpg* и *captain_amazing_250x250.jpg* (их можно скачать с сайта *http://www.headfirstlabs.com/hfcsharp*) и сохраните их. В окне Solution Explorer щелкните правой кнопкой мыши на папке Assets, выберите в меню Add→Existing Item и добавьте туда файлы. Теперь посмотрим на метод `CreateImageFromAssets`:

```
private static BitmapImage CreateImageFromAssets(string imageFilename) {
     return new BitmapImage(new Uri("ms-appx:///Assets/" + imageFilename));
}
```

Все файлы проекта обладают уникальными именами, находящимися в пространстве имен `ms-appx`. Файл *purple_250x250.jpg* из папки *Assets* имеет уникальный идентификатор *ms-appx:///Assets/purple_250x250.jpg*. Воспользуемся им для загрузки файла в объект `BitmapImage`, и через минуту вы увидите, как объект связывается с элементом <Image> в коде XAML.

6 **Завершим XAML и программный код главной страницы.**

Откройте файл *MainPage.xaml*. Вот ресурсы для страницы:

```
<Page.Resources>
    <local:ComicQueryManager x:Name="comicQueryManager"/>
    <x:String x:Key="AppName">Jimmy's Comics</x:String>
</Page.Resources>
```

Элемент управления ListView автоматически добавляет к спискам полосы прокрутки. Добавьте ко второму атрибуту RowDefinition в определении строк сетки `Height="*"`. Полосы прокрутки исчезнут! Ведь теперь строка расширяется в соответствии с размером элементов списка.

Для компоновки содержимого возьмем `StackPanel`:

```
<Grid Grid.Row="1" Margin="120,0"
    DataContext="{StaticResource ResourceKey=comicQueryManager}">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <TextBlock Style="{StaticResource SubheaderTextStyle}"
        Text="Choose a query to run" Margin="10,0,0,20"/>
    <ListView Grid.Row="1" Margin="0,-10,0,0" ItemsSource="{Binding AvailableQueries}"
        ItemTemplate="{StaticResource Standard130ItemTemplate}"
        SelectionMode="None" IsItemClickEnabled="True" ItemClick="ListView_ItemClick"/>
</Grid>
```

Присвоение свойству SelectionMode значения None отключает возможность выделения списка.

Свойство IsItemClickEnabled заставляет элемент ListView при щелчке на элементе вызывать событие ItemClick.

Добавьте обработчик событий к коду. `SelectionChanged` для элемента ListView может обращаться к выделенным элементам через `e.AddedItems`. ListView связан с `ObservableCollection` объектов `ComicQuery`, поэтому `e.AddedItems[0]` всегда будет содержать `ComicQuery`, на котором щелкнул пользователь. Это передается новой странице в качестве параметра методом `Frame.Navigate()`.

```
private void ListView_ItemClick(object sender, ItemClickEventArgs e) {
    ComicQuery query = e.ClickedItem as ComicQuery;
    if (query != null)
        this.Frame.Navigate(typeof(QueryDetail), query);
}
```

Можно добавить аргумент к методу Frame.Navigate() для передачи странице, на которую вы хотите перейти, объекта в виде параметра.

7 Завершим XAML и код страницы с подробностями запроса.

Откройте файл *QueryDetail.xaml*. Вот ресурсы для этой страницы:

```
<Page.Resources>
    <local:ComicQueryManager x:Name="comicQueryManager"/>
    <x:String x:Key="AppName">Query Detail</x:String>
</Page.Resources>
```

Для компоновки содержимого возьмем еще одну сетку:

```
<Grid Grid.Row="1" Margin="120,0" DataContext="{StaticResource ResourceKey=comicQueryManager}">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/> <RowDefinition/>
    </Grid.RowDefinitions>
    <TextBlock Style="{StaticResource SubheaderTextStyle}"
            Text="Query results" Margin="10,0,0,20"/>
    <ListView Grid.Row="1" Margin="0,-10,0,0" ItemsSource="{Binding CurrentQueryResults}"
            ItemTemplate="{StaticResource Standard130ItemTemplate}" SelectionMode="None"/>
</Grid>
```

На одну строку можно поместить набор тегов <RowDefinition>.

Когда главная страница вызывает метод `Frame.Navigate()` для перехода на страницу с подробностями, она передает объект `ComicQuery` в качестве параметра. Для доступа к этому параметру перекроем метод `OnNavigatedTo()` в программном коде и используем `e.Parameter`:

```
protected override void OnNavigatedTo(NavigationEventArgs e) {
    ComicQuery comicQuery = e.Parameter as ComicQuery;
    if (comicQuery != null) {
        comicQueryManager.UpdateQueryResults(comicQuery);
        pageTitle.Text = comicQueryManager.Title;
    }
    base.OnNavigatedTo(e);
}
```

Свойства объекта ComicQuery и созданный запросами LINQ анонимный тип соответствуют связям в DataTemplate, поэтому элемент ListView отобразит их.

8 Запустим программу! Воспользуемся IDE для анализа принципов ее работы.

Каким же образом программа показывает изображения в каждом элементе ListView? Эти элементы управления используют ItemTemplate `Standard130ItemTemplate`. Через IDE найдите этот шаблон в файле *StandardStyles.xaml*. Это шаблон данных, аналогичный тому, что вы использовали в главе 10:

```
<Border Background="{StaticResource ListViewItemPlaceholderBackgroundThemeBrush}" Width="110" Height="110">
    <Image Source="{Binding Image}" Stretch="UniformToFill" AutomationProperties.Name="{Binding Title}"/>
</Border>
<StackPanel Grid.Column="1" VerticalAlignment="Top" Margin="10,0,0,0">
    <TextBlock Text="{Binding Title}" Style="{StaticResource TitleTextStyle}" TextWrapping="NoWrap"/>
    <TextBlock Text="{Binding Subtitle}" Style="{StaticResource CaptionTextStyle}" TextWrapping="NoWrap"/>
    <TextBlock Text="{Binding Description}" Style="{StaticResource BodyTextStyle}" MaxHeight="60"/>
</StackPanel>
```

Expensive comics
Comics over $500
Comics whose value is over 500 bucks. Jimmy can use this to figure out which comics are most coveted!

Универсальность LINQ

Вы можете не только извлекать отдельные элементы коллекции, но и редактировать их. Сгенерировав результат, LINQ предоставляет набор методов для работы с ним. То есть вы получаете инструменты для управления вашими данными.

Все коллекции реализуют интерфейс **IEnumerable<T>** — обратное неверно. Для коллекции нужно реализовывать еще и интерфейс **ICollection<T>**, то есть методы **Add()**, **Clear()**, **Contains()**, **CopyTo()** и **Remove()**...Разумеется, **ICollection<T>** расширяет **IEnumerable<T>**. LINQ же работает с последовательностями значений или объектов, а не с коллекциями, а значит, вам требуется объект, реализующий **IEnumerable<T>**.

Отредактируем результаты запроса

Добавив в конец каждой строки в этом массиве дополнительную строку, вы **создадите набор** модифицированных строк.

```
string[] sandwiches = { "ham and cheese", "salami with mayo",
                        "turkey and swiss", "chicken cutlet" };
var sandwichesOnRye =
    from sandwich in sandwiches
    select sandwich + " on rye";

foreach (var sandwich in sandwichesOnRye)
Console.WriteLine(sandwich);
```

Добавим строку « on rye» (на ржаном хлебе) ко всем элементам, вошедшим в результат запроса.

Добавив on rye в конец каждой строки, мы положили все элементы сэндвичей на ржаной хлеб.

Результат:

```
ham and cheese on rye
salami with mayo on rye
turkey and swiss on rye
chicken cutlet on rye
```

Изменения касаются результатов запроса... но никак не затрагивают элементы исходной коллекции или базы данных.

Вычисления внутри коллекции

Помните, мы говорили, что LINQ обеспечивает коллекции методами расширения? Некоторые из них весьма полезны.

```
Random random = new Random();
List<int> listOfNumbers = new List<int>();
int length = random.Next(50, 150);
for (int i = 0; i < length; i++)
    listOfNumbers.Add(random.Next(100));

Console.WriteLine("Есть {0} чисел",
                  listOfNumbers.Count());
Console.WriteLine("Самое маленькое {0}",
                  listOfNumbers.Min());
Console.WriteLine("Самое большое {0}",
                  listOfNumbers.Max());
Console.WriteLine("Их сумма равна {0}",
                  listOfNumbers.Sum());
Console.WriteLine("Среднее арифметическое {0:F2}",
                  listOfNumbers.Average());
```

Ни один из этих методов не имеет отношения к .NET... все они определены в LINQ.

Это методы расширения для IEnumerable<T> в пространстве имен System.Linq, где используется статический класс Enumerable. Щелкните на любом из них правой кнопкой мыши и выберите команду Go to Definition, чтобы убедиться в этом лично.

Последовательностью называется упорядоченный набор объектов или значений, которые LINQ возвращает в IEnumerable<T>.

Сохраните результат в виде последовательности

Иногда нужно, чтобы результаты выполнения запроса LINQ были под рукой. Воспользуемся для этого командой `ToList()`:

```
var under50sorted =
  from number in listOfNumbers
  where number < 50
  orderby number descending
  select number;

List<int> newList = under50sorted.ToList();
```

На этот раз числа сортируются по убыванию.

Метод ToList() преобразует переменную типа var в объект List<T>, давая вам возможность сохранить результаты запроса. Аналогичную функцию выполняют методы ToArray() и ToDictionary().

С помощью метода `Take()` можно сформировать подмножество результатов:

```
var firstFive = under50sorted.Take(5);

List<int> shortList = firstFive.ToList();
foreach (int n in shortList)
    Console.WriteLine(n);
```

Метод Take() берет указанное число элементов из результатов запроса. Их можно поместить в другую переменную типа var, а потом преобразовать в список.

Будьте осторожны!

Запросы LINQ не выполняются заранее!

Это называется «отложенными вычислениями» — сначала должен сработать оператор, использующий результаты запроса. Поэтому так важен метод ToList() — он заставляет LINQ немедленно выполнить запрос.

Посетите официальную страницу 101 LINQ Samples от Microsoft

Здесь вы найдете дополнительные материалы по работе с LINQ:

http://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b

В: Так много новых слов: `from`, `where`, `orderby`, `select`... как будто совершенно другой язык. Почему он так отличается от C#?

О: Потому что он служит другой цели. Большая часть синтаксиса C# предназначена для выполнения одной небольшой операции за один раз. Можно начать цикл, присвоить значение переменной, произвести математический расчет или вызвать метод... все это будут единичные операции.

Единичный же запрос LINQ может выполнять целый ряд функций. Рассмотрим пример:

```
var under10 =
 from number in numberArray
 where number < 10
 select number;
```

Несмотря на кажущуюся простоту, это довольно сложный фрагмент кода. Подумайте, сколько действий должна совершить программа для выбора из массива numberArray всех элементов, не превышающих 10. Нужно циклически просмотреть массив, сравнить каждый его элемент с 10 и вывести результат в форме, пригодной для дальнейшего применения. Именно поэтому LINQ выглядит так странно с точки зрения C#. Ведь многочисленным операциям соответствует совсем короткая запись.

LINQ позволяет коротко писать очень сложные запросы.

Это принцип разделения ответственности. Можно добавлять запросы, редактируя класс ComicQueryManager и не касаясь ни XAML, ни кода программной части, так как код всех запросов инкапсулирован в этом классе.

Новые запросы

Джимми интересно, как LINQ поможет в управлении его данными. Добавьте к приложению три запроса с предыдущей страницы, чтобы продемонстрировать возможности LINQ. Вам нужно обновить класс `ComicQueryManager` (и добавить изображение в папку *Assets*). Добавьте три объекта `ComicQuery` к инициализатору объекта для свойства `AvailableQueries`:

Упражнение!

```
    private void UpdateAvailableQueries() {
        AvailableQueries = new ObservableCollection<ComicQuery> {
            new ComicQuery("LINQ упрощает запросы", "Пример запроса",
                "Продемонстрируем Джимми гибкость LINQ",
                CreateImageFromAssets("purple_250x250.jpg")),

            new ComicQuery("Дорогие комиксы", "Комиксы дороже $500",
                "Комиксы, цена которыз превышает 500 долларов."

                    + " Эта штука позволит Джимми понять, какие комиксы он хочет больше",
                CreateImageFromAssets("captain_amazing_250x250.jpg")),

            new ComicQuery("Универсальный LINQ 1", "Изменим все возвращаемое запросами",
                "Этот код добавит строку в конец каждой строки в массиве.",
                CreateImageFromAssets("bluegray_250x250.jpg")),

            new ComicQuery("Универсальный LINQ 2", "Вычисления в коллекциях",
                "LINQ предоставляет методы расширения коллекциям (и всему прочему"
                  + " что реализует интерфейс IEnumerable<T>).",
                CreateImageFromAssets("purple_250x250.jpg")),

            new ComicQuery("Универсальный LINQ 3",
                "Сохраним все результаты в новую последовательность",
                "Иногда результаты запросов LINQ нужно сохранить отдельно.",
                CreateImageFromAssets("bluegray_250x250.jpg")),
        };
    }
```

Добавьте эти три запроса, чтобы они появились на главной странице.

Теперь нужно обновить оператор `switch`, чтобы он начал запускать выбираемые в элементе управления ListView запросы:

```
public void UpdateQueryResults(ComicQuery query) {
    Title = query.Title;

    switch (query.Title) {
        case "LINQ упрощает запросы": LinqMakesQueriesEasy(); break;
        case "Дорогие комиксы": ExpensiveComics(); break;
        case "Универсальный LINQ 1": LinqIsVersatile1(); break;
        case "Универсальный LINQ 2": LinqIsVersatile2(); break;
        case "Универсальный LINQ 3": LinqIsVersatile3(); break;
    }
}
```

Добавьте эти три строки в оператор switch. Они будут выполняться при переходе на страницу с деталями запроса.

Следует добавить эти три метода. Сравните их с запросами LINQ на двух предыдущих страницах:

```
private void LinqIsVersatile1() {
    string[] sandwiches = { "ham and cheese", "salami with mayo",
             "turkey and swiss", "chicken cutlet" };
    var sandwichesOnRye =
        from sandwich in sandwiches
        select sandwich + " on rye";

    CurrentQueryResults.Clear();
    foreach (var sandwich in sandwichesOnRye)
        CurrentQueryResults.Add(CreateAnonymousListViewItem(sandwich, "bluegray_250x250.jpg"));
}

private void LinqIsVersatile2() {
    Random random = new Random();
    List<int> listOfNumbers = new List<int>();
    int length = random.Next(50, 150);
    for (int i = 0; i < length; i++)
        listOfNumbers.Add(random.Next(100));

    CurrentQueryResults.Clear();
    CurrentQueryResults.Add(CreateAnonymousListViewItem(
        String.Format("There are {0} numbers", listOfNumbers.Count())));
    CurrentQueryResults.Add(
        CreateAnonymousListViewItem(String.Format("The smallest is {0}", listOfNumbers.Min())));
    CurrentQueryResults.Add(
       CreateAnonymousListViewItem(String.Format("The biggest is {0}", listOfNumbers.Max())));
    CurrentQueryResults.Add(
       CreateAnonymousListViewItem(String.Format("The sum is {0}", listOfNumbers.Sum())));
    CurrentQueryResults.Add(CreateAnonymousListViewItem(
            String.Format("The average is {0:F2}", listOfNumbers.Average())));
}

private void LinqIsVersatile3() {
    List<int> listOfNumbers = new List<int>();
    for (int i = 1; i <= 10000; i++)
        listOfNumbers.Add(i);

    var under50sorted =
            from number in listOfNumbers
            where number < 50
            orderby number descending
            select number;

    var firstFive = under50sorted.Take(6);

    List<int> shortList = firstFive.ToList();
    foreach (int n in shortList)
       CurrentQueryResults.Add(CreateAnonymousListViewItem(n.ToString(), "bluegray_250x250.jpg"));
}
```

Скачайте файл bluegray_250x250.jpg с сайта Head First Labs и добавьте его в папку Assets.

Упражнение!

Микро Упражнение

Чтобы код заработал, требуется еще один метод. Каждый из трех методов `LinqIsVersatile` вызывает метод `CreateAnonymousListViewItem()`. Первым его параметром является заголовок, который следует использовать как свойство `Title` нового анонимного объекта. Второй параметр **необязателен**. Это имя файл, загруженного в свойство `Image` анонимного объекта. Если его опустить, будет взято значение по умолчанию *purple_250x250.jpg*. Попробуйте написать такой метод. Ответ дан на следующей странице.

КЛЮЧЕВЫЕ МОМЕНТЫ

- **from** указывает IEnumerable<T>, что мы осуществляем запрос. За ним всегда следует имя переменной, потом **in** и имя входных данных (from value in values).
- **where** в общем случае следует за from. Это предложение использует условные операторы C# для фильтрации элементов (where value < 10).
- **orderby** указывает порядок сортировки результата. За ним следует критерий сортировки и иногда ключевое слово **descending** (orderby value descending).
- **select** указывает, что входит в конечный результат (select value).
- **Take** позволяет получить первые несколько результатов запроса (results.Take(10)). Для каждой последовательности в LINQ существуют и другие методы: Min(), Max(), Sum() и Average().
- Предложение select работает не только с именем, созданным в предложении from. Скажем, если запрос выбирает цены из массива значений int, которому в предложении from присвоили имя value, строку с ценами можно вернуть так: select String.Format("{0:c}", value).

Это напоминает запись {0:x}, которую вы использовали в главе 9 при построении шестнадцатиричного дампа. Есть еще варианты {0:d} и {0:D} для длинной и короткой дат, и {0:P} или {0:Pn} для вывода процентов (с n десятичных знаков).

Микро Упражнение Решение

```
private object CreateAnonymousListViewItem(string title,
                         string imageFilename = "purple_250x250.jpg") {
    return new {
        Title = title,
        Image = CreateImageFromAssets(imageFilename),
    };
}
```

Это необязательный параметр. Если его опустить, загрузится пурпурный прямоугольник.

Часто задаваемые вопросы

В: Как работает предложение from?

О: Оно напоминает первую строчку цикла foreach. Запросы LINQ поначалу сложно воспринимать, так как они соответствуют нескольким операциям.

Запрос делает одно и то же с каждым элементом коллекции. Предложение from, во-первых, указывает, к какой коллекции осуществляется запрос, во-вторых, присваивает этой коллекции имя.

Новое имя создается почти так же, как в случае цикла foreach. Вот первая строчка этого цикла:

```
foreach (int i in values)
```

Он на время создает переменную i, которой по очереди присваиваются элементы коллекции values. А теперь посмотрим на предложение from в аналогичной ситуации:

```
from i in values
```

Предложение тоже создает временную переменную i и по очереди присваивает ей элементы коллекции values. Цикл foreach запускает для каждого из элеметов расположенный снизу блок кода, в то время как запрос LINQ применяет к каждому элементу критерии из предложения where. Но следует помнить, что запросы LINQ — это всего лишь методы расширения. Всю работу делают вызываемые ими методы, которые вы можете вызвать, и не прибегая к LINQ.

В: Как LINQ определяет, что включать в результат?

О: Это определяет предложение select. Каждый запрос возвращает последовательность из элементов одного типа. При этом четко указывается, что она должна содержать. Если запрос делается к массиву или коллекции элементов одного типа, результат очевиден. А что делать при запросе к списку объектов Comic? Можно выбрать весь класс, как это сделал Джимми. А можно заменить последнюю строчку запроса на select comic.Name и получить результат в виде набора строк. А написав select comic.Issue, вы получите последовательность целых чисел.

Магниты LINQ

Расположите магниты таким образом, чтобы получить показанный в нижней части страницы результат.

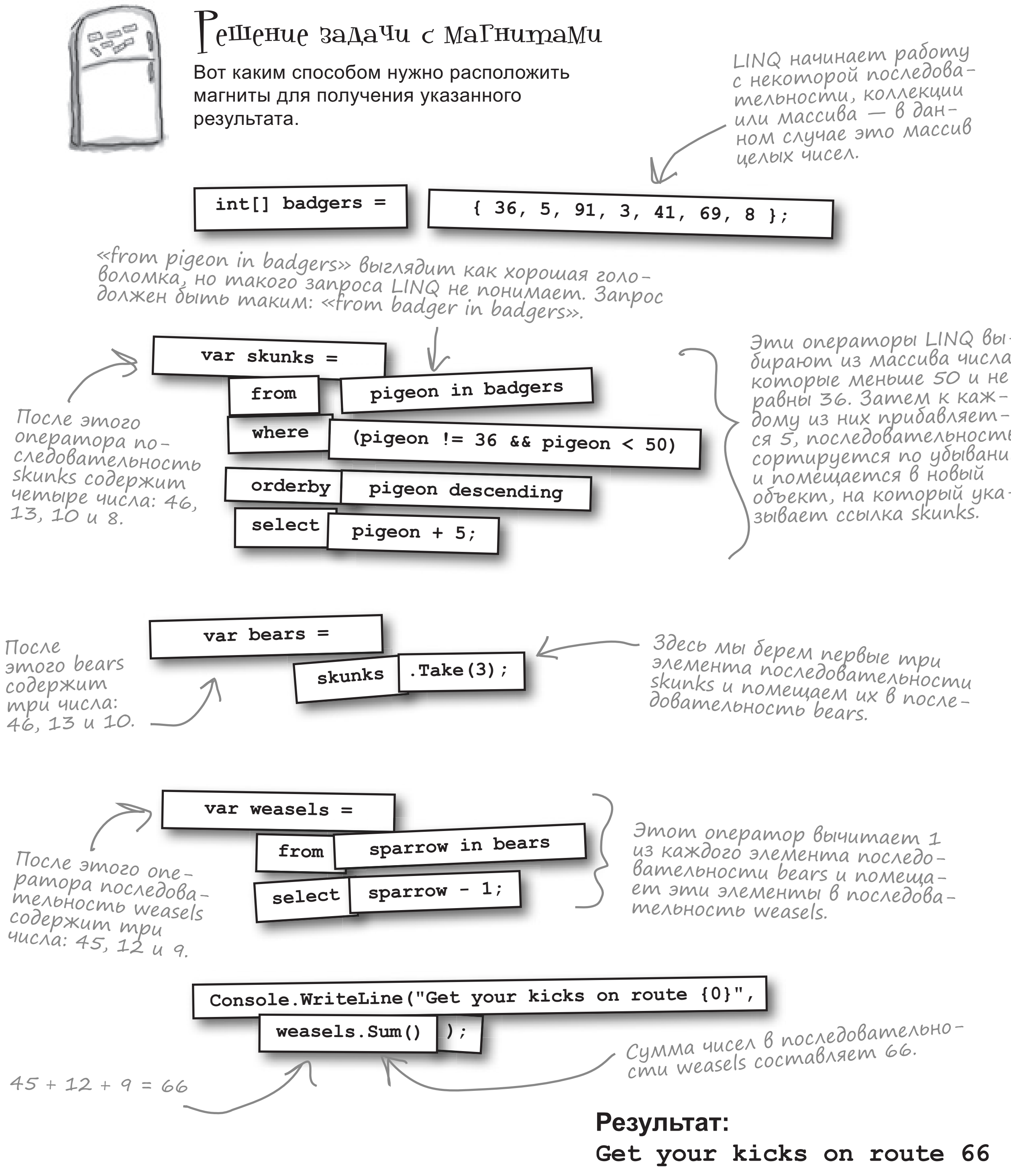
```
pigeon descending
Console.WriteLine("Get your kicks on route {0}",
weasels.Sum()
sparrow in bears
pigeon in badgers
where
from
from
orderby
select
select
var weasels =
int[] badgers =
var skunks =
var bears =
);
skunks
.Take(3);
pigeon + 5;
sparrow - 1;
(pigeon != 36 && pigeon < 50)
{ 36, 5, 91, 3, 41, 69, 8 };
```

Результат:

```
Get your kicks on route 66
```

Решение задачи с магнитами

Вот каким способом нужно расположить магниты для получения указанного результата.



Результат:
Get your kicks on route 66

Группировка результатов запроса

Вы уже знаете, что LINQ позволяет разбивать результаты запроса на группы, так как именно это вы делали в симуляторе улья. Посмотрим более подробно на то, как это работает.

Посмотреть на запрос LINQ в действии (и больше узнать о принципах работы приложений WinForms) вы можете, создав анимацию симулятора. Скачайте бесплатную главу GDI+ со страницы http://headfirstlabs.com/hfcsharp.

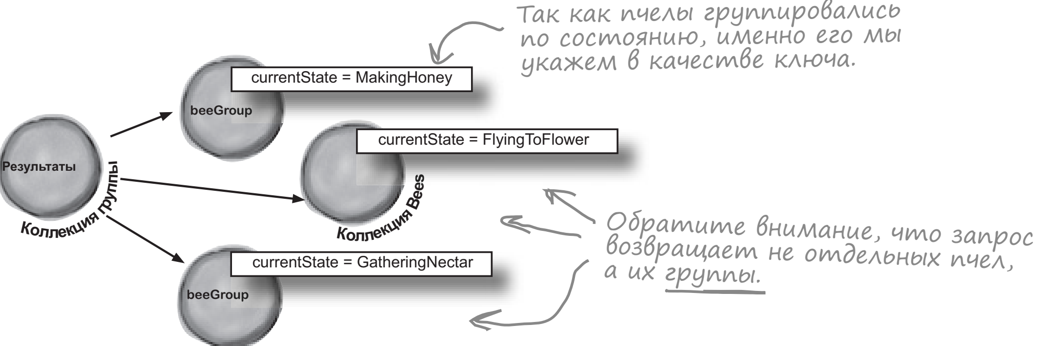
```
var beeGroups =
  from bee in world.Bees
  group bee by bee.CurrentState
  into beeGroup
  orderby beeGroup.Key
  select beeGroup;
```

1. Начало запроса ничем не отличается от других — вы извлекаете отдельных пчел из коллекции `world.Bees` объекта `List<Bee>`.

2. Следующая строка запроса содержит новое ключевое слово **`group`**. Оно приказывает запросу возвращать ***группы*** пчел. То есть вместо одной последовательности у нас будет целый набор. Запись **`group bee by bee.CurrentState`** говорит, что пчел нужно разбить на группы в соответствии со свойством `CurrentState`. Ну и наконец, мы указываем имя новых групп: **`into beeGroup`**.

3. Теперь, когда группы готовы, ими можно управлять. Например, при помощи предложения `orderby` упорядочить их по значениям перечисления `CurrentState` (`Idle`, `FlyingToFlower` и т. п.). Строчка **`orderby beeGroup.Key`** сортирует последовательности по ключу. В качестве ключа в данном случае будет использоваться свойство `CurrentState`.

4. Теперь нужно при помощи ключевого слова `select` указать, какой результат возвращает запрос. В данном случае указывается имя группы: **`select beeGroup;`**



Сгруппируем результаты Джимми

Джимми покупает много дешевых комиксов, чуть меньше комиксов средней ценовой категории и несколько дорогих, и он хочет научиться оценивать свои финансовые возможности перед покупкой. Он помещает цены из каталога Грега в перечисление `Dictionary<int, int>` при помощи метода `GetPrices()`. Воспользуемся LINQ, чтобы разбить их на три группы: комиксы со стоимостью до $100, со стоимостью от $100 до $1000 и комиксы, стоящие дороже $1000. Мы создадим перечисление `PriceRange`, которое будет использоваться в качестве ключа, и метод `EvaluatePrice()`, определяющий цену и возвращающий перечисление `PriceRange`.

В качестве ключа используем перечисление

Ключом группы выступает некое общее свойство. Это может быть строка, число или даже ссылка на объект. В нашем случае каждая группа, которую возвращает запрос, будет последовательностью из номеров выпусков, ключом же группы станет перечисление `PriceRange`. Метод `EvaluatePrice()` будет возвращать это перечисление на основе такого параметра, как цена:

```
enum PriceRange { Cheap, Midrange, Expensive }

static PriceRange EvaluatePrice(decimal price) {
    if (price < 100M) return PriceRange.Cheap;
    else if (price < 1000M) return PriceRange.Midrange;
    else return PriceRange.Expensive;
}
```

Добавьте код с этой страницы в новое консольное приложение и посмотрите, сможете ли вы заставить его работать! В конце этой главы мы добавим данный запрос для магазина Windows.

Сгруппируем комиксы по ценовым категориям

Запрос вернет **последовательность последовательностей**. Каждая из них будет иметь свойство `Key`, совпадающее с элементом перечисления `PriceRange`, возвращенным методом `EvaluatePrice()`. Обратите внимание на предложение `group by` — мы выбираем из словаря пары и используем для них имя `pair`: `pair.Key` — это номер выпуска, а `pair.Value` — его цена. Запись `group pair.Key` объединяет в группы номера выпусков, ориентируясь на их цену:

```
Dictionary<int, decimal> values = GetPrices();

var priceGroups =
    from pair in values
    group pair.Key by EvaluatePrice(pair.Value)
        into priceGroup
        orderby priceGroup.Key descending
        select priceGroup;

foreach (var group in priceGroups) {
    Console.Write("Найдены {0} {1} комикса: выпуски ", group.Count(), group.Key);
    foreach (var issueNumber in group)
        Console.Write(issueNumber.ToString() + " ");
    Console.WriteLine();
}
```

Запрос определяет, к какой группе относится выбранный комикс, передавая его цену методу EvaluatePrice(). Метод же возвращает перечисление PriceRange, используемое в качестве ключа группы.

Каждая из групп является последовательностью, поэтому мы добавили внутренний цикл foreach для просмотра цен внутри группы.

Результат:

```
Найдены 2 дорогих комикса: выпуски 6 57
Найдены 3 комикса по средней цене: выпуски 19 36 68
Найдены 3 дешевых комикса: выпуски 74 83 97
```

Ребус в бассейне

Поместите фрагменты кода из бассейна на пустые строчки. Каждый фрагмент может быть использован несколько раз. Есть и лишние фрагменты. Вам нужно получить следующий ***результат***:

Horses enjoy eating carrots, but they love eating apples.

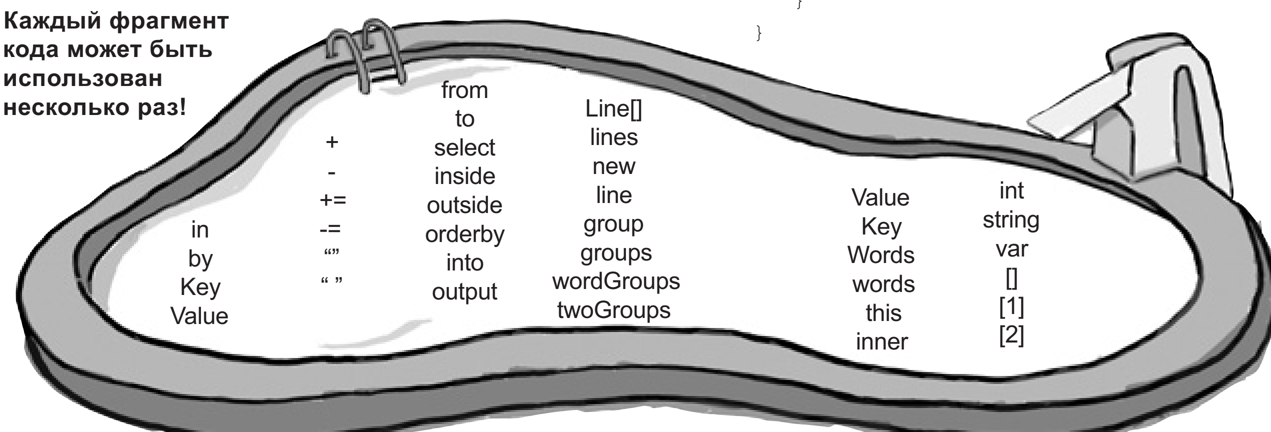
```
class Line {
    public string[] Words;
    public int Value;
    public Line(string[] Words, int Value) {
        this.Words = Words; this.Value = Value;
    }
}
```

Подсказка: LINQ сортирует строки в алфавитном порядке.

```
Line[] lines = {
  new Line( new string[] { "eating", "carrots,",
          "but", "enjoy", "Horses" } , 1),
  new Line( new string[] { "zebras?", "hay",
         "Cows", "bridge.", "bolted" } , 2),
  new Line( new string[] { "fork", "dogs!",
         "Engine", "and" }, 3 ) ,
  new Line( new string[] { "love", "they",
         "apples.", "eating" }, 2 ) ,
  new Line( new string[] { "whistled.", "Bump" }, 1 ) };
```

```
var ______ =
    from ______ in ______
    ______ line by line.______
    into wordGroups
    orderby ________.______
    select ________;

____ _________ = words.______(2);

foreach (var group in twoGroups)
{
   int i = 0;
   foreach (______ inner in ______) {
      i++;
      if (i == ______.Key) {
         var poem =
            ______ word in ______.______
            ________ word descending
            ______ word + ____;
         foreach (var word in ______)
            Console.Write(word);
      }
   }
}
```

Каждый фрагмент кода может быть использован несколько раз!



Решение ребуса в бассейне

```
class Line {
    public string[] Words;
    public int Value;
    public Line(string[] Words, int Value) {
        this.Words = Words; this.Value = Value;
    }
}

Line[] lines = {
  new Line( new string[] { "eating", "carrots,", "but", "enjoy", "Horses" } , 1),
  new Line( new string[] { "zebras?", "hay", "Cows", "bridge.", "bolted" } , 2),
  new Line( new string[] { "fork", "dogs!", "Engine", "and" }, 3 ) ,
  new Line( new string[] { "love", "they", "apples.", "eating" }, 2 ) ,
  new Line( new string[] { "whistled.", "Bump" }, 1 )
};

var words =
    from line in lines
    group line by line.Value
    into wordGroups
    orderby wordGroups.Key
    select wordGroups;

var twoGroups = words.Take(2);

foreach (var group in twoGroups)
{
    int i = 0;
    foreach (var inner in group) {
        i++;
        if (i == group.Key) {
            var poem =
                from word in inner.Words
                orderby word descending
                select word + " ";
            foreach (var word in poem)
                Console.Write(word);
        }
    }
}
```

Первый запрос делит объекты Line из массива lines[] на группы в соответствии со значением поля Value, располагая их по возрастанию.

Первые две группы — это строки со значениями Value 1 и 2.

Этот цикл запрашивает первый объект Line в первой группе и второй объект Line во второй группе.

Вы поняли, почему выражения «Horses enjoy eating carrots, but» («Лошади получают удовольствие от моркови, но») и «they love eating apples» («они любят яблоки») расположены в алфавитном порядке по убыванию?

Результат: **Horses enjoy eating carrots, but they love eating apples.**

Предложение join

Джимми собрал целую коллекцию комиксов и хотел бы сравнить цены на них с ценами в каталоге Грега, чтобы понять, не переплатил ли он. Для записи своих расходов он создал класс `Purchase` с двумя автоматическими свойствами `Issue` и `Price`. Перечень купленных им комиксов находится в коллекции `List<Purchase>`, которая называется `purchases`. Как же ему теперь осуществить сравнение с ценами из каталога Грега?

Предложение `join` позволит **скомбинировать данные из двух коллекций** в единый запрос. Это делается путем поиска в первой коллекции совпадающих значений со второй. (LINQ делает это эффективно — сравнивает только те пары, которые нужно.) В качестве результата выводятся совпадающие пары.

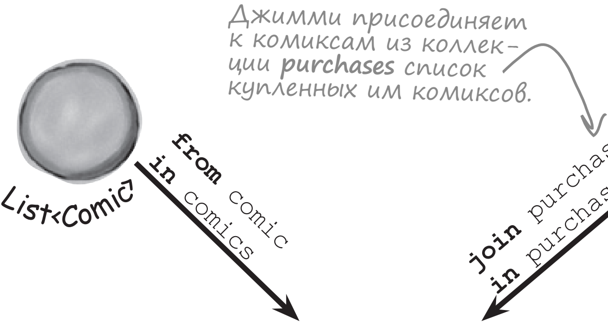
Данные Джимми представлены в виде коллекции объектов Purchase, которая называется **purchases**.

```
class Purchase {
  public int Issue
            { get; set; }
  public decimal Price
            { get; set; }
}
```

1 После предложения `from` вместо критерия отбора результатов напишите:

join *name* **in** *collection*

Имя *name* назначается членам, которые извлекаются из объединенной коллекции на каждой итерации цикла. Затем вы используете его в предложении `where`.



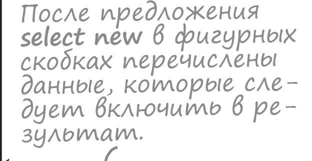
List<Purchases>

join purchase
in purchases

```
on comic.Issue
equals purchase.Issue
```

2 Добавим предложение **on**, указывающее способ объединения коллекций. Затем укажем имя первой коллекции, ключевое слово **equals** и имя второй коллекции.

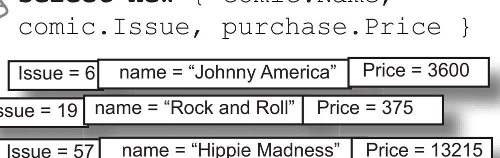
3 Затем следуют предложения `where` и `orderby`. Так как в результат обычно требуется включить часть данных из одной коллекции, а часть из другой, в конце используется предложение **select new**, создающее пользовательский набор результатов при помощи **анонимного типа**.




```
select new { comic.Name,
comic.Issue, purchase.Price }
```

результат

LINQ Sequence



Джимми изрядно сэкономил

Кажется, Джимми заключает выгодную сделку. Он создал список классов `Purchase`, в котором перечислены его покупки, и сравнил его с ценами из каталога Грега.

Сначала Джимми создал дополнительную коллекцию

Он использовал свой старый метод `BuildCatalog()`. Джимми осталось только написать метод `FindPurchases()` и построить коллекцию классов `Purchase`.

```
public static IEnumerable<Purchase> FindPurchases() {
    List<Purchase> purchases = new List<Purchase>() {
        new Purchase() { Issue = 68, Price = 225M },
        new Purchase() { Issue = 19, Price = 375M },
        new Purchase() { Issue = 6, Price = 3600M },
        new Purchase() { Issue = 57, Price = 13215M },
        new Purchase() { Issue = 36, Price = 660M },
    };
    return purchases;
}
```

Это статический метод из класса Purchase.

За выпуск #57 Джимми заплатил $13,215.

Все готово для слияния!

Часть этого запроса вы уже видели. Осталось добавить к нему недостающий фрагмент.

```
IEnumerable<Comic> comics = BuildCatalog();
Dictionary<int, decimal> values = GetPrices();
IEnumerable<Purchase> purchases = FindPurchases();
var results =
    from comic in comics
    join purchase in purchases
    on comic.Issue equals purchase.Issue
    orderby comic.Issue ascending
    select new { comic.Name, comic.Issue, purchase.Price };
decimal gregsListValue = 0;
decimal totalSpent = 0;
foreach (var result in results) {
    gregsListValue += values[result.Issue];
    totalSpent += result.Price;
    Console.WriteLine("Выпуск #{0} ({1}) куплен за {2:c}",
              result.Issue, result.Name, result.Price);
}
Console.WriteLine("Я потратил {0:c} на комиксы, стоящие {1:c}",
              totalSpent, gregsListValue);
```

Предложение join инициирует сравнение каждого элемента из коллекции comics с элементами коллекции purchases для поиска тех, у которых comic.Issue совпадает с purchase.Issue.

Предложение select создает результат, комбинируя значения Name и Issue члена comic со значением Price от члена purchase.

Джимми счастлив, что он знает LINQ, так как это позволило ему подсчитать, сколько же он сэкономил.

Результат:

```
Выпуск #6 (Johnny America vs. the Pinko) куплен за $3,600.00
Выпуск #19 (Rock and Roll (limited edition)) куплен за $375.00
Выпуск #36 (Woman's Work) куплен за $660.00
Выпуск #57 (Hippie Madness (misprinted)) куплен за $13 215.00
Выпуск #68 (Revenge of the New Wave Freak (damaged)) куплен за
$225.00
Я потратил $18,075.00 на комиксы, стоящие $18 525.00
```

КЛЮЧЕВЫЕ МОМЕНТЫ

- Предложение **group** разбивает результат на группы — только создает последовательность из последовательностей.
- Каждая группа содержит член, являющийся общим для всех остальных членов. Он называется **ключом** и задается ключевым словом **by**. Каждая последовательность обладает членом **Key**, содержащим ключ группы.
- Ключевые слова **on** ... **equals** задают критерий сравнения в предложении **join**.
- Предложение **join** объединяет две коллекции в одном запросе. Члены обеих коллекций сравниваются друг с другом и из совпадающих пар формируется результат.
- В результаты запроса **join** обычно требуется выборочно включить элементы из обеих коллекций. Это реализуется при помощи предложения **select**.
- Предложение `select new` позволяет сконструировать пользовательский запрос LINQ, результаты которого включают только элементы, которые мы хотим видеть в итоговой последовательности.

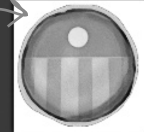
Упражнение

Добавьте к приложению Джимми два последних запроса LINQ.

Добавим объекты ComicQuery к методу UpdateAvailableQueries().
Обновите инициализатор объекта `AvailableQueries`, чтобы создать экземпляры двух новых объектов ComicQuery, которые нужно добавить на главную страницу. Вот как должны выглядеть новые кнопки:

Group comics by price range
Combine Jimmy's values into groups
Jimmy buys a lot of cheap comic books, some midrange comic books, and a few expensive ones, and he wants to know what his options are before he decides what comics to buy.

Можете сделать так, чтобы в заголовке страницы Джимми видел, сколько он потратил и сколько стоили комиксы?



Join purchases with prices
Let's see if Jimmy drives a hard bargain
This query creates a list of Purchase classes that contain Jimmy's purchases and compares them with the prices he found on Greg's List.

Добавим метод, выполняющий запросы и обновляющий результат.
Вам потребуется класс `Purchases` и метод `EvaluatePrice()`, которые рассматривались ранее. Не забудьте добавить перечисление `PriceRange`. Добавьте `EvaluatePrice()` в класс `Purchases` как статический метод.

Добавим в метод UpdateQueryResults() новые запросы.
Как только два новых метода будут добавлены к оператору `switch` в методе `UpdateQueryResults`, дело можно считать сделанным.

Часто задаваемые вопросы

В: Я так и не понял, как работает предложение join.

О: Предложение `join` работает с двумя последовательностями. Предположим, у вас есть коллекция футбольных игроков `players`. Ее элементами являются объекты со свойствами `Name`, `Position` и `Number`. Выбрать игроков, на футболке которых номер больше 10, можно запросом:

```
var results =
 from player in players
 where player.Number > 10
 select player;
```

У нас есть и коллекция jerseys, элементы которой обладают свойствами Number и Size. Чтобы определить размер футболки каждого игрока, запишем:

```
var results =
 from player in players
 where player.Number > 10
 join shirt in jerseys
    on player.Number
    equals shirt.Number
 select shirt;
```

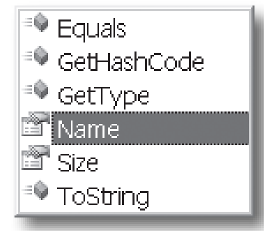
В: Этот запрос даст мне множество футболок. А как быть, если меня не волнуют номера игроков, но хотелось бы узнать размер футболки каждого из них?

О: Здесь вам пригодятся анонимные типы — в них можно положить любые нужные вам данные. Они позволяют и выбирать из объединенных коллекций. И ничто не мешает вам выбрать только имена игроков и размеры их футболок:

```
var results =
 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 в состоянии самостоятельно разобраться с результатом, который выдает запрос. При создании цикла, нумерующего результаты, сразу после ввода переменной появится окно IntelliSense со списком.

```
foreach (var r in results)
   r.
```



В списке присутствуют свойства Name и Size. Добавленные к предложению `select` дополнительные пункты тоже появятся в этом списке. Это связано с тем, что запрос создает различные анонимные типы для различных членов.

В: Объясните, пожалуйста, что означает `var`.

О: Ключевое слово `var` решает сложную проблему, возникающую в LINQ. Обычно при вызове метода или выполнении оператора сразу ясно, с каким типом данных вы работаете. К примеру, если метод возвращает значения типа `string`, результат его работы нужно сохранить в переменную или поле именно этого типа.

А вот запрос LINQ может вернуть данные анонимного типа, который нигде не определен. Вы знаете, что это какая-то последовательность. Но тип содержащихся в ней объектов полностью зависит от содержания запроса LINQ. Например, рассмотрим вот такой запрос:

```
var mostExpensive =
  from comic in comics
  where values[comic.
Issue] > 500
  orderby values[comic.
Issue] descending
  select comic;
```

Если вместо последней строчки написать:

```
  select new
   { Name = comic.Name,
     IssueNumber = "#" +
comic.Issue };
```

запрос вернет данные анонимного типа с двумя членами — строкой Name и строкой IssueNumber. Но определение класса для этого типа в нашей программе отсутствует, при том что переменная mostExpensive должна быть объявлена как принадлежащая к какому-то типу.

Здесь на помощь приходит ключевое слово `var`, которое как бы объясняет компилятору: «Это корректный тип, просто мы пока не знаем, какой именно. Определи это, пожалуйста, самостоятельно»..

Вот код, который нужно добавить в приложение Джимми, чтобы задействовать два последних запроса LINQ.

```
private void UpdateAvailableQueries() {
    AvailableQueries = new ObservableCollection<ComicQuery> {
        new ComicQuery("LINQ упрощает запросы", "Пример запроса",
            "Продемонстрируем Джимми гибкость LINQ",
            CreateImageFromAssets("purple_250x250.jpg")),

        new ComicQuery("Дорогие комиксы", "Комиксы дороже $500",
            "Комиксы, цена которых превышает 500 долларов."
                + "Эта штука позволит Джимми понять, какие комиксы он хочет больше.",
            CreateImageFromAssets("captain_amazing_250x250.jpg")),

        new ComicQuery("Универсальный LINQ 1", "Изменим все возвращаемое запросами",
            "Этот код добавит строку в конец каждой строки в массиве.",
            CreateImageFromAssets("bluegray_250x250.jpg")),

        new ComicQuery("Универсальный LINQ 2", "Вычисления в коллекциях",
            "LINQ предоставляет методы расширения коллекциям (и всему прочему"
              + " что реализует интерфейс IEnumerable<T>).",
            CreateImageFromAssets("purple_250x250.jpg")),

        new ComicQuery("Универсальный LINQ 3",
            "Сохраним все результаты в новую последовательность",
            "Иногда результаты запросов LINQ нужно сохранять отдельно.",
            CreateImageFromAssets("bluegray_250x250.jpg")),

        new ComicQuery("Группировка комиксов по диапазону цен",
            "Объединим данные Джимми в группы",
            "Джимми покупает много дешевых комиксов, несколько более дорогих и"
                 + " немного дорогих и хочет знать, какие варианты ему доступны"
                 + " перед окончательной покупкой.",
                CreateImageFromAssets("captain_amazing_250x250.jpg")),

        new ComicQuery("Объединим покупки с ценами",
            "Посмотрим, умеет ли Джимми торговаться",
            "Этот запрос создает список классов Purchase с приобретениями Джимми"
                 + " и сравнивает их с ценами в списке Грега.",
            CreateImageFromAssets("captain_amazing_250x250.jpg")),
    };
}
```

Добавление этих двух объектов ComicQuery к инициализатору объекта AvailableQueries делает их видимыми на главной странице.

Упражнение Решение

```
public void UpdateQueryResults(ComicQuery query) {
    Title = query.Title;

    switch (query.Title) {
        case "LINQ упрощает запросы": LinqMakesQueriesEasy(); break;
        case "Дорогие комиксы": ExpensiveComics(); break;
        case "Универсальный LINQ 1": LinqIsVersatile1(); break;
        case "Универсальный LINQ 2": LinqIsVersatile2(); break;
        case "Универсальный LINQ 3": LinqIsVersatile3(); break;
        case "Группировка комиксов по диапазону цен":
             CombineJimmysValuesIntoGroups();
             break;
        case "Объединим покупки с ценами":
             JoinPurchasesWithPrices();
             break;
    }
}
```

Это новые варианты для оператора switch, вызывающего методы запросов.

Не забудьте перечисление PriceRange.

```
enum PriceRange { Cheap, Midrange, Expensive }

class Purchase {
    public int Issue { get; set; }
    public decimal Price { get; set; }

    public static IEnumerable<Purchase> FindPurchases()
    {
        List<Purchase> purchases = new List<Purchase>() {
            new Purchase() { Issue = 68, Price = 225M },
            new Purchase() { Issue = 19, Price = 375M },
            new Purchase() { Issue = 6, Price = 3600M },
            new Purchase() { Issue = 57, Price = 13215M },
            new Purchase() { Issue = 36, Price = 660M },
        };
        return purchases;
    }

    public static PriceRange EvaluatePrice(decimal price)
    {
        if (price < 100M) return PriceRange.Cheap;
        else if (price < 1000M) return PriceRange.Midrange;
        else return PriceRange.Expensive;
    }
}
```

Это класс Purchase. Теперь EvaluatePrice() — статический метод этого класса.

```
private void CombineJimmysValuesIntoGroups() {
    Dictionary<int, decimal> values = GetPrices();
    var priceGroups =
        from pair in values
        group pair.Key by Purchase.EvaluatePrice(pair.Value)
            into priceGroup
            orderby priceGroup.Key descending
            select priceGroup;
    foreach (var group in priceGroups) {
        string message = String.Format("Найдены {0} {1} комиксов: выпуски ",
                                       group.Count(), group.Key);
        foreach (var price in group)
            message += price.ToString() + " ";
        CurrentQueryResults.Add(
           CreateAnonymousListViewItem(message, "captain_amazing_250x250.jpg"));
    }
}

private void JoinPurchasesWithPrices() {
    IEnumerable<Comic> comics = BuildCatalog();
    Dictionary<int, decimal> values = GetPrices();
    IEnumerable<Purchase> purchases = Purchase.FindPurchases();
    var results =
        from comic in comics
        join purchase in purchases
        on comic.Issue equals purchase.Issue
        orderby comic.Issue ascending
        select new {
            Comic = comic,
            Price = purchase.Price,
            Title = comic.Name,
            Subtitle = "Выпуск #" + comic.Issue,
            Description = String.Format("Куплен за {0:c}", purchase.Price),
            Image = CreateImageFromAssets("captain_amazing_250x250.jpg"),
        };

    decimal gregsListValue = 0;
    decimal totalSpent = 0;
    foreach (var result in results) {
        gregsListValue += values[result.Comic.Issue];
        totalSpent += result.Price;
        CurrentQueryResults.Add(result);
    }

    Title = String.Format("Я потратил {0:c} на комиксы, стоящие {1:c}",
             totalSpent, gregsListValue);
}
```

Это методы с запросами LINQ.

Заголовок страницы привязан к свойству Title объекта ComicQueryManager, поэтому данная строка поменяет его на сообщение, информирующее Джимми, сколько он потратил и сколько это стоило.

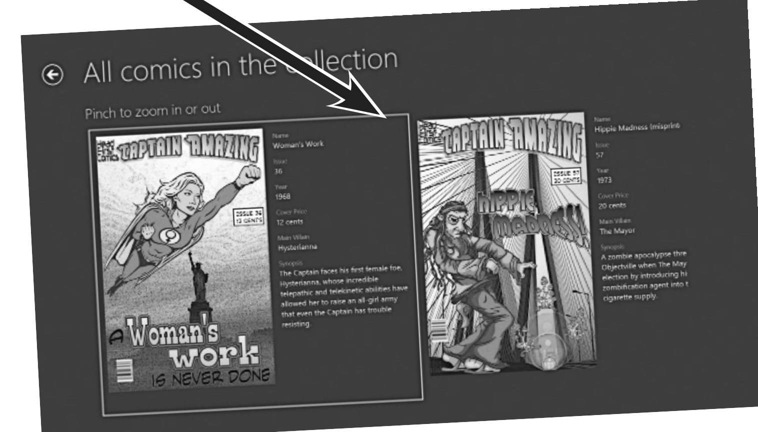
Контекстное масштабирование

Мы подготовили для Джимми обзор коллекции, почему бы не предоставить и способ более детализированного рассмотрения? Существует очень полезный элемент управления, добавляющий к навигационной системе приложения дополнительное измерение. **Контекстное масштабирование (semantic zoom)** позволяет пользователю переключаться между двумя представлениями данных: «режим приближения», обеспечивающий общий вид, и «режим удаления», позволяющий рассмотреть детали.

Кнопка переведет симулятор в режим масштабирования пальцами. Для имитации такого масштабирования удерживая кнопку мыши, вращайте ее колесико.

Воздействие на элемент управления, отвечающий за контекстное масштабирование, осуществляется щипком, как при просмотре фотографии на телефоне или планшете. Также можно использовать колесо прокрутки или щелкать на элементах.

Контекстное масштабирование позволяет отобразить два представления одних и тех же данных: общий вид с множеством элементов и увеличенный вид, дающий возможность рассмотреть детали.



Больше узнать про контекстное масштабирование можно на странице: http://msdn.microsoft.com/ru-ru/library/windows/apps/hh465319.aspx

Вот базовый шаблон XAML для элемента управления, отвечающего за контекстное масштабирование. Один элемент ListView или GridView обеспечивает «режим приближения», а другой «режим удаления»:

```
<SemanticZoom IsZoomedInViewActive="False" >
    <SemanticZoom.ZoomedOutView>
        <ListView>
            <!-- Этот раздел содержит ListView или GridView
                 для отображения общего вида -->
        </ListView>
    </SemanticZoom.ZoomedOutView>
    <SemanticZoom.ZoomedInView>
        <GridView>
            <!-- Этот ListView или GridView показывает
                 детали детализированного вида -->
            <GridView.ItemTemplate>
                <DataTemplate>
                    <!-- ListView или GridView этого раздела
                         отображают детализированный вид -->
                </DataTemplate>
            </GridView.ItemTemplate>
        </GridView>
    </SemanticZoom.ZoomedInView>
</SemanticZoom>
```

Элемент GridView во многом напоминает ListView. Просто ListView добавляет элементам вертикальную прокрутку, а GridView — горизонтальную.

Оба представления содержат ListView или GridView с шаблоном данных, содержащим отвечающие за масштабирование элементы управления.

Мы использовали ListView для общего представления, а GridView для детализированного, но вы можете переписать код по-своему.

ListView и GridView реализуют ISemanticZoomInformation

Элемент SemanticZoom может содержать только реализующие интерфейс ISemanticZoomInformation элементы. Именно там находятся методы, позволяющие элементу SemanticZoom инициировать и завершить изменение масштаба. К счастью, вам не нужно реализовывать этот интерфейс самостоятельно. В приведенном примере мы использовали ListView для отображения общего вида, а GridView для детализированного показа элементов.

Усовершенствуем приложение Джимми

Джимми хотел бы видеть все предметы своей коллекции, увеличивать масштаб и читать подробную информацию об отдельных комиксах.

Добавим на главную страницу новый элемент.
Джимми должен на чем-то щелкать, поэтому первым делом добавим элемент, возвращающий все комиксы коллекции. Их будет отображать метод ComicQueryManager:

```
private void AllComics() {
    foreach (Comic comic in BuildCatalog()) {
        var result = new {
            Image = CreateImageFromAssets("captain_amazing_zoom_250x250.jpg"),
            Title = comic.Name,
            Subtitle = "Выпуск #" + comic.Issue,
            Description = "The Captain versus " + comic.MainVillain,
            Comic = comic,
        };
        CurrentQueryResults.Add(result);
    }
}
```

Теперь добавим еще одну строку к оператору switch в методе UpdateQueryResults():

```
case "All comics in the collection": AllComics(); break;
```

Наконец, добавим новый объект ComicQuery к инициализатору коллекции в методе UpdateAvailableQueries(). Не забудьте поместить файл *captain_amazing_zoom_250x250.jpg* в папку *Assets/*.

```
new ComicQuery("Все комиксы коллекции",
               "Получить все комиксы, входящие в коллекцию",
               "Этот запрос возвращает все комиксы",
               CreateImageFromAssets("captain_amazing_zoom_250x250.jpg")),
```

Скачайте изображение с сайта Head First C#.

Добавим свойства в класс Comic.
Контекстное масштабирование имеет смысл только при наличии деталей. При нем также будут отображаться объекты Comic из коллекции ComicQueryManager.CurrentQueryResults, поэтому добавим в класс Comic описание всех комиксов и свяжем с этими новыми свойствами режим приближения.

```
using Windows.UI.Xaml.Media.Imaging;

class Comic {
    public string Name { get; set; }
    public int Issue { get; set; }
    public int Year { get; set; }
    public string CoverPrice { get; set; }
    public string Synopsis { get; set; }
    public string MainVillain { get; set; }
    public BitmapImage Cover { get; set; }
}
```

Добавим подробную информацию о комиксах.

Отредактируем метод `BuildCatalog()`, вставив описание каждого комикса. Не забудьте скопировать в папку Assets проекта изобржения обложек. Их можно скачать со страницы http://www.headfirstlabs.com/hfcsharp.

```
public static IEnumerable<Comic> BuildCatalog() {
  return new List<Comic> {
     new Comic { Name = "Johnny America vs. the Pinko", Issue = 6, Year = 1949, CoverPrice = "10 cents",
         MainVillain = "The Pinko", Cover = CreateImageFromAssets("Captain Amazing Issue 6 cover.png"),
         Synopsis = "Captain Amazing must save America from Communists as The Pinko and his"
             + " communist henchmen threaten to take over Fort Knox and steal all of the nation's gold." },

     new Comic { Name = "Rock and Roll (limited edition)", Issue = 19, Year = 1957, CoverPrice = "10 cents",
         MainVillain = "Doctor Vortran", Cover = CreateImageFromAssets("Captain Amazing Issue 19 cover.png"),
         Synopsis = "Doctor Vortran wreaks havoc with the nation's youth with his radio wave device that"
             + " uses the latest dance craze to send rock and roll fans into a mind-control trance." },

        new Comic { Name = "Woman's Work", Issue = 36, Year = 1968, CoverPrice = "12 cents",
            MainVillain = "Hysterianna", Cover = CreateImageFromAssets("Captain Amazing Issue 36 cover.png"),
            Synopsis = "The Captain faces his first female foe, Hysterianna, whose incredible telepathic"
                + " and telekinetic abilities have allowed her to raise an all-girl army that"
                        + " even the Captain has trouble resisting." },

        new Comic { Name = "Hippie Madness (misprinted)", Issue = 57, Year = 1973, CoverPrice = "20 cents",
            MainVillain = "The Mayor", Cover = CreateImageFromAssets("Captain Amazing Issue 57 cover.png"),
            Synopsis = "A zombie apocalypse threatens Objectville when The Mayor rigs the election by"
                + " introducing his zombification agent into the city's cigarette supply." },

        new Comic { Name = "Revenge of the New Wave Freak (damaged)", Issue = 68, Year = 1984,
            CoverPrice = "75 cents", MainVillain = "The Swindler",
            Cover = CreateImageFromAssets("Captain Amazing Issue 68 cover.png"),
            Synopsis = "A tainted batch of eye makeup turns Dr. Alvin Mudd into the Captain's new nemesis,"
                + " in The Swindler's first appearance in a Captain Amazing comic." },

        new Comic { Name = "Black Monday", Issue = 74, Year = 1986, CoverPrice = "75 cents",
            MainVillain = "The Mayor", Cover = CreateImageFromAssets("Captain Amazing Issue 74 cover.png"),
            Synopsis = "The Mayor returns to throw Objectville into a financial crisis by directing his"
                + " zombie creation powers to the floor of the Objectville Stock Exchange." },

        new Comic { Name = "Tribal Tattoo Madness", Issue = 83, Year = 1996, CoverPrice = "Two bucks",
            MainVillain = "Mokey Man", Cover = CreateImageFromAssets("Captain Amazing Issue 83 cover.png"),
            Synopsis = "Monkey Man escapes from his island prison and wreaks havoc with his circus sideshow"
                + " of tattooed henchmen that and their deadly grunge ray." },

        new Comic { Name = "The Death of an Object", Issue = 97, Year = 2013, CoverPrice = "Four bucks",
            MainVillain = "The Swindler", Cover = CreateImageFromAssets("Captain Amazing Issue 97 cover.png"),
            Synopsis = "The Swindler's clone army attacks Objectville in a ruse to trap and kill the "
                + " Captain. Can the scientists of Objectville find a way to bring him back?" },
   };
}
```

Переверните страницу, чтобы завершить приложение →

Добавим новый шаблон Basic Page для элементов управления масштабированием.
Джимми доволен нововведениями, поэтому вместо редактирования существующей страницы мы создадим новую. **Добавьте новый шаблон Basic Page *QueryDetailZoom.xaml*.**

После этого **вернитесь к странице *MainPage.xaml* и отредактируйте обработчик события `ItemClick`** в коде программной части. При щелчке пользователя на новом результате запроса он должен направлять нас на страницу `QueryDetailZoom`:

```
private void ListView_ItemClick(object sender, ItemClickEventArgs e) {
    ComicQuery query = e.ClickedItem as ComicQuery;
    if (query != null) {
        if (query.Title == "All comics in the collection")
            this.Frame.Navigate(typeof(QueryDetailZoom), query);
        else
            this.Frame.Navigate(typeof(QueryDetail), query);
    }
}
```

Это новый обработчик события ItemClick для страницы MainPage.xaml. Он проверяет заголовок запроса, чтобы понять, на какую страницу следует перейти — QueryDetail или QueryDetailZoom.

5 **Добавим на новую страницу статический ресурс ComicQueryManager.**
Новая страница `QueryDetailZoom` работает так же, как `QueryDetail`. Нужно добавить `ComicQueryManager` в раздел `<Page.Resources>` файла *QueryDetailZoom.xaml*. Обновлять ресурс `AppName` не следует, так как страница настроена на использование этого кода C#:

```
<Page.Resources>
    <local:ComicQueryManager x:Name="comicQueryManager"/>
    <!-- TODO: Удалите эту строку, если ключ AppName объявлен в App.xaml -->
    <x:String x:Key="AppName">My Application</x:String>
</Page.Resources>
```

6 **Добавим программный код для страницы с описаниями комиксов.**
И такой же метод `OnNavigatedTo()` следует добавить в файл *QueryDetailZoom.xaml.cs*:

```
protected override void OnNavigatedTo(NavigationEventArgs e) {
    ComicQuery comicQuery = e.Parameter as ComicQuery;
    if (comicQuery != null) {
        comicQueryManager.UpdateQueryResults(comicQuery);
        pageTitle.Text = comicQueryManager.Title;
    }
    base.OnNavigatedTo(e);
}
```

Создадим код XAML для страницы с описаниями комиксов.

Осталось написать код XAML для страницы *QueryDetailZoom.xaml*, включив в него элемент управления контекстным масштабированием, отображающий описания комиксов. Это самая большая из всех созданных вами до этого момента страниц, поэтому мы поделили код на две части, чтобы было проще понять, что происходит.

```
<Grid Grid.Row="1" Margin="120,0"
                  DataContext="{StaticResource ResourceKey=comicQueryManager}">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
    </Grid.RowDefinitions>

    <TextBlock Style="{StaticResource SubheaderTextStyle}" Margin="0,0,0,20"
               Text="Pinch to zoom in or out" />

    <SemanticZoom IsZoomedInViewActive="False" Grid.Row="1">

        <SemanticZoom.ZoomedOutView>
            <ListView ItemsSource="{Binding CurrentQueryResults}" Margin="0,0,20,0"
                        ItemTemplate="{StaticResource Standard500x130ItemTemplate}"
                        SelectionMode="None" />
        </SemanticZoom.ZoomedOutView>

        <SemanticZoom.ZoomedInView>
          <GridView ItemsSource="{Binding CurrentQueryResults}"
                     Margin="0,0,20,0" SelectionMode="None" x:Name="detailGridView">
            <GridView.ItemTemplate>
              <DataTemplate>
                <Grid Height="780" Width="600" Margin="10">
                  <Grid.ColumnDefinitions>
                     <ColumnDefinition Width="Auto"/>
                     <ColumnDefinition/>
                  </Grid.ColumnDefinitions>

                  <Image Source="{Binding Comic.Cover}" Margin="0,0,20,0"
                         Stretch="UniformToFill" Width="326" Height="500"
                         VerticalAlignment="Top"/>
```

Мы поместили SemanticZoom в строку сетки, чтобы обеспечить возможность прокрутки у элементов ListView и GridView.

За общий вид отвечает элемент ListView, совпадающий с элементом страницы QueryDetail.

Основой детализированного представления является GridView. Его шаблон данных — это сетка, содержащая элемент управления Image для обложки и набор StackPanel элементов TextBlock для остальных свойств.

Шаблон данных GridView работает так же, как в ListView. Чтобы вспомнить принципы работы шаблонов данных, вернитесь к главе 10.

Переверните страницу, чтобы найти остальной код XAML.

```
                <StackPanel Grid.Column="1">

                    <TextBlock Text="Name"
                               Style="{StaticResource CaptionTextStyle}" />
                    <TextBlock Text="{Binding Comic.Name}"
                               Style="{StaticResource ItemTextStyle}" />

                    <TextBlock Text="Issue"
                              Style="{StaticResource CaptionTextStyle}" Margin="0,10,0,0" />
                    <TextBlock Text="{Binding Comic.Issue}"
                               Style="{StaticResource ItemTextStyle}" />

                    <TextBlock Text="Year"
                               Style="{StaticResource CaptionTextStyle}" Margin="0,10,0,0" />
                    <TextBlock Text="{Binding Comic.Year}"
                               Style="{StaticResource ItemTextStyle}" />

                    <TextBlock Text="Cover Price"
                               Style="{StaticResource CaptionTextStyle}" Margin="0,10,0,0" />
                    <TextBlock Text="{Binding Comic.CoverPrice}"
                               Style="{StaticResource ItemTextStyle}" />

                    <TextBlock Text="Main Villain"
                               Style="{StaticResource CaptionTextStyle}" Margin="0,10,0,0" />
                    <TextBlock Text="{Binding Comic.MainVillain}"
                               Style="{StaticResource ItemTextStyle}" />

                    <TextBlock Text="Synopsis"
                               Style="{StaticResource CaptionTextStyle}" Margin="0,10,0,0" />
                    <TextBlock Text="{Binding Comic.Synopsis}"
                               Style="{StaticResource ItemTextStyle}" />
                </StackPanel>
            </Grid>
          </DataTemplate>
        </GridView.ItemTemplate>
      </GridView>
    </SemanticZoom.ZoomedInView>
  </SemanticZoom>
</Grid>
```

Эта часть шаблона GridView отвечает за подробное представление. Она показывает остальные детали в наборе элементов TextBlock, связанные со свойствами объекта Comic.

Редактируйте запросы в LINQPad

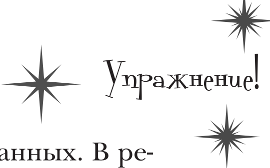
Существует замечательный бесплатный инструмент LINQ. Он называется LINQPad и предоставлен Джо Албахари (из уникальной команды научных редакторов «Head First C#»). Его можно скачать с сайта: http://www.linqpad.net/

Вы осчастливили Джимми

Благодаря созданному приложению коллекция Джонни теперь в полном порядке. Отличная работа!

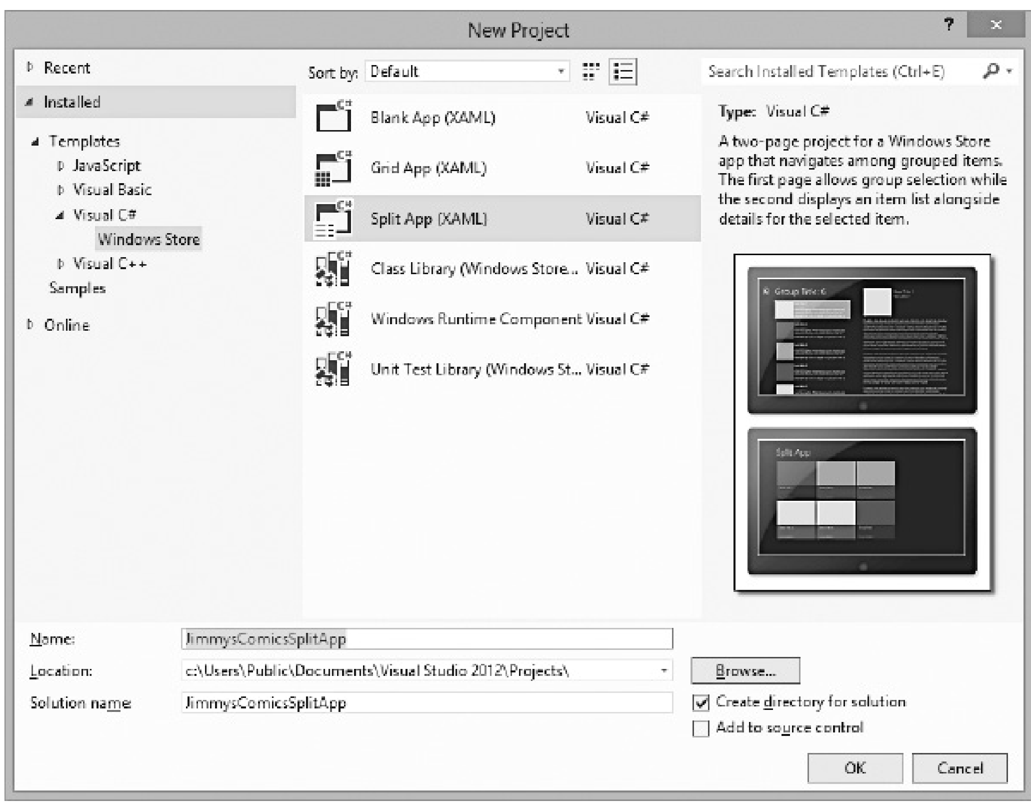
Шаблон Split App

Существует простой способ создания двустраничного приложения, переключающегося между общим и детализированным представлениями. Если при создании нового проекта воспользоваться **шаблоном Split App**, IDE автоматически сгенерирует приложение нужного типа. Познакомимся с этим шаблоном поближе, обновив построенное нами для Джимми приложение.



Создадим и запустим проект Split App.
Шаблон проекта Split App содержит класс, генерирующий образцы данных. В результате вы можете запустить приложение сразу же после его создания.

Откройте новый **проект Split App (XAML) и назовите его JimmysComicsSplitApp**, чтобы пространство имен было похоже на код, приведенный на следующих страницах.

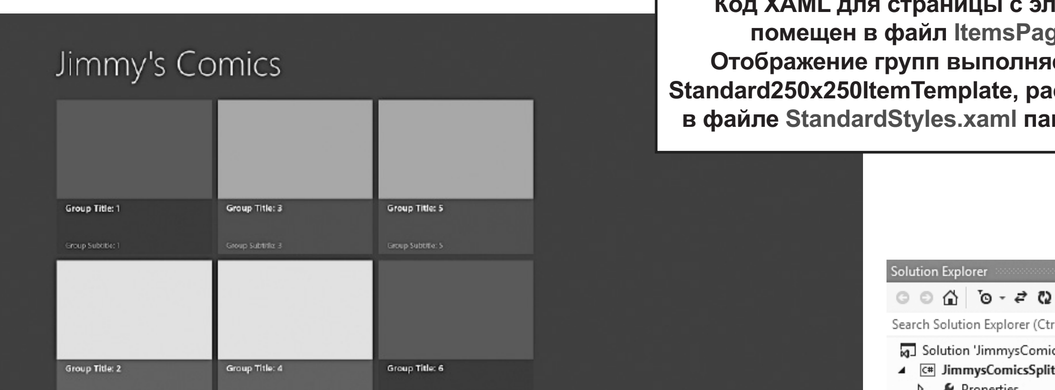


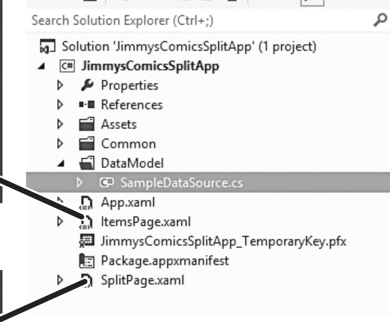
Через ресурс `AppName` измените имя на «Jimmy's Comics». В проектах на основе шаблона Split App **ресурс AppName находится в файле *App.xaml***:

```
<x:String x:Key="AppName">Jimmy's Comics</x:String>
```

Запустите приложение. Оно состоит из двух страниц. Первая демонстрирует вам группы элементов, которые можно рассмотреть подробнее:

Код XAML для страницы с элементами помещен в файл ItemsPage.xaml. Отображение групп выполняет шаблон Standard250x250ItemTemplate, расположенный в файле StandardStyles.xaml папки Common.





Щелкните на любом элементе для перехода на страницу с деталями:

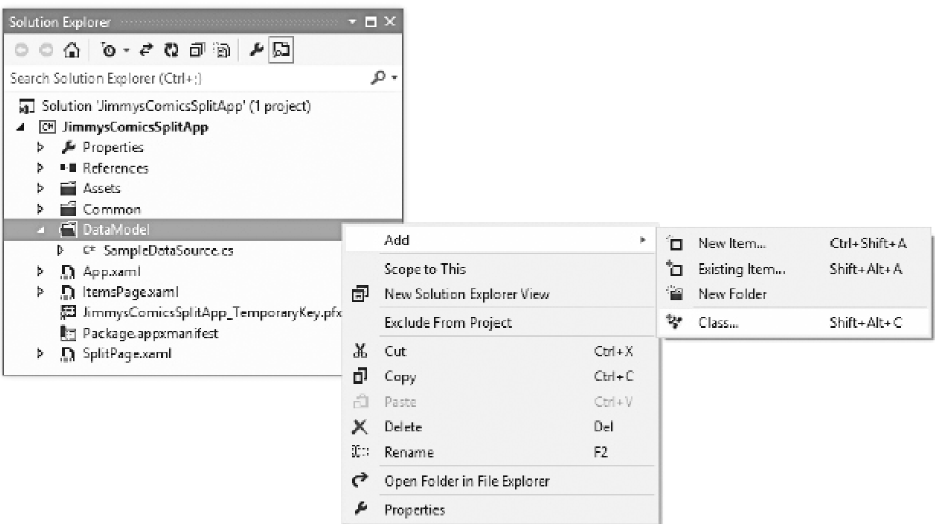
Щелчок на элементе общего представления заставляет приложение перейти к разделенному представлению, находящемуся в файле SplitPage.xaml. Содержимое групп показывается слева при помощи шаблона Standard130ItemTemplate. Описание выбранного элемента отображается справа. Эту функциональность обеспечивает XAML со связыванием данных, а не шаблон.

Большой фрагмент текста — это один TextBlock, который вы на шаге 6 заместите XAML-кодом с описанием комиксов из вашего приложения.

Код XAML разделенного представления отображает раздел с подробностями только в альбомной ориентации. Проверьте сами: запустите приложение в симуляторе и воспользуйтесь кнопками и для поворота экрана виртуального устройства.

Добавим классы данных в папку DataModel.

Щелкните правой кнопкой мыши на папке *DataModel* в окне Solution Explorer и выберите команду Add→Class... для добавления в папку нового класса.

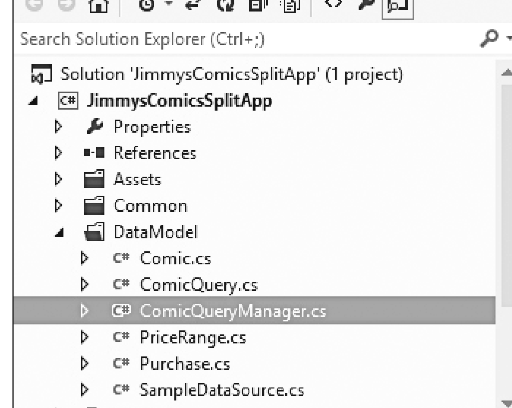


Создайте класс `ComicQueryManager`. При создании класса внутри папки IDE использует пространство имен, в которое входит имя папки:

```
namespace JimmysComicsSplitApp.DataModel
{
    class ComicQueryManager
    {

    }
}
```

Скопируйте содержимое класса `ComicQueryManager` из приложения Джимми в новый файл *Comic.cs* в папке *DataModel*. Обязательно сохраните пространство имен `JimmysComicsSplitApp.DataModel` (не забудьте оператор `using`).

Аналогичным образом **создайте классы `Comic`, `ComicQuery` и `Purchase`, а также перечисление `PriceRange`**. Все они должны создаваться внутри папки *DataModel*, чтобы оказаться в пространстве имен `JimmysComicsSplitApp.DataModel`. Вот как будет выглядеть окно Solution Explorer после завершения работы:



В папке *DataModel* находится файл *SampleDataSource.cs* с кодом генерации образцов данных, которые вы видели при запуске приложения.

Откройте его, он функционирует аналогично классам данных в приложении для Джимми. Файл содержит набор классов, в том числе `SampleDataGroup` (представляющий группы высокого уровня, как и ваш класс `ComicQuery`) и класс `SampleDataItem` (представляющий отдельные элементы, как ваш класс `Comic`). При этом образцы данных создаются в конструкторе класса `SampleDataSource`, расположенном в нижней части файла.

Код программной части страницы с элементами создает новый экземпляр класса `SampleDataSource` и использует его для заполнения словаря `DefaultViewModel`.

Подробно класс ViewModel и способ его построения будут рассмотрены в главе 16.

Подождите!
Если данные из моего приложения структурированы аналогично образцам, мы легко добавим их в разделенное приложение.

Все правильно. Шаблон Split App спроектирован так, чтобы вам было легко добавлять данные.

Все, что вам требуется для вставки данных в разделенное приложение, — небольшое редактирование кода программной части на страницах общего и детализированного представлений, чем мы сейчас и займемся. Еще мы отредактируем разделенную страницу, чтобы для отображения обложек комиксов и их описаний она использовала один и тот же код XAML.

Отредактируем программный код в файле ItemsPage.xaml.cs.

Откройте файл *ItemsPanel.xaml.cs* и командами Edit→Find и Replace найдите там «TODO:». Почитайте комментарии: шаблон рассказывает, где именно производится замещение образцов данных. Превратите в комментарии две строки, задающие значение Items в словаре `DefaultViewModel` и **вставьте под ними собственный код**, считывающий свойство `AvailableQueries` нового объекта `ComicQueryManager`:

Превратите эти строки в комментарии.

```
// TODO: Создайте подходящую модель данных для замены образца данных
//var sampleDataGroups = SampleDataSource.GetGroups((String)navigationParameter);
//this.DefaultViewModel["Items"] = sampleDataGroups;
this.DefaultViewModel["Items"] = new DataModel.ComicQueryManager().AvailableQueries;
```

Добавьте эту строку.

Также нужно превратить в комментарии код в методе обработки события `ItemView_ItemClick()`, который после щелчка пытается привести элемент к типу `SampleDataGroup` (это передается в аргумент события как `e.ClickedItem`). Свойство `AvailableQueries` возвращает коллекцию объектов ComicQuery. Вот новый обработчик события `ItemClick()`:

```
void ItemView_ItemClick(object sender, ItemClickEventArgs e) {
    // Перебрасывает на корректную страницу, конфигурирует эту новую страницу
    // передавая нужную информацию как параметр навигации
    //var groupId = ((SampleDataGroup)e.ClickedItem).UniqueId;
    //this.Frame.Navigate(typeof(SplitPage), groupId);
     DataModel.ComicQuery query = e.ClickedItem as DataModel.ComicQuery;
     if (query != null)
         this.Frame.Navigate(typeof(SplitPage), query);
}
```

ComicQuery и прочие классы создавались в папке DataModel, поэтому они находятся в пространстве имен DataModel.

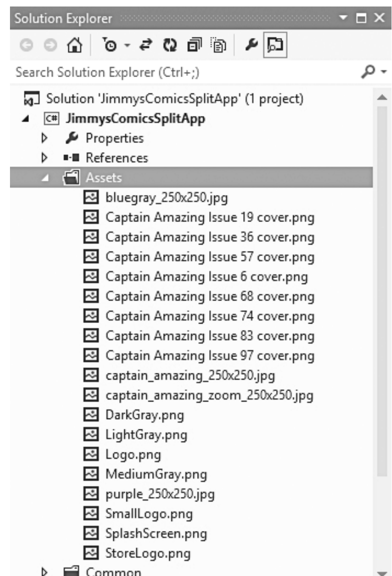
Отредактируем код программной части в файле SplitPage.xaml.cs.

На разделенной странице также присутствует комментарий с заголовком «TODO:», сразу над операторами задания значений Group и Items в словаре `DefaultViewModel`. Вставьте свой код, чтобы присвоить группам и элементам данные из модели данных для комиксов:

```
// TODO: Создайте подходящую модель данных для замены образца данных
// var group = SampleDataSource.GetGroup((String)navigationParameter);
// this.DefaultViewModel["Group"] = group;
// this.DefaultViewModel["Items"] = group.Items;
DataModel.ComicQueryManager comicQueryManager = new DataModel.ComicQueryManager();
DataModel.ComicQuery query = navigationParameter as DataModel.ComicQuery;
comicQueryManager.UpdateQueryResults(query);
this.DefaultViewModel["Group"] = query;
this.DefaultViewModel["Items"] = comicQueryManager.CurrentQueryResults;
```

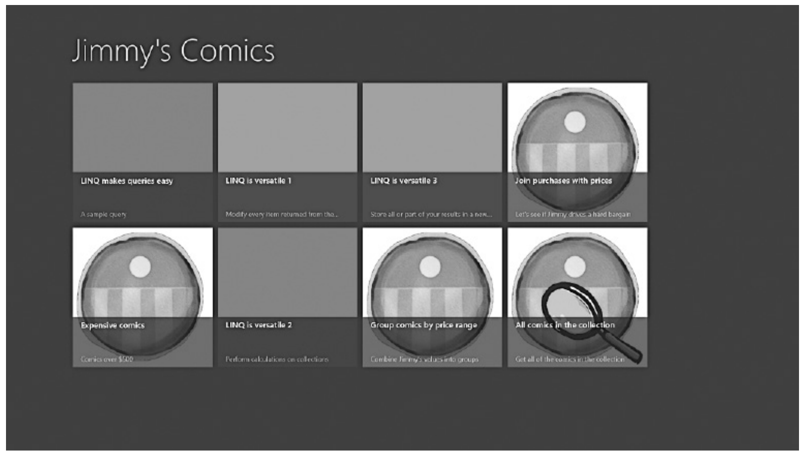
Остался всего один момент. Разделенная страница перекрывает метод `SaveState`, позволяющий вспомнить, на каком именно элементе щелкал пользователь. Генерируемый код приводит выделенный элемент к типу `SampleDataItem`, поэтому превратите весь код метода в комментарии, чтобы избежать исключений, связанных с приведением типов.

```
protected override void SaveState(Dictionary<String, Object> pageState) {
    // Превратите в комментарии весь код метода
}
```

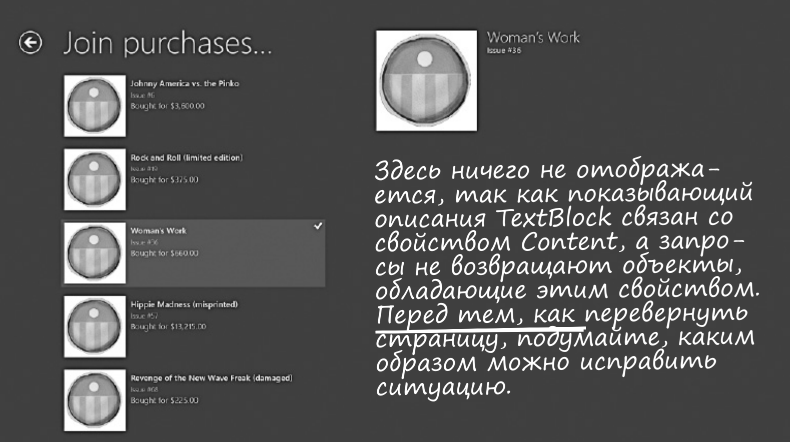


5 **Добавим изображения в папку Assets.**
Нам потребуются все файлы изображений из приложения Джимми. **Щелкните правой кнопкой мыши на папке *Assets* и выберите команду Add→Existing Item...** для вызова окна диалога Add Existing Item. Найдите папку, содержащую код ранее созданного вами приложения, и щелчками при нажатой клавише Control выделите все файлы, кроме *Logo.png*, *SmallLogo.png*, *SplashScreen.png* и *StoreLogo.png*. Щелкните на кнопке Add, чтобы поместить все файлы в папку *Assets* вашего проекта.

Теперь ваше приложение работает! Страница с элементами отображает доступные запросы объекта ComicQueryManager...



...а разделенная страница демонстрирует результаты запросов и позволяет прочитать описание каждого элемента.



Отредактируем страницу SplitPage.xaml.

Код XAML в файле *SplitPage.xaml* для отображения элементов в левой части страницы использует шаблоны, а вот для показа справа описания выбранного элемента нужен обычный XAML со связыванием даных. Это связанный со свойством `Content` элемент TextBlock:

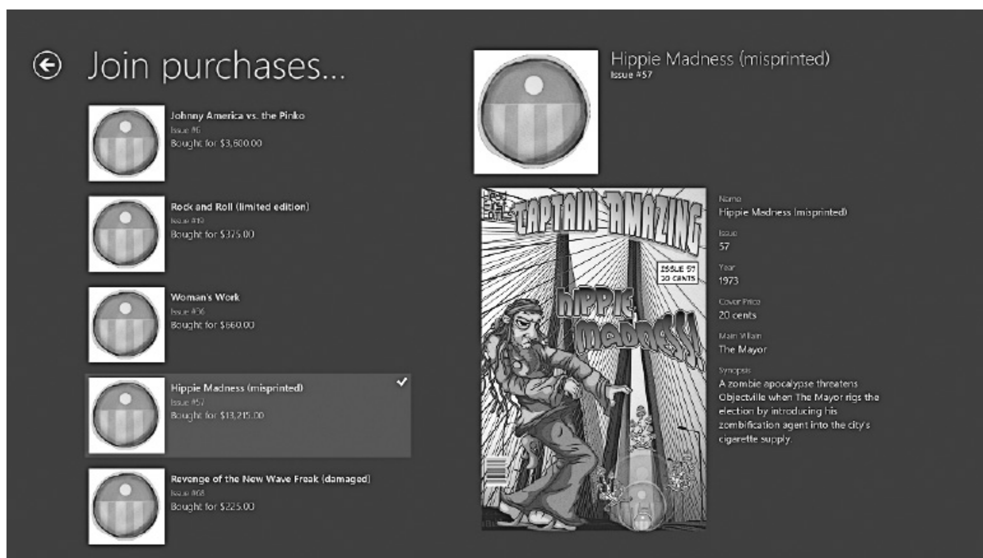
```
<TextBlock Grid.Row="2" Grid.ColumnSpan="2" Margin="0,20,0,0"
          Text="{Binding Content}" Style="{StaticResource BodyTextStyle}"/>
```

Мы используем эти свойства повторно, поэтому сведения о комиксах появятся на том же месте.

Образец данных в файле *SampleDataSource.cs* включает свойство `Content`, содержащее большой блок текста. Нам нужно, чтобы приложение отображало сведения о комиксах из коллекции Джимми. К счастью, у нас есть код XAML, который делает это, когда связан с объектом `Comic`. **Замените TextBlock кодом XAML с описанием комиксов**. Добавьте к внешней сетке свойства `Grid.Row`, `Grid.ColumnSpan` и `Margin`:

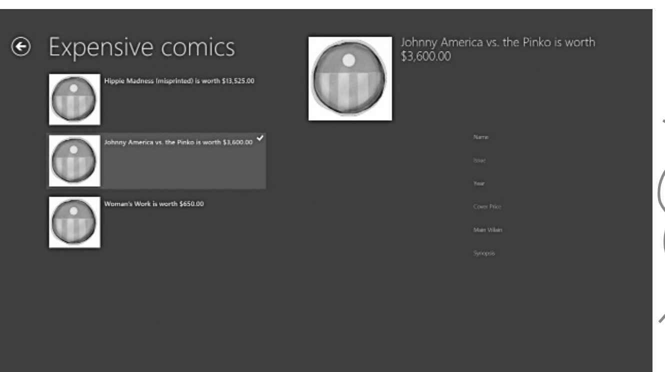
```
<Grid Height="780" Width="600" Grid.Row="2" Grid.ColumnSpan="2" Margin="0,20,0,0">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Image Source="{Binding Comic.Cover}" Margin="0,0,20,0"
           Stretch="UniformToFill" Width="326" Height="500"
           VerticalAlignment="Top"/>

    <StackPanel Grid.Column="1">

        <TextBlock Text="Name"
                   Style="{StaticResource CaptionTextStyle}" />
        <TextBlock Text="{Binding Comic.Name}"
                   Style="{StaticResource ItemTextStyle}" />

        <TextBlock Text="Issue"
                   Style="{StaticResource CaptionTextStyle}" Margin="0,10,0,0" />
        <TextBlock Text="{Binding Comic.Issue, Target}"
                   Style="{StaticResource ItemTextStyle}" />

        <TextBlock Text="Year"
                   Style="{StaticResource CaptionTextStyle}" Margin="0,10,0,0" />
        <TextBlock Text="{Binding Comic.Year}"
                   Style="{StaticResource ItemTextStyle}" />

        <TextBlock Text="Cover Price"
                   Style="{StaticResource CaptionTextStyle}" Margin="0,10,0,0" />
        <TextBlock Text="{Binding Comic.CoverPrice}"
                   Style="{StaticResource ItemTextStyle}" />

        <TextBlock Text="Main Villain"
                   Style="{StaticResource CaptionTextStyle}" Margin="0,10,0,0" />
        <TextBlock Text="{Binding Comic.MainVillain}"
                   Style="{StaticResource ItemTextStyle}" />

        <TextBlock Text="Synopsis"
                   Style="{StaticResource CaptionTextStyle}" Margin="0,10,0,0" />
        <TextBlock Text="{Binding Comic.Synopsis}"
                   Style="{StaticResource ItemTextStyle}" />
    </StackPanel>
</Grid>
```

Теперь приложение позволяет углубляться в детали возвращенных запросом результатов, отображая описание выбранного комикса на разделенной странице.



Некоторые запросы не возвращают объект Comic, поэтому любое связанное с данным поле окажется пустым. В главе 16 вы познакомитесь с конвертерами значений, которые позволяют прятать такие поля или отображать вместо них значения по умолчанию.

Запрос Expensive Comics возвращает последовательность анонимных объектов, обладающих свойствами Title и Image.



Эти элементы управления связаны со свойствами, не входящими в контекст данных, поэтому на странице они отображаются как пустые поля. Позднее вы познакомитесь с инструментами, скрывающими ярлыки или отображающими значение по умолчанию.

К проектам на основе шаблонов Items Page и Split Page страницы добавляются той же командой Add New Item, которой вы пользовались при работе со страницами на основе Basic Page. Существует еще один шаблон Grid App, с тремя уровнями навигации. Подробнее почитать о шаблонах Grid App и Split App можно на сайте:
http://msdn.microsoft.com/ru-ru/library/windows/apps/hh768232.aspx

15 события и делегаты

Что делает ваш код, когда вы на него не смотрите

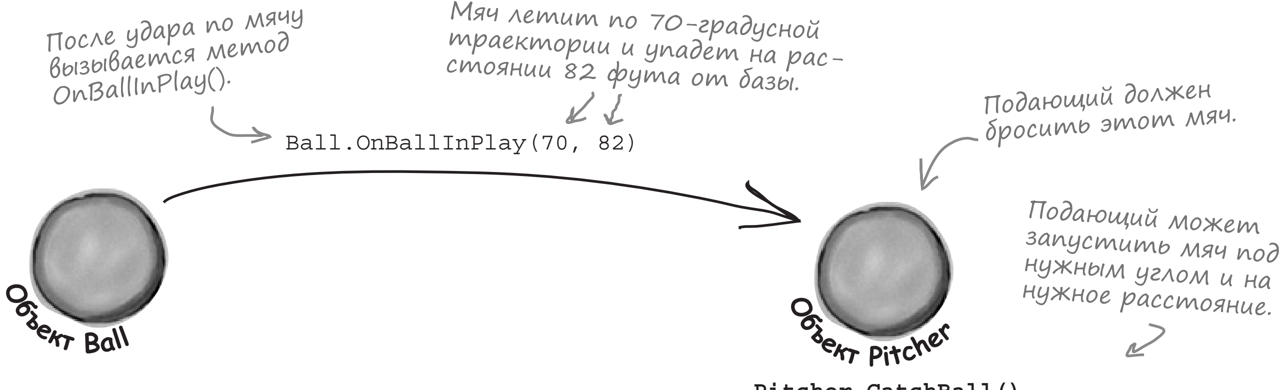
Невозможно все время контролировать созданные объекты. Иногда что-то… происходит. И хотелось бы, чтобы объекты умели **реагировать на происходящее**. Здесь вам на помощь приходят события. Один объект их *публикует*, другие объекты на них *подписываются*, и система работает. А для контроля подписчиков вам пригодится метод **обратного вызова**.

Хотите, чтобы объекты научились думать сами?

Предположим, вы пишете симулятор игры в бейсбол, чтобы продать это приложение команде «Янки» и заработать миллион долларов. Вы создали объекты Ball (Мяч), Pitcher (Подающий), Umpire (Судья), Fan (Фанат) и многие другие. Вы даже написали код, моделирующий перебрасывание мяча объектом Pitcher.

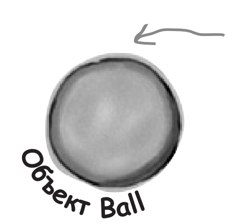
Это повсеместно распространенный способ именования методов. Мы обсудим его чуть позже.

Осталось собрать все вместе. Вы добавляете мячу метод OnBallInPlay() (Мяч в игре), и теперь нужно, чтобы объект Pitcher ответил методом своего обработчика событий. Методы уже написаны, их требуется только связать друг с другом:



Как объекты узнают, что произошло?

Сформулируем проблему. Вы хотите, чтобы объект Ball беспокоился об отправляющем его в полет ударе, а объект Pitcher — о поимке летящих на него мячей. При этом не нужно, чтобы объект Ball сообщал объекту Pitcher: «Я лечу».



Объект Ball не знает, кто из игроков будет иметь с ним дело... может быть, Pitcher, а может быть, Catcher или даже ThirdBaseman, решивший реализовать метод ScootIn() и очень быстро бегающий.

Объекты должны заботиться о себе, а не о других. Мы за разделение сферы влияния объектов.

Это не означает невозможности взаимодействия объектов. Мы говорим всего лишь о том, что не объект Ball определяет, кто будет его кидать. Это не входит в его обязанности.

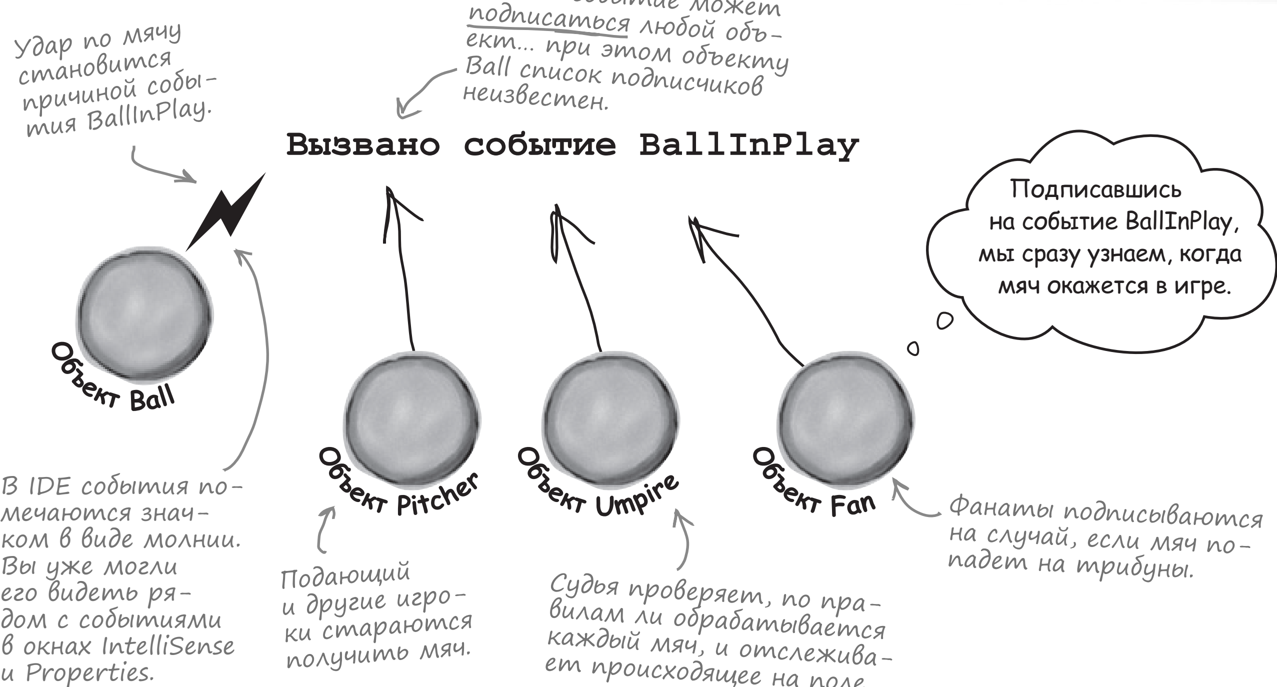
События

После удара по мячу вам потребуется **событие (event)**. Этим термином называется ***что-то происходящее*** в вашей программе. На событие могут прореагировать объекты, например `Pitcher`.

Разумеется, это могут быть и объекты `Catcher`, `ThirdBaseman`, `Umpire` и даже `Fan`. При этом реакция для каждого объекта будет своя.

То есть объект `Ball` должен **вызывать событие**. Остальные же объекты будут **подписываться на событие этого типа**... и реагировать на его возникновение.

со-бы-ти-е, сущ.
то, что случается.
*Солнечное затмение – это **событие**, которое нельзя пропустить.*



Обработчик событий

Логичным результатом оповещения объектов о событии должен быть запуск некоего кода. Этот код называют **обработчиком событий (event handler)**.

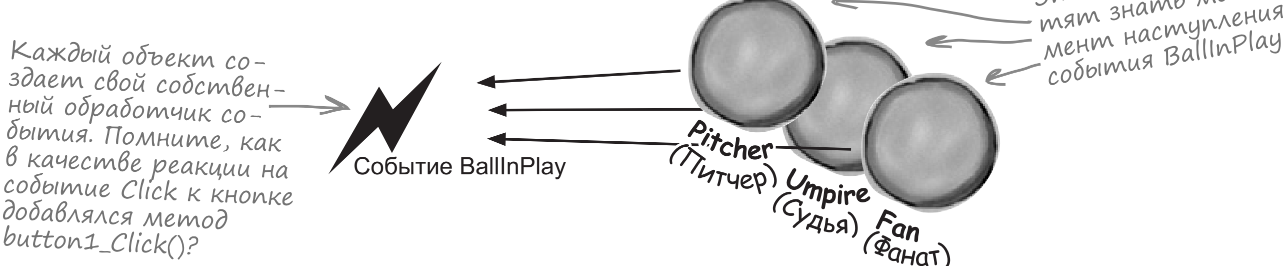
Все это происходит во время работы программы ***без вашего вмешательства***. Вы пишете код, вызывающий событие, затем код для его обработки и запускаете приложение. При возникновении события обработчик начинает свою деятельность... *вы при этом не делаете ничего.* И лучше всего то, что объекты при этом заботятся только о себе, а не о других объектах.

Вы уже делали это много раз. Каждый щелчок на кнопке становился причиной события, на которое каким-то образом отвечал ваш код.

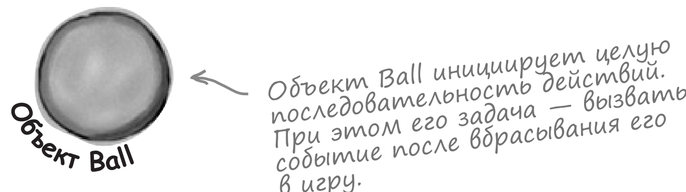
Один объект инициирует событие, другой реагирует на него

Посмотрим, как в C# функционируют события, их обработчики и подписки:

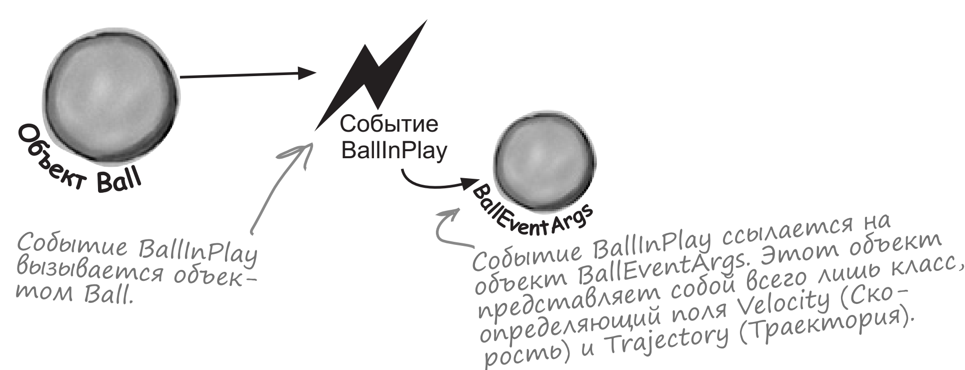
1. **Сначала объекты подписываются на событие.**
До момента, когда объект `Ball` сможет вызвать событие `BallInPlay`, на него должны подписаться другие объекты. Этим они как бы сообщают о своем желании знать, что это событие наступило.



2. **Событие запускается.**
По мячу наносится удар. Именно в этот момент объект `Ball` вызывает новое событие.



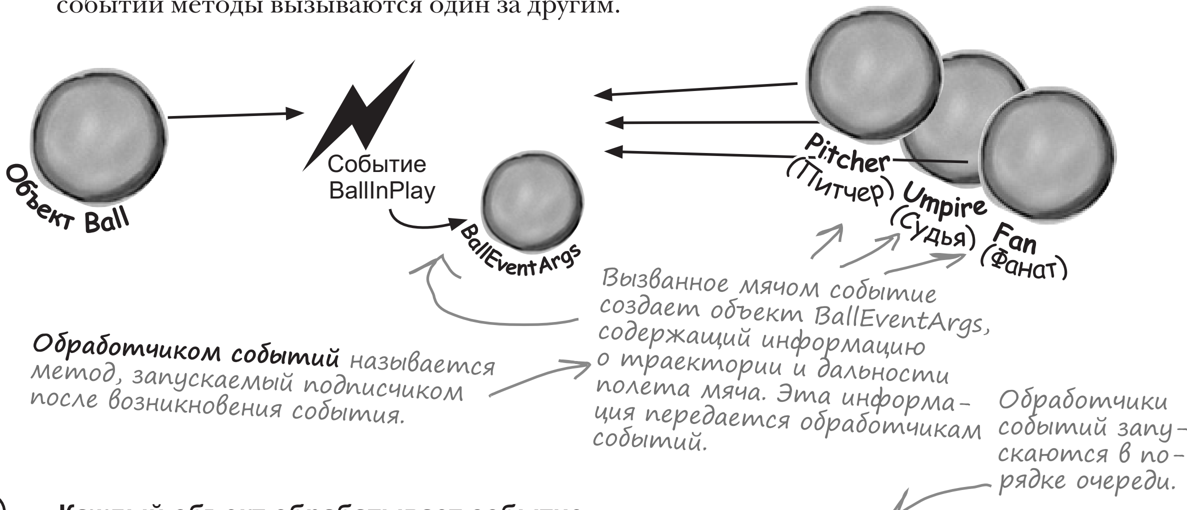
3. **Мяч вызывает событие.**
Появилось новое событие (о том, как именно это происходит, мы поговорим через минуту). Его аргументами являются скорость и траектория движения мяча. Эти аргументы присоединены к событию как экземпляр объекта `EventArgs`. Затем информация о событии отправляется подписчикам.



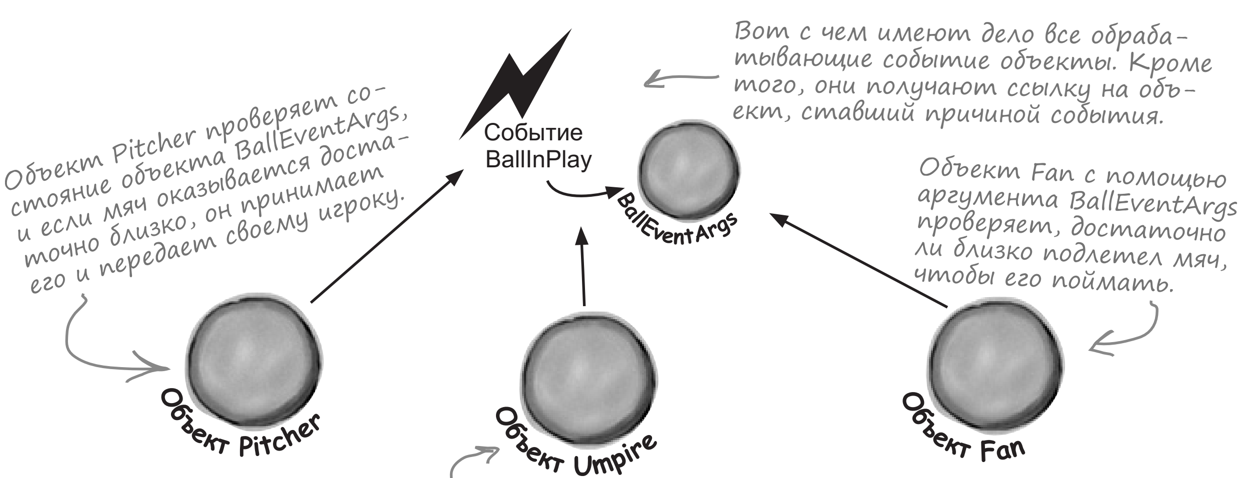
Обработка события

После возникновения события все подписанные на него объекты получают уведомления и могут совершать различные действия.

4 **Подписчики получают уведомление.**
Так как объекты `Pitcher`, `Umpire` и `Fan` подписаны на событие `BallInPlay` объекта `Ball`, они получают уведомление, и их предназначенные для обработки событий методы вызываются один за другим.



5 **Каждый объект обрабатывает событие.**
Теперь объекты `Pitcher`, `Umpire` и `Fan` начинают каждый по-своему реагировать на событие `BallInPlay`. Их обработчики событий вызываются в порядке очереди путем передачи им в качестве параметра ссылки на объект `BallEventArgs`.



Соединим все вместе

Теперь, когда вы имеете общее представление о том, что происходит, рассмотрим более подробно процесс соединения разрозненных фрагментов.

Имеет смысл (хотя это и не обязательно) делать объекты аргументов события производными от класса EventArgs.

EventArgs

BallEventArgs
Trajectory
Distance

Это означает возможность ***восходящего приведения*** *объекта EventArgs в случае, когда его нужно переслать событию, не умеющему его обрабатывать.*

Эти свойства позволят мячу передать обработчикам событий информацию о месте вброса в игру.

1 Нам нужен объект для аргументов события.
Помните, что событие `BallInPlay` имеет несколько аргументов? Для них нам потребуется объект. В .NET для этой цели существует стандартный класс **EventArgs**, но он не имеет членов. Он предназначен исключительно для передачи аргументов объекта обработчикам события. Вот объявление этого класса:

```
class BallEventArgs : EventArgs
```

2 Нужно объявить событие внутри вызвавшего его класса.
В классе Ball присутствует строка с ключевым словом `event`. Она может располагаться в произвольном месте класса, обычно ее помещают рядом с объявлением свойств. Благодаря этому другие объекты могут подписываться на событие. Вот как это выглядит:

```
public event EventHandler BallInPlay;
```

Доступ к событиям обычно открыт. Наше событие определено в классе Ball, но нужно, чтобы на него могли ссылаться объекты Pitcher, Umpire и т. п. Если вы хотите ограничить доступ к событию экземплярами из его класса, событие можно закрыть.

Следующее после ключевого слова event ключевое слово EventHandler является частью .NET. Оно показывает подписывающимся на событие объектам, как должны выглядеть их обработчики событий.

Присутствие EventHandler означает, что обработчики событий должны иметь два параметра: sender и e.
sender — это ссылка на объект, вызвавший событие, а e — ссылка на объект EventArgs.

3 **Классам-подписчикам нужны обработчики событий.**
Вы уже знаете, как функционируют обработчики событий. Каждый раз, когда вы создавали, к примеру, метод для обработки события `Click`, IDE добавляла его в класс. То же самое произойдет с событием `BallInPlay` объекта `Ball`. Его обработчик событий будет иметь уже знакомый вам вид:

```
void ball_BallInPlay(object sender, EventArgs e)
```

В C# нет правила именования обработчиков событий, но обычно имена формируются следующим образом: имя объекта, нижнее подчеркивание, имя события.

Класс, которому принадлежит данный обработчик события, имеет ссылочную переменную ball на объект Ball, поэтому имя обработчика события BallInPlay начинается со слова ball_, за которым следует имя обрабатываемого события BallInPlay.

В объявлении события BallInPlay его тип указан как EventHandler, что означает присутствие двух параметров: объекта sender и ссылки EventArgs с именем e, а также то, что это событие не возвращает значение.

4 **На событие подписываются объекты.**
Теперь, когда обработчик события задан, объектам `Pitcher`, `Umpire`, `ThirdBaseman` и `Fan` нужно подключить свои обработчики событий. Каждый из них будет иметь собственный метод `ball_BallInPlay`. Так что при наличии у вас ссылочной переменной на объект `Ball` или поля `ball` оператор += привяжет к ним обработчик события:

```
ball.BallInPlay += new EventHandler(ball_BallInPlay);
```

Мы связываем обработчик с событием BallInPlay объекта, на который указывает ссылка ball.

Оператор += осуществляет подписку обработчика на событие.

Эта часть определяет, какой метод обработчика будет подписан на событие.

Сигнатура метода обработчика событий (его параметры и возвращаемое значение) должна **совпадать** с указанной в EventHandler, иначе программа **не будет компилироваться**.

Переверните страницу и продолжим →

5 **Объект `Ball` оповещает подписчиков, что он в игре, с помощью события.**

Теперь, когда все настроено, объект `Ball` может вызвать свое событие в ответ на определенные действия симулятора. Вызвать событие легко:

```
EventHandler ballInPlay = BallInPlay;
if (ballInPlay != null)
    ballInPlay(this, e);
```

BallInPlay копируется в переменную ballInPlay, которая гарантированно не имеет значения null и используется для вызова события.

e — это новый объект BallEventArgs.

По мячу наносится удар, и объект Ball начинает действовать...

...создавая новый объект BallEventArgs с правильными данными...

...и передавая его возникшему событию.

BallEventArgs

BallInPlay(this, e)

Объект Ball

Событие активно. Кто на него подписан?

Событие BallInPlay

Объект Pitcher связывает свой обработчик с событием BallInPlay.

ball_BallInPlay()

Объект Pitcher

Поэтому метод объекта Pitcher вызывается с правильными данными и может работать с событием.

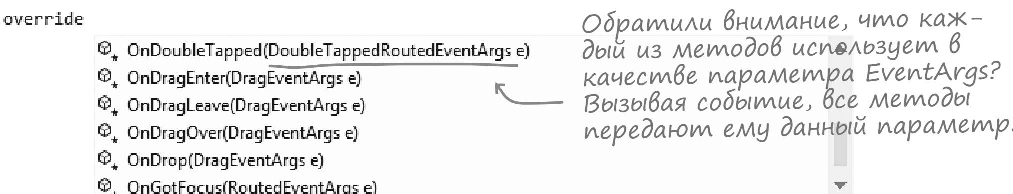
Будьте осторожны!

Событие, не имеющее обработчика, становится причиной исключения.

Если другие объекты не добавят событию свои обработчики, оно получает значение null, и появляется исключение `NullReferenceException`. *Именно поэтому нужно копировать событие в переменную перед проверкой на равенство null. В крайне редких случаях событие успевает приобрести это значение уже после проверки.*

Стандартные имена методов, вызывающих событие

Откройте код любой страницы и введите слово **`override`** в строку объявления метода. После нажатия пробела появится окно IntelliSense:



Обратили внимание, что каждый из методов использует в качестве параметра EventArgs? Вызывая событие, все методы передают ему данный параметр.

Объект XAML Page может вызывать множество событий, причем каждое собственным методом. Метод `OnDoubleTapped()` вызывает событие `DoubleTappedEvent` (которое наследует от суперкласса `UIElement`). Поэтому объект `Ball` **имеет метод `OnBallInPlay`**, использующий в качестве параметра объект `BallEventArgs`. Симулятор вызывает этот метод, когда для мяча требуется событие `BallInPlay`, так что когда симулятор обнаруживает удар биты по мячу, он создает экземпляр `BallEventArgs`, содержащий информацию о траектории и дальности полета, который и передается методу `OnBallInPlay()`.

Часто задаваемые вопросы

В: Зачем в объявлении события писать слово `EventHandler`? Я думал, что обработчиком называют способ других объектов подписаться на событие.

О: Это верно, для подписки на событие пишется метод, называемый обработчиком событий. Но вы заметили, каким именно образом `EventHandler` использовался в объявлении события (на шаге #2) ***и*** в строке подписки (на шаге #4)? `EventHandler` определяет **сигнатуру** события, он сообщает объектам, подписывающимся на событие, каким именно образом они должны определить методы-обработчики. А для подписки метода на события он должен иметь два параметра (`object` и ссылку `EventArgs`) и не возвращать значения.

В: Что произойдет при попытке воспользоваться методом, не совпадающим с определенным при помощи `EventHandler`?

О: Программа не будет компилироваться. Компилятор следит за тем, чтобы вы случайно не подписались на метод-обработчик, несовместимый с событием. Стандартный обработчик событий `EventHandler` показывает, как именно должны выглядеть ваши методы.

В: Что значит «стандартный» обработчик событий? Разве есть и другие?

О: Да! Ваши события ***вместо*** объекта и ссылки `EventArgs` могут отправлять что угодно или вообще ничего! Посмотрите на окно IntelliSense в нижней части страницы. Обратили внимание, как метод `OnDragDrop` принимает ссылку `DragEventArgs` вместо `EventArgs`? `DragEventArgs` наследует от `EventArgs` точно так же, как и `BallEventArgs`. Событие формы `DragDrop` не использует `EventHandler`. Он берет `DragEventArgs`, и для его обработки ваш метод должен брать объект и ссылку `DragEventArgs`.

Параметры события определяются делегатами. Примеры делегатов — EventHandler и DragEventArgs. О том, что это такое, мы поговорим чуть позже,

В: Можно ли сделать так, чтобы обработчик событий возвращал значение?

О: Можно, но делать этого не следует. Ведь именно отсутствие возвращаемых значений позволяет соединять обработчики событий в цепочки, присоединяя к одному событию несколько обработчиков.

В: Зачем нужны цепочки?

О: Они позволяют подписать на одно событие несколько обработчиков. Эта процедура будет рассмотрена чуть позже.

В: И поэтому для добавления обработчика события я использовал оператор +=? Как будто добавляя новый обработчик к уже существующим?

О: Именно так! Благодаря оператору `+=` ваш обработчик событий не замещает предыдущий. Он становится еще одним в цепочке обработчиков одного и того же события.

В: Почему при вызове события `BallInPlay()` использовалось слово this?

О: Это первый параметр стандартного обработчика событий. Вы заметили, что любой обработчик события `Click` имеет параметр object sender? Это **ссылка на вызывающий событие объект**. То есть когда вы обрабатываете щелчок на кнопке, объект `sender` указывает на эту кнопку. При обработке события `BallInPlay` параметр `sender` будет указывать на объект `Ball`, а мяч при вызове события обозначит этот параметр ключевым словом `this`.

ОДНО событие всегда вызывается ОДНИМ объектом. Но отвечать на него могут МНОГИЕ объекты.

Автоматическое создание обработчиков событий

Большинство программистов присваивают имена обработчикам событий по одному и тому же принципу. Скажем, если объект `Ball` вызывает событие `BallInPlay`, а переменная, ссылающаяся на этот объект, называется `ball`, обработчику события присваивается имя `ball_BallInPlay()`. Соблюдать это правило не обязательно, но следование ему облегчает другим пользователям чтение вашего кода.

К счастью, IDE позволяет без проблем следовать этому неписаному правилу, ведь она умеет автоматически добавлять обработчики событий. Вы уже сталкивались с этой функцией, но давайте рассмотрим ее еще раз.

1. **Создайте пустое приложение Windows Store и добавьте `Ball` и `BallEventArgs`.**
Вот код для класса `Ball`:

```
class Ball {
    public event EventHandler BallInPlay;
    public void OnBallInPlay(BallEventArgs e) {
        EventHandler ballInPlay = BallInPlay;
        if (ballInPlay != null)
            ballInPlay(this, e);
    }
}
```

А это класс `BallEventArgs`:

```
class BallEventArgs : EventArgs {
    public int Trajectory { get; private set; }
    public int Distance { get; private set; }
    public BallEventArgs(int trajectory, int distance) {
        this.Trajectory = trajectory;
        this.Distance = distance;
    }
}
```

2. **Добавим конструктор для класса `Pitcher`.**
Конструктор класса `Pitcher` будет содержать одну строчку, добавляющую обработчик событий к `ball.BallInPlay`. Начните вводить оператор, но пока не набирайте **+=**.

```
public Pitcher(Ball ball) {
    ball.BallInPlay
}
```

3 IDE поможет вам сэкономить время

Как только вы введете оператор +=, появится окно:

```
public Pitcher(Ball ball) {
    ball.BallInPlay +=
}
```

ball_BallInPlay;   (Press TAB to insert)

Нажмите tab для завершения ввода оператора. Вот как он выглядит:

```
public Pitcher(Ball ball) {
    ball.BallInPlay += ball_BallInPlay;
}
```

4 Обработчик событий тоже будет добавлен автоматически

Нужно добавить еще один метод в цепочку события. К счастью, это тоже можно сделать средствами IDE.

```
ball.BallInPlay += ball_BallInPlay;
```

Press TAB to generate handler 'ball_BallInPlay' in this class

Нажмите клавишу tab, чтобы добавить этот обработчик событий в класс `Pitcher`. Выбор имени будет проведен по схеме `objectName_HandlerName()`:

```
void ball_BallInPlay(object sender, EventArgs e) {
    throw new NotImplementedException();
}
```

IDE по умолчанию вставляет исключение NotImplementedException() в качестве заполнителя. Если вы не введете сюда нужный код и запустите программу, появится сообщение о том, что метод не реализован.

5 Завершение обработчика событий для класса pitcher

Вы добавили скелет, теперь нужно нарастить на него мышцы. Нападающий подает низко летящие мячи или защищает первую базу.

```
void ball_BallInPlay(object sender, EventArgs e) {
    if (e is BallEventArgs) {
        BallEventArgs ballEventArgs = e as BallEventArgs;
        if ((ballEventArgs.Distance < 95) && (ballEventArgs.Trajectory < 60))
            CatchBall();
        else
            CoverFirstBase();
    }
}
```

Так как класс BallEventArgs производный по отношению к классу EventArgs, мы прибегнем к нисходящему приведению при помощи ключевого слова as для получения доступа к его свойствам.

Эти методы будут добавлены через минуту.

Пришло время применить полученные знания на практике. Вам нужно закончить классы `Ball` и `Pitcher`, добавить класс `Fan` и убедиться, что они работают правильно.

1 **Завершение класса `Pitcher`.**

Ниже приведен код для класса `Pitcher`. Добавьте методы `CatchBall()` и `CoverFirstBase()`. Эти методы должны создавать строку, информирующую о том, что защитник поймал мяч и перебежал на первую базу. Данная строка добавляется в открытую коллекцию `ObservableCollection<string>`, которая называется `PitcherSays`.

```
class Pitcher {
    public Pitcher(Ball ball) {
        ball.BallInPlay += new EventHandler(ball_BallInPlay);
    }

    void ball_BallInPlay(object sender, EventArgs e) {
        if (e is BallEventArgs){
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if ((ballEventArgs.Distance < 95) && (ballEventArgs.Trajectory < 60))
                CatchBall();
            else
                CoverFirstBase();
        }
    }
}
```

Для добавления строки в коллекцию PitcherSays ObservableCollection нужно реализовать эти два метода.

Объект Pitcher

2 **Создание класса `Fan`.**

Конструктор класса `Fan` должен быть подписан на событие `BallInPlay`. Его обработчик событий должен проверять, не превышает ли расстояние 400 футов, а траектория — 30 (хоумран), и при превышении значений пытаться поймать мяч. В противном случае болельщик кричит. Выведите результат на консоль. Все крики фанатов следует добавить в коллекцию `ObservableCollection<string>`, которая называется `FanSays`.

Хоум-ран — удар, при котором мяч пролетает все поле и вылетает за его пределы.

Точный список вывода вы увидите на рисунке на следующей странице.

Объект Fan

3 **Построение простого симулятора.**

Если вы этого еще не сделали, создайте новое приложение для магазина Windows, замените страницу *MainPage.xaml* шаблоном Basic Page и добавьте следующий класс `BaseballSimulator`. Превратите его в статический ресурс страницы.

```
using System.Collections.ObjectModel;

class BaseballSimulator {
    private Ball ball = new Ball();
    private Pitcher pitcher;
    private Fan fan;

    public ObservableCollection<string> FanSays { get { return fan.FanSays; } }
    public ObservableCollection<string> PitcherSays { get { return pitcher.PitcherSays; } }

    public int Trajectory { get; set; }
    public int Distance { get; set; }

    public BaseballSimulator()     {
        pitcher = new Pitcher(ball);
        fan = new Fan(ball);
    }

    public void PlayBall() {
        BallEventArgs ballEventArgs = new BallEventArgs(Trajectory, Distance);
        ball.OnBallInPlay(ballEventArgs);
    }
}
```

4 **Построение главной страницы.**

Можете написать код XAML для представленного справа экрана? Два элемента управления TextBox привязаны к свойствам `Trajectory` и `Distance` статического ресурса `BaseballSimulator`. Реплики подающего и фанатов отображаются в элементах ListView, связанных с коллекциями `ObservableCollections`.

Trajectory
0
Distance
0
Play ball!
Не забудьте обработчик события Click для кнопки.

Pitcher says
Pitch #1: I covered first base
Pitch #2: I caught the ball
Pitch #3: I covered first base

Fan says
Pitch #1: Woo-hoo! Yeah!
Pitch #2: Woo-hoo! Yeah!
Pitch #3: Home run! I'm going for the ball!

Симулятор должен сгенерировать реплики подающего и фанатов для трех вброшенных мячей. Запишите, какие значения вы использовали для получения указанного результата:

Мяч 1:	**Мяч 2:**	**Мяч 3:**
Траектория:	Траектория:	Траектория: ....
Расстояние:	Расстояние:	Расстояние: ....

Упражнение Решение

Вот как выглядит код с написанными классами `Ball` и `Pitcher` и добавленным классом `Fan`.

Вот уже знакомые объекты `Ball` и `BallEventArgs` и новый класс `Fan`:

```
class Ball {
    public event EventHandler BallInPlay;
    public void OnBallInPlay(BallEventArgs e) {
        EventHandler ballInPlay = BallInPlay;
        if (ballInPlay != null)
            ballInPlay(this, e);
    }
}
```

Метод OnBallInPlay() вызывает событие BallInPlay. Не забудьте проверить, не равно ли его значение null, иначе он станет причиной исключения.

```
class BallEventArgs : EventArgs {
    public int Trajectory { get; private set; }
    public int Distance { get; private set; }
    public BallEventArgs(int trajectory, int distance)
    {
        this.Trajectory = trajectory;
        this.Distance = distance;
    }
}
```

В качестве аргументов события прекрасно подходят автоматически добавляемые, предназначенные только для чтения свойства. Ведь обработчики событий только читают передаваемые им данные.

```
using System.Collections.ObjectModel;
class Fan {
    public ObservableCollection<string> FanSays = new ObservableCollection<string>();
    private int pitchNumber = 0;

    public Fan(Ball ball) {
        ball.BallInPlay += new EventHandler(ball_BallInPlay);
    }

    void ball_BallInPlay(object sender, EventArgs e)  {
        pitchNumber++;
        if (e is BallEventArgs) {
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if (ballEventArgs.Distance > 400 && ballEventArgs.Trajectory > 30)
                FanSays.Add("Pitch #" + pitchNumber
                            + ": Home run! I'm going for the ball!");
            else
                FanSays.Add("Pitch #" + pitchNumber + ": Woo-hoo! Yeah!");
        }
    }
}
```

Конструктор объекта Fan привязывает свой обработчик события к событию BallInPlay.

Обработчик события BallInPlay класса fan высматривает высоко летящие мячи, брошенные на слишком длинную дистанцию.

Из программного кода следовало добавить только **метод обработчика события `Button_Click()`**:

```
private void Button_Click(object sender, RoutedEventArgs e) {
    baseballSimulator.PlayBall();
}
```

Этот статический ресурс идет в раздел Page.Resources.

Это XAML-код страницы. Ему требуется: `<local:BaseballSimulator x:Name="baseballSimulator"/>`

```
<Grid Grid.Row="1" Margin="120,0" DataContext="{StaticResource ResourceKey=baseballSimulator}">
  <Grid.ColumnDefinitions>
      <ColumnDefinition Width="200" />
      <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <StackPanel Margin="0,0,40,0">
    <TextBlock Text="Trajectory" Style="{StaticResource GroupHeaderTextStyle}" Margin="0,0,0,20"/>
    <TextBox Text="{Binding Trajectory, Mode=TwoWay}"  Margin="0,0,0,20"/>
    <TextBlock Text="Distance" Style="{StaticResource GroupHeaderTextStyle}" Margin="0,0,0,20"/>
    <TextBox Text="{Binding Distance, Mode=TwoWay}" Margin="0,0,0,20"/>
    <Button Content="Play ball!" Click="Button_Click"/>
  </StackPanel>
  <StackPanel Grid.Column="1">
    <TextBlock Text="Pitcher says" Style="{StaticResource GroupHeaderTextStyle}" Margin="0,0,0,20"/>
    <ListView ItemsSource="{Binding PitcherSays}" Height="150"/>
    <TextBlock Text="Fan says" Style="{StaticResource GroupHeaderTextStyle}" Margin="0,0,0,20"/>
    <ListView ItemsSource="{Binding FanSays}" Height="150"/>
  </StackPanel>
</Grid>
```

А это класс `Pitcher` (на верх файла нужно добавить строку `using System.Collections.ObjectModel;`):

```
class Pitcher {
    public ObservableCollection<string> PitcherSays = new ObservableCollection<string>();
    private int pitchNumber = 0;

    public Pitcher(Ball ball)     {
        ball.BallInPlay += ball_BallInPlay;
    }

    void ball_BallInPlay(object sender, EventArgs e) {
        pitchNumber++;
        if (e is BallEventArgs) {
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if ((ballEventArgs.Distance < 95) && (ballEventArgs.Trajectory < 60))
                CatchBall();
            else
                CoverFirstBase();
        }
    }
    private void CatchBall() {
        PitcherSays.Add("Pitch #" + pitchNumber + ": I caught the ball");
    }
    private void CoverFirstBase() {
        PitcherSays.Add("Pitch #" + pitchNumber + ": I covered first base");
    }
}
```

Мы предоставили вам обработчик события BallInPlay подающего. Он высматривает все низко летящие мячи.

Вот значения, которые мы использовали, чтобы получить показанный на рисунке результат. Вы можете попробовать и другие комбинации.

Мяч 1:	**Мяч 2:**	**Мяч 3:**
Траектория: 75	Траектория: 48	Траектория: 40
Расстояние: 105	Расстояние: 80	Расстояние: 435

Обобщенный EventHandler

Посмотрим на объявление события в классе Ball:

```
public event EventHandler BallInPlay;
```

Теперь откройте любое приложение Windows Forms и обратите внимание на объявление события Click у формы кнопки и других элементов управления, которые вы использовали в первой части книги:

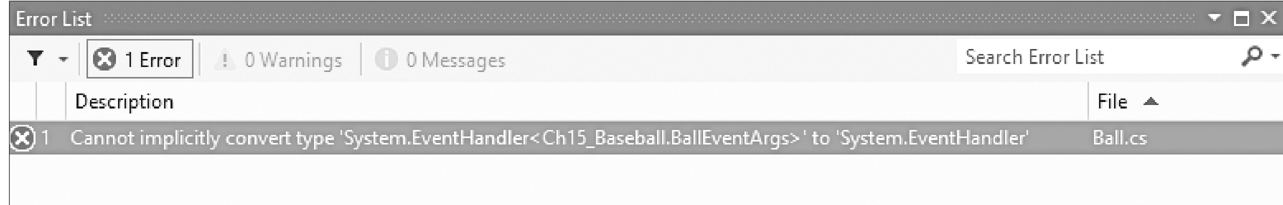
```
public event EventHandler Click;
```

Они называются по-разному, но объявляются одним и тем же способом. А так как все это прекрасно работает, посторонние могут и не знать, что обработчик BallEventHandler передает BallEventArgs при возникновении события. К счастью, в .NET имеется инструмент, позволяющий сообщить эту информацию, — обобщенный EventHandler. Измените обработчик события BallInPlay вот таким образом:

Обобщенный аргумент EventHandler должен быть производным от EventArgs.

```
public event EventHandler<BallEventArgs> BallInPlay;
```

Кроме того, в методе OnBallInPlay следует заменить EventHandler на EventHandler<BallEventArgs>. Перестройте код. Появится сообщение об ошибке:



Вы отредактировали объявление события, значит, нужно обновить и ссылку на него в классе Ball:

```
EventHandler<BallEventArgs> ballInPlay = BallInPlay;
if (ballInPlay != null)
    ballInPlay(this, e);
```

Неявное приведение при отсутствии ключевого слова new и типа

Несколько страниц назад IDE автоматически создала вот этот метод обработки события:

```
ball.BallInPlay += ball_BallInPlay;
```

В подобных случаях C# выполняет **неявное приведение** и самостоятельно определяет тип. Попробуйте вместо этой строки ввести в классе Pitcher или Fan:

```
ball.BallInPlay += new EventHandler<BallEventArgs>(ball_BallInPlay);
```

Программа все равно будет работать, так как IDE автоматически сгенерировала код с неявным приведением. А значит, вам не нужно редактировать тип при изменении типа события.

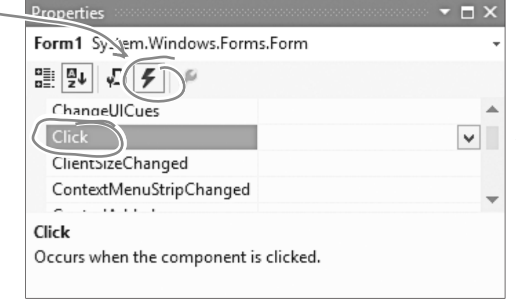
Все формы используют события

Создавая кнопку, дважды щелкая на ней в конструкторе формы и прописывая код для метода `button1_Click()`, вы работаете с событиями.

1. Создайте проект **Windows Forms Application**. Щелкните на кнопке Events (она помечена значком молнии) в окне Properties формы. Откроется **список событий**:

Для просмотра событий, связанных с элементом управления, выделите этот элемент и щелкните на кнопке со значком молнии.

Чтобы создать событие, возникающее при щелчке на форме, нужно выбрать в раскрывающемся списке в строчке Click вариант Form1_Click.



Найдите строку Click и дважды щелкните на ней. Будет добавлен новый обработчик события, который активизируется при любом щелчке на форме. В файле Form1.Designer.cs появится новая строка, связывающая обработчик и событие.

2. После двойного щелчка на строке Click к форме будет автоматически добавлен обработчик события `Form1_Click`. Введите для него код:

```
private void Form1_Click(object sender, EventArgs e) {
    MessageBox.Show("You just clicked on the form");
}
```

3. Visual Studio не только пишет за вас объявление метода, но и связывает обработчик с событием `Click` формы. Откройте файл `Form1.Designer.cs` и воспользуйтесь функцией Quick Find (Edit >> Find and Replace >> Quick Find) для поиска текста `Form1_Click`. Вы найдете строчку:

```
this.Click += new System.EventHandler(this.Form1_Click);
```

Запустите программу и убедитесь, что код работает!

Переверните страницу и продолжим!

Несколько обработчиков одного события

Часто Задаваемые Вопросы

В: Почему после добавления нового обработчика событий к объекту Pitcher появилось сообщение об исключении?

О: IDE добавила код, вызывающий исключение NotImplementedException, чтобы напомнить вам, что метод пока не реализован. Данное исключение обычно используется как местозаполнитель, когда вы создаете скелет класса, но пока не готовы писать код целиком. Появление этого исключения говорит, что вам просто нужно закончить работу над кодом.

События можно соединять в **цепочки**, чтобы одно событие вызывало один за другим несколько методов. Добавим к вашей форме несколько кнопок, чтобы посмотреть, как это работает.

4. Добавьте к форме два метода:

```
private void SaySomething(object sender, EventArgs e) {
    MessageBox.Show("Something");
}
private void SaySomethingElse(object sender, EventArgs e) {
    MessageBox.Show("Something else");
}
```

5. Добавьте к форме две кнопки и двойным щелчком на каждой из них добавьте обработчики событий:

```
private void button1_Click(object sender, EventArgs e) {
    this.Click += new EventHandler(SaySomething);
}
private void button2_Click(object sender, EventArgs e) {
    this.Click += new EventHandler(SaySomethingElse);
}
```

Вспомните о том, что каждая из кнопок **связывает новый обработчик с событием Click формы**. В первых трех шагах вы использовали IDE, чтобы добавить обработчик событий, вызывающий окно диалога при щелчке на форме, — код с оператором +=, осуществляющим привязку к обработчику, вставлялся в файл `Form1.Designer.cs`.

Теперь вы добавили кнопки, которые используют аналогичный синтаксис, чтобы сформировать цепочку дополнительных обработчиков события Click. Подумайте, что будет, если запустить программу и по очереди щелкнуть на первой, потом на второй кнопке, а потом на самой форме.

Для этого проекта мы взяли приложение Windows Forms, чтобы воспользоваться работой форм с событиями. Это применимо к любым событиям, но проще всего рассмотреть событие Click кнопки.

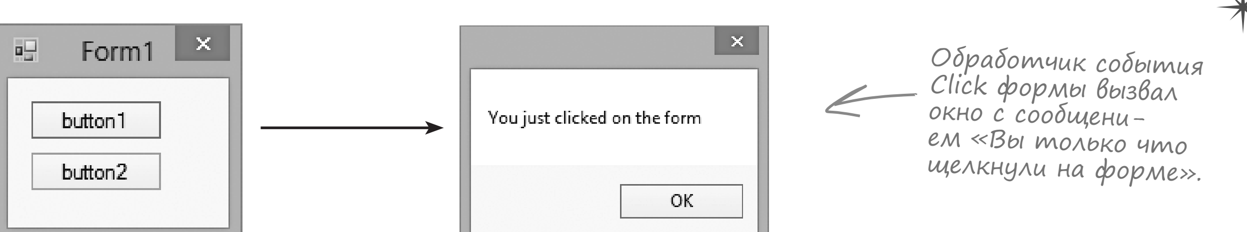
Будьте осторожны!

Обработчики событий должны быть «связаны».

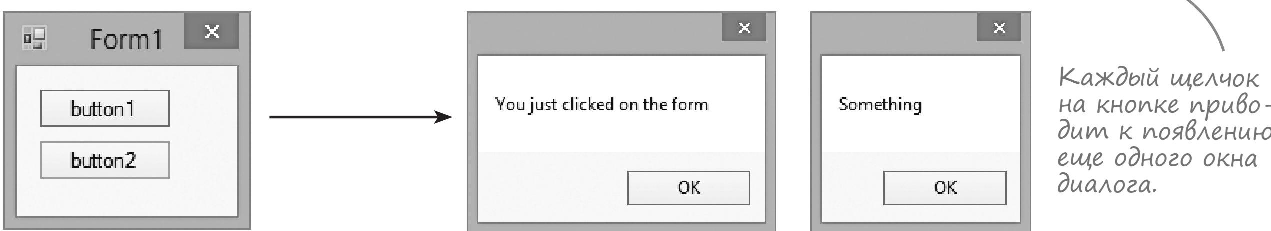
Если перетащить кнопку на форму, добавить метод `button1_Click()` *с правильными параметрами, но* ***не связанный с кнопкой****, он не будет вызываться. Дважды щелкните на кнопке в конструкторе, IDE возьмет заданный по умолчанию обработчик* `button1_Click_1()` *и добавит его к кнопке.*

Запустите программу и выполните следующие действия:

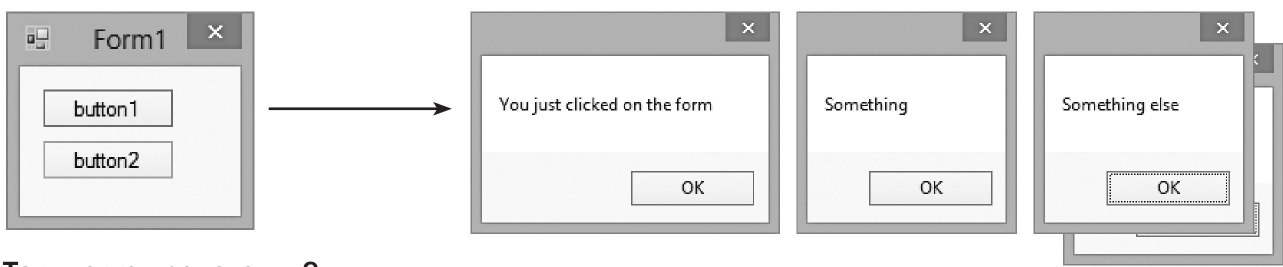
★ **Щелкните на форме.** Появится окно с текстом «You just clicked on the form».



★ Теперь **щелкните на кнопке button1**, а затем **снова на форме**. Появятся два окна с текстом: «You just clicked on the form» и «Something».



★ **Дважды щелкните на кнопке button2**, а затем **снова на форме**. Появятся четыре окна: «You just clicked on the form», «Something», «Something else» и «Something else».

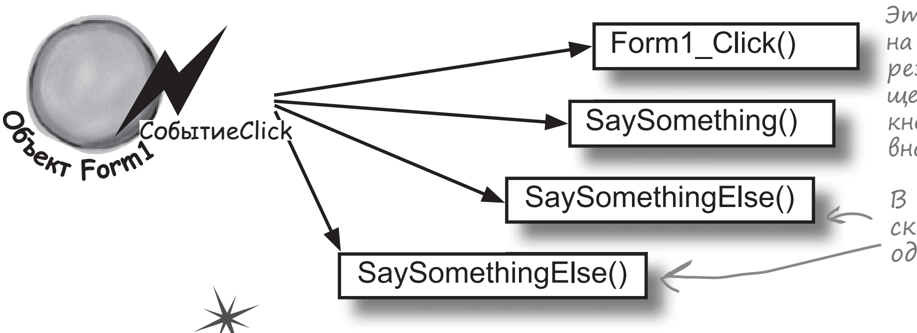


Так что же происходит?

Каждый щелчок на одной из кнопок по цепочке вызывает другой метод — `Something()` или `SomethingElse()` в ответ на событие `Click` формы. Если продолжить щелкать на кнопках, продолжится **вызов тех же методов**. Событию безразлично количество методов в цепочке, вы даже можете несколько раз подсоединить один и тот же метод. Все они будут вызываться в том порядке, в котором были добавлены, при каждом появлении события.

При щелчке на кнопках по цепочке вызываются другие обработчики события Click формы.

Это означает, что щелчок на кнопках не даст никакого результата! Сначала нужно щелкнуть на форме, так как кнопки меняют ее поведение, внося изменения в событие Click.



В цепочку можно несколько раз добавить один и тот же метод.

Приложение для магазина Windows можно завершить, вызвав Application.Current.Exit(), но хорошо спроектированному приложению это не требуется благодаря управлению жизненным циклом.

Приложения для магазина Windows и события

Возможно, вы уже обратили внимание на фундаментальное отличие приложений для магазина Windows от приложений рабочего стола: у первых отсутствует средство, позволяющее закрыть приложение. Но давайте подумаем... зачем нам нужно закрывать приложение? Вы можете переключиться на что-то другое. Представьте, что компьютер обладает достаточным количеством памяти и ресурсами процессора, чтобы держать приложение открытым на случай, если вы захотите к нему вернуться. После переключения на другую задачу Windows **приостанавливает** приложение, оставляя его в памяти со всеми объектами и ресурсами, которые требуются для работы. Когда Windows потребуется освободить память, она **завершает** работу приложения, выгружая его и все используемые им ресурсы. Но разве вам как пользователю не все равно, приостановило или прекратило свою работу приложение? Ведь после возобновления работы приложение оказывается в нужном пользователю состоянии. Приостановка и возобновление работы приложения операционной системой Windows называются **управлением жизненным циклом**.

События, связанные с управлением жизненным циклом

Каждый раз, когда Windows приостанавливает приложение для магазина Windows, возникает событие Suspending, сохраняющее состояние приложения.

Откройте любое приложение для магазина Windows и дважды щелкните на строке *App.xaml.cs* в окне Solution Explorer. Найдите конструктор `App`:

```
public App()
{
    this.InitializeComponent();
    this.Suspending += OnSuspending;
}
```

Вам должно быть понятно, что здесь происходит. Класс `App`, который является подклассом `Application` в пространстве имен `Windows.UI.Xaml`, обладает событием `Suspending`, которое в конструкторе связано с обработчиком событий OnSuspending. Щелкните правой кнопкой мыши на строке `Suspending` и выберите команду Go To Definition для перехода на вкладку Application [from metadata] с членами класса `Application`. Перейдите к событию **Suspending**:

```
//
// Summary:
//     Occurs when the application transitions to Suspended state from some other
//     state.
public event SuspendingEventHandler Suspending;
```

Это событие возникает всякий раз, когда пользователь куда-то переключается. То есть при каждой приостановке работы приложения вызывается метод `OnSuspending()` в файле *App.xaml.cs.* Загрузку приложения сопровождает вызов метода `OnLaunched()` в файле *App.xaml.cs.*

Windows может завершить работу приостановленного приложения в любой момент. Поэтому оно должно быть сконструировано так, чтобы при **приостановке работы состояние сохранялось**. Метод `OnLaunched()` проверяет, после приостановки или с нуля запускается приложение.

Управление жизненным циклом приложения Джимми

Заставим приложение для работы с каталогом комиксов сохранять и восстанавливать текущую страницу. Обработчик события Suspending при переключении на другую задачу должен записывать имя текущего запроса в файл в папке *local* приложения. Если Windows завершит работу приложения, повторная загрузка вернет нас на последнюю использовавшуюся страницу.

Добавим класс для сохранения и загрузки состояния.
Добавьте класс SuspensionManager. Он содержит статическое свойство для слежения за текущим запросом и два статических метода для чтения и записи имени запроса в файл *_sessionState.txt* в папке *local* приложения.

```
using Windows.Storage;
class SuspensionManager {
    public static string CurrentQuery { get; set; }

    private const string filename = "_sessionState.txt";

    static async public Task SaveAsync() {
        if (String.IsNullOrEmpty(CurrentQuery))
            CurrentQuery = String.Empty;
        IStorageFile storageFile =
               await ApplicationData.Current.LocalFolder.CreateFileAsync(
                          filename, CreationCollisionOption.ReplaceExisting);
        await FileIO.WriteTextAsync(storageFile, CurrentQuery);
    }

    static async public Task RestoreAsync() {
        IStorageFile storageFile =
               await ApplicationData.Current.LocalFolder.GetFileAsync(filename);
        CurrentQuery = await FileIO.ReadTextAsync(storageFile);
    }
}
```

2 **Пусть главная страница обновляет SuspensionManager при загрузке запроса.**
Элемент ListView в файле *MainPage.xaml* заставляет приложение при щелчке на выбранном элементе переходить на другую страницу, поэтому **добавьте к обработчику события ItemClick** строку, присваивающую статическому свойству CurrentQuery объекта SuspensionManager заголовок запроса, который будет загружаться:

```
private void ListView_ItemClick(object sender, ItemClickEventArgs e) {
    ComicQuery query = e.ClickedItem as ComicQuery;
    if (query != null) {
        SuspensionManager.CurrentQuery = query.Title;
        if (query.Title == "All comics in the collection")
            this.Frame.Navigate(typeof(QueryDetailZoom), query);
        else
            this.Frame.Navigate(typeof(QueryDetail), query);
    }
}
```

При каждом щелчке на запросе обработчик события обновляет статическое свойство CurrentQuery.

3 Перекроем метод OnNavigatedFrom() для удаления сохраненного запроса.

Щелчок пользователя на стрелке для возвращения к предыдущей странице сопровождается событием **NavigatedFrom**. Обновите программный код для ***обеих*** страниц — `QueryDetail` и `QueryDetailZoom`, перекрыв метод `OnNavigatedFrom()`, который вызывается сразу после ухода со страницы. **Откройте файл *QueryDetail.xaml.cs*** и добавьте в класс ключевое слово `override`, затем при помощи функции IntelliSense создайте заглушку метода:

```
override
    OnManipulationStarting(ManipulationStartingRoutedEventArgs e)
    OnNavigatedFrom(NavigationEventArgs e)          void Common.LayoutAwarePage.OnNavigatedFrom(NavigationEventArgs e)
                                                    Invoked when this page will no longer be displayed in a Frame.
    OnNavigatedTo(NavigationEventArgs e)
    OnNavigatingFrom(NavigatingCancelEventArgs e)
    OnPointerCanceled(PointerRoutedEventArgs e)
    OnPointerCaptureLost(PointerRoutedEventArgs e)
    OnPointerEntered(PointerRoutedEventArgs e)
    OnPointerExited(PointerRoutedEventArgs e)
    OnPointerMoved(PointerRoutedEventArgs e)
```

При выборе в окне IntelliSense варианта `OnNavigatedFrom()` IDE добавит заглушку метода, вызывающего метод `OnNavigatedFrom()` базового класса. Добавьте строку, очищающую запрос. Обязательно **проделайте то же самое для файла *QueryDetailZoom.xaml.cs***.

```
protected override void OnNavigatedFrom(NavigationEventArgs e) {
    SuspensionManager.CurrentQuery = null;
    base.OnNavigatedFrom(e);
}
```

Когда страницы QueryDetail и QueryDetailZoom вызывают свои события OnNavigatedFrom для возващения на главную страницу, свойство CurrentQuery объекта SuspensionManager очищается.

4 Отредактируем обработчик события Suspending.

Откройте файл *App.xaml.cs* и найдите метод обработчика события `OnSuspending()`, связанный с событием `Suspending`. Он снабжен комментарием, начинающимся с `TODO`:

```
private void OnSuspending(object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    //TODO: Save application state and stop any background activity
    deferral.Complete();
}
```

Строки «TODO» входят в некоторые шаблоны IDE, указывая, куда можно добавлять код.

Вставьте вместо TODO вызов метода `SaveAsync()`. Обязательно добавьте к объявлению **async**, чтобы делать асинхронный вызов с помощью ключевого слова `await`:

```
async private void OnSuspending(object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    await SuspensionManager.SaveAsync();
    deferral.Complete();
}
```

Обновим метод OnLaunched.

До сих пор наши измнения были направлены на поддержание актуального состояния статического свойства CurrentQuery объекта SuspensionManager. Теперь осталось обновить обработчик события Launched в файле *App.xaml.cs*, чтобы вернуться в сохраненное состояние.

```
async protected override void OnLaunched(LaunchActivatedEventArgs args) {
    SettingsPane.GetForCurrentView().CommandsRequested += OnCommandsRequested;

    Frame rootFrame = Window.Current.Content as Frame;

    // Не повторяем инициализацию при наличии контента,
    // но гарантируем активность окна
    if (rootFrame == null)
    {
        // Создаем Frame как контекст навигации и переходим к первой странице
        rootFrame = new Frame();

        if (args.PreviousExecutionState == ApplicationExecutionState.Terminated) {
            await SuspensionManager.RestoreAsync();
        }

        // Place the frame in the current Window
        Window.Current.Content = rootFrame;
    }

    if (rootFrame.Content == null) {
        // Когда стек навигации не может перебросить на первую страницу,
        // конфигурируем новую страницу, передавая нужную информацию
        // как навигационный параметр
        if (!rootFrame.Navigate(typeof(MainPage), args.Arguments)) {
            throw new Exception("Failed to create initial page");
        }
        if (!String.IsNullOrEmpty(SuspensionManager.CurrentQuery)) {
            var currentQuerySequence =
                from query in new ComicQueryManager().AvailableQueries
                where query.Title == SuspensionManager.CurrentQuery
                select query;

            if (currentQuerySequence.Count() == 1) {
                ComicQuery query = currentQuerySequence.First();
                if (query != null) {
                    if (query.Title == "All comics in the collection")
                        rootFrame.Navigate(typeof(QueryDetailZoom), query);
                    else
                        rootFrame.Navigate(typeof(QueryDetail), query);
                }
            }
        }
    }
    // Гарантируем активность текущего окна
    Window.Current.Activate();
}
```

Чтобы применить оператор await.

Еще один оператор TODO загружает состояние ранее приостановленного приложения. Замените его вызовом метода RestoreAsync() объекта SuspensionManager.

Добавьте этот код для сравнения сохраненного состояния со списком запросов. При совпадении с известным запросом приложение переходит на страницу с его подробным описанием.

Обратите внимание на этот запрос LINQ. Понимаете, как он работает?

Suspend

Suspend

Resume

Suspend and shutdown

Для тестирования приложения можно воспользоваться раскрывающимся списком Suspend, который находится на панели Debug Location, видимой при запуске приложения в отладчике (если вы ее не видите, воспользуйтесь командой View→Toolbars). Щелкните Suspend and shutdown для завершения работы приложения и вызовите событие OnSuspending.

Элементы XAML пользуются перенаправленными событиями

Вернитесь на несколько страниц назад и посмотрите на окно IntelliSense, появляющееся при вводе слова `override`. Два типа аргументов события немного отличаются от остальных. Второй аргумент события `DoubleTapped` относится к типу `DoubleTappedRoutedEventArgs`, а у события `GotFocus` есть `RoutedEventArgs`. Дело в том, что `DoubleTapped` и `GotFocus` — **перенаправленные события (routed events)**. Когда управляющий объект реагирует на такое событие, он запускает метод обработчика событий. Если обработки не происходит, **перенаправленное событие отправляется контейнеру управляющего объекта**. Последний запускает событие, и если оно не обрабатывается, отправляет своему контейнеру. Событие **поднимается вверх**, пока не будет обработано или пока не достигнет **корня** или самого верхнего контейнера. Вот типичная сигнатура метода обработки перенаправленного события.

```
private void EventHandler(object sender, RoutedEventArgs e)
```

Объект `RoutedEventArgs` обладает свойством **`Handled`**, при помощи которого обработчик указывает на успешную обработку события. Присвоение этому свойству значения `true` **завершает подъем события вверх**.

Как для обычных, так и для перенаправленных событий параметр `sender` содержит ссылку на объект, который вызывает обработчик события. И если событие поднимается от элемента управления к контейнеру, например Grid, при вызове сеткой обработчика праметр `sender` превращается в ссылку на эту сетку. Как найти элемент, вызвавший событие первым? Объект `RoutedEventArgs` обладает свойством **`OriginalSource`**, содержащим ссылку именно на этот элемент. Если `OriginalSource` и `sender` указывают на один и тот же объект, значит, именно элемент, вызвавший обработчик, стал причиной появления события и его подъема вверх.

IsHitTestVisible и «видимость» элементов

Обычно на любом элементе страницы можно «щелкнуть» указателем мыши — при условии, что он отвечает определенным критериям. Он должен быть видимым (это зависит от свойства `Visibility`), должен иметь отличные от null свойства `Background` или `Fill` (но обладать свойством `Transparent`), должен быть доступен (через свойство `IsEnabled`) и обладать отличными от нуля свойствами `height` и `width`. При соблюдении этих условий **свойство `IsHitTestVisible` возвращает значение `True`**, и элемент отвечает на события указателя или мыши.

Это свойство особенно полезно, если элемент нужно сделать «невидимым» для мыши. Присвойте `IsHitTestVisible` значение `False`, и щелчки мыши будут **проникать сквозь элемент**. Реагировать на событие будет элемент, расположенный под ним.

Список перенаправленных событий находится здесь:
http://msdn.microsoft.com/ru-ru/library/windows/apps/Hh758286.aspx

Структура в виде элементов управления, содержащих другие элементы управления, называется деревом объектов, и перенаправленные события поднимаются от потомка к предку, пока не достигнут корневого элемента.

Исследуем перенаправленные события

Не забудьте заменить MainPage.xaml на новый шаблон Basic Page.

Вот приложение для магазина Windows, которое мы используем для экспериментов. Элемент StackPanel содержит Border, который содержит Grid с элементами Ellipse и Rectangle внутри. Видите на рисунке прямоугольник поверх эллипса? Это два элемента, положенные в одну ячейку. Но при этом у них общий предок: сетка Grid, имеющая предка Border, который является потомком элемента StackPanel. Перенаправленные события элемента Rectangle или Ellipse будут подниматься по **дереву объектов**.

Элемент **ToggleSwitch** позволяет перейти от включенного к выключенному состоянию и обратно. Текст заголовка определяется свойством `Header`, а для задания и получения этого значения пользуйтесь свойством `IsOn`.

The rectangle was pressed
The grid was pressed
The border was pressed
The panel was pressed
Border sets handled
Off
Grid sets handled
Off
Ellipse sets handled
Off
Rectangle sets handled
Off
Update Rectangle IsHitTestVisible
New IsHitTestVisible value
On

Перенаправленные события поднимаются по дереву объектов.

```
<Grid Grid.Row="1" Margin="120,0">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <StackPanel x:Name="panel" PointerPressed="StackPanel_PointerPressed">
        <Border BorderThickness="10" BorderBrush="Blue" Width="155" x:Name="border"
                Margin="20" PointerPressed="Border_PointerPressed">
            <Grid x:Name="grid" PointerPressed="Grid_PointerPressed">
                <Ellipse Fill="Red"  Width="100" Height="100"
                      PointerPressed="Ellipse_PointerPressed"/>
                <Rectangle Fill="Gray" Width="50" Height="50"
                      PointerPressed="Rectangle_PointerPressed" x:Name="grayRectangle"/>
            </Grid>
        </Border>
        <ListBox BorderThickness="1" Width="300" Height="250" x:Name="output" Margin="0,0,20,0"/>
    </StackPanel>

    <StackPanel Grid.Column="1">
        <ToggleSwitch Header="Border sets handled" x:Name="borderSetsHandled"/>
        <ToggleSwitch Header="Grid sets handled" x:Name="gridSetsHandled" />
        <ToggleSwitch Header="Ellipse sets handled" x:Name="ellipseSetsHandled"/>
        <ToggleSwitch Header="Rectangle sets handled" x:Name="rectangleSetsHandled"/>
        <Button Content="Update Rectangle IsHitTestVisible"
              Click="UpdateHitTestButton" Margin="0,20,20,0"/>
        <ToggleSwitch IsOn="True" Header="New IsHitTestVisible value"
              x:Name="newHitTestVisibleValue" />
    </StackPanel>
</Grid>
```

IsOn по умолчанию False. Переключатель присваивает ему значение True, так как свойство IsHitTestVisible элементов управления по умолчанию имеет значение true.

Переверните страницу, чтобы завершить приложение

ObservableCollection для отображения результатов в элементе ListBox.
Создайте поле `outputItems` и задайте свойство `ListBox.ItemsSource` в конструкторе страницы. Добавьте к коллекции `ObservableCollection<T>` оператор `using System.Collections.ObjectModel;`.

```
public sealed partial class MainPage : RoutedEvents.Common.LayoutAwarePage {
    ObservableCollection<string> outputItems = new ObservableCollection<string>();

    public MainPage() {
        this.InitializeComponent();

        output.ItemsSource = outputItems;
    }
```

Вот программный код. Обработчик события `PointerPressed` каждого элемента управления очищает результат в случае исходного источника и добавляет к результату строку. При включенном состоянии «обработанного» переключателя для обработки события используется `e.Handled`.

```
private void Ellipse_PointerPressed(object sender, PointerRoutedEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The ellipse was pressed");
    if (ellipseSetsHandled.IsOn) e.Handled = true;
}

private void Rectangle_PointerPressed(object sender, PointerRoutedEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The rectangle was pressed");
    if (rectangleSetsHandled.IsOn) e.Handled = true;
}

private void Grid_PointerPressed(object sender, PointerRoutedEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The grid was pressed");
    if (gridSetsHandled.IsOn) e.Handled = true;
}

private void Border_PointerPressed(object sender, PointerRoutedEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The border was pressed");
    if (borderSetsHandled.IsOn) e.Handled = true;
}

private void StackPanel_PointerPressed(object sender, PointerRoutedEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The panel was pressed");
}

private void UpdateHitTestButton(object sender, RoutedEventArgs e) {
    grayRectangle.IsHitTestVisible = newHitTestVisibleValue.IsOn;
}
```

Некоторые перенаправленные обработчики событий получат для события PointerPressed подкласс RoutedEventArgs, например PointerRoutedEventArgs.

Обработчик события Click кнопки использует свойство IsOn переключателя для изменения состояния свойства IsHitTestVisible элемента управления Rectangle.

Это граф объектов главной страницы.
Класс MainPage является корнем дерева объектов. Исходная страница *MainPage.xaml* является экземпляром объекта Page, но применение шаблона Basic Page расширяет **подкласс Page, который называется LayoutAwarePage**.

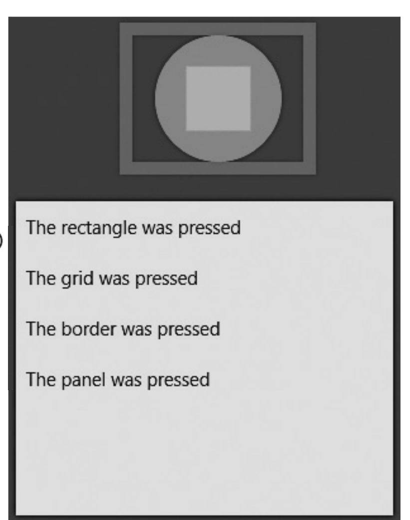
Эта сетка добавлена на главную страницу как часть шаблона Basic Page.

Это элемент StackPanel, содержащий элементы Border, Grid, Ellipse и Rectangle.

Эта сетка получает перенаправленные события PointerPressed, но не может заставить их подниматься. Ее свойство IsHitTestVisible по умолчанию имеет значение False, так как у него отсутствуют свойства Background и Fill. Если обновить код XAML, добавив к нему свойство Background, свойство IsHitTestVisible по умолчанию true, даже если присвоить ему значение Transparent. Это даст возможность реагировать на нажатия указателя.

Переверните страницу, чтобы начать эксперимент →

Запустите приложение и коснитесь прямоугольника.



Результат показан справа. ⟶

Чтобы точно понять, что происходит, поместите точку останова на первую строку метода `Rectangle_PointerPressed()`, обработчика события `PointerPressed` элемента Rectangle:

```
private void Rectangle_PointerPressed(object sender, PointerRoutedEventArgs e)
{
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The rectangle was pressed");
    if (rectangleSetsHandled.IsOn) e.Handled = true;
}
```

Снова щелкните на сером прямоугольнике. После остановки используйте функцию Step Over (F10) для **пошагового прохода**. Сначала будет выполняться блок `if`, очищающий связанную с элементом ListBox коллекцию `outputItems ObservableCollection`. Это происходит потому, что `sender` и `e.OriginalSource` ссылаются на один и тот же элемент Rectangle, что верно только внутри метода обработки события вызвавшего это событие элемента (в данном случае элемента, на котором был сделан щелчок), поэтому выражение `sender == e.OriginalSource` имеет значение true.

После завершения метода **продолжайте пошаговый проход программы**. Событие будет подниматься по дереву объектов, запустив сначала обработчик события элемента Rectangle, затем элемента Grid, затем элемента Border, затем элемента Panel. Напоследок будет запущен метод обработки события, относящийся к классу `LayoutAwarePage`, он расположен вне вашего кода, не входит в перенаправленное событие и будет запущен в любом случае. Так как ни один из этих элементов управления не является вызвавшим событие, ни один из отправителей не совпадет с `e.OriginalSource` и не очистит результат работы программы.

Отключите IsHitTestVisible, нажмите кнопку «Update» и щелкните на прямоугольнике.

The ellipse was pressed

The grid was pressed

The border was pressed

The panel was pressed

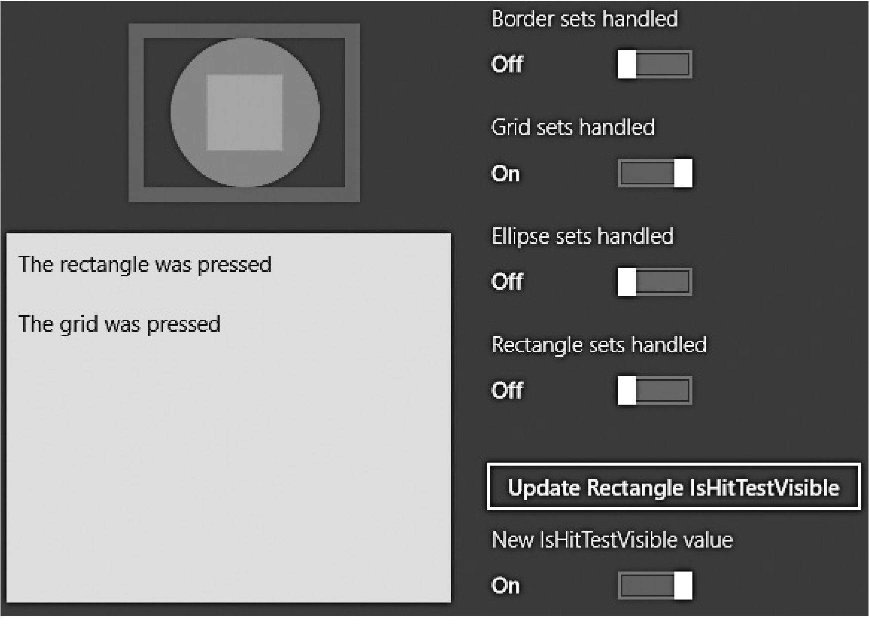
⟵ Результат.

Странно. Вы щелкнули на элементе Rectangle, а сработал обработчик события `PointerPressed` элемента Ellipse. Что происходит?

При нажатии кнопки обработчик события `Click` изменил свойство `IsHitTestVisible` элемента Rectangle на `false`, и элемент стал «невидимым» для событий, связанных с действиями мыши или указателя. Поэтому щелчок подействовал на самый верхний элемент, у которого `IsHitTestVisible` имеет значение `true`, а `Background` назначен цвет или значение `Transparent`. В нашем случае это элемент Ellipse, для которого и генерируется событие `PointerPressed`.

Включите переключатель «Grid sets handled» и щелкните на Rectangle.

Вот что получится. ⟶



Почему в ListBox появились только две строки? **Снова выполните пошаговый просмотр кода**. На этот раз свойство `gridSetsHandled.IsOn` имеет значение true, так как `gridSetsHandled` имеет значение `On`, и последняя строка **обработчика события элемента Grid присваивает свойству `e.IsHandled` значение `true`**. Как только метод обработчика перенаправленного события это делает, **событие перестает подниматься вверх**. После завершения обработчика события элемента Grid приложение видит, что событие успешно обработано, поэтому вместо методов обработчика событий элементов Border или Panel переходит к методу в классе `LayoutAwarePage`, не входящему в добавленный вами код.

Поэкспериментируйте с перенаправленными событиями.

Попробуйте выполнить следующие действия:

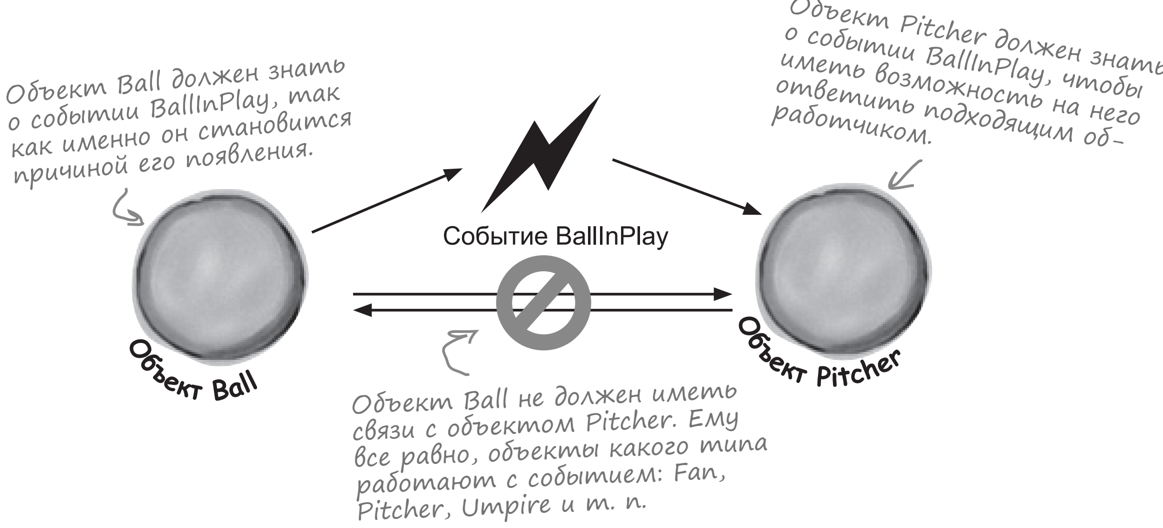
- Щелкните на сером прямоугольнике, потом на красном эллипсе и посмотрите, как будут подниматься события.
- Отключайте все переключатели, начиная с верхнего, чтобы заставить обработчики событий присвоить свойству `e.Handled` значение `true`. Обратите внимание, как обработанные события перестают подниматься.
- Ставьте точки останова и выполняйте отладку всех методов обработки событий.
- Установите точку останова на метод обработки события элемента Ellipse, включите свойство `IsHitTestVisible` прямоугольника нижним переключателем и нажатием кнопки. Выполните пошаговый проход кода для прямоугольника, когда свойство `IsHitTestVisible` имеет значение `false`.
- Остановите программу и добавьте элементу Grid свойство `Background`, чтобы сделать сетку чувствительной к действиям указателя.

Перенаправленное событие сначала вызывает обработчик для элемента, ставшего причиной события, затем или поднимает его по иерархии до верха, или присваивает свойству e.Handled значение true.

Связь между издателями и слушателями

Одним из самых сложных для понимания при изучении событий является обстоятельство, что **издатель (sender)** должен знать, какое событие он инициирует, в том числе какие аргументы передает. И **слушатель (listener)** должен знать, какой тип возвращаемого значения и аргументы следует использовать для методов обработчика событий.

Но связи между издателем и подписчиком нет. Издатель инициирует событие, и *ему все равно, кто на него подписан*. А подписчику важно, что это за событие, *но все равно, какой объект стал его причиной*. Получается, что как издатель, так и подписчик сфокусированы на событии, но не друг на друге.



«Мой народ хочет вступить в контакт с вашим народом».

Вы знаете, что делает этот код:

```
Ball currentBall;
```

Он создает **ссылочную переменную** на любой объект `Ball`. Эта переменная может указывать даже на значение null.

Событиям нужна такая же ссылка, но указывающая не на объект, а **на метод**. Событиям нужно отслеживать список подписанных на них методов. Как вы уже видели, эти методы могут принадлежать другим классам и даже быть закрытыми. Как же отследить все обработчики событий, которые будут вызываться? Для этого используются **делегаты (delegate)**.

> **Де-ле-гат, сущ.**
> человек, имеющий полномочия представлять других. *Президент отправил **делегата** на саммит.*

Делегат замещает методы

Для создания делегата нужно только указать сигнатуру методов, на которые он может ссылаться.

Возникающее событие **не знает**, методы-обработчики событий каких объектов будут вызваны. Каким же образом событие может управлять этим процессом?

Для этого в C# существуют **делегаты (delegate)**. Это ссылочный тип, позволяющий **ссылаться на методы внутри класса**... также делегаты являются основой для событий.

Данный делегат может ссылаться на любой метод, использующий в качестве параметров объект и EventArgs и не возвращающий значения.

В этой главе вам уже приходилось с ними сталкиваться. При создании события `BallInPlay` вы работали с делегатом `EventHandler`. Щелкните на нем правой кнопкой мыши и выберите команду «Go to definition», и вот что вы увидите.

```
public delegate void EventHandler(object sender, EventArgs e);
```

Этот элемент сигнатуры делегата показывает, что EventHandler может ссылаться только на методы, не возвращающие значения.

Имя делегата EventHandler.

Добавляем к проекту новый тип данных

Добавляя к проекту делегаты, вы добавляете данные нового типа. Используя их для создания поля или переменной, вы создаете **экземпляр** этого типа. **Откройте новый проект Console Application** и добавьте к нему новый файл классов `ConvertsIntToString.cs`. Введите одну строчку:

```
    delegate string ConvertsIntToString(int i);
```

Еще одним добавленным в проект делегатом является ConvertsIntToString. Его можно использовать для объявления переменных точно так же, как вы делали бы это для класса или интерфейса.

Добавьте в класс `Program` метод `HiThere()`:

```
private static string HiThere(int i)
{
    return "Hi there! #" + (i * 100);
}
```

Сигнатура этого метода совпадает с ConvertsIntToString.

Заполните метод `Main()`:

someMethod — это переменная типа ConvertsIntToString. От обычной ссылочной переменной она отличается тем, что указывает не на объект в куче, а на метод.

```
static void Main(string[] args)
{
    ConvertsIntToString someMethod = new ConvertsIntToString(HiThere);
    string message = someMethod(5);
    Console.WriteLine(message);
    Console.ReadKey();
}
```

Переменной someMethod можно присваивать значения, как и любой другой. При ее вызове вызывается метод, на который она ссылается.

Переменная someMethod указывает на метод `HiThere()`. Записью `someMethod(5)` вызывается метод `HiThere()`, которому передается аргумент 5. В итоге возвращается строка `Hi there! #500`. Просмотрите программу в режиме отладки, чтобы понять, что именно происходит.

Делегаты в действии

Еще одна программа, в которой используются Windows Forms. На этот раз нам нужен метод MessageBox.Show(), позволяющий увидеть, что происходит.

Работать с делегатами легко, они не требуют большого объема кода. Попробуем с их помощью помочь владельцу ресторана рассортировать секретные ингредиенты с кухни шеф-повара.

Добавьте делегат в новый проект

Делегаты обычно располагаются вне классов, поэтому добавьте к проекту новый файл классов `GetSecretIngredient.cs` и введите в него строку:

Удалите из нового файла класса объявление класса и замените его этой строкой кода.

```
delegate string GetSecretIngredient(int amount);
```

(Полностью удалите объявление класса.) Этот делегат станет основой переменной, указывающей на метод, который, взяв параметр типа `int`, возвращает строку.

Добавьте класс для первого шеф-повара, Сюзанны

`Suzanne.cs` содержит класс, следящий за секретными ингредиентами, которые использует Сюзанна. Он снабжен закрытым методом `SuzannesSecretIngredient()` с сигнатурой, совпадающей с `GetSecretIngredient`. Имеется и предназначенное только для чтения свойство, возвращающее `GetSecretIngredient`. С его помощью остальные объекты могут получить ссылку на метод `SuzannesIngredientList()`.

```
class Suzanne {
  public GetSecretIngredient MySecretIngredientMethod {
    get {
      return new GetSecretIngredient(SuzannesSecretIngredient);
    }
  }
  private string SuzannesSecretIngredient(int amount) {
      return amount.ToString() + " ounces of cloves";
  }
}
```

Взяв переменную amount типа int, метод возвращает строку, описывающую секретный ингредиент.

Свойство GetSecretIngredient возвращает новый экземпляр делегата GetSecretIngredient, указывающего на метод получения секретных ингредиентов.

Добавьте класс для второго шеф-повара, Эми

Методы Эми работают так же, как методы Сюзанны:

```
class Amy {
    public GetSecretIngredient AmysSecretIngredientMethod {
        get {
            return new GetSecretIngredient(AmysSecretIngredient);
        }
    }
    private string AmysSecretIngredient(int amount) {
        if (amount < 10)
            return amount.ToString()
                      + " cans of sardines -- you need more!";
        else
            return amount.ToString() + " cans of sardines";
    }
}
```

Метод получения секретных ингредиентов, с которыми работает Эми, также берет переменную amount типа int и возвращает строку. Но это уже другая строка.

Secret Ingredients

Get the ingredient | 5

Get Suzanne's delegate

Get Amy's delegate

4 **Постройте форму.**

Вот код формы:

```
GetSecretIngredient ingredientMethod = null;
Suzanne suzanne = new Suzanne();
Amy amy = new Amy();

private void useIngredient_Click(object sender, EventArgs e) {
    if (ingredientMethod != null)
        MessageBox.Show("I'll add " + ingredientMethod((int)amount.Value));
    else
        MessageBox.Show("I don't have a secret ingredient!");
}

private void getSuzanne_Click(object sender, EventArgs e) {
    ingredientMethod = new GetSecretIngredient(suzanne.MySecretIngredientMethod);
}

private void getAmy_Click(object sender, EventArgs e) {
    ingredientMethod = new GetSecretIngredient(amy.AmysSecretIngredientMethod);
}
```

Убедитесь, что элементу NumericUpDown дано имя «amount».

5 **Посмотрите на работу делегатов при помощи отладчика**

Воспользуйтесь отладчиком, чтобы понять принцип работы делегатов:

- Запустите программу. Щелкните на кнопке «Get the ingredient», на консоль будет выведена строка «I don't have a secret ingredient!»
- Щелкните на кнопке «Get Suzanne's delegate», которая присваивает полю `ingredientMethod` (являющемуся делегатом `GetSecretIngredient`) формы значение, возвращаемое свойством `GetSecretIngredient`. Это новый экземпляр типа `GetSecretIngredient`, указывающий на метод `SuzannesSecretIngredient()`.
- Снова щелкните на кнопке «Get the ingredient». Теперь, когда поле `ingredientMethod` указывает на `SuzannesSecretIngredient()`, произойдет вызов этого метода, и его значение будет передано `numericUpDown` (убедитесь, что он называется **amount**), а затем выведено на консоль.
- Щелкните на кнопке «Get Amy's delegate». Она использует свойство `Amy.GetSecretIngredient`, чтобы присвоить полю `ingredientMethod` значение `AmysSecretIngredient()`.
- Снова щелкните на кнопке «Get the ingredient». Будет вызван метод Эми.
- Поместите точки останова в первые строки каждого из трех методов формы. **Перезапустите программу** (чтобы метод `ingredientMethod` стал равен null) и снова выполните все вышеуказанные шаги. Используйте функцию Step Into (F11). Отслеживайте, что происходит при щелчке на кнопке «Get the ingredient».

Ребус в бассейне

Возьмите фрагменты кода из бассейна и заполните пропуски. Каждый фрагмент может быть использован несколько раз. В бассейне есть лишние фрагменты. Вам нужно, чтобы при щелчке на кнопке `button1` на консоль выводился следующий результат:

Результат:

`Fingers is coming to get you!`

```
public Form1() {
    InitializeComponent();
    this.______ += new EventHandler(Minivan);
    this.______ += new EventHandler(____________);
}
void Towtruck(object sender, EventArgs e) {
    Console.Write("is coming ");
}
void Motorcycle(object sender, EventArgs e) {
    button1.______ += new EventHandler(____________);
}
void Bicycle(object sender, EventArgs e) {
    Console.WriteLine("to get you!");
}
void ____________(object sender, EventArgs e) {
    button1.______ += new EventHandler(Dumptruck);
    button1.______ += new EventHandler(____________);
}
void ____________(object sender, EventArgs e) {
    Console.Write("Fingers ");
}
```

Каждый фрагмент может быть использован несколько раз.

+
++
==
-=
!=

Van
Car
Minivan
Motorcycle
Tricycle

Load
Save
Open
Close
Click
Scroll

event
delegate
int
private
public

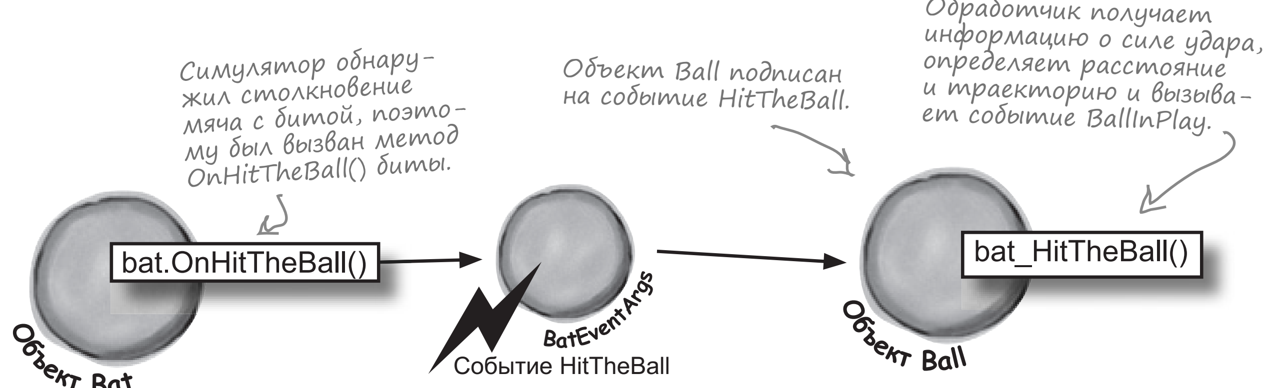
Airplane
Bicycle
Dumptruck
Towtruck
Flatbed

Ответ на с. 727 →

Объект может подписаться на событие...

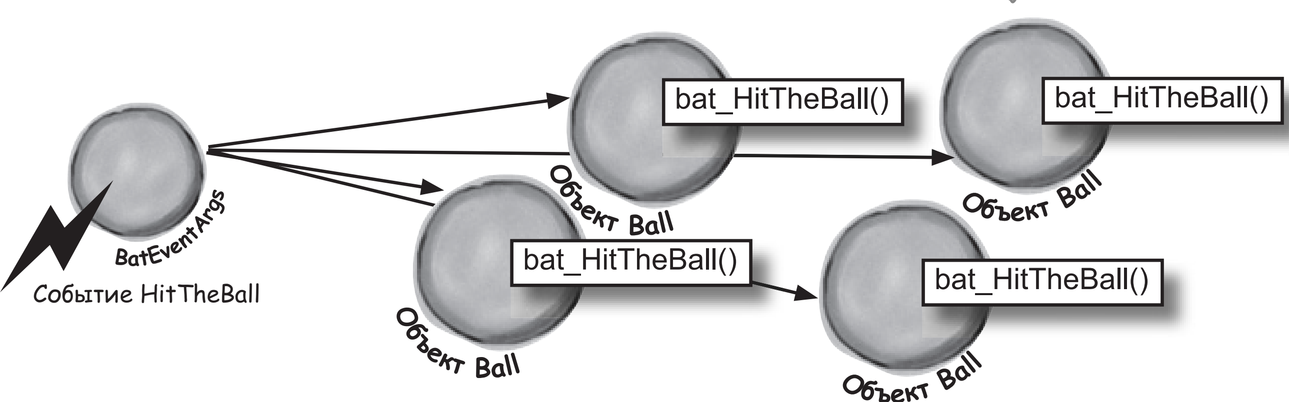
Предположим, для симулятора был создан класс `Bat` (Бита), который добавляет событие `HitTheBall`. Обнаружив, что игрок ударил по мячу, симулятор вызывает метод `OnHitTheBall()` объекта `Bat`, который вызывает событие `HitTheBall`.

Поэтому к классу `Ball` можно добавить метод `bat_HitTheBall`, который подпишется на событие `HitTheBall`. При ударе по мячу уже его собственный обработчик вызовет метод `OnBallInPlay()` для уже его собственного события `BallInPlay`, и цепочка начнет работу. Игроки играют, болельщики визжат, судьи кричат... всё как на реальном матче.



...но это не всегда хорошо!

В игре может присутствовать только один мяч. Но если объект `Bat` при помощи события передаст информацию об ударе по мячу, подписаться на него сможет любой объект `Ball`. Представьте, что программист случайно добавил еще три объекта `Ball`. Что произойдет в этом случае? После удара битой по полю разлетятся **четыре мяча**!



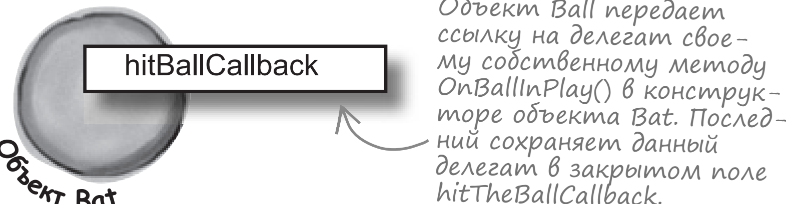
Обратный вызов

События в системе работают корректно, если объекты `Ball` и `Bat` имеются в единственном числе. В ситуации, когда мячей больше одного, все они оказываются подписаны на событие `HitTheBall` и при его возникновении вбрасываются в игру, что не имеет никакого смысла. Другими словами, нам нужно связать с битой всего один мяч, исключив возможность привязки других мячей.

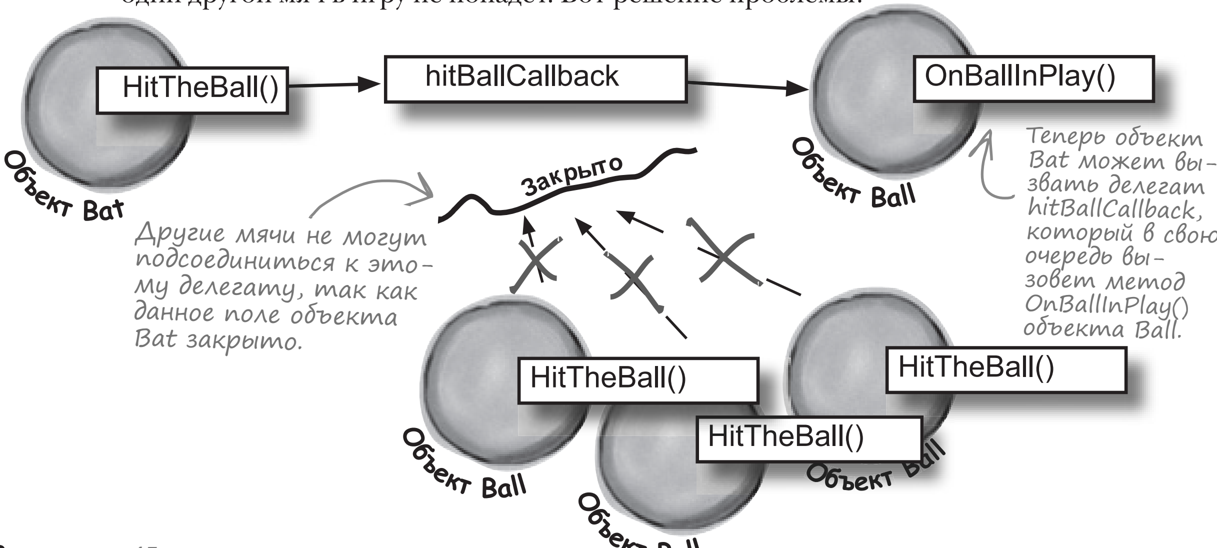
В этом нам поможет **обратный вызов (callback)**. Так называется техника работы с делегатами, при которой вместо события, доступного для подписки любым объектам, используется метод (часто конструктор) с хранящимся в закрытом поле делегатом в качестве аргумента. Обратный вызов позволит нам гарантировать, что объект `Bat` оповещает всего один объект `Ball`:

1. **Объект `Bat` сохраняет поле делегата закрытым**
   Предотвратить возможность создания цепочек из объектов `Ball` поможет закрытие хранящего эту информацию поля. В этом случае объект `Bat` управляет вызовом нужного метода объекта `Ball`.

2. **Конструктор объекта `Bat`**
   Когда мяч оказывается в игре, конструктор создает экземпляр биты и передает указатель на него методу `OnBallInPlay()`. Это **метод обратного вызова**, так как объект `Bat` использует его для вызова объекта, который его создал.



3. **Когда Bat ударяет по мячу, он использует метод обратного вызова**
   Но пока Bat скрывает делегат, можно быть полностью уверенным в том, что ни один другой мяч в игру не попадет. Вот решение проблемы!



Случай с золотым крабом

Пятиминутная тайна

Генри «Простак» Ходкингс — охотник за сокровищами (TreasureHunter). Сейчас он выслеживает одно из самых желанных и редких ювелирных изделий на морскую тему — инструктированного нефритами золотого краба. Но охотников за сокровищами много. И в их конструкторах также присутствует ссылка на этого краба. Генри же хочет найти сокровище ***первым***.

Из украденных диаграмм классов Генри узнал, что как только кто-то приближается к крабу, класс GoldenCrab вызывает событие RunForCover (Побег в укрытие). Более того, событие включает NewLocationArgs (Новое место), указывающее, куда переместился краб. Но так как остальные охотники не знают о событии, Генри считает, что сможет получить сокровище первым.

Генри добавляет в конструктор код, превращающий метод treasure_RunForCover() в обработчик события RunForCover. Затем кто-то посылается за крабом, чтобы тот спрятался и вызвал событие RunForCover, — это даст методу treasure_RunForCover() всю нужную информацию.

Все шло по плану, пока Генри не прибыл на нужное место и не обнаружил, что там за краба уже дерутся три его конкурента.

Каким же образом они там оказались? → Ответ на с. 731.

Решение ребуса в бассейне

```
public Form1() {
    InitializeComponent();
    this.Load += new EventHandler(Minivan);
    this.Load += new EventHandler(Motorcycle);
}
void Towtruck(object sender, EventArgs e) {
    Console.Write("is coming ");
}
void Motorcycle(object sender, EventArgs e) {

    button1.Click += new EventHandler(Bicycle);
}
void Bicycle(object sender, EventArgs e) {
    Console.WriteLine("to get you!");
}
void Minivan(object sender, EventArgs e) {
    button1.Click += new EventHandler(Dumptruck);
    button1.Click += new EventHandler(Towtruck);
}
void Dumptruck(object sender, EventArgs e) {
    Console.Write("Fingers ");
}
```

Конструктор последовательно добавляет два обработчика к событиям Load. В итоге они вызываются сразу после загрузки формы.

Два обработчика события Load привязывают к обработчику события Click кнопки три других, отдельных обработчика.

Щелчок на кнопке вызывает цепочку из трех связанных с ней обработчиков событий.

Напоминаем, что в приложении WinForms метод Console.WriteLine() выводит результат в окно Output внутри IDE.

Обратный вызов как способ работы с делегатами

Обратный вызов является **еще одним способом работы с делегатами**. Он описывает **шаблон** — способ, при котором вы используете делегатов в своих классах таким образом, что один объект может сказать другому: «Сообщи мне, когда это случится!»

1 **Добавим в проект еще один делегат.**
Так как объект `Bat` хранит делегата в закрытом поле, указывающем на метод, новый делегат должен иметь совпадающую сигнатуру:

```
delegate void BatCallback(BallEventArgs e);
```

Создавать для делегатов отдельные файлы не обязательно. Этот делегат поместите в файл с классом Bat.

Обратный вызов объекта Bat будет указывать на метод OnBallInPlay() этого объекта, поэтому делегат обратного вызова должен иметь аналогичную сигнатуру — использовать в качестве параметра BallEventArgs и не возвращать значения.

2 **Добавим к проекту класс `Bat`.**
Это очень простой класс. Он содержит метод `HitTheBall()`, запускающийся при каждом ударе по мячу, который с помощью делегата `hitBallCallback()` вызывает метод `OnBallInPlay()` (передаваемый в конструктор).

```
class Bat {
    private BatCallback hitBallCallback;
    public Bat(BatCallback callbackDelegate) {
        this.hitBallCallback = new BatCallback(callbackDelegate);
    }
    public void HitTheBall(BallEventArgs e) {
        if (hitBallCallback != null)
            hitBallCallback(e);
    }
}
```

Для предотвращения исключения нужно проверять, не ссылается ли какой-либо делегат на значение null.

Мы воспользовались оператором =, так как в данном случае нужно, чтобы объект bat получал сообщения только от одного объекта ball, соответственно данный делегат настраивается всего один раз. Но ничто не запрещает написать обратный вызов с оператором +=, который будет адресован нескольким методам. Основной смысл обратного вызова — <u>в отслеживании вызывающим объектом адресатов</u>. В случае события объекты требуют оповещения, добавляя обработчики, в то время как при обратном вызове объекты перебирают делегаты и просят об оповещении.

3 **Свяжем биту с мячом.**
Каким же образом конструктор объекта `Bat` получает ссылку на метод `OnBallInPlay()` определенного мяча? Он вызывает метод `GetNewBat()` (Получить новую биту) этого объекта, который мы сейчас добавим:

```
public Bat GetNewBat()
{
    return new Bat(new BatCallback(OnBallInPlay));
}
```

Метод GetNewBat() создает объект Bat и использует делегат BatCallBack для передачи ссылки на этот новый объект собственному методу OnBallInPlay(). Именно этот метод обратного вызова будет применен битой в момент удара по мячу.

Мы настроили обратный вызов в конструкторе класса Bat. Но бывают случаи, когда имеет смысл поместить его в открытый метод или метод записи.

4 **Инкапсулируем класс `Ball`.**
Вызывающие события методы, название которых начинается с `On...` не бывают открытыми. Поэтому зададим уровень доступа к методу `OnBallInPlay()`:

```
protected void OnBallInPlay(BallEventArgs e) {
    EventHandler<BallEventArgs> ballInPlay = BallInPlay;
    if (ballInPlay != null)
        ballInPlay(this, e);
}
```

Это стандартный шаблон, с которым вы еще не раз столкнетесь при работе с классами .NET. Если в таком классе имеется событие, вы практически гарантированно найдете защищенный метод, название которого начинается с «On».

5 **Остается только разобраться с классом `BaseballSimulator`.**
`BaseballSimulator` больше не может вызывать метод `OnBallInPlay()` объекта `Ball`, как мы и хотели (и поэтому IDE не сигнализирует об ошибке). Вместо этого форма просит новую биту для удара по мячу. И при этом объект `Ball` гарантирует, что его метод `OnBallInPlay()` связан с обратным вызовом биты.

```
public void PlayBall() {
    Bat bat = ball.GetNewBat();
    BallEventArgs ballEventArgs =
                 new BallEventArgs(Trajectory, Distance);
    bat.HitTheBall(ballEventArgs);
}
```

Если форма (или симулятор) хочет ударить по мячу, ей потребуется новый объект Bat. Мяч гарантирует связь обратного вызова с битой. При вызове метода HitTheBall() биты он вызывает метод OnBallInPlay() мяча, инициирующий событие BallInPlay.

Запустите программу, она должна работать, как и раньше. Но теперь в ней невозможно появление набора мячей, реагирующих на одно и то же событие.

Не верьте нам на слово, посмотрите на работу программы в режиме отладки!

КЛЮЧЕВЫЕ МОМЕНТЫ

- Добавляя к проекту делегат, вы **создаете новый тип,** хранящий ссылки на методы.
- С помощью делегатов события оповещают объекты о произведенных действиях.
- Объекты подписываются на события, если им нужно реагировать на происходящее с другими объектами.
- `EventHandler` — это вид делегата, часто работающий с событиями.
- С одним событием можно связать несколько обработчиков. Именно поэтому для присвоения обработчика событию используется оператор +=.
- Всегда проверяйте события и делегаты на равенство null.
- Все элементы с панели toolbox используют события.
- Когда один объект передает другому ссылку на метод таким образом, что этот второй объект может вернуть информацию, это называется обратным вызовом.
- Методы могут подписываться на события анонимно, в то время как обратные вызовы позволяют объектам контролировать делегаты.
- Обратные вызовы и события используют делегаты для ссылки и вызова методов других объектов.
- Чтобы понять, как работают делегаты, используйте отладчик.

Обратные вызовы с командами класса MessageDialog

При создании UICommand для MessageDialog можно создать обратный вызов через делегат UICommandInvokedHandler. Ему передается необязательный объект-идентификатор, который вместе с меткой доступен через параметр делегата IUICommand.

```
MessageDialog dialog = new MessageDialog("Here's a dialog.");
dialog.Commands.Add(new UICommand("My label", MyUiCommandCallback, "My identifier"));
await dialog.ShowAsync();
```

Здесь может быть не только строка, но и любой объект.

Добавьте эти строки в приложение Воспользуйтесь командой IDE Generate Method Stub, чтобы сгенерировать заглушку для метода обратного вызова.

Часто Задаваемые Вопросы

В: Чем обратный вызов отличается от события?

О: События и делегаты — это часть .NET. Это способ, которым одни объекты оповещают другие о произведенных действиях. На событие может подписаться произвольное количество объектов, при этом издатель лишен возможность узнать о них. При запуске события все подписанные на него объекты запускают обработчики.

Обратные вызовы не принадлежат .NET — это всего лишь название способа использования делегатов (или событий, ничто не мешает вам создать обратный вызов из закрытого события). Этим термином всего лишь называются отношения между двумя классами, при которых объект запрашивает оповещение. В случае же событий объекты требуют оповещений.

В: То есть обратные вызовы не принадлежат .NET?

О: Нет. Обратный вызов — это ***шаблон***, способ использования существующих типов, ключевых слов и инструментов. Рассмотрите внимательно код обратного вызова, написанный для биты и мяча. Присутствуют ли там неизвестные вам ключевые слова? Нет! Но при этом там не используются делегаты, которые **являются** типом .NET.

Шаблонов существует множество. Есть даже отдельная область программирования — паттерны проектирования. Ведь многие проблемы, с которыми вы можете столкнуться, уже решены, и вам достаточно воспользоваться подходящим паттерном.

В: Получается, что обратные вызовы — это всего лишь закрытые события?

О: Не совсем. Проще всего представлять их таким образом, но закрытые события имеют свою специфику. Помните, что на самом деле означает модификатор доступа `private`? Доступ к помеченному им члену класса имеют только экземпляры этого класса. Поэтому, пометив событие как `private`, вы запрещаете подписываться на него из других классов. В то время как обратные вызовы допускают анонимную подписку.

В: Но обратный вызов выглядит как событие, не снабженное ключевым словом event.

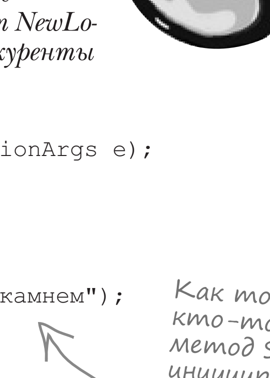
О: Обратный вызов похож на событие, потому что они оба используют **делегаты**. Это имеет смысл, так как обратный вызов является инструментом, позволяющим одному объекту передать другому ссылку на свой метод.

Но событие — это способ, которым класс оповещает мир о неких действиях. В случае же обратных вызовов оповещения отсутствуют. Они являются закрытыми, и метод, осуществляющий вызов, отслеживает, кто именно вызывается.

Посмотрите раздел «Head First Design Patterns» на сайте лаборатории Head First. Там вы можете познакомиться с различными паттернами. Например, первым идет уже встречавшийся вам паттерн `Observer` (или Publisher-Subscriber). Один объект публикует информацию, а другие объекты подписываются на нее. События в C# являются способом реализации паттерна Observer.

Случай с золотым крабом

Пятиминутная тайна раскрыта



Почему другие охотники опередили Генри?

Разгадка в том, как именно наш охотник искал каменоломню. Впрочем, начнем с изучения украденных Генри диаграмм.

Генри обнаружил, что при попытке приблизиться к крабу класс GoldenCrab вызывает событие RunForCover. Событие включает NewLocationArgs с информацией о новом месте укрытия. Так как конкуренты о событии не знают, Генри решил, что победа будет за ним.

```
class GoldenCrab {
    public delegate void Escape(object sender, NewLocationArgs e);
    public event Escape RunForCover;
    public void SomeonesNearby() {
        Escape runForCover = RunForCover;
        if (runForCover != null)
            runForCover(this, new NewLocationArgs("Под камнем");
    }
}
class NewLocationArgs {
    public NewLocationArgs(HidingPlace newLocation) {
        this.newLocation = newLocation;
    }
    private HidingPlace newLocation;
    public HidingPlace NewLocation { get { return newLocation; } }
}
```

Как только к крабу кто-то приближается, метод SomeonesNearby() инициирует событие RunForCover, и краб прячется в другом месте.

Как же Генри воспользовался полученной информацией?

Благодаря полученной ссылке на краба Генри добавил к своему конструктору метод treasure_RunForCover() в качестве обработчика события RunForCover. Затем он послал за крабом помощника, зная, что это станет причиной побега краба и появления события RunForCover, которое даст методу treasure_RunForCover() всю нужную информацию.

```
class TreasureHunter {
    public TreasureHunter(GoldenCrab treasure) {
        treasure.RunForCover += treasure_RunForCover;
    }
    void treasure_RunForCover(object sender, NewLocationArgs e) {
        MoveHere(e.NewLocation);
    }
    void MoveHere(HidingPlace location) {
        // ... код перемещения в новое место ...
    }
}
```

Генри думал, что добавление к конструктору обработчика, вызывающего метод MoveHere() при каждом возникновении события RunForCover у краба, — это хитрый ход. Но он забыл, что все охотники за сокровищами наследуют от одного и того же класса, поэтому его код добавит в цепочку и их обработчики событий!

Вот почему Генри потерпел поражение. Добавив обработчик событий конструктору `TreasureHunter`, он нечаянно ***сделал это для всех охотников***! Обработчики событий всех охотников оказались связанными с одним и тем же событием `RunForCover`. И сообщение о перемещении краба в новое укрытие пришло всем. Можно было бы обернуть дела в свою пользу, если бы Генри получил сообщение первым. Но Генри не мог узнать, в каком порядке проводится оповещение. В итоге все подписавшиеся до него получили новость о местоположении краба раньше.

Настройка панели чудо-кнопок в Windows

Откройте начальную страницу Windows 8 и щелкните на значке Internet Explorer, затем откройте панель чудо-кнопок и выберите Settings. Internet Explorer заставит Windows 8 добавить в меню Settings пункты Internet Options и About. Но при щелчке на команде IE About вы получите страницу, отличающуюся от страниц About для программ Maps, Mail или приложений для магазина Windows. Дело в том, что каждое приложение — и даже каждая страница — сообщает Windows о своих параметрах Settings панели чудо-кнопок и регистрирует обратный вызов, срабатывающий при выборе пользователем соответствующей команды. Написанные на C# приложения для магазина Windows используют для этого делегатов. Посмотрим, как это работает, и **добавим команду About к панели Settings приложения Джимми**.

Откройте файл *MainPage.xaml.cs*, добавьте два оператора `using` и одну строку кода в конструктор:

```
using Windows.UI.ApplicationSettings;
using Windows.UI.Popups;

/// <summary>
/// Основная страница, обеспечивающая общие для большинства приложений характеристики.
/// </summary>
public sealed partial class MainPage : JimmysComics3.Common.LayoutAwarePage
{
    public MainPage()
    {
        this.InitializeComponent();

        SettingsPane.GetForCurrentView().CommandsRequested += MainPage_CommandsRequested;
    }
```

Статический класс `SettingsPane` позволяет приложению добавлять команды на панель Settings и удалять их оттуда. Он находится в пространстве имен `Windows.UI.ApplicationSettings`.

Как только вы введете символы «+=», IDE предложит создать заглушку метода обработки событий. Вот что должно туда входить. Мы воспользуемся делегатом `UICommandInvokedHandler`, поэтому добавим метод `AboutInvokedHandler()`. Этот метод будет вызываться параметром About.

```
void MainPage_CommandsRequested(SettingsPane sender, SettingsPaneCommandsRequestedEventArgs args) {
    UICommandInvokedHandler invokedHandler =
                                new UICommandInvokedHandler(AboutInvokedHandler);
    SettingsCommand aboutCommand = new SettingsCommand("About", "About Jimmy's Comics",
                                                       invokedHandler);
    args.Request.ApplicationCommands.Add(aboutCommand);
}

async void AboutInvokedHandler(IUICommand command) {
    await new MessageDialog("An app to help Jimmy manage his comic collection",
                    "Jimmy's Comics").ShowAsync();
}
```

Запустите приложение. Откройте панель чудо-кнопок, нажмите Settings и выберите команду About. Приложение вызовет `AboutInvokedHandler` и отобразит окно `MessageDialog`.

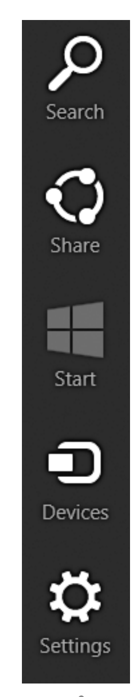
После перехода на главную страницу нажмите Settings, приложение отобразит «About option» в окне MessageDialog.

Для доступа к чудо-кнопкам и панели приложения используйте кнопку ⊞.

- ★ Панель чудо-кнопок: ⊞ + I
- ★ Панель Settings: ⊞ + I
- ★ Панель приложения: ⊞ + Z

Посмотрим, как все это работает. Остановите программу и воспользуйтесь командой Go To Definition для получения **определения `SettingsCommand`** из метаданных. Вот его вид:

```
...public sealed class SettingsCommand : IUICommand
{
    ...public SettingsCommand(object settingsCommandId, string label, UICommandInvokedHandler handler);

    ...public object Id { get; set; }
    ...public UICommandInvokedHandler Invoked { get; set; }
    ...public string Label { get; set; }
}
```

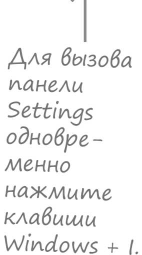


Теперь посмотрим на определение `UICommandInvokedHandler`:

```
...public delegate void UICommandInvokedHandler(IUICommand command);
```

Взгляните на разные объекты, чтобы понять, как они работают:

- ★ Метод `SettingsPane.GetForCurrentView()` возвращает объект, обладающий событием `CommandsRequested`. Вернитесь к коду и посмотрите определение события `CommandsRequested`.
- ★ У обработчика события есть аргумент `SettingsPaneCommandsRequestedEventArgs`. Перейдите к его определению, чтобы увидеть объект `Request`, используемый в третьей строке обработчика событий.
- ★ Объект `Request` обладает одним свойством: коллекцией `ApplicationCommands`, содержащей объекты `SettingsCommand`.
- ★ Снова вернитесь к обработчику событий. Когда пользователь касается панели Settings, она вызывает событие `CommandsRequested`, которое просит у приложения команды и обратные вызовы. Вы связали с этим событием объект-получатель и заставили его возвращать `SettingsCommand`, который определяет параметр About, с делегатом, указывающим на метод, вызывающий окно `MessageDialog`. При нажатии на About панель через этого делегата выполняет обратный вызов метода `AboutInvokedHandler()`.
- ★ Все еще непонятно? Используйте кнопки панели инструментов для перемещения по определениям. Поместите точки останова в конструктор и оба метода. Иногда нужно несколько раз посмотреть определения, чтобы мозг усвоил информацию.



16 проектирование с шаблоном mvvm

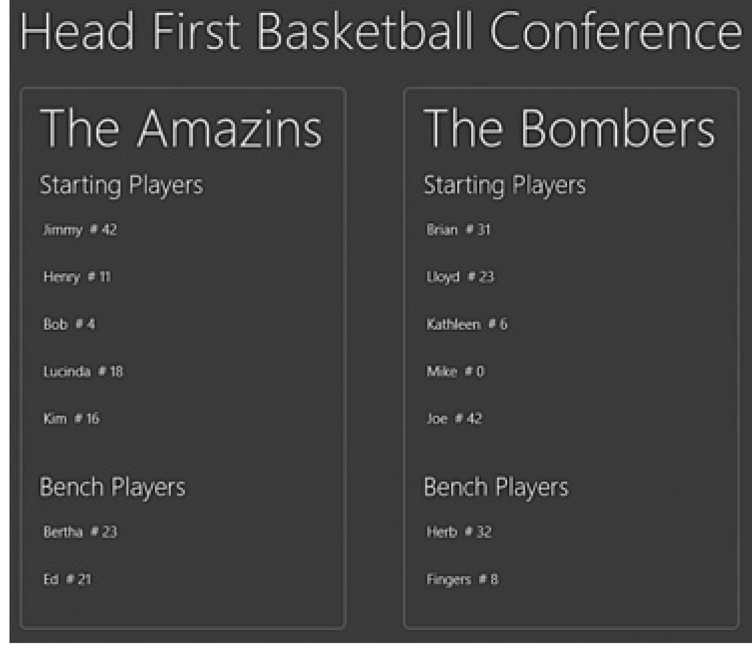
Прекрасные изнутри и снаружи

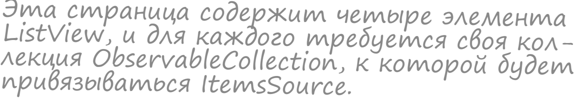
Для приложения важна не только визуальная привлекательность. Что первым делом приходит на ум, когда вы думаете о дизайне? Грамотная архитектура? Прекрасная компоновка? Эстетически приятный и хорошо работающий продукт? Все это относится и к вашим приложениям. В этой главе вы познакомитесь с **шаблоном Model-View-ViewModel** и узнаете, как получить хорошо спроектированные, слабо связанные приложения. Попутно мы поговорим об **анимации** и **шаблонах элементов управления**, о том, как при помощи **преобразователей** упростить связывание данных и как связать все воедино, ***заложив твердый фундамент C#*** под свою способность написать любое приложение.

Приложение для баскетбольного матча Head First

Джимми и Брайан — капитаны двух ведущих баскетбольных команд на организованном Head First матче Объектвильской баскетбольной лиги. В команды входят великолепные игроки, которым требуется первоклассное приложение для наблюдения за тем, кто принимает участие в игре, а кто сидит на скамейке запасных.

В каждой команде есть активные и запасные игроки, и у каждого из них есть имя и номер.





Как прийти к согласию?

Увы, Брайан и Джимми по-разному представляют создание приложения и поэтому ведут жаркие дебаты. Брайн хочет, чтобы отображаемыми на странице данными было легко управлять, в то время как Джимми больше озабочен упрощением процедуры связывания данных. Вряд ли подобные споры облегчат создание приложения.

Мне совершенно очевидно, что следует упростить процесс добавления данных. Разве не так?

Джимми: Погоди, дружище. Это звучит несколько недальновидно.

Брайан: Ты не понимаешь, о чем я говорю, поэтому я повторю еще раз. Мы начнем с простого класса `Player`, который содержит свойства с именем, номером и текущим состоянием игрока.

Джимми: Я *понимаю*, о чем ты. А вот ты меня не слышишь. Ты думал, как смоделировать данные.

Брайан: Именно. Ведь с этого все начинается.

Джимми: Создавать данные будет удобно.

Брайан: Ты уловил мою мысль...

Джимми: Я не закончил. А что с остальной частью приложения? У нас есть отображающие данные элементы ListView и TextBlock. И без связанных с ними коллекций они не будут работать.

Брайан: Хм...

Джимми: Именно так. Поэтому следует принять пару тактических решений по поводу нашей объектной модели.

Брайан: Намекаешь, что нам придется создать посредственную, сложную в использовании объектную модель, так как нам нужно нечто для привязки?

Джимми: Да, если у тебя нет идеи получше.

Мозговой штурм

Как создать класс, с которым будет просто связать данные, сохранив объектную модель, позволяющую легко работать с этими данными?

Проектирование для связывания или для работы с данными?

Вы уже знаете, насколько важна объектная модель, позволяющая легко работать с данными. Но что, если с этими объектами следует сделать **две разные вещи**? Это одна из самых распространенных проблем, стоящих перед проектировщиками приложений. Объекты должны обладать открытыми свойствами и коллекцией `ObservableCollections`, с которой будут связываться элементы управления XAML. Но иногда это усложняет работу с данными, так как вам приходится строить объектную модель, которая не относится к интуитивно понятным и с которой сложно работать.

Player
Name: string Number: int Starter: bool

Roster
TeamName: string Players: IEnumerable<string>

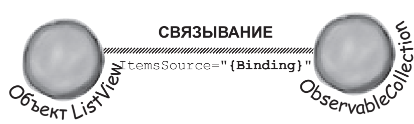
Сложно оптимизировать классы для облегченного анализа данных запросами LINQ...

```
var benchPlayers =
    from player in _roster.Players
    where player.Starter == false
    select player;
```

Подобные модели данных ограничивают возможности создания страниц.

При такой модели данных создавать страницы легко, но написание кода запросов и управления данными усложняется.

...если данные требуется связать с элементами управления XAML на страницах приложения.



Player
Name: string Number: int

Roster
TeamName: string Starters: ObservableCollection Bench: ObservableCollection

League
JimmysTeam: Roster BriansTeam: Roster
Здесь были бы удобны закрытые методы, создающие фиктивные данные.

Проектирование для связывания и для данных

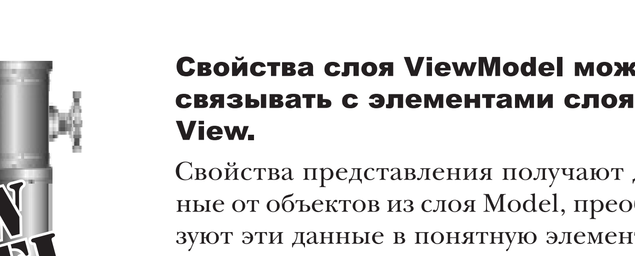
Практически все приложения с большой и достаточно сложной объектной моделью сталкиваются с необходимостью жертвовать или структурой класса, или доступными для связывания объектами. К счастью, существует шаблон проектирования, позволяющий решить данную проблему. Он называется **Model-View-ViewModel** (или **MVVM**) и разбивает приложение на три слоя: ***Модель*** (содержит данные и состояние приложения), ***Представление*** (содержит страницы и элементы управления, с которыми взаимодействует пользователь), и ***Модель представления***, которая преобразует данные Модели в объекты, допускающие связывание и восприимчивые к событиям в Представлении, о которых следует знать Модели.

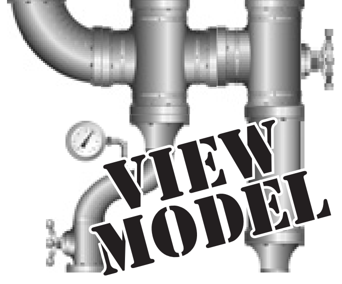
Шаблон MVVM использует уже имеющиеся у вас инструменты, аналогично рассмотренным в предыдущей главе обратным вызовам и шаблонам Observer.

Объекты, с которыми взаимодействует пользователь, попадают в слой View.

Это страницы, кнопки, текст, сетки, StackPanels, ListView и другие элементы, компонуемые в коде XAML. Они связываются с объектами слоя ViewModel, а их обработчики событий вызывают методы объектов ViewModel.

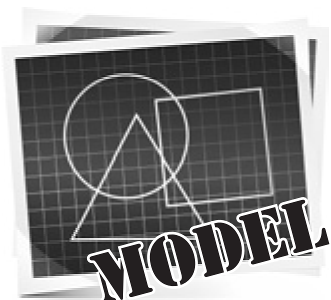
Слой ViewModel подобен трубопроводу, соединяющему объекты слоя View с объектами слоя Model.





Свойства слоя ViewModel можно связывать с элементами слоя View.

Свойства представления получают данные от объектов из слоя Model, преобразуют эти данные в понятную элементам управления слоя View форму и уведомляют слой View обо всех изменениях данных.



Объекты, содержащие сведения о состоянии приложения, находятся в слое Model.

Именно здесь приложение хранит данные. Слой ViewModel вызывает свойства и методы из слоя Model. В этот слой попадают объекты, меняющиеся за время жизни приложения, а также данные, сохраняемые в файл и загружаемые из файла.

Начнем строить приложение для баскетболистов

Создайте новое приложение для магазина Windows и **присвойте ему имя <u>BasketballRoster</u>** (так как в коде будет использоваться пространство имен `BasketballRoster` и данное имя гарантирует совпадение вашего кода с написанным на следующих страницах).

Создадим для нашего проекта папки Model, View и ViewModel.
Щелкните правой кнопкой мыши на имени проекта в окне Solution Explorer и выберите в меню Add команду New Folder:

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'BasketballRoster' (1 project)
BasketballRoster
Build
Rebuild
Deploy
Run Code Analysis
Scope to This
New Solution Explorer View
Add
Add Reference...
Add Service Reference...
Store
Manage NuGet Packages...
Set as StartUp Project
Debug
Add Solution to Source Control...
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename F2
Unload Project
Open Folder in File Explorer
Open in Blend...
Properties

New Item... Ctrl+Shift+A
Existing Item... Shift+Alt+A
New Folder
Class... Shift+Alt+C

Не заменяйте *MainPage.xaml* на Basic Page. Мы сделаем это на шаге #4.

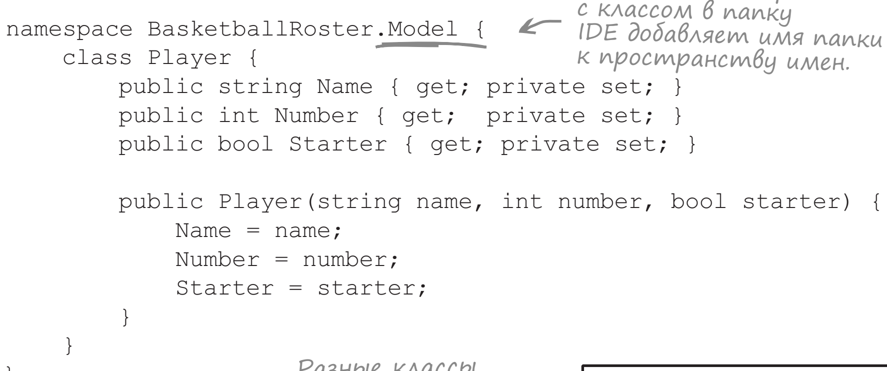
> **При добавлении к проекту новой папки через окно Solution Explorer IDE создает пространство имен на основе имени этой папки. Это заставляет команду the Add→Class... создавать классы с этим пространством имен. Так, при добавлении класса в папку Model IDE добавит в верхнюю часть файла строку `BasketballRoster.Model`.**

Добавьте папку *Model*. Аналогичным образом добавьте папки *View* и *ViewModel*, чтобы проект выглядел так:

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'BasketballRoster' (1 project)
BasketballRoster
Properties
References
Assets
Common
Model
View
ViewModel
App.xaml
BasketballRoster_TemporaryKey.pfx
MainPage.xaml
Package.appxmanifest
Solution Explorer | Class View

В этих папках будут содержаться классы, элементы управления и страницы вашего приложения.

2 **Начнем построение модели с добавления класса Player.**
Щелкните правой кнопкой мыши на папке *Model* и **добавьте класс `Player`**. При этом IDE обновит пространство имен. Вот класс `Player`:



```
namespace BasketballRoster.Model {
    class Player {
        public string Name { get; private set; }
        public int Number { get;  private set; }
        public bool Starter { get; private set; }

        public Player(string name, int number, bool starter) {
            Name = name;
            Number = number;
            Starter = starter;
        }
    }
}
```

При добавлении файла с классом в папку IDE добавляет имя папки к пространству имен.

Player
Name: string
Number: int
Starter: bool

Разные классы отвечают за разные вещи? Звучит знакомо...

Эти классы малы, так как их задача — следить за графиком игроков. Классы слоя Model не занимаются отображением данных, они только управляют ими.

3 **Завершим модель, добавив класс Roster.**
Теперь **добавьте в папку *Model* класс `Roster`**. Вот его код.

```
namespace BasketballRoster.Model {
    class Roster {
        public string TeamName { get; private set; }

        private readonly List<Player> _players = new List<Player>();
        public IEnumerable<Player> Players {
            get { return new List<Player>(_players); }
        }

        public Roster(string teamName, IEnumerable<Player> players) {
            TeamName = teamName;
            _players.AddRange(players);
        }
    }
}
```

Это указывает на закрытость поля.

Roster
TeamName: string
Players: IEnumerable<string>

Вот как должна выглядеть папка *Model*:

В начало имени поля `_players` мы добавили нижнее подчеркивание. Это стандартная запись имен закрытых полей, и мы будем ее использовать, так что привыкайте.

Слой View на следующей странице

4 Добавим в папку View главную страницу.

Щелкните правой кнопкой мыши на папке *View* и **добавьте новый шаблон Basic Page с именем** *LeaguePage.xaml*. Вам предложат добавить отсутствующие страницы, кроме того, потребуется перестроить решение. Отредактируйте код XAML и назовите страницу «Head First Basketball Conference», отредактировав (как обычно) статический ресурс `AppName`. Страница *MainPage.xaml* нам не нужна, поэтому на следующем шаге мы ее удалим.

5 Заменим главную страницу на LeaguePage.xaml.

Удалите из папки проекта файл *MainPage.xaml*. Перестройте проект. Это приведет к ошибке:

Error List

1 Error | 0 Warnings | 0 Messages | Search Error List

	Description	File	Line	Column	Project
1	The type or namespace name 'MainPage' could not be found (are you missing a using directive or an assembly reference?)	App.xaml.cs	67	48	BasketballRoster

Error List | Output

Дважды щелкните на ошибке для перехода к строке, в которой возникла проблема:

```
if (!rootFrame.Navigate(typeof(MainPage), args.Arguments))
{
    throw new Exception("Failed to create initial page");
}
```

Знакомый код! Вы редактировали его при создании приложения для Джимми. При загрузке приложения он ищет класс `MainPage`, а вы только что удалили определяющий данный класс файл XAML. Ничего страшного! Просто укажите класс, который следует загрузить:

```
if (!rootFrame.Navigate(typeof(LeaguePage), args.Arguments))
{
    throw new Exception("Failed to create initial page");
}
```

Странно. Добавленный к проекту `LeaguePage` не распознается. Дело в том, что вы добавили его в папку, и **IDE поместила его в пространство имен View**. Поэтому при ссылке на данный класс требуется в явном виде указывать пространство имен:

```
if (!rootFrame.Navigate(typeof(View.LeaguePage), args.Arguments))
{
    throw new Exception("Failed to create initial page");
}
```

Теперь перестраиваемое приложение компилируется! Запустите его, чтобы увидеть новую страницу.

Ваши собственные элементы управления

В создаваемом приложении каждая команда пользуется одинаковым набором элементов управления: TextBlock, еще TextBlock, ListView, еще TextBlock плюс ListView, помещенные в контейнер StackPanel внутри контейнера Border. А зачем нам два одинаковых набора на одной странице? Добавление третьей и четвертой команд потребует множественного копирования. Но этого можно избежать при помощи **пользовательских элементов управления**. Данный класс позволяет создавать собственные элементы. Для этого используется XAML и программный код как при построении страниц. Добавим такой элемент к проекту BasketballRoster.

UserControl — базовый класс, позволяющий инкапсулировать связанные друг с другом элементы управления, добавляя логику, определяющую поведение получившейся структуры.

1. **Добавим новый элемент в папку View.**

   Щелкните правой кнопкой мыши на папке *View* и добавьте элемент. Выберите в окне диалога User Control и назовите его *RosterControl.xaml*.

2. **Программный код нового элемента.**

   Откройте файл *RosterControl.xaml.cs*. Новый элемент расширяет базовый класс `UserControl`. Весь программный код, определяющий его поведение, попадает сюда.

```
namespace BasketballRoster.View
{
    public sealed partial class RosterControl : UserControl
    {
        public RosterControl()
        {
            this.InitializeComponent();
        }
    }
}
```

3. **Код XAML нового элемента.**

   IDE добавила новый элемент с пустой сеткой `<Grid>`. Сюда следует вписать ваш код XAML.

Перед тем как перевернуть страницу, посмотрите на снимок программы и подумайте, какой код XAML будет у элемента `RosterControl`.

★ `<StackPanel>` для элементов, которые окаймляет синий `<Border>`. Подумайте, какое свойство делает скругленные углы у элемента Border?

★ Два элемента ListView отображают данные об игроках, значит, понадобится `<UserControl.Resources>`, содержащий `DataTemplate`. Мы назвали его `PlayerItemTemplate`.

★ Свяжите элементы ListView со свойствами `Starters` и `Bench`, а верхний TextBlock со свойством `TeamName`.

★ Элемент Border вложен в `<Grid>` с одной строкой, с параметром `Height=«Auto»`, чтобы она не выходила за нижнюю границу ListView, заполняя всю страницу.

> «Дать не рыбу, но удочку...»
>
> **Книга приближается к концу, поэтому мы хотим предложить вам реальные задачи. Хороший программист делает множество обоснованных предположений, поэтому мы даем вам минимум информации о том, как работает `UserControl`. Даже связывание пока не работает, поэтому данных в конструкторе вы не увидите! Какую часть кода XAML вы сможете написать до того, как перевернете страницу и найдете код для `RosterControl`?**

4 **Завершим XAML-код элемента RosterControl.**

Вот код пользовательского элемента `RosterControl`, добавленного в папку *View*. Вы обратили внимание, что мы предоставили свойства для связывания, но не дали контекст данных? Тому есть причина. Два элемента управления на странице будут отображать разные данные, поэтому страница установит для каждого из них свой контекст.

```
<UserControl
    x:Class="BasketballRoster.View.RosterControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:BasketballRoster.View"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300"
    d:DesignWidth="400">

    <UserControl.Resources>
        <DataTemplate x:Key="PlayerItemTemplate">
            <TextBlock Style="{StaticResource ItemTextStyle}">
                <Run Text="{Binding Name}"/>
                <Run Text=" #"/>
                <Run Text="{Binding Number}"/>
            </TextBlock>
        </DataTemplate>
    </UserControl.Resources>

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>

        <Border BorderThickness="2" BorderBrush="Blue" CornerRadius="6" Margin="0,0,40,0">
            <StackPanel Margin="20">
                <TextBlock Text="{Binding TeamName}"
                           Style="{StaticResource HeaderTextStyle}"/>
                <TextBlock Text="Starting Players"
                           Style="{StaticResource GroupHeaderTextStyle}" Margin="0,20,0,0"/>
                <ListView ItemsSource="{Binding Starters}"
                          ItemTemplate="{StaticResource PlayerItemTemplate}" Margin="0,20,0,0"/>
                <TextBlock Text="Bench Players"
                           Style="{StaticResource GroupHeaderTextStyle}" Margin="0,20,0,0"/>
                <ListView ItemsSource="{Binding Bench}"
                          ItemTemplate="{StaticResource PlayerItemTemplate}" Margin="0,20,0,0"/>
            </StackPanel>
        </Border>
    </Grid>
</UserControl>
```

Вы уже знаете, что размер элементов управления зависит от параметров Height и Width. С их помощью меняется вид элемента в конструкторе IDE.

Это шаблон для пунктов элементов управления ListView. В каждой строке один TextBlock с тремя разделами для отображения имени и номера игрока.

Придать элементу Border скругленные углы позволяет свойство CornerRadius.

Оба элемента ListView используют один шаблон, определенный как статический ресурс.

Упражнение

Постройте слой ViewModel для приложения BasketballRoster по виду данных в слое Model и по связыванию в слое View. Определите, какую «трубу» следует добавить к приложению, чтобы связать их друг с другом.

Обновим LeaguePage.xaml, добавив элементы Roster.

Добавьте к странице эти свойства `xmlns`, чтобы она начала распознавать новое пространство имен:

```
xmlns:view="using:BasketballRoster.View"
xmlns:viewmodel="using:BasketballRoster.ViewModel"
```

Добавьте экземпляр `LeagueViewModel` как статический ресурс:

```
<Page.Resources>
    <viewmodel:LeagueViewModel x:Name="LeagueViewModel"/>
    <x:String x:Key="AppName">Head First Basketball Conference</x:String>
</Page.Resources>
```

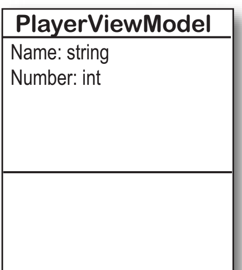
Теперь к странице можно добавить StackPanel с двумя элементами `RosterControls`:

```
<StackPanel Orientation="Horizontal" Margin="120,0,0,0" Grid.Row="1"
DataContext="{StaticResource ResourceKey=LeagueViewModel}" >
    <view:RosterControl DataContext="{Binding JimmysTeam}" Margin="0,0,20,0"/>
    <view:RosterControl DataContext="{Binding BriansTeam}" Margin="0,0,20,0"/>
</StackPanel>
```

Создадим классы ViewModel.

Создайте три класса в папке *ViewModel*.

Убедитесь, что созданные классы и страницы находятся в корректных папках; иначе пространства имен не совпадут с кодом в решении упражнения.



RosterViewModel
TeamName: string
Starters: ObservableCollection<PlayerViewModel>
Bench: ObservableCollection<PlayerViewModel>
---
constructor: RosterViewModel(Model.Roster)
private UpdateRosters()

LeagueViewModel
JimmysTeam: RosterViewModel
BriansTeam: RosterViewModel
---
private GetBomberPlayers(): Model.Roster
private GetAmazinPlayers(): Model.Roster

Заставим классы ViewModel работать.

- ★ Класс `PlayerViewModel` — простой объект данных с двумя свойствами только для чтения.
- ★ `LeagueViewModel` содержит два закрытых метода, создающих фиктивные данные, и создает для каждой команды объекты `Model.Roster`, передаваемые в конструктор RosterViewModel.
- ★ `RosterViewModel` имеет конструктор, принимающий объект `Model.Roster`. Он задает свойство `TeamName` и вызывает закрытый метод `UpdateRosters()`, через запрос LINQ извлекающий активных и запасных игроков и обновляющий свойства `Starters` и `Bench`. Добавьте строку **`using Model;`** для доступа к объектам пространства имен `Model`.

Про запрос LINQ говорилось несколькими страницами ранее...

Если в конструкторе XAML появляется сообщение, что `LeagueViewModel` отсутствует в пространстве имен `ViewModel`, а вы уверены, что добавили все корректно, щелкните правой кнопкой мыши на строке BasketballRoster, выберите команду Unload Project, затем щелкните правой кнопкой и перезагрузите проект.

В слое ViewModel приложения BasketballRoster три класса: `LeagueViewModel`, `PlayerViewModel` и `RosterViewModel`. Все они находятся в папке *ViewModel*.

Если вы забыли строку `using Model;`, вместо `Roster` везде нужно писать `Model.Roster`.

```
namespace BasketballRoster.ViewModel {
    using Model;
    using System.Collections.ObjectModel;

    class LeagueViewModel {
        public RosterViewModel BriansTeam { get; private set; }
        public RosterViewModel JimmysTeam { get; private set; }

        public LeagueViewModel() {
            Roster briansRoster = new Roster("The Bombers", GetBomberPlayers());
            BriansTeam = new RosterViewModel(briansRoster);

            Roster jimmysRoster= new Roster("The Amazins", GetAmazinPlayers());
            JimmysTeam = new RosterViewModel(jimmysRoster);
        }

        private IEnumerable<Player> GetBomberPlayers() {
            List<Player> bomberPlayers = new List<Player>() {
                new Player("Brian", 31, true),
                new Player("Lloyd", 23, true),
                new Player("Kathleen",6, true),
                new Player("Mike", 0, true),
                new Player("Joe", 42, true),
                new Player("Herb",32, false),
                new Player("Fingers",8, false),
            };
            return bomberPlayers;
        }

        private IEnumerable<Player> GetAmazinPlayers()  {
            List<Player> amazinPlayers = new List<Player>() {
                new Player("Jimmy",42, true),
                new Player("Henry",11, true),
                new Player("Bob",4, true),
                new Player("Lucinda", 18, true),
                new Player("Kim", 16, true),
                new Player("Bertha", 23, false),
                new Player("Ed",21,  false),
            };
            return amazinPlayers;
        }
    }
}
```

LeagueViewModel предоставляет объекты RosterViewModel, которые RosterControl может использовать как контекст данных. Он создает объектную модель Roster для объектов RosterViewModel.

Этот закрытый метод генерирует фиктивные данные для класса Bombers, создавая новый список объектов Player.

Фиктивные данные обычно попадают в слой ViewModel, так как управление состоянием приложения MVVM осуществляется через экземпляры класса Model, инкапсулированные внутри объектов ViewModel.

Классы слоя View используются для хранения данных, поэтому метод возвращает объекты Player, а не объекты PlayerViewModel.

PlayerViewModel — это простой объект данных, свойства которого связываются с шаблоном данных.

```
namespace BasketballRoster.ViewModel {
    class PlayerViewModel {
        public string Name { get; private set; }
        public int Number { get; private set; }

        public PlayerViewModel(string name, int number) {
            Name = name;
            Number = number;
        }
    }
}
```

В типичном приложении MVVM INotifyPropertyChanged реализуются только классы слоя ViewModel, так как это единственные объекты, связанные с элементами XAML.

```
namespace BasketballRoster.ViewModel {
    using Model;
    using System.Collections.ObjectModel;
    using System.ComponentModel;

    class RosterViewModel : INotifyPropertyChanged {
        public ObservableCollection<PlayerViewModel> Starters { get; private set; }
        public ObservableCollection<PlayerViewModel> Bench { get; private set; }

        private Roster _roster;

        private string _teamName;
        public string TeamName {
            get { return _teamName; }
            set {
                _teamName = value;
                OnPropertyChanged("TeamName");
            }
        }

        public RosterViewModel(Roster roster) {
            _roster = roster;

            Starters = new ObservableCollection<PlayerViewModel>();
            Bench = new ObservableCollection<PlayerViewModel>();

            TeamName = _roster.TeamName;

            UpdateRosters();
        }

        private void UpdateRosters() {
            var startingPlayers =
                from player in _roster.Players
                where player.Starter
                select player;
            Starters.Clear();
            foreach (Player player in startingPlayers)
                Starters.Add(new PlayerViewModel(player.Name, player.Number));

            var benchPlayers =
                from player in _roster.Players
                where player.Starter == false
                select player;
            Bench.Clear();
            foreach (Player player in benchPlayers)
                Bench.Add(new PlayerViewModel(player.Name, player.Number));
        }

        public event PropertyChangedEventHandler PropertyChanged;

        protected void OnPropertyChanged(string propertyName) {
            PropertyChangedEventHandler propertyChanged = PropertyChanged;
            if (propertyChanged != null)
                propertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Приложение сохраняет состояние в объекты Roster, инкапсулированные в слое ViewModel. Остальная часть класса преобразует данные слоя Model в свойства, с которыми можно связать слой View.

При изменении свойства TeamName объект RosterViewModel вызывает событие PropertyChanged, что приводит к обновлению всех связанных с ним объектов.

Этот запрос LINQ обнаруживает всех активных игроков и добавляет их в свойство Starters коллекции ObservableCollection.

Аналогичный запрос LINQ для поиска запасных игроков.

Не являются ли пользовательские элементы управления всего лишь способом распределения XAML по файлам?

Пользовательские элементы управления полностью функциональны.

И подобно любым другим элементам управления они являются объектами. В данном случае объектами, расширяющими базовый класс `UserControl`, содержащий знакомые свойства `Height` и `Visibility` и перенаправленные события `Tapped` и `PointerEntered`. Вы можете добавлять собственные свойства или пользоваться другими элементами управления XAML для создания сложных и визуально симпатичных пользовательских интерфейсов. Но важнее всего способность пользовательских элементов управления **инкапсулировать** все дополнительные элементы внутри одного элемента XAML, доступного для повторного использования.

Подождите... разговор об инкапсуляции и разделении объектов по слоям я уже где-то слышал. Имеет ли все это какое-либо отношение к разделению ответственности?

Именно так! Слои Model, View и ViewModel разделяют зоны ответственности программы.

При проектировании больших, надежных приложений сложнее всего определить, что будут делать те или иные объекты. Спроектировать приложение можно бесконечным количеством способов. Это означает, что C# предоставляет вам гибкий инструментарий. И именно поэтому принятые сегодня решения могут осложнить жизнь в будущем. Шаблон MVVM помогает отделить данные приложения от его UI, что упрощает проектирование, так как вы легко можете определить место для данных и для элементов интерфейса, связывая их друг с другом через предоставляемый шаблон.

Когда изменения одного класса приводят к необходимости редактирования еще нескольких классов, программисты называют это «эффектом дробовика». Крайне неприятная ситуация, особенно если вы торопитесь.

Разделение ответственности позволяет предотвратить подобные проблемы, а такой полезный инструмент, как MVVM, помогает отделять друг от друга важные элементы, присутствующие практически в каждом приложении.

Часто Задаваемые Вопросы

В: Что мешает поместить элементы управления в слой ViewModel или коллекцию `ObservableCollections` в слой Model?

О: Ничего, просто после этого вы перестанете работать с шаблоном MVVM. Элементы управления и страницы отвечают за отображение данных. Поместив их в слой View, вы упростите работу с кодом по мере роста приложения. Доверившись шаблону MVVM, вы улучшите свою жизнь в будущем, так как редактирование кода упростится.

В: Я не пойму, что такое *состояние*.

О: Говоря о *состоянии*, люди подразумевают объекты в памяти, определяющие работу приложения: текст в редакторе, положение врагов и игрока и счет игры, значения в ячейках электронной таблицы. Это сложная концепция, так как иногда сложно определить «этот объект часть состояния», а «тот объект нет». Следующий проект этой главы поможет вам на практике разобраться, что на самом деле означает *состояние*.

В: Зачем нужна строка `using Model;` в классах ViewModel?

О: При создании классов в папке *Model* IDE автоматически создает пространство имен `BasketballRoster.Model`. Точка в центре указывает, что Model находится ниже, чем BasketballRoster. Любой другой класс в пространстве имен под BasketballRoster может обратиться к классам Model, добавив в начало **`Model.`** или введя указанную строку. Вне пространства имен `BasketballRoster` классам потребуется добавлять `using BasketballRoster.Model;`.

В: Могут ли мои элементы управления включать другие элементы?

О: Да. Вы можете добавлять содержимое, как и к любому другому элементу:

```
<view:RosterControl>
    <TextBlock>Hello!</TextBlock>
</view:RosterControl>
```

Здесь свойству Content элемента присваивается объект TextBlock. Но при этом RosterControl будет заменен на TextBlock!

Чтобы этого избежать, добавьте раздел <UserControl.ContentTemplate> в код XAML объекта UserControl. Вставьте в раздел DataTemplate с кодом XAML элемента. Эта строка:

```
<ContentPresenter
   Content="{TemplateBinding
Content}"/>
```

будет заменена контентом.

В: На моей странице продолжает появляться треугольник с восклицательным знаком. Что это?

О: Конструктор XAML — сложный механизм. Он работает так хорошо, что мы забываем, какого труда стоит отобразить страницу и обновлять ее по мере редактирования кода XAML. В завершенной программе BasketballRoster конструктор показывает фиктивные данные по обеим командам. Но они генерируются в закрытых методах слоя ViewModel. То есть при каждом обновлении страницы конструктор должен запускать эти методы. Для корректной работы конструктора методы нужно скомпилировать. Если после редактирования элементов управления код C# не скомпилирован, конструктор сообщает о том, что страница устарела. Перестройте код, и восклицательный знак исчезнет.

В: В BasketballRoster отображаются только генерируемые при запуске фиктивные данные. Как будет работать функция, редактирующая данные в слое Model?

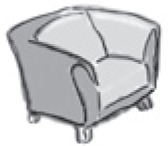
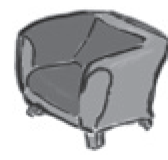
О: Хотите, чтобы программа BasketballRoster позволила Джимми и Брайану менять игроков? Элементы ListView в слое View связаны с объектами `ObservableCollection`, и слой ViewModel общается со слоем View посредством событий `PropertyChanged` и `CollectionChanged`. Аналогичным способом можно заставить слой Model взаимодействовать с ViewModel. К объекту `Roster` можно добавить событие `RosterUpdated`. Следить за этим событием будет `RosterViewModel`, а обработчики события начнут обновлять коллекции `Starters` и `Bench`, что приведет к событиям `CollectionChanged`, вызывающим обновление элементов ListView.

Для слоя Model — события — это хороший способ взаимодействовать с остальным приложением, не зная, реагируют ли на событие другие классы. Слой управляет состоянием, позволяя другим классам обрабатывать вводимые данные и обновлять интерфейс, ведь он отсоединен от классов в слоях ViewModel и View.

Доверившись шаблону MVVM сегодня, вы облегчите себе жизнь завтра, так как управлять кодом приложения будет проще.

Шаблон Model-View-ViewModel является разновидностью шаблона Model-View-Controller. Почитать о MVC можно в GDI+ PDF, доступных для скачивания на сайте Head First Labs.

Беседа у камина

Слои Model и ViewModel ведут горячие дебаты на актуальную тему: «Кто нужнее?»

Model:

Я не понимаю, о чем тут спорить. Где бы ты был без меня? Я содержу данные; я содержу логику, определяющую функционирование приложения. Без меня тебе было бы нечего делать.

ViewModel:

Ну вот, опять ты считаешь себя центром вселенной.

Model:

Ну раз уж ты сам затронул эту тему, подобное мнение вполне может быть правдой.

ViewModel:

Ха! А что случится, если я перестану работать?

Model:

Ты не посмеешь.

ViewModel:

Откуда ты знаешь? Без меня ты ничто. Слой View не знает, как с тобой разговаривать. Элементы управления останутся пустыми, и пользователь ничего не увидит.

Model:

Теперь ты понимаешь, почему я общаюсь с тобой только через события? Ты такой вредный!

ViewModel:

Давай-ка поговорим об этом. Почему ты не показываешь мне, что у тебя внутри? Я вижу только методы и свойства, а ты отправляешь мне сообщения через аргументы событий.

Model:

Разумеется! Если не инкапсулировать данные, кто знает, какой вред ты можешь причинить?

ViewModel:

Кажется, у ***кого-то*** проблемы с доверием.

Model:

Да! Я доверяю управление данными только моим закрытым методам; иначе состояние нашего приложения может пойти вразнос. Но я не один такой! Почему ты не даешь мне поговорить со слоем View? Он кажется хорошим парнем.

ViewModel:

Да ты не сможешь говорить на его языке! Я никогда не видел, чтобы ты вызывал событие `PropertyChanged`, более того, не думаю, чтобы кто-то из твоих объектов реализовывал `INotifyPropertyChanged`.

Model:

Как ты смеешь! Зачем нужны события `PropertyChanged`? Ни один уважающий себя слой Model не будет их вызывать! Ты оскорбил меня предположением, что меня касается что-то кроме данных. За кого ты меня принимаешь?

Секундомер для судьи

Последнюю игру Джимми и Брайану пришлось отменить, так как судья забыл секундомер. Можем ли мы воспользоваться шаблоном MVVM и создать приложение, имитирующее секундомер?

Как судье без секундомера обеспечить выполнение правила трех секунд?

MVVM как забота о состоянии приложения

Приложения на основе MVVM используют слои Model и View, чтобы отделить состояние от пользовательского интерфейса. Поэтому при построении приложения MVVM первым делом нужно подумать, что означает управление состоянием приложения. Как только вы возьмете этот аспект под контроль, можно приступать к созданию слоя Model, в котором поля и свойства используются для фиксации состояния, то есть всего, за чем приложение должно следить для успешного функционирования. Большинству приложений нужно модифицировать состояние, поэтому в слой Model входят открытые методы, отвечающие за этот аспект. Остальная часть приложения должна видеть текущее состояние, поэтому в слое Model присутствуют открытые свойства.

Итак, что же это такое «управление состоянием секундомера»?

Секундомер знает, работает он или нет.

Вы с первого взгляда видите, работает или нет секундомер, поэтому в слое Model должен присутствовать способ определения этого состояния.

Всегда можно посмотреть время работы.

Как в ручном, аналоговом секундомере, так и в его цифровой версии можно увидеть, сколько времени он работал.

Можно зафиксировать и посмотреть промежуточное время.

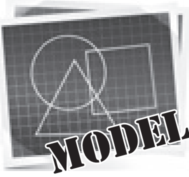
Большинство секундомеров позволяют сохранить текущее время, не прекращая отсчет. В аналоговых секундомерах для этого используется дополнительная стрелка, а у цифровых применяется отдельный вывод.

Секундомер можно оставить, запустить и обнулить.

Приложение должно запустить секундомер, остановить его и сбросить показания, то есть слой Model должен предоставить остальной части приложения инструменты для выполнения этих задач.

Слой Model фиксирует состояние приложения: то есть информацию, которой приложение обладает в конкретный момент. Он также предоставляет функции редактирования состояния и свойства, позволяющие остальной части приложения увидеть сведения о состояниии.

Начало работы над слоем Model



Теперь у нас достаточно информации для построения слоя Model приложения, имитирующего секундомер. Создайте новое приложение для магазина Windows. **Присвойте ему имя *Stopwatch***, чтобы пространство имен совпало с фигурирующим в коде решения. Затем **создайте папки *Model*, *View* и *ViewModel***. Добавьте в папку *Model* класс StopwatchModel:

Упражнение!

```
class StopwatchModel {
    private DateTime? _started;

    private TimeSpan? _previousElapsedTime;

    public bool Running {
        get { return _started.HasValue; }
    }

    public TimeSpan? Elapsed {
        get {
            if (_started.HasValue) {
                if (_previousElapsedTime.HasValue)
                    return CalculateTimeElapsedSinceStarted() + _previousElapsedTime;
                else
                    return CalculateTimeElapsedSinceStarted();
            }
            else
                return _previousElapsedTime;
        }
    }

    private TimeSpan CalculateTimeElapsedSinceStarted() {
        return DateTime.Now - _started.Value;
    }

    public void Start() {
        _started = DateTime.Now;
        if (!_previousElapsedTime.HasValue)
            _previousElapsedTime = new TimeSpan(0);
    }

    public void Stop() {
        if (_started.HasValue)
            _previousElapsedTime += DateTime.Now - _started.Value;
        _started = null;
    }

    public void Reset() {
        _previousElapsedTime = null;
        _started = null;
    }

    public StopwatchModel() {
        Reset();
    }
}
```

Убедитесь, что класс Stopwatch создан в папке Model. Мы опустили остальные строки, касающиеся пространств имен { }, так как вы уже знаете, как они выглядят.

В этих двух закрытых полях хранится состояние приложения. Их можно обнулять.

Дополнительное булево поле будет следить за тем, работает секундомер или нет. Но это поле будет принимать значение true только у поля _started, если существует значение. Может, стоит сразу использовать _started.HasValue?

Это предназначенное только для чтения свойство использует два закрытых поля для вычисления времени работы. Вы понимаете, как оно функционирует?

Подсказка: при добавлении или вычитании значений DateTime или TimeSpan всегда будет получаться TimeSpan.

Остальной части приложения требуется метод, запускающий и останавливающий секундомер, поэтому в слой Model добавляются соответствующие методы.

Сброс состояния означает присвоение полям значния null.

Каждый новый экземпляр объекта StopwatchModel инициализируется как сброшенный или остановленный.

Структуры TimeSpan и DateTime

Для управления временем существуют две полезные структуры. Вы уже работали с DateTime, в которой хранится дата. Структура TimeSpan представляет собой временной интервал. Он измеряется в тиках (одна десятимиллионная секунды), и структура TimeSpan обладает методами предобразования этой величины в секунды, миллисекунды, дни и т. п.

События как сигнал об изменении состояния

Секундомер должен фиксировать промежуточное время, сохраняя его как часть состояния. Также нам нужен метод, позволяющий узнать промежуточное время. А теперь представьте, что мы хотим, чтобы по истечении этого времени приложение выполняло некие действия. Слой ViewModel может включить индикатор или показать небольшую анимацию. Слой Model **сообщает остальному приложению о важных изменениях состояния через события**. Поэтому добавим в слой Model событие, возникающие по истечении промежуточного времени. Начнем с добавления класса `LapEventArgs` в папку *Model*:

```
class LapEventArgs : EventArgs {

    public TimeSpan? LapTime { get; private set; }

    public LapEventArgs(TimeSpan? lapTime) {

        LapTime = lapTime;

    }

}
```

Это класс LapEventArgs. Обязательно поместите его в папку Model, чтобы он оказался в корректном пространстве имен.

При обновлении промежуточного времени приложению нужно узнать время работы. Для хранения этой информации у нас есть свойство TimeSpan.

Слой Model может вызвать событие, сообщающее приложению о важных изменениях без ссылок на внешние классы. Его просто строить, так как оно не связано с остальными слоями MVVM.

Добавьте в класс `StopwatchModel` метод `Lap()`, задающий свойство `LapTime` и вызвающий событие `LapTimeUpdated`.

```
public void Reset() {
    _previousElapsedTime = null;
    _started = null;
    LapTime = null;
}

public TimeSpan? LapTime { get; private set; }

public void Lap() {
    LapTime = Elapsed;
    OnLapTimeUpdated(LapTime);
}

public event EventHandler<LapEventArgs> LapTimeUpdated;

private void OnLapTimeUpdated(TimeSpan? lapTime) {
    EventHandler<LapEventArgs> lapTimeUpdated = LapTimeUpdated;
    if (lapTimeUpdated != null) {
        lapTimeUpdated(this, new LapEventArgs(lapTime));
    }
}
```

Убедитесь, что свойство LapTime сбрасывается вместе со сбросом состояния секундомера.

Здесь подойдет автоматическое свойство. Закрытое вспомогательное поле не требуется, так как здесь нет вычислений, которые нужно было бы инкапсулировать.

Метод Lap() обновляет свойство и вызывает событие.

Это обычный код вызова события.

Приятным последствием несвязанных слоев является возможность построения приложения сразу после завершения слоя Model.

Если IDE сообщает об отсутствии `StopwatchViewModel` в пространстве имен `ViewModel`, хотя вы стопроцентно уверены в его наличии, выгрузите и снова загрузите проект.

Представление для простого секундомера

Добавьте в папку *View* пользовательский элемент управления *BasicStopwatch.xaml* и введите этот код. В пользовательский элемент входят два элемента TextBlock для времени работы и промежуточного времени и кнопки запуска, остановки, сброса и просмотра промежуточного времени.

```
<UserControl
    x:Class="Stopwatch.View.BasicStopwatch"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Stopwatch.View"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300"
    d:DesignWidth="400"
    xmlns:viewmodel="using:Stopwatch.ViewModel">

    <UserControl.Resources>
        <viewmodel:StopwatchViewModel x:Name="viewModel"/>
    </UserControl.Resources>

    <Grid DataContext="{StaticResource ResourceKey=viewModel}">
        <StackPanel>
            <TextBlock>
                <Run>Elapsed time: </Run>
                <Run Text="{Binding Hours}"/>
                <Run>:</Run>
                <Run Text="{Binding Minutes}"/>
                <Run>:</Run>
                <Run Text="{Binding Seconds}"/>
            </TextBlock>
            <TextBlock>
                <Run>Lap time: </Run>
                <Run Text="{Binding LapHours}"/>
                <Run>:</Run>
                <Run Text="{Binding LapMinutes}"/>
                <Run>:</Run>
                <Run Text="{Binding LapSeconds}"/>
            </TextBlock>
            <StackPanel Orientation="Horizontal">
                <Button Click="StartButton_Click">Start</Button>
                <Button Click="StopButton_Click">Stop</Button>
                <Button Click="ResetButton_Click">Reset</Button>
                <Button Click="LapButton_Click">Lap</Button>
            </StackPanel>
        </StackPanel>
    </Grid>
</UserControl>
```

Этот элемент управления находится в папке View под основным пространством имен вашего проекта.

Это свойство xmlns добавляет пространство имен. Мы назвали проект Stopwatch, поэтому для ViewModel пространство имен будет называться Stopwatch.ViewModel.

Этот пользовательский элемент управления хранит ViewModel как статический ресурс и пользуется им как контекстом данных. Он не нуждается в задании контекста со стороны контейнера и самостоятельно фиксирует состояние.

Этот элемент TextBlock связан со свойствами класса ViewModel, возвращающими время работы.

Этот элемент TextBlock связан со свойствами, показывающими промежуточное время.

Чтобы эти значения были актуальными, ViewModel должен вызывать события PropertyChanged.

Чтобы код компилировался, нужно добавить к элементу управления обработчики события `Click` и класс `StopwatchViewModel` в пространство имен `ViewModel`.

Подсказка: воспользуйтесь классом DispatcherTimer для непрерывной проверки Model и обновления его свойств

Код для ViewModel на следующей странице. Можете написать его, зная код слоев View и Model и не заглядывая в ответ? Добавьте на главную страницу элемент управления BasicStopwatch *(для начала)* и посмотрите, насколько далеко вы сможете продвинуться самостоятельно.

Но будьте аккуратны и не считайте, что не права именно IDE. Иногда ошибка в коде XAML одной страницы (например, некорректное свойство `xmlns`) нарушает работу всей системы.

Слой ViewModel для секундомера

Слой ViewModel должен находиться в пространстве имен `ViewModel`.

```
class StopwatchViewModel : INotifyPropertyChanged {
    private StopwatchModel _stopwatchModel = new StopwatchModel();

    private DispatcherTimer _timer = new DispatcherTimer();

    public bool Running { get { return _stopwatchModel.Running; } }

    public StopwatchViewModel() {
        _timer.Interval = TimeSpan.FromMilliseconds(50);
        _timer.Tick += TimerTick;
        _timer.Start();
        Start();

        _stopwatchModel.LapTimeUpdated += LapTimeUpdatedEventHandler;
    }

    public void Start() {
        _stopwatchModel.Start();
    }

    public void Stop() {
        _stopwatchModel.Stop();
    }

    public void Lap() {
        _stopwatchModel.Lap();
    }

    public void Reset() {
        bool running = Running;
        _stopwatchModel.Reset();
        if (running)
            _stopwatchModel.Start();
    }

    int _lastHours;
    int _lastMinutes;
    decimal _lastSeconds;
    void TimerTick(object sender, object e) {
        if (_lastHours != Hours) {
            _lastHours = Hours;
            OnPropertyChanged("Hours");
        }
        if (_lastMinutes != Minutes) {
            _lastMinutes = Minutes;
            OnPropertyChanged("Minutes");
        }
        if (_lastSeconds != Seconds) {
            _lastSeconds = Seconds;
            OnPropertyChanged("Seconds");
        }
    }

    public int Hours {
        get { return _stopwatchModel.Elapsed.HasValue ? _stopwatchModel.Elapsed.Value.Hours : 0; }
    }
```

Свойство Running проверяет класс Model, чтобы понять, работает ли секундомер.

Без этих операторов using класс компилироваться не будет.

```
using Model;
using System.ComponentModel;
using Windows.UI.Xaml;
```

Эти методы Start(), Stop() и Lap() просто передают управление аналогичным методам класса model.

Сначала метод Reset() вызывает метод Reset() класса Model, а затем, если секундомер включен, метод Start().

При каждом отсчете объекта DispatcherTimer ViewModel проверяет, изменились ли часы, минуты и секунды. В случае положительного результата проверки вызывается событие PropertyChanged, чтобы класс View смог обновиться.

Синтаксис ?: позволяет поместить условие в одну строку и работает аналогично оператору if.

```
    public int Minutes {
        get { return _stopwatchModel.Elapsed.HasValue ? _stopwatchModel.Elapsed.Value.Minutes : 0; }
    }

    public decimal Seconds {
        get {
            if (_stopwatchModel.Elapsed.HasValue) {
                return (decimal)_stopwatchModel.Elapsed.Value.Seconds
                    + (_stopwatchModel.Elapsed.Value.Milliseconds * .001M);
            }
            else
                return 0.0M;
        }
    }

    public int LapHours {
        get { return _stopwatchModel.LapTime.HasValue ? _stopwatchModel.LapTime.Value.Hours : 0; }
    }

    public int LapMinutes {
        get { return _stopwatchModel.LapTime.HasValue ? _stopwatchModel.LapTime.Value.Minutes : 0; }
    }

    public decimal LapSeconds {
        get {
            if (_stopwatchModel.LapTime.HasValue) {
                return (decimal)_stopwatchModel.LapTime.Value.Seconds
                    + (_stopwatchModel.LapTime.Value.Milliseconds * .001M);
            }
            else
                return 0.0M;
        }
    }

    int _lastLapHours;
    int _lastLapMinutes;
    decimal _lastLapSeconds;
    private void LapTimeUpdatedEventHandler(object sender, LapEventArgs e) {
        if (_lastLapHours != LapHours) {
            _lastLapHours = LapHours;
            OnPropertyChanged("LapHours");
        }
        if (_lastLapMinutes != LapMinutes) {
            _lastLapMinutes = LapMinutes;
            OnPropertyChanged("LapMinutes");
        }
        if (_lastLapSeconds != LapSeconds) {
            _lastLapSeconds = LapSeconds;
            OnPropertyChanged("LapSeconds");
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged(string propertyName) {
        PropertyChangedEventHandler propertyChanged = PropertyChanged;
        if (propertyChanged != null)
            propertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Elapsed.Value возвращает структуру TimeSpan, а его свойство Minutes возвращает значение типа int.

Свойство Seconds возвращает секунды плюс сотые секунд как значение типа decimal. Установите здесь точку останова и воспользуйтесь отладчиком, чтобы понять, как это работает.

Эти свойства функционируют аналогично свойствам для времени работы, но на основе значений LapTime вместо Elapsed.

Обработчик события LapTimeUpdated класса Model. Функционирует аналогично обработчику событий объекта DispatcherTimer, проверяя свойства объекта, отвечающего за промежуточное время и вызывая для изменившихся свойств события PropertyChanged.

Уже знакомый код для событий PropertyChanged.

Завершение работы над секундомером

Осталось завершить всего несколько вещей: добавить обработчики событий к элементу управления BasicStopwatch и поместить элемент на главную страницу.

1. **Вернитесь к файлу *BasicStopwatch.xaml.cs*** и добавьте обработчики событий:

```
private void StartButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Start();
}
private void StopButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Stop();
}
private void ResetButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Reset();
}
private void LapButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Lap();
}
```

Кнопки в слое view вызывают методы класса ViewModel. Это типичный способ работы слоя View.

2. **Удалите файл *MainPage.xaml* и *замените его* шаблоном Basic Page**, как в остальных проектах (не забудьте перестроить решение).

3. Откройте новый файл *MainPage.xaml* и **добавьте пространство имен XML** к тегу верхнего уровня:

```
xmlns:view="using:Stopwatch.View"
```

4. Отредактируйте ресурс `AppName` в файле *MainPage.xaml*, **задав имя страницы**:

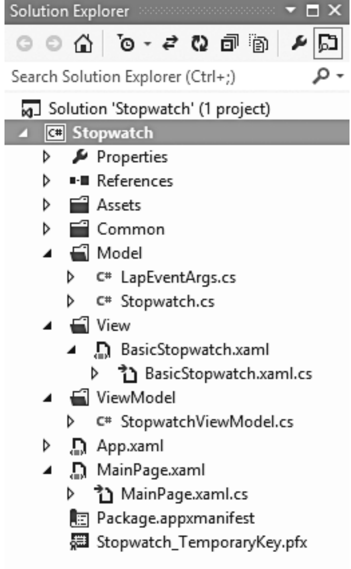
```
<Page.Resources>
    <x:String x:Key="AppName">Stopwatch</x:String>
</Page.Resources>
```

Это поведение встроено в элемент управления, поэтому программный код для главной страницы отсутствует.

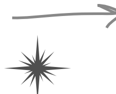
5. **Добавьте элемент BasicStopwatch в код XAML** файла *MainPage.xaml*:

```
<view:BasicStopwatch Grid.Row="1" Margin="120,0"/>
```

Теперь приложение работает. Нажимайте кнопки Start, Stop, Reset и Lap и смотрите, что происходит.



Чего-то не хватает? Вот как должно выглядеть ваше окно Solution Explorer.

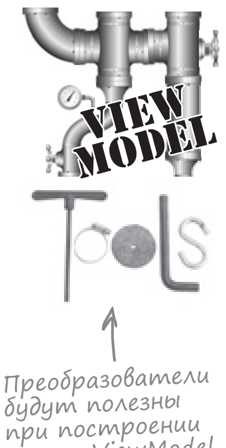


Работа с такими шаблонами, как MVVM, подразумевает принятие решений. MVVM — это шаблон, использующий соглашения, а не строгие правила, которые можно проверить компилятором. И это ***гибкий*** шаблон, который может реализовываться множеством способов. В примерах этой главы мы показываем наиболее распространенные вещи, которые можно встретить в MVVM-приложениях. И там, где решение варьируется, мы объясняем причину такого подхода. Нашей основной целью является демонстрация гибкости *или ее отсутствия*, чтобы при построении ваших собственных приложений вы оказались в состоянии **принять оптимальное решение**.

Вот правила, которых мы придерживаемся при построении MVVM-приложений:

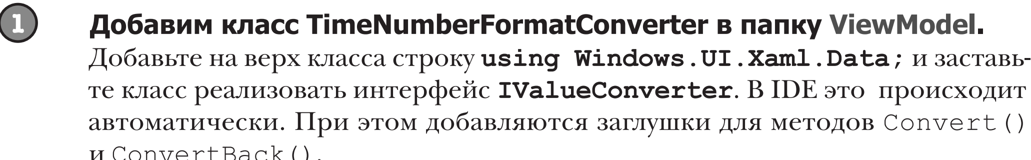
- Классы Model, ViewModel и View находятся в **отдельных пространствах имен**.
- Элементы и страницы класса View могут **сохранять ссылки на ViewModel**, а значит, вызывать его методы и связываться с его свойствами.
- Объекты класса ViewModel **не** сохраняют ссылки на объекты класса View.
- Для передачи данных из класса ViewModel в класс View служат **события `PropertyChanged` и `CollectionChanged`,** чтобы связанные элементы могли обновляться автоматически.
- Объекты ViewModel имеют **ссылки на объекты Model** и могут вызывать их методы, получать и задавать их свойства.
- Для передачи данных из класса Model в ViewModel **вызывается событие**.
- Объекты класса Model **не** имеют ссылок на объекты класса ViewModel.
- Класс Model **следует хорошо инкапсулировать**, чтобы он зависел только от своих объектов. Все содержимое папки Model должно компилироваться даже после удаления всего остального кода программы.
- `DispatcherTimers` и асинхронный код обычно попадают в слой ViewModel, а не в слой Model. Связанный с отсчетом времени код отвечает за то, ***как*** меняется состояние приложения, но большую часть времени не является ***частью этого состояния***.

Автоматическое преобразование значений для связывания



Любой владелец электронных часов знает, что минуты они отображают с ведущими нулями. Наш секундомер также должен отображать минуты и секунды в виде двух цифр и округлять сотые доли секунд до ближайшего целого. *Можно было бы* заставить ViewModel показывать отформатированные строковые значения, но это добавление новых свойств при каждом изменении формата. Здесь нам помогут **преобразователи значений**. Это объекты, которыми XAML-связывание пользуется для редактирования данных перед отправкой их элементу управления. Такой объект можно построить, реализовав интерфейс `IValueConverter` (он находится в пространстве имен `Windows.UI.Xaml.Data`). Добавим к нашему секундомеру такой преобразователь.

Преобразователи будут полезны при построении класса ViewModel.



1 **Добавим класс TimeNumberFormatConverter в папку ViewModel.**
Добавьте на верх класса строку **`using Windows.UI.Xaml.Data;`** и заставьте класс реализовать интерфейс **`IValueConverter`**. В IDE это происходит автоматически. При этом добавляются заглушки для методов `Convert()` и `ConvertBack()`.

2 **Реализуйте метод Convert() в преобразователе значений.**
`Convert()` принимает несколько параметров, мы используем только два: **`value`** — исходное значение, передаваемое в связывание, а **`parameter`** позволяет указать параметр в коде XAML.

```
using Windows.UI.Xaml.Data;

class TimeNumberFormatConverter : IValueConverter {
    public object Convert(object value, Type targetType,
                          object parameter, string language) {
        if (value is decimal)
            return ((decimal)value).ToString("00.00");
        else if (value is int) {
            if (parameter == null)
                return ((int)value).ToString("d1");
            else
                return ((int)value).ToString(parameter.ToString());
        }
        return value;
    }

    public object ConvertBack(object value, Type targetType,
                          object parameter, string language) {
        throw new NotImplementedException();
    }
}
```

Преобразователь умеет работать со значениями типа decimal и int. Во втором случае можно также передать параметр.

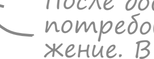
Метод ConvertBack() применяется для двустороннего связывания. В этом проекте мы не будем его использовать, поэтому оставьте заглушку метода без изменений.

Стоит ли оставлять в коде `NotImplementedException`? В этом проекте мы не собираемся запускать этот код. Но если код запущен, стоит ли скрыть от пользователя ошибку? Или лучше показать исключение, дав возможность отследить проблему? В каком случае приложение будет более надежным? Тут возможны несколько правильных ответов.

3 Добавим преобразователь к элементу как статический ресурс.

Этот код нужно поместить под объектом ViewModel:

```
<UserControl.Resources>
    <viewmodel:StopwatchViewModel x:Name="viewModel"/>
    <viewmodel:TimeNumberFormatConverter x:Name="timeNumberFormatConverter"/>
</UserControl.Resources>
```



После добавления этой строки может потребоваться перестроить приложение. В некоторых случаях придется выгрузить и заново загрузить проект.

4 Обновим код XAML.

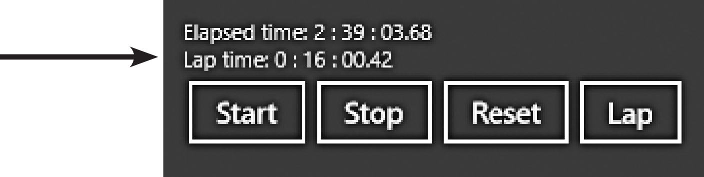
Добавьте в каждый из тегов <Run> разметки {Binding} строку Converter=.

```
<TextBlock>
    <Run>Elapsed time: </Run>
    <Run Text="{Binding Hours,
        Converter={StaticResource timeNumberFormatConverter}}"/>
    <Run>:</Run>
    <Run Text="{Binding Minutes,
        Converter={StaticResource timeNumberFormatConverter}, ConverterParameter=d2}"/>
    <Run>:</Run>
    <Run Text="{Binding Seconds,
        Converter={StaticResource timeNumberFormatConverter}}"/>
</TextBlock>
<TextBlock>
    <Run>Lap time: </Run>
    <Run Text="{Binding LapHours,
        Converter={StaticResource timeNumberFormatConverter}}"/>
    <Run>:</Run>
    <Run Text="{Binding LapMinutes,
        Converter={StaticResource timeNumberFormatConverter}, ConverterParameter=d2}"/>
    <Run>:</Run>
    <Run Text="{Binding LapSeconds,
        Converter={StaticResource timeNumberFormatConverter}}"/>
</TextBlock>
```

При отсутствии параметров не забудьте дополнительную закрывающуюся скобку }}.

Для передачи параметра в преобразователь используйте синтаксис ConverterParameter.

Теперь перед передачей значений элементам TextBlock секундомер прогоняет их через преобразователь, и на странице числа отображаются в корректном формате.



Преобразователи работают с разными типами

Элементы управления TextBlock и TextBox работают с текстом, поэтому строки и цифры имеет смысл связывать со свойством `Text`. Но существуют и другие допускающие связывание свойства. При наличии у объекта ViewModel свойства типа Boolean его можно связать с любым свойством, имеющим значение `true/false`. Связывание возможно даже для свойств, использующих перечисления, свойство `IsVisible` работает с перечислением `Visibility`, и вы можете написать для него преобразователь значений. Добавим к нашему секундомеру новые типы связывания.

Нам пригодятся два преобразователя.

Иногда требуется связать свойства типа Boolean, такие как `IsEnabled`, чтобы элемент управления был доступен при значении `false` связанного свойства. Добавим новый преобразователь `BooleanNotConverter`, который при помощи оператора ! инвертирует целевое свойство.

```
IsEnabled="{Binding Running, Converter={StaticResource notConverter}}"
```

Иногда нужно, чтобы элементы отображались в зависимости от логического свойства в контексте данных. Связывание допускает только свойство Visibility элемента с целевым свойством типа `Visibility` (возвращающее такие значения, как `Visibility.Collapsed`). Добавим преобразователь `BooleanVisibilityConverter`, который свяжет эти два свойства, обеспечив управление видимостью элемента.

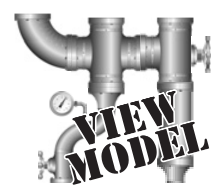
```
Visibility="{Binding Running, Converter={StaticResource visibilityConverter}}"
```

Отредактируйте обработчик события Tick класса ViewModel.

Заставьте обработчик события `Tick` объекта `DispatcherTimer` вызывать событие `PropertyChanged` при изменении значения свойства `Running`:

```
int _lastHours;
int _lastMinutes;
decimal _lastSeconds;
bool _lastRunning;
void TimerTick(object sender, object e) {
    if (_lastRunning != Running) {
        _lastRunning = Running;
        OnPropertyChanged("Running");
    }
    if (_lastHours != Hours) {
        _lastHours = Hours;
        OnPropertyChanged("Hours");
    }
    if (_lastMinutes != Minutes) {
        _lastMinutes = Minutes;
        OnPropertyChanged("Minutes");
    }
    if (_lastSeconds != Seconds) {
        _lastSeconds = Seconds;
        OnPropertyChanged("Seconds");
    }
}
```

Мы добавили проверку свойства Running к таймеру. Может быть, вместо этого следовало заставить класс Model вызывать событие?



Theme	Light
	Default
	Dark
	Light
	High Contrast (default)
	High Contrast White
	High Contrast Black
	High Contrast #1
	High Contrast #2

Если вы не видите элемент управления в конструкторе, попробуйте выбрать в окне Device другую тему.

2 **Добавим преобразователь значений типа Boolean.**

Вот преобразователь, меняющий значение `true` на `false` и наоборот. Он используется с такими свойствами элементов, как `IsEnabled`.

```
using Windows.UI.Xaml.Data;

class BooleanNotConverter : IValueConverter {
    public object Convert(object value, Type targetType, object parameter, string language) {
        if ((value is bool) && ((bool)value) == false)
            return true;
        else
            return false;
    }
    public object ConvertBack(object value, Type targetType, object parameter, string language) {
        throw new NotImplementedException();
    }
}
```

3 **Преобразователь значений типа Booleans в перечисления Visibility.**

Вы уже умеете управлять видимостью элемента, присваивая свойству `Visibility` значение `Visible` или `Collapsed`. Это значения расположенного в пространстве имен *Windows.UI.Xaml* перечисления `Visibility`. Вот преобразователь значений Boolean в значения Visibility:

```
using Windows.UI.Xaml;
using Windows.UI.Xaml.Data;

class BooleanVisibilityConverter : IValueConverter {
    public object Convert(object value, Type targetType, object parameter, string language) {
        if ((value is bool) && ((bool)value) == true)
            return Visibility.Visible;
        else
            return Visibility.Collapsed;
    }
    public object ConvertBack(object value, Type targetType, object parameter, string language) {
        throw new NotImplementedException();
    }
}
```

4 **Заставим базовый элемент управления секундомером использовать преобразователи.**

Добавьте в файл *BasicStopwatch.xaml* экземпляры преобразователей как статические ресурсы:

```
<viewmodel:BooleanVisibilityConverter x:Key="visibilityConverter"/>
<viewmodel:BooleanNotConverter x:Key="notConverter"/>
```

Теперь свойства `IsEnabled` и `Visibility` элементов управления можно связать со свойством `Running` класса ViewModel:

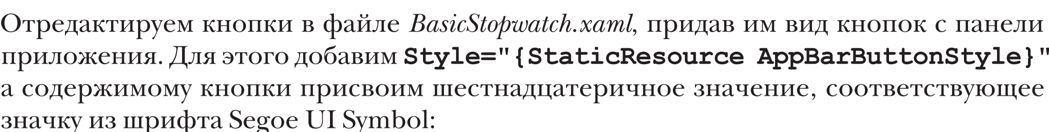
```
<StackPanel Orientation="Horizontal">
    <Button IsEnabled="{Binding Running, Converter={StaticResource notConverter}}"
            Click="StartButton_Click">Start</Button>
    <Button IsEnabled="{Binding Running}" Click="StopButton_Click">Stop</Button>
    <Button Click="ResetButton_Click">Reset</Button>
    <Button IsEnabled="{Binding Running}" Click="LapButton_Click">Lap</Button>
</StackPanel>
<TextBlock Text="Stopwatch is running"
           Visibility="{Binding Running, Converter={StaticResource visibilityConverter}}"/>
```

Это дает доступ к кнопке Start только при остановленном секундомере.

Это делает видимым элемент TextBlock при работающем секундомере.

Меняем вид кнопок

При построении слоя View обычно пишется код XAML. Эти элементы управления XAML являются обычными объектами, поэтому слой View *возможно* целиком построить при помощи кода C#, но XAML оптимизирован под эту работу. Посмотрим, как все это работает, на примере кнопок панели приложения.



Отредактируем кнопки в файле *BasicStopwatch.xaml*, придав им вид кнопок с панели приложения. Для этого добавим **`Style="{StaticResource AppBarButtonStyle}"`** а содержимому кнопки присвоим шестнадцатеричное значение, соответствующее значку из шрифта Segoe UI Symbol:

```
<Button Style="{StaticResource AppBarButtonStyle}"
        IsEnabled="{Binding Running, Converter={StaticResource notConverter}}"
        AutomationProperties.Name="Start"
        Click="StartButton_Click">&#xE102;</Button>

<Button Style="{StaticResource AppBarButtonStyle}"
        AutomationProperties.Name="Stop"
        IsEnabled="{Binding Running}" Click="StopButton_Click">&#xE103;</Button>

<Button Style="{StaticResource AppBarButtonStyle}"
        AutomationProperties.Name="Reset"
        Click="ResetButton_Click">&#xE10E;</Button>

<Button Style="{StaticResource AppBarButtonStyle}"
        AutomationProperties.Name="Lap"
        IsEnabled="{Binding Running}" Click="LapButton_Click">&#xE16D;</Button>
```

Теперь кнопки стали круглыми, на них появились значки, а под ними имена:

В главе 11 рассматривался статический ресурс `AppBarButtonStyle`, определенный в файле *StandardStyles.xaml* и добавляемый к проекту вместе с шаблоном Basic Page. Но что именно при этом происходит? Как и во всех остальных случаях с приложениями C#, никакой магии нет: кнопки панели приложения при помощи **стилей и шаблонов элементов управления** определяют свой вид, и потом его можно применять многократно.

① Откройте в IDE файл *StandardStyles.xaml*. Открывающий тег указывает на наличие `ResourceDictionary`, объекта, обеспечивающего приложение статическими ресурсами.

```
<ResourceDictionary

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
```

② Запросом `AppBarButtonStyle` найдите примененный к кнопкам статический ресурс. Вы обнаружите, что его задает тег **`<Style>`**. Он обладает механизмом, **задающим свойства всех элементов, к которым применяется стиль**. Свойство стиля `TargetType` определяет, какому элементу он назначается. В нашем случае это `ButtonBase`, от которого наследует класс `Button`. Стиль содержит теги `<Setter>` задающие свойства элементов управления, которым он назначается.

```
<Style x:Key="AppBarButtonStyle" TargetType="ButtonBase">

    <Setter Property="Foreground"
            Value="{StaticResource AppBarItemForegroundThemeBrush}"/>

    <Setter Property="VerticalAlignment" Value="Stretch"/>

    <Setter Property="FontFamily" Value="Segoe UI Symbol"/>

    <Setter Property="FontWeight" Value="Normal"/>

    <Setter Property="FontSize" Value="20"/>

    <Setter Property="AutomationProperties.ItemType" Value="App Bar Button"/>
```

Эти механизмы задают цвет, выравнивание и шрифт всех кнопок, которым назначается стиль.

Этому статическому ресурсу присваивается светлый или темный цвет, в зависимости от того, какая тема используется для отображения элемента управления.

Другие элементы управления могут на основе этих свойств определить, что это кнопки панели приложения.

③ Следующий `<Setter>` присваивает свойству `Template` значение `<ControlTemplate>`, определяя **шаблон** для элемента управления. При рисовании кнопок на странице Windows ищет способ их отображения именно в шаблоне и отображает все входящие в шаблон элементы управления.

```
<Setter Property="Template">
    <Setter.Value>
        <ControlTemplate TargetType="ButtonBase">
```

Этот шаблон элемента управления применим к объектам ButtonBase или их подклассам (например, Button).

Что-то знакомое... Кажется, в главе 1 я использовал ControlTemplate?

Да! Мы пользовались IDE для создания шаблона врага.

И вернувшись к тому коду, вы сможете понять, как все работает. IDE добавила шаблон элемента управления как статический ресурс `EnemyTemplate`, а вы придали элементу сходство с инопланетянином, заставив его свойство `Template` указывать на шаблон. IDE создала шаблон с `x:Key` вместо `x:Name`, и код смог осуществлять поиск по имени в коллекции `Resources`.

4 Для рисования кнопки шаблон использует элемент StackPanel, содержащий элементы Grid и TextBlock. Сетка не имеет строк и столбцов, в данном случае мы пользуемся тем, что элементы в ячейке сетки располагаются друг над другом (как в предыдущей главе при рассмотрении перенаправленных событий). Для рисования круглой кнопки применяются два глифа из шрифта Segoe UI Symbol: `` (залитый круг) и `` (пустой круг). Вы можете сами посмотреть в таблице Charmap. Поверх глифов находится элемент управления ContentPresenter. При создании объекта `Button` все это замещается содержимым, которое вы поместите между открывающим и закрывающим тегами или в свойство `Content`, — эта система в буквальном смысле представляет собой содержимое.

```
<ControlTemplate TargetType="ButtonBase">

    <Grid x:Name="RootGrid" Width="100" Background="Transparent">

        <StackPanel VerticalAlignment="Top" Margin="0,12,0,11">

            <Grid Width="40" Height="40" Margin="0,0,0,5" HorizontalAlignment="Center">

                <TextBlock x:Name="BackgroundGlyph" Text="&#xE0A8;"
                           FontFamily="Segoe UI Symbol" FontSize="53.333" Margin="-4,-19,0,0"
                           Foreground="{StaticResource AppBarItemBackgroundThemeBrush}"/>

                <TextBlock x:Name="OutlineGlyph" Text="&#xE0A7;" FontFamily="Segoe UI Symbol"
                           FontSize="53.333" Margin="-4,-19,0,0"/>

                <ContentPresenter x:Name="Content" HorizontalAlignment="Center"
                                  Margin="-1,-1,0,0" VerticalAlignment="Center"/>

            </Grid>

            <TextBlock
                x:Name="TextLabel" Text="{TemplateBinding AutomationProperties.Name}"
                Foreground="{StaticResource AppBarItemForegroundThemeBrush}"
                Margin="0,0,2,0" FontSize="12" TextAlignment="Center"
                Width="88" MaxHeight="32" TextTrimming="WordEllipsis"
                Style="{StaticResource BasicTextStyle}"/>

        </StackPanel>
```

Stop

На последней странице вы добавили глиф E103 «кнопка паузы» как картинку кнопки, и именно она попадает в элемент TextBlock. На шаге 2 шрифту присваивается Segoe UI Symbol.

Разметка `TemplateBinding` позволяет привязывать свойства шаблона элемента управления к свойствам элемента, которому этот шаблон назначается, и при наличии привязки к свойствам `Text`, `Width`, `Foreground` и пр. можно задать значение в объекте `Button` и увидеть его в шаблоне. Класс `AutomationProperties` предоставляет еще несколько вариантов связывания.

Дополнительные примеры можно найти в файле StandardStyles.xaml, введя строку поиска AutomationProperties.

Класс `AutomationProperties` является удобным способом передачи в шаблон элемента управления дополнительных значений, хотя изначально он предназначен для специальных возможностей. Подробнее узнать о нем можно тут: http://msdn.microsoft.com/ru-ru/library/windows/apps/xaml/hh868160.aspx

⑤ Последние два элемента управления рисуют вокруг элемента прямоугольники. Первый называется FocusVisualWhite и рисует пунктирные линии. Второй называется FocusVisualBlack и тоже рисует пунктирные линии, но более частым пунктиром. Эти прямоугольники отображаются, если запустить секундомер и нажать клавишу Tab для переключения между кнопками.

```
<Rectangle
        x:Name="FocusVisualWhite" IsHitTestVisible="False"
        Stroke="{StaticResource FocusVisualWhiteStrokeThemeBrush}"
        StrokeEndLineCap="Square" StrokeDashArray="1,1"
        Opacity="0" StrokeDashOffset="1.5"/>

<Rectangle
        x:Name="FocusVisualBlack IsHitTestVisible="False"
        Stroke="{StaticResource FocusVisualBlackStrokeThemeBrush}"
        StrokeEndLineCap="Square" StrokeDashArray="1,1"
        Opacity="0" StrokeDashOffset="0.5"/>
```

Стили меняют любой элемент указанного типа

Посмотрите, как в файле *StandardStyles.xaml* определен стиль `AppBarButtonStyle`:

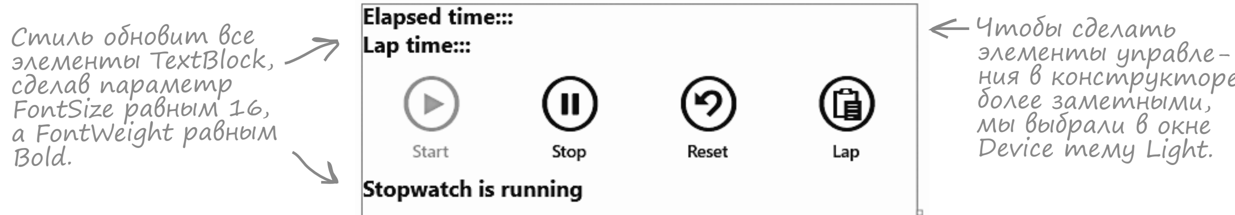
```
<Style x:Key="AppBarButtonStyle" TargetType="ButtonBase">
```

Он добавляется как статический ресурс и благодаря ключу `AppBarButtonStyle` может назначаться кнопкам (или любому расширяющему класс `ButtonBase` классу) через свойство `Style`. А что будет, если не указывать ключ? Тогда стиль будет **автоматически применяться ко всем классам, совпадающим с `TargetType`**. Посмотрим на практике, как это выглядит.

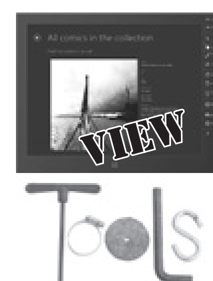
⑥ Откройте файл *BasicStopwatch.xaml* и добавьте в раздел `<UserControl.Resources>` стиль как статический ресурс со значением `TargetType` элемента `TextBlock`. Задайте кегль и насыщенность шрифта:

```
<Style TargetType="TextBlock">

    <Setter Property="FontSize" Value="16"/>

    <Setter Property="FontWeight" Value="Bold"/>

</Style>
```

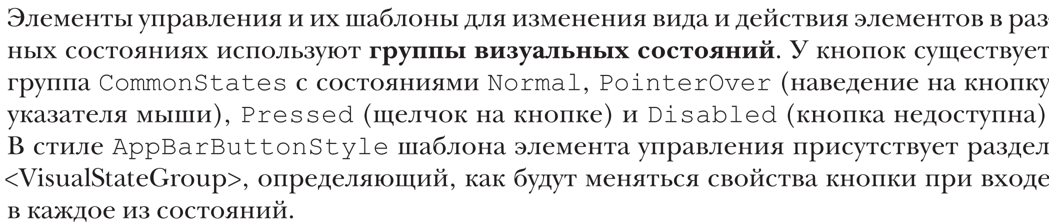
Сразу же после задания стиля все элементы управления TextBlock в классе `BasicStopwatch` соответствующим образом изменят свойства `FontSize` и `FontWeight`:



Визуальные состояния и элементы управления



При наведении указателя мыши кнопка становится непрозрачной. При переносе фокуса на кнопку вокруг нее появляется пунктирная линия. Это происходит, так как **вы меняете состояние кнопки**. Состояние с наведенным указателем называется `PointerOver`. Существует множество различных состояний, и большинство элементов управления вовсе не обязано реагировать на каждое из них.



Элементы управления и их шаблоны для изменения вида и действия элементов в разных состояниях используют **группы визуальных состояний**. У кнопок существует группа `CommonStates` с состояниями `Normal`, `PointerOver` (наведение на кнопку указателя мыши), `Pressed` (щелчок на кнопке) и `Disabled` (кнопка недоступна). В стиле `AppBarButtonStyle` шаблона элемента управления присутствует раздел <VisualStateGroup>, определяющий, как будут меняться свойства кнопки при входе в каждое из состояний.

```
<VisualStateGroup x:Name="CommonStates">
    <VisualState x:Name="Normal"/>
    <VisualState x:Name="PointerOver">
        <Storyboard>
            Анимация при наведении указателя мыши на кнопку
            Другая анимация для другого свойства в том же состоянии
        </Storyboard>
    </VisualState>
    <VisualState x:Name="Pressed">
        <Storyboard>
            Анимация при нажатии кнопки
            Другая анимация для другого свойства в том же состоянии
            И еще одна... Анимаций для свойств может быть много
        </Storyboard>
    </VisualState>
    <VisualState x:Name="Disabled">
        <Storyboard>
            Анимация при нажатии кнопки
            Другая анимация для другого свойства в том же состоянии
            и т. п.
        </Storyboard>
    </VisualState>
</VisualStateGroup>
```

В обычном состоянии элемент управления не делает ничего особенного.

Каждое состояние обрабатывается с тегом `<VisualState>`, содержащим объект `Storyboard`, который управляет анимациями и может использовать временную шкалу для определения начала и конца анимации.

Все объекты `Storyboard` используют для редактирования свойств анимации. При входе элемента управления в состояние объект `Storyboard` запускает анимации, меняющие значения свойств.

Класс DoubleAnimation

При задании численных свойств, например, `Width` или `Height` элемента управления в XAML задается также значение свойства `double` соответствующего объекта. Класс `DoubleAnimation` позволяет **постепенно менять это свойство от одного значения к другому за заданный промежуток времени**. Шаблон элемента управленяи в стиле `AppBarButtonStyle` использует класс `DoubleAnimation` для анимации «сфокусированного» состояния путем изменения прозрачности двух прямоугольников вокруг элемента управления с 0 (прозрачный) до 1 (непрозрачный). Продолжительность анимации равна нулю, то есть изменение происходит мгновенно:

```
<VisualState x:Name="Focused">
    <Storyboard>
        <DoubleAnimation
            Storyboard.TargetName="FocusVisualWhite"
            Storyboard.TargetProperty="Opacity"
            To="1"
            Duration="0"/>
        <DoubleAnimation
            Storyboard.TargetName="FocusVisualBlack"
            Storyboard.TargetProperty="Opacity"
            To="1"
            Duration="0"/>
    </Storyboard>
</VisualState>
```

Эта анимация применяется к элементу FocusVisualWhite.

Свойство Opacity медленно меняется от значения по умолчанию к 1.

При выходе кнопки из сфокусированного состояния объект storyboard возвращается в исходное состояние, а с ним и вся анимация, что означает возвращение прозрачности значения 0.

Поэкспериментируем с этой анимацией, скопировав стиль целиком в пользовательский элемент управления, заставив кнопку пользоваться этим стилем и отредактировав анимацию:

1. **Скопируйте стиль**, начиная с `<Style x:Key="AppBarButtonStyle" TargetType="ButtonBase">` и заканчивая тегом `<Style>`. **Вставьте его** в раздел `<UserControl.Resources>` файла *BasicStopwatch.xaml* сразу за преобразователями и **поменяйте ключ** с `AppBarButtonStyle` на `StopwatchButtonStyle`.

Вид кнопок пока не будет отличаться, так как добавленный в виде статического ресурса стиль был скопирован из уже имеющегося стиля.

2. Примените новый стиль ко всем кнопками, присвоив свойству `Style` значение `Style="{StaticResource StopwatchButtonStyle}"`.

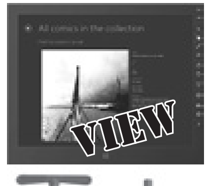
3. Отредактируйте теги `DoubleAnimation` в состоянии `Focused`, поменяв продолжительность анимации на 5 секунд. Этот параметр всегда имеет форму **`hours:minutes:seconds`**, поэтому поменяйте его на **`Duration="0:0:5"`** (в обеих анимациях, чтобы все работало, как в светлой, так и в темной теме).

4. Запустите программу и **нажмите клавишу Tab** для перехода фокуса с кнопки на кнопку. Теперь пунктирный контур должен исчезать в течеие 5 секунд.

5. Снова отредактируйте анимацию, на этот раз указав параметры: **`Duration="0:0:0.5" AutoReverse="true" RepeatBehavior="Forever"`**

6. Запустите программу. Теперь прямоугольник в течение половины секунды будет становиться более четким, а за следующие полсекунды плавно исчезать.

Анимация параметров через анимацию объектов



Некоторые свойства объектов завязаны на двойные значения, а некоторые на объекты. Присваивая свойству `Foreground` значение Black, вы на самом деле присваиваете объект `SolidColorBrush`. Вспомините анимацию состояния `Pressed` для стиля StopwatchButtonStyle, в которой `ObjectAnimationUsingKeyFrames` при нажатии кнопки меняет цвет круглого фонового глифа:

```
<VisualState x:Name="Pressed">
  <Storyboard>
    <ObjectAnimationUsingKeyFrames
         Storyboard.TargetName="BackgroundGlyph" Storyboard.TargetProperty="Foreground">
        <DiscreteObjectKeyFrame KeyTime="0"
             Value="{StaticResource AppBarItemPointerOverBackgroundThemeBrush}"/>
    </ObjectAnimationUsingKeyFrames>
    <ObjectAnimationUsingKeyFrames Storyboard.TargetName="Content"
         Storyboard.TargetProperty="Foreground">
        <DiscreteObjectKeyFrame KeyTime="0"
             Value="{StaticResource AppBarItemPointerOverForegroundThemeBrush}"/>
    </ObjectAnimationUsingKeyFrames>
  </Storyboard>
</VisualState>
```

Анимация по ключевым кадрам работает за счет создания **ключевых кадров (key frames)** или дискретных событий, происходящих в определенные моменты времени. Давайте **добавим третью анимацию к состоянию `Pressed`**. Вставьте этот код перед закрывающим тегом `</Storyboard>`:

```
<ObjectAnimationUsingKeyFrames
        Storyboard.TargetName="Content" Storyboard.TargetProperty="Visibility">
    <DiscreteObjectKeyFrame KeyTime="0:0:0" Value="Visible"/>
    <DiscreteObjectKeyFrame KeyTime="0:0:0.2" Value="Collapsed"/>
    <DiscreteObjectKeyFrame KeyTime="0:0:0.4" Value="Visible"/>
    <DiscreteObjectKeyFrame KeyTime="0:0:0.6" Value="Collapsed"/>
    <DiscreteObjectKeyFrame KeyTime="0:0:0.8" Value="Visible"/>
</ObjectAnimationUsingKeyFrames>
```

Запустите программу. При нажатии на кнопку глиф `Content` начнет мигать. Как только вы отпустите кнопку, анимация остановится на середине. Кнопка возвращается в состояние `Normal`, и анимация сбрасывается в начальную точку.

Пример применения визуальных состояний в шаблоне элемента управления для флажка:
http://msdn.microsoft.com/ru-ru/library/windows/apps/xaml/hh465374.aspx

Аналоговый секундомер на основе того же класса ViewModel

Помните, как вы без изменений перенесли классы данных, созданные в главе 14 для каталога комиксов, в приложение Split? Здесь мы поступим так же.

Шаблон MVVM **изолирует** слой View от слоя ViewModel, а слой ViewModel — от слоя Model. Это полезно в ситуации, когда изменения требуется внести только в один слой. Вы можете быть уверены, что «эффект дробовика» не появится в остальных слоях. Хорошо ли мы выполнили изоляцию слоя View от слоя ViewModel? Ответить на этот вопрос можно только одним способом: давайте построим новый слой View, не затрагивая классы ViewModel. Единственным изменением кода C# будет **новый преобразователь в слое ViewModel**, превращающий минуты и секунды в углы.

Упражнение!

Добавим преобразователь времени в углы.
Добавьте в папку *ViewModel* класс `AngleConverter`. Он нужен для имитации стрелок.

```
using Windows.UI.Xaml.Data;
class AngleConverter : IValueConverter {
    public object Convert(object value, Type targetType, object parameter, string language) {
        double parsedValue;
        if ((value != null)
            && double.TryParse(value.ToString(), out parsedValue)
            && (parameter != null))
            switch (parameter.ToString()) {
                case "Hours":
                    return parsedValue * 30;
                case "Minutes":
                case "Seconds":
                    return parsedValue * 6;
            }
        return 0;
    }
    public object ConvertBack(object value, Type targetType, object parameter, string language) {
        throw new NotImplementedException();
    }
}
```

Параметр hour меняется от 0 до 11, поэтому для преобразования в угол мы умножаем его на 30.

Диапазон минут и секунд меняется от 0 до 60, поэтому преобразование в угол означает умножение на 6.

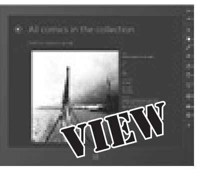
Добавим новый UserControl.
Добавьте **в папку *View* новый пользовательский элемент управления `AnalogStopwatch`,** вставьте `ViewModel` в тег `<UserControl>` и поменяйте расчетную ширину и высоту:

```
d:DesignHeight="300"
d:DesignWidth="400"
xmlns:viewmodel="using:Stopwatch.ViewModel">
```

Добавьте ViewModel, два преобразователя и стиль к статическим ресурсам пользовательского элемента управления.

```
<UserControl.Resources>
    <viewmodel:StopwatchViewModel x:Name="viewModel"/>
    <viewmodel:BooleanNotConverter x:Key="notConverter"/>
    <viewmodel:AngleConverter x:Key="angleConverter"/>
    <Style TargetType="TextBlock">
        <Setter Property="RenderTransformOrigin" Value="0.5,0.5"/>
        <Setter Property="Foreground" Value="Black"/>
        <Setter Property="FontSize" Value="20"/>
    </Style>
</UserControl.Resources>
```

Добавим к сетке циферблат и стрелки.
Отредактируйте тег <Grid>, добавив циферблат секундомера и четыре прямоугольника для стрелок.




```
<Grid x:Name="baseGrid" DataContext="{StaticResource ResourceKey=viewModel}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="400"/>
    </Grid.ColumnDefinitions>
    <Ellipse Width="300" Height="300" Stroke="Black" StrokeThickness="2">
        <Ellipse.Fill>
            <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
                <LinearGradientBrush.RelativeTransform>
                    <CompositeTransform CenterY="0.5" CenterX="0.5" Rotation="45"/>
                </LinearGradientBrush.RelativeTransform>
                <GradientStop Color="#FFB03F3F"/>
                <GradientStop Color="#FFE4CECE" Offset="1"/>
            </LinearGradientBrush>
        </Ellipse.Fill>
    </Ellipse>
    <Rectangle RenderTransformOrigin="0.5,0.5" Width="2" Height="150" Fill="Black">
        <Rectangle.RenderTransform>
            <TransformGroup>
                <TranslateTransform Y="-60"/>
                <RotateTransform Angle="{Binding Seconds,
                    Converter={StaticResource ResourceKey=angleConverter},
                    ConverterParameter=Seconds}"/>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Rectangle RenderTransformOrigin="0.5,0.5" Width="4" Height="100" Fill="Black">
        <Rectangle.RenderTransform>
            <TransformGroup>
                <TranslateTransform Y="-50"/>
                <RotateTransform Angle="{Binding Minutes,
                    Converter={StaticResource ResourceKey=angleConverter},
                    ConverterParameter=Minutes}"/>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
   <Rectangle RenderTransformOrigin="0.5,0.5" Width="1" Height="150" Fill="Yellow">
         Rectangle.RenderTransform>
            <TransformGroup>
                <TranslateTransform Y="-60"/>
                <RotateTransform Angle="{Binding LapSeconds,
                    Converter={StaticResource ResourceKey=angleConverter},
                    ConverterParameter=Seconds}"/>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Rectangle RenderTransformOrigin="0.5,0.5" Width="2" Height="100" Fill="Yellow">
        <Rectangle.RenderTransform>
            <TransformGroup>
                <TranslateTransform Y="-50"/>
                <RotateTransform Angle="{Binding LapMinutes,
                    Converter={StaticResource ResourceKey=angleConverter},
                    ConverterParameter=Minutes}"/>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
   <Ellipse Width="10" Height="10" Fill="Black"/>
</Grid>
```

Задав ширину столбца, вы помешаете ему увеличиться до размеров контейнера.

Циферблат секундомера с черным ободком и серым градиентом в качестве фона.

Секундная стрелка — длинный, тонкий элемент Rectangle с преобразованиями translate и rotate

Минутная стрелка

У каждого элемента управления может быть только один раздел RenderTransform.

Две желтые стрелки для промежуточной отсечки

Тег TransformGroup позволяет применять к одному элементу управления несколько преобразований.

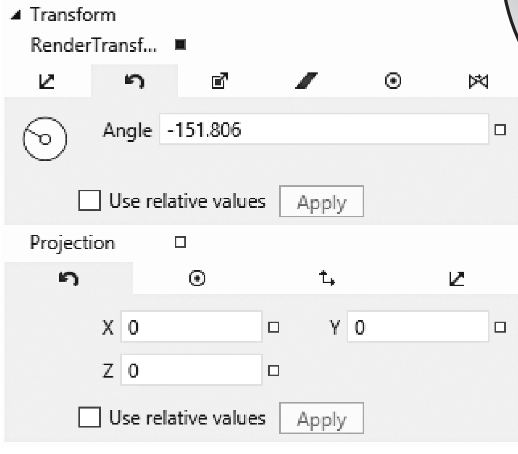
Мы рисуем дополнительный круг в центре, который прикроет место пересечения стрелок. Так как его код находится в нижней части контейнера Grid, он будет нарисован последним и окажется сверху.

Для поверхности секундомера мы используем градиент как для фона в проекте *Save the Humans*.

Каждая стрелка подвергается преобразованиям два раза. Она появляется в центре, поэтому сначала мы сдвигаем ее в нужное положение.

```
<TranslateTransform Y="-60"/>

<RotateTransform Angle="{Binding Seconds,
    Converter={StaticResource ResourceKey=angleConverter},
    ConverterParameter=Seconds}"/>
```

Второе преобразование поворачивает стрелку на нужный угол. Свойство `Angle` преобразования поворота связано с секундами или минутами в классе ViewModel, а преобразователь превращает его значение в угол.



Все элементы управления обладают элементом `RenderTransform`, меняющим способ их отображения. Сюда входят поворот, смещение, сжатие и т. п.

В проекте *Save the Humans* мы пользовались преобразованиями для придания эллипсу формы, напоминающей инопланетянина.

Секундомер начнет работу сразу же после появления второй стрелки, так как экземпляр ViewModel будет создан в виде статического ресурса для визуализации элемента управления в конструкторе. Конструктор может прекратить обновление секундомера, но его легко перезапустить, на время покинув окно конструктора и затем вернувшись в него.

Добавим к секундомеру кнопки.
Кнопки аналогового секундомера будут в стиле, скопированном в файл *BasicStopwatch.xaml*, но копировать их и снова вставлять не нужно. К счастью, вы знаете, как поступать в подобной ситуации: воспользуйтесь словарем ресурсов, например, как в файле *StandardStyles.xaml*. **Добавьте в папку *View* новый элемент Resource Dictionary с именем *StopwatchStyles.xaml*** (окно New Item, через которое добавлялся User Control). Вырежьте стиль `StopwatchButtonStyle` из файла *BasicStopwatch.xaml* и **вставьте его в новый файл *StopwatchStyles.xaml*.**

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Stopwatch.View">

    <Style x:Key="StopwatchButtonStyle" TargetType="ButtonBase">
        <Setter Property="Foreground" Value="{StaticResource AppBarItemForegroundThemeBrush}"/>

    ...

    </Style>
</ResoucreDictionary>
```

В файле *App.xaml* добавьте словарь к ресурсам вашего приложения. При создании нового приложения для магазина Windows IDE генерирует файл *App.xaml* с единственным тегом `<Application.Resources>`, и именно через него приложение узнает о стилях в *StandardStyles.xaml*. Отредактируйте тег, добавив новый словарь ресурсов:

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>

            <!--
                Styles that define common aspects of the platform look and feel
                Required by Visual Studio project and item templates
                -->
            <ResourceDictionary Source="Common/StandardStyles.xaml"/>
            <ResourceDictionary Source="View/StopwatchStyles.xaml"/>
        </ResourceDictionary.MergedDictionaries>

    </ResourceDictionary>
</Application.Resources>
```

При добавлении этой строки в файл App.xaml стили из нового файла StopwatchStyles.xaml добавляются в ресурсы вашего приложения.

Теперь можно добавить кнопки. Скопируйте StackPanel и вставьте в новый элемент.

```
<StackPanel Orientation="Horizontal" VerticalAlignment="Bottom">
    <Button Style="{StaticResource StopwatchButtonStyle}"
       IsEnabled="{Binding Running, Converter={StaticResource notConverter}}"
       AutomationProperties.Name="Start" Click="StartButton_Click">&#xE102;</Button>
    <Button Style="{StaticResource StopwatchButtonStyle}" AutomationProperties.Name="Stop"
      IsEnabled="{Binding Running}" Click="StopButton_Click">&#xE103;</Button>
    <Button Style="{StaticResource StopwatchButtonStyle}" AutomationProperties.Name="Reset"
      Click="ResetButton_Click">&#xE10E;</Button>
    <Button Style="{StaticResource StopwatchButtonStyle}" AutomationProperties.Name="Lap"
      IsEnabled="{Binding Running}" Click="LapButton_Click">&#xE16D;</Button>
</StackPanel>
```

Вертикальное выравнивание помещает кнопку вниз.

Нам нравится, когда кнопки прячутся за циферблатом. Можно добавить их во вторую строку сетки, разместив под циферблатом.

Отредактируем программный код и обновим страницу.

Вы добавили кнопки, но методы обработки их событий пока отсутствуют. Программный код кнопок будет тем же самым, что и в основной программе:

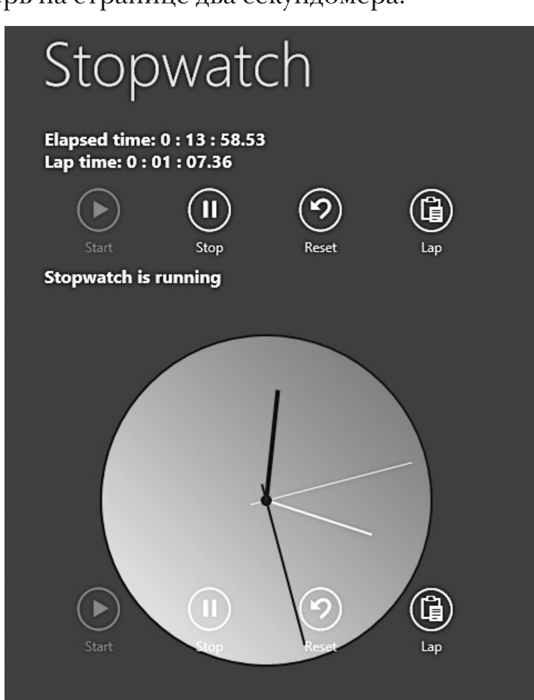
```
private void StartButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Start();
}
private void StopButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Stop();
}
private void ResetButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Reset();
}
private void LapButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Lap();
}
```

Осталось добавить в файл *MainPage.xaml* элемент `AnalogStopwatch`:

```
<StackPanel Orientation="Vertical" Grid.Row="1" Margin="120,0">
    <view:BasicStopwatch Margin="0,0,0,40" />
    <view:AnalogStopwatch/>
</StackPanel>
```

Запустите приложение. Теперь на странице два секундомера.

Каждый секундомер фиксирует собственное время, так как для каждого из них существует свой, отдельный экземпляр класса ViewModel в виде статического ресурса.



Попробуйте отредактировать класс ViewModel, сделав поле _stopwatchModel статическим. Как изменилось поведение приложения? Можете ли вы объяснить, что произошло?

Экземпляры элементов UI

Вы уже знаете, что код XAML создает экземпляры классов в пространстве имен `Windows.UI`, и даже исследовали их в главе 10 при помощи окна Watch. Но что, если элемент управления потребуется создать прямо в коде? Элементы управления являются объектами, и работать с ними можно так же, как и с любыми другими объектами. Поэтому **добавим через программный код разметку на циферблат аналогового секундомера**.

```
using Windows.UI;
using Windows.UI.Xaml.Shapes;
using Windows.UI.Xaml.Media;
public sealed partial class AnalogStopwatch : UserControl {
    public AnalogStopwatch() {
        this.InitializeComponent();
        AddMarkings();
    }

    private void AddMarkings() {
        for (int i = 0; i < 360; i += 3) {
            Rectangle rectangle = new Rectangle();
            rectangle.Width = (i % 30 == 0) ? 3 : 1;
            rectangle.Height = 15;
            rectangle.Fill = new SolidColorBrush(Colors.Black);
            rectangle.RenderTransformOrigin = new Point(0.5, 0.5);

            TransformGroup transforms = new TransformGroup();
            transforms.Children.Add(new TranslateTransform() { Y = -140 });
            transforms.Children.Add(new RotateTransform() { Angle = i });
            rectangle.RenderTransform = transforms;
            baseGrid.Children.Add(rectangle);
        }
    }
    // ... обработчики событий кнопок остались без изменений
```

Нам потребуется пространство имен Windows.UI для класса Colors, Windows.UI.Xaml.Shapes для элемента Rectangle и преобразований и Windows.UI.Xaml.Media для кисти SolidColorBrush.

Вставим в конструктор вызов добавляющего разметку метода.

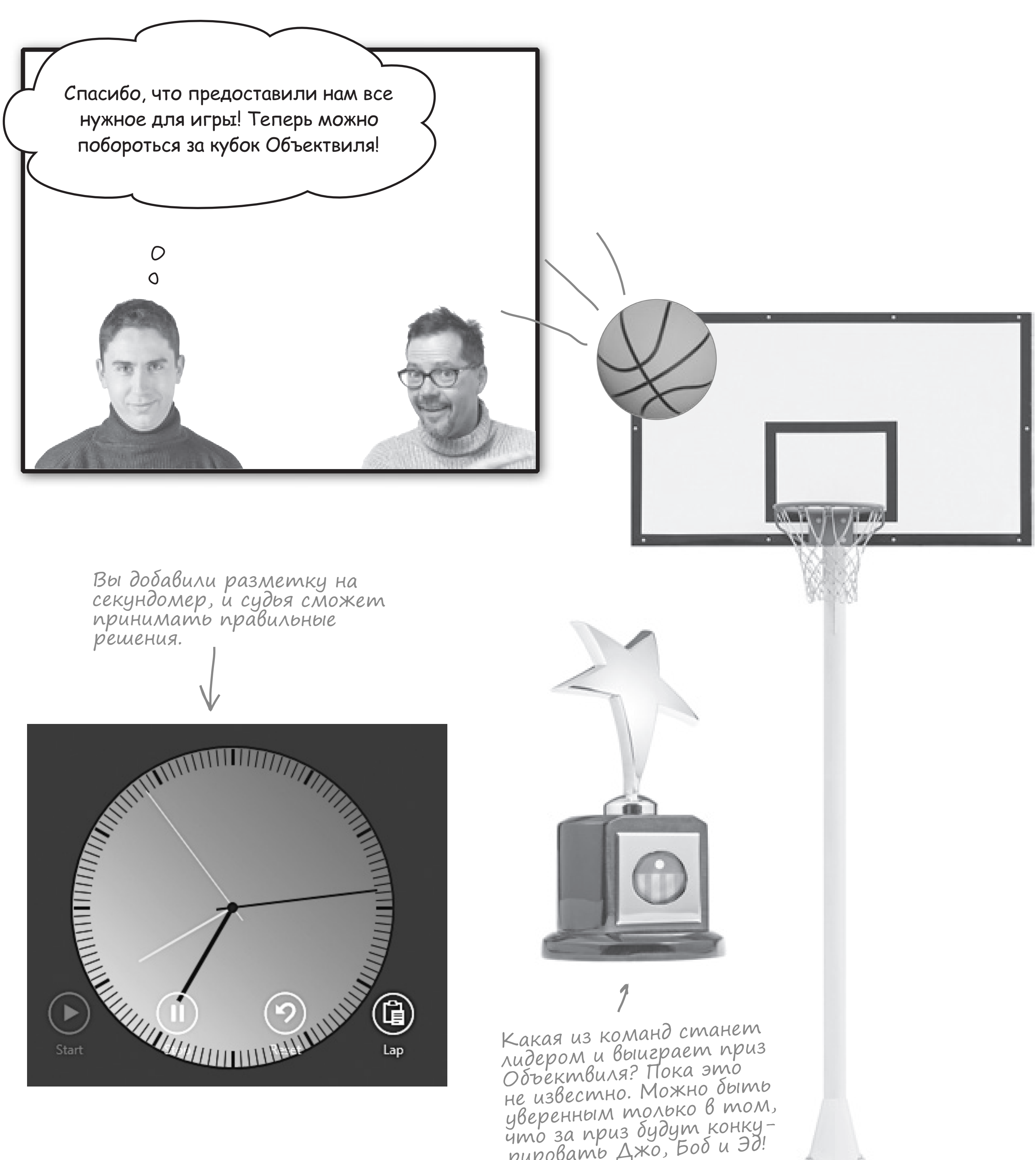
Оператор % позволяет сделять риски часов из рисок для минут. i % 30 возвращает 0, только если i делится на 30.

Здесь создаются экземпляры объекта Rectangle, созданного тегом <Rectangle>.

Вернитесь к коду XAML для часовой и минутной стрелок. Этот код задает те же преобразования, но вместо связывания свойства Angle он присваивает ему значения.

Такие элементы, как Grid, StackPanel и Canvas, обладают коллекцией `Children` со ссылками на все содержащиеся внутри них элементы управления. Для добавления элементов к сетке используйте ее метод `Add()`, а для удаления — метод `Clear()`. Аналогичным образом преобразования добавляются в `TransformGroup`.

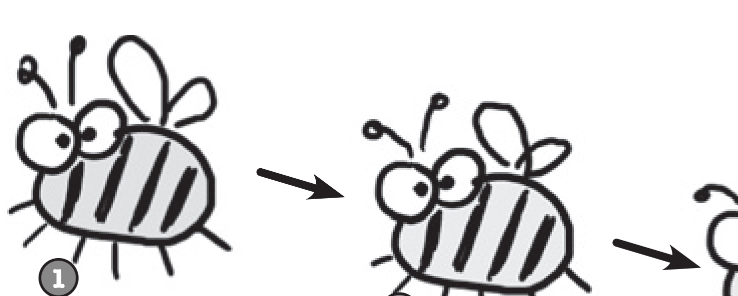
Для связывания кода C# в главе 11 использовался объект `Binding`. Можете ли вы удалить код XAML, создающий элементы управления Rectangle для часовой и минутной стрелок, и заменить его выполняющим ту же задачу кодом C#?


Спасибо, что предоставили нам все нужное для игры! Теперь можно побороться за кубок Объектвиля!
Вы добавили разметку на секундомер, и судья сможет принимать правильные решения.
Start
Lap
Какая из команд станет лидером и выиграет приз Объектвиля? Пока это не известно. Можно быть уверенным только в том, что за приз будут конкурировать Джо, Боб и Эд!

C# и «реальная» анимация

В языках C# и XAML слово *анимация* может относиться к любому меняющемуся за определенный период времени свойству. А в реальном мире оно подразумевает движущиеся картинки. Давайте попробуем написать простую программу, воспроизводящую «реальную» анимацию.

Каждая из этих пчел представляет собой один кадр, немного отличающийся от предыдущего и последующего. Прокрутка кадров в прямом и обратном порядке (кадр #1, затем #2, #3, #4 и обратно к #3 и #2) создает впечатление, что крылья пчелы движутся.



Создайте проект и добавьте картинки

Начнем с **нового проекта для магазина Windows под названием `AnimatedBee`**. Скачайте четыре картинки (это файлы *.png*) с сайта лаборатории Head First. **Добавьте их в папку *Assets*.** Также потребуется создать папки *View*, *Model* и *ViewModel*.

Скачайте изображения с сайта Head First: www.headfirstlabs.com/books/hfcsharp/

Когда вы перевернете страницу, пчелы начнут радостно махать крыльями.

Ищите анимацию везде.

Посмотрите, что происходит, когда вы заходите на стартовую страницу Windows, открываете окно About, наводите указатель мыши на кнопку и т. п. Анимация подстерегает нас повсюду, смотрите внимательно, и вы сами ее увидите.

Элемент управления для анимации картинки

Инкапсулируем покадровый код анимации. **Добавьте в папку View элемент управления *AnimatedImage*.** Он практически не содержит XAML, вся логика находится в программном коде. Вот все содержимое тега `<UserControl>` в коде XAML:

```
<Grid>
    <Image x:Name="image" Stretch="Fill"/>
</Grid>
```

Работа выполняется в программном коде. Обратите внимание на перекрытый конструктор, вызывающий метод `StartAnimation()`, который **создает объекты storyboard и keyframe**, анимируя свойство `Source` элемента управления Image.

```
using Windows.UI.Xaml.Media.Animation;
using Windows.UI.Xaml.Media.Imaging;

public sealed partial class AnimatedImage : UserControl {
    public AnimatedImage() {
        this.InitializeComponent();
    }

    public AnimatedImage(IEnumerable<string> imageNames, TimeSpan interval)
        : this()
    {
        StartAnimation(imageNames, interval);
    }

    public void StartAnimation(IEnumerable<string> imageNames, TimeSpan interval) {
        Storyboard storyboard = new Storyboard();
        ObjectAnimationUsingKeyFrames animation = new ObjectAnimationUsingKeyFrames();
        Storyboard.SetTarget(animation, image);
        Storyboard.SetTargetProperty(animation, "Source");

        TimeSpan currentInterval = TimeSpan.FromMilliseconds(0);
        foreach (string imageName in imageNames) {
            ObjectKeyFrame keyFrame = new DiscreteObjectKeyFrame();
            keyFrame.Value = CreateImageFromAssets(imageName);
            keyFrame.KeyTime = currentInterval;
            animation.KeyFrames.Add(keyFrame);
            currentInterval = currentInterval.Add(interval);
        }

        storyboard.RepeatBehavior = RepeatBehavior.Forever;
        storyboard.AutoReverse = true;
        storyboard.Children.Add(animation);
        storyboard.Begin();
    }

    private static BitmapImage CreateImageFromAssets(string imageFilename) {
        return new BitmapImage(new Uri("ms-appx:///Assets/" + imageFilename));
    }
}
```

BitmapImage находится в пространстве имен Media.Imaging, а Storyboard и остальные отвечающие за анимацию классы попали в пространство имен Media.Animation.

Для создания экземпляра элемента управления сраствами XAML у элемента должен быть не имеющий параметров конструктор. Добавление перекрытых конструкторов помогает только при написании кода создания элемента.

Статические методы `SetTarget()` и `SetTargetProperty()` класса `Storyboard` задают анимируемый объект (`"image"`) и свойство, которое будет изменяться (`"Source"`).

После настройки объекта `Storyboard` и добавления анимации в его коллекцию `Children` вызовите его метод `Begin()` для запуска анимации.

Пусть пчелы летают по странице

Для тестового полета используем элемент управления AnimatedImage.

1 **Замените MainPage.xaml шаблоном базовой страницы.**
Добавьте **в папку *View* шаблон Basic Page** с именем *FlyingBees.xaml*. Удалите из проекта страницу *MainPage.xaml*. В файле *App.xaml.cs* укажите новую начальную страницу:

```
if (!rootFrame.Navigate(typeof(View.FlyingBees), args.Arguments))
```

2 **Пчелы будут летать по элементу Canvas.**
Присвойте статическому ресурсу `AppName` имя `Flying  Bees`. Добавьте к тегу `<common:LayoutAwarePage>` новой страницы свойство `xmlns`, чтобы обеспечить доступ к пространству имен `View`:

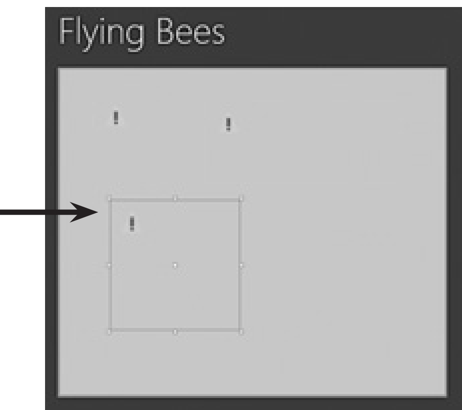
```
xmlns:view="using:AnimatedBee.View"
```

Если проект не компилируется, выгрузите и снова загрузите его. Если вы дали проекту другое имя, убедитесь, что вместо AnimatedBee указано корректное пространство имен.

Добавьте в файл *FlyingBees.xaml* элемент управления Canvas — контейнер, в который можно вставлять элементы, например Grid или StackPanel. Он позволяет задавать координаты вставленных элементов через `Canvas.Left` и `Canvas.Top`. Именно он стал основой игрового поля в проекте *Save the Humans*. Вот код XAML, который нужно вставить в файл *FlyingBees.xaml*:

```
<Canvas Grid.Row="1" Background="SkyBlue" Width="600"
        HorizontalAlignment="Left" Margin="120,0,120,120">
    <view:AnimatedImage Canvas.Left="55" Canvas.Top="40"
                        x:Name="firstBee" Width="50" Height="50"/>
    <view:AnimatedImage Canvas.Left="80" Canvas.Top="260"
                        x:Name="secondBee" Width="200" Height="200"/>
    <view:AnimatedImage Canvas.Left="230" Canvas.Top="100"
                        x:Name="thirdBee" Width="300" Height="125"/>
</Canvas>
```

Элемент управления AnimatedImage невидим, пока не вызван метод `CreateFrameImages()`, поэтому вставленные в контейнер Canvas элементы отображаются в виде контуров. Для их выделения пользуйтесь окном Document Outline. Подвигайте элемент по холсту, чтобы посмотреть, как меняются свойства `Canvas.Left` и `Canvas.Top`.



3 Добавим программный код страницы.

`using` требуется для пространства имен с объектами `Storyboard` и `DoubleAnimation`:

```
using Windows.UI.Xaml.Media.Animation;
```

Теперь **отредактируем конструктор в файле *FlyingBees.xaml.cs***, чтобы запустить анимацию пчел. Создадим объект `DoubleAnimation` для анимации свойства `Canvas.Left`. Сравните этот код построения раскадровки и анимации с содержащим тег `<DoubleAnimation>` кодом XAML, который мы писали раньше.

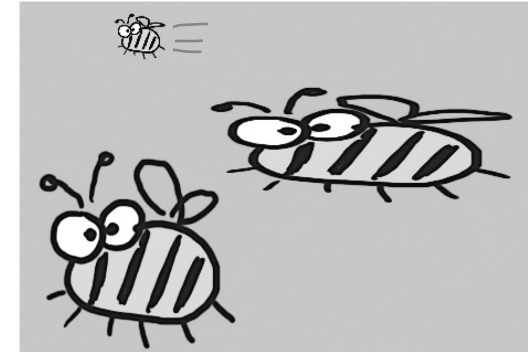
```
public FlyingBees() {
    this.InitializeComponent();

    List<string> imageNames = new List<string>();
    imageNames.Add("Bee animation 1.png");
    imageNames.Add("Bee animation 2.png");
    imageNames.Add("Bee animation 3.png");
    imageNames.Add("Bee animation 4.png");

    firstBee.StartAnimation(imageNames, TimeSpan.FromMilliseconds(50));
    secondBee.StartAnimation(imageNames, TimeSpan.FromMilliseconds(10));
    thirdBee.StartAnimation(imageNames, TimeSpan.FromMilliseconds(100));

    Storyboard storyboard = new Storyboard();
    DoubleAnimation animation = new DoubleAnimation();
    Storyboard.SetTarget(animation, firstBee);
    Storyboard.SetTargetProperty(animation, "(Canvas.Left)");
    animation.From = 50;
    animation.To = 450;
    animation.Duration = TimeSpan.FromSeconds(3);
    animation.RepeatBehavior = RepeatBehavior.Forever;
    animation.AutoReverse = true;
    storyboard.Children.Add(animation);
    storyboard.Begin();
}
```

Метод CreateFrameImages() берет в качестве параметров набор имен из ресурсов и TimeSpan и задает частоту обновления кадров.

Вместо создания тегов <Storyboard> и <DoubleAnimation> создаются объекты Storyboard и DoubleAnimation и в коде задаются их свойства.

После завершения анимации объект `Storyboard` удаляется сборщиком мусора. В этом можно убедиться с помощью функции Make Object ID и щелчка на кнопке для обновления страницы.

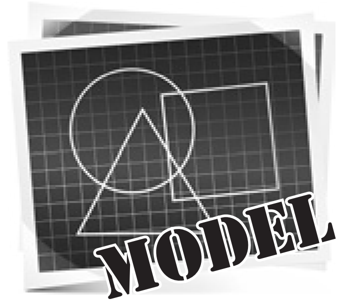
Запустите программу. Теперь крылья пчел двигаются. Мы взяли разные интервалы, поэтому пчелы машут крыльями с разной частотой, ведь их таймеры меняют кадры через разные промежутки времени. Свойство Canvas.Left верхней пчелы анимировано от 50 до 450 и обратно, что заставляет ее двигаться по холсту. Внимательно посмотрите на свойства объекта DoubleAnimation и сравните их со свойствами XAML, которые мы использовали ранее.



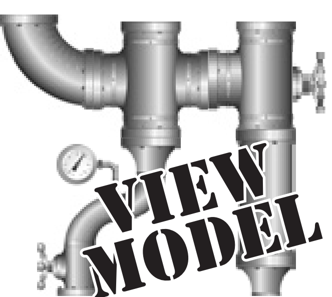
Что-то в этом проекте не так. Видите?

Кое-что не так: ничего нет в папке *Model* или *ViewModel*, а вы создаете фиктивные данные в слое View. Это не MVVM!

Чтобы получить больше пчел, нужно создать дополнительные элементы управления в классе View и инициализировать их по отдельности. А что, если нам нужны пчелы разных размеров и видов? Или требуется еще что-нибудь анимировать? При наличии оптимизированного под данные слоя Model эти задачи решались бы намного проще. Как выстроить проект в соответствии с шаблоном MVVM?



???



Очень легко. Добавьте коллекцию элементов управления ObservableCollection и свяжите с ней свойство Children элемента Canvas. Почему это кажется вам таким сложным?

Не получится. Связывание данных не работает со свойством `Children` контейнеров, и это не случайно.

Связывание данных предназначено для работы с **прикрепляемыми свойствами**, то есть со свойствами, которые появляются в коде XAML. Объект `Canvas` *не* имеет открытого свойства `Children`, и если попытаться задать связывание через XAML (`Children="{Binding ...}"`), код ***компилироваться не будет.***

Но вы уже умеете связывать коллекции объектов с элементами управления XAML, так как вы проделывали эту операцию для элементов ListView и GridView через свойство `ItemsSource`. Воспользуемся преимуществом такого связывания и добавим к элементу Canvas дочерние элементы управления.

Связывание через ItemsPanelTemplate

При связывании с элементами ListView, GridView или ListBox через свойство`ItemsSource` не имело значения, с чем именно осуществляется связывание, так как это свойство всегда работает одинаково. Для построения трех классов с одинаковым поведением нужно поместить это поведение в базовый класс и превратить три класса в его расширения, не так ли? Именно так поступила команда Microsoft, создав элементы выбора. ListView, GridView и ListBox, все они расширяют класс `Selector`, который является подклассом **класса `ItemsControl`, отображающего коллекции**.

1. Воспользуемся свойством `ItemsPanel` и **настроим шаблон панели, управляющей компоновкой элементов**. Начнем с добавления пространства имен `ViewModel` в файл *FlyingBees.xaml*:

```
xmlns:viewmodel="using:AnimatedBee.ViewModel"
```

Если ваш проект называется по-другому, замените AnimatedBee на корректное пространство имен.

2. **Добавьте *пустой* класс `BeeViewModel` в папку *ViewModel***, а экземпляр этого класса как статический ресурс вставьте в файл *FlyingBees.xaml*:

```
<viewmodel:BeeViewModel x:Key="viewModel"/>
```

В файле *FlyingBees.xaml.cs* **удалите весь код, добавленный в конструктор `FlyingBees()`** в элементе управления `FlyingBees`. Но вы ***не должны*** удалять метод `InitializeComponents()`!

3. Вот код XAML класса ItemsControl. Откройте файл *FlyingBees.xaml*, **удалите добавленный тег <Canvas>** и **и замените его классом ItemsControl**:

Этот статический ресурс ViewModel используется в качестве контекста данных и связывает ItemsSource со свойством Sprites.

```
<ItemsControl DataContext="{StaticResource viewModel}"
    ItemsSource="{Binding Path=Sprites}"
    Grid.Row="1" Margin="120,0,120,120">
    <ItemsControl.ItemsPanel>
        <ItemsPanelTemplate>
            <Canvas Background="SkyBlue" />
        </ItemsPanelTemplate>
    </ItemsControl.ItemsPanel>
</ItemsControl>
```

Вы можете настроить панель по своему вкусу. Мы взяли элемент Canvas с небесно-голубым фоном.

Для настройки `ItemsPanelTemplate` используйте свойство `ItemsPanel`. Оно содержит один элемент Panel, а элементы Grid и Canvas расширяют класс `Panel`. Любой связанный с `ItemsSource` элемент будет добавлен в коллекцию `Children` класса Panel.

Создание класса ItemsControl сопровождается созданием класса Panel, который содержит все элементы и использует `ItemsPanelTemplate` как шаблон элементов управления.

Шаблон «фабричный метод»

MVVM — всего лишь один из многих шаблонов проектирования. Один из наиболее распространенных и наиболее используемых шаблонов — «фабричный метод», обладает методом создания объектов. Это статический метод, имя которого заканчивается словом «Factory», что однозначно указывает на его назначение.

4 Создайте в **в папке *View* новый класс `BeeHelper`**. Он должен быть статическим, ведь чтобы помочь ViewModel управлять пчелами, требуются статические методы.

```
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media.Animation;

static class BeeHelper {
    public static AnimatedImage BeeFactory(
                double width, double height, TimeSpan flapInterval) {
        List<string> imageNames = new List<string>();
        imageNames.Add("Bee animation 1.png");
        imageNames.Add("Bee animation 2.png");
        imageNames.Add("Bee animation 3.png");
        imageNames.Add("Bee animation 4.png");

        AnimatedImage bee = new AnimatedImage(imageNames, flapInterval);
        bee.Width = width;
        bee.Height = height;
        return bee;
    }

    public static void SetBeeLocation(AnimatedImage bee, double x, double y) {
        Canvas.SetLeft(bee, x);
        Canvas.SetTop(bee, y);
    }

    public static void MakeBeeMove(AnimatedImage bee,
                                   double fromX, double toX, double y) {
        Canvas.SetTop(bee, y);
        Storyboard storyboard = new Storyboard();
        DoubleAnimation animation = new DoubleAnimation();
        Storyboard.SetTarget(animation, bee);
        Storyboard.SetTargetProperty(animation, "(Canvas.Left)");
        animation.From = fromX;
        animation.To = toX;
        animation.Duration = TimeSpan.FromSeconds(3);
        animation.RepeatBehavior = RepeatBehavior.Forever;
        animation.AutoReverse = true;
        storyboard.Children.Add(animation);
        storyboard.Begin();
    }
}
```

Этот фабричный метод создает элементы управления Bee. Поместите его в слой View, так как его код связан с UI.

Небольшой постоянно используемый блок кода, который вы помещаете в свой (зачастую статический) метод, иногда называют вспомогательным методом. Поместив такой метод в статический класс и добавив «Helper» в конце его имени, вы упростите чтение своего кода.

Это тот же самый код, который присутствовал в конструкторе страницы. Теперь он помещен в статический вспомогательный метод.

Все элементы XAML наследуют от базового класса `UIElement` в пространстве имен `Windows.UI.Xaml`. Мы в явном виде пишем `Windows.UI.Xaml.UIElement` в теле класса, а не добавляем строку `using`, чтобы ограничить количество добавленного в класс ViewModel кода UI.

Мы используем `UIElement` как наиболее абстрактный класс, расширяемый всеми спрайтами. Для некоторых проектов больше подходит подкласс `FrameworkElement`, в котором определяются многие свойства, в том числе `Width`, `Height`, `Opacity`, `HorizontalAlignment` и пр.

5 Вот код для добавленного в папку *ViewModel* пустого класса `BeeViewModel`. Поместив весь связанный с UI код в слой View, мы оставим в слое ViewModel простой и связанный только с управлением поведением пчел код.

При добавлении элемента управления AnimatedImage к связанной со свойством ItemsSource элемента ItemsControl коллекции _sprites класса ObservableCollection этот элемент добавляется также к панели, созданной на основе класса ItemsPanelTemplate.

```
using View;
using System.Collections.ObjectModel;
using System.Collections.Specialized;

class BeeViewModel {
    private readonly ObservableCollection<Windows.UI.Xaml.UIElement>
            _sprites = new ObservableCollection<Windows.UI.Xaml.UIElement>();
    public INotifyCollectionChanged Sprites { get { return _sprites; } }

    public BeeViewModel() {
        AnimatedImage firstBee =
             BeeHelper.BeeFactory(50, 50,
                            TimeSpan.FromMilliseconds(50));
        _sprites.Add(firstBee);

        AnimatedImage secondBee =
             BeeHelper.BeeFactory(200, 200, TimeSpan.FromMilliseconds(10));
        _sprites.Add(secondBee);

        AnimatedImage thirdBee =
             BeeHelper.BeeFactory(300, 125, TimeSpan.FromMilliseconds(100));
        _sprites.Add(thirdBee);

        BeeHelper.MakeBeeMove(firstBee, 50, 450, 40);
        BeeHelper.SetBeeLocation(secondBee, 80, 260);
        BeeHelper.SetBeeLocation(thirdBee, 230, 100);
    }
```

Для инкапсуляции свойства Sprites предпримем два шага. Пометим вспомогательное поле как поле только для чтения, чтобы его было невозможно переопределить, и превратим его в свойство INotifyCollectionChanged, чтобы другие классы могли его видеть, но не редактировать.

Спрайтом называется любая картинка или анимация, вставляемая в большую игру или анимацию.

Вы изменили свойства элементов управления и добавили анимацию после их добавления в `ObservableCollection`. Почему это возможно?

6 Запустите приложение. Оно выглядит как раньше, но теперь поведения распределены по слоям. Связанный с UI код перенесен во View, а код, относящийся к пчелам и движению, — во ViewModel.

Ключевое слово readonly

Мы используем инкапсуляцию, чтобы не дать одному классу переопределить данные другого класса. А как помешать классу переопределять собственные данные? Нам поможет ключевое слово **readonly**. Такое поле можно отредактировать только в его объявлении или конструкторе.

Это последнее упражнение в книге. Вам нужно создать программу, анимирующую пчел и звезды. Потребуется написать большой код, но вы вполне можете это сделать... и после этого у вас будут все инструменты для создания видеоигры.

1. Вот приложение, которое нужно создать.

Пчелы с двигающимися крыльями летают по темно-синему фону, а звезды меняют свою яркость. Вы построите слой View, содержащий пчел, звезды и страницу для их отображения, слой Model, фиксирующий положение и вызывающий события при перемещении пчел и изменении звезд, и слой ViewModel, соединяющий все это в единую программу.

2. Создадим новый проект для магазина Windows.

Создайте проект StarryNight. Добавьте **папки *Model*, *View* и *ViewModel*.** После этого нужно будет создать **пустой класс `BeeStarViewModel`** в папке *ViewModel*.

3. Добавим в папку View новый шаблон Basic Page.

Поместите в **папку *View* шаблон Basic Page с именем** *BeesOnAStarryNight.xaml*. Добавьте к верхнему тегу в файле *BeesOnAStarryNight.xaml* пространство имен (оно должно совпасть с именем проекта `StarryNight`):

```
xmlns:viewmodel="using:StarryNight.ViewModel"
```

Добавьте ViewModel как статический ресурс и поменяйте имя страницы:

```
<Page.Resources>
    <viewmodel:BeeStarViewModel x:Name="viewModel"/>
    <x:String x:Key="AppName">Bees on a Starry Night</x:String>
</Page.Resources>
```

Код XAML страницы **полностью совпадает** с кодом *FlyingBees.xaml* в последнем проекте, но фон элемента Canvas **`Blue`**, а сам элемент имеет обработчик события `SizeChanged`:

```
<Canvas Background="Blue" SizeChanged="SizeChangedHandler" />
```

Событие `SizeChanged` возникает при изменении размеров элемента управления, а свойства `EventArgs` испльзуются для нового размера.

Visual Studio — фантастический инструмент для экспериментов с формами! Запустите Blend для Visual Studio 2012 и при помощи ручки, карандаша и панели инструментов создавайте формы XAML для вставки в проекты C#.

Код из шага 4 не будет компилироваться, пока вы не добавите классу ViewModel свойство PlayAreaSize на шаге 9. А сейчас можно воспользоваться IDE и сгенерировать заглушку свойства.

4 **Добавим программный код страницы и приложения.**

Добавьте `SizeChanged` в файл *BeesOnAStarryNight.xaml.cs* в папке *View*:

```
private void SizeChangedHandler(object sender, SizeChangedEventArgs e) {
    viewModel.PlayAreaSize = new Size(e.NewSize.Width, e.NewSize.Height);
}
```

Отредактируйте файл *App.xaml.cs*, изменив вызов метода `rootFrame.Navigate()` таким образом, чтобы приложение запускалось на новой странице:

```
if (!rootFrame.Navigate(typeof(View.BeesOnAStarryNight), args.Arguments))
```

5 **Добавим в папку View элемент управления AnimatedImage.**

Вернитесь к папке *View* и добавьте элемент AnimatedImage. Он уже встречался вам в этой главе. Обязательно **добавьте графические файлы** с кадрами анимации в папку *Assets*.

6 **Добавим в папку View пользовательский элемент управления StarControl.**

Этот элемент рисует звезду. Он обладает двумя раскадровками: одна для увеличения, а вторая для уменьшения яркости. Для запуска раскадровок **добавьте в программный код методы `FadeIn()` и `FadeOut()`**.

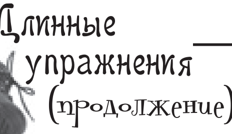
Элемент Polygon рисует многоугольник из набора точек. Наш UserControl использует его для рисования звезды.

```
<UserControl
    // Подойдет обычный, сгенерированный IDE код XAML,
    // в дополнительных пространствах имен User Control не нуждается
    >

    <UserControl.Resources>
        <Storyboard x:Name="fadeInStoryboard">
            <DoubleAnimation From="0" To="1" Storyboard.TargetName="starPolygon"
                       Storyboard.TargetProperty="Opacity" Duration="0:0:1.5" />
        </Storyboard>
        <Storyboard x:Name="fadeOutStoryboard">
           <DoubleAnimation From="1" To="0" Storyboard.TargetName="starPolygon"
                       Storyboard.TargetProperty="Opacity" Duration="0:0:1.5" />
        </Storyboard>
    </UserControl.Resources>

    <Grid>
        <Polygon Points="0,75 75,0 100,100 0,25 150,25" Fill="Snow"
                 Stroke="Black" x:Name="starPolygon"/>
    </Grid>
</UserControl>
```

Нужно добавить открытые методы FadeIn() и FadeOut() в запускающий эти раскадровки программный код. После этого яркость звезд начнет меняться.

Это звезда. В качестве эксперимента можно заменить ее на другую форму.

Кроме эллипсов, прямоугольников и многоугольников существуют и другие формы: http://msdn.microsoft.com/ru-ru/library/windows/apps/xaml/hh465055.aspx



Добавим класс BeeStarHelper в папку View.
Вот полезный вспомогательный класс. Он содержит уже знакомые вам инструменты и парочку новых. Поместите его в папку *View*.

```
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media.Animation;
using Windows.UI.Xaml.Shapes;

static class BeeStarHelper {
    public static AnimatedImage BeeFactory(double width, double height, TimeSpan flapInterval) {
        List<string> imageNames = new List<string>();
        imageNames.Add("Bee animation 1.png");
        imageNames.Add("Bee animation 2.png");
        imageNames.Add("Bee animation 3.png");
        imageNames.Add("Bee animation 4.png");

        AnimatedImage bee = new AnimatedImage(imageNames, flapInterval);
        bee.Width = width;
        bee.Height = height;
        return bee;
    }

    public static void SetCanvasLocation(UIElement control, double x, double y) {
        Canvas.SetLeft(control, x);
        Canvas.SetTop(control, y);
    }

    public static void MoveElementOnCanvas(UIElement uiElement, double toX, double toY) {
        double fromX = Canvas.GetLeft(uiElement);
        double fromY = Canvas.GetTop(uiElement);

        Storyboard storyboard = new Storyboard();
        DoubleAnimation animationX = CreateDoubleAnimation(uiElement,
                                        fromX, toX, "(Canvas.Left)");
        DoubleAnimation animationY = CreateDoubleAnimation(uiElement,
                                        fromY, toY, "(Canvas.Top)");
        storyboard.Children.Add(animationX);
        storyboard.Children.Add(animationY);
        storyboard.Begin();
    }

    public static DoubleAnimation CreateDoubleAnimation(UIElement uiElement,
                                    double from, double to, string propertyToAnimate) {
        DoubleAnimation animation = new DoubleAnimation();
        Storyboard.SetTarget(animation, uiElement);
        Storyboard.SetTargetProperty(animation, propertyToAnimate);
        animation.From = from;
        animation.To = to;
        animation.Duration = TimeSpan.FromSeconds(3);
        return animation;
    }

    public static void SendToBack(StarControl newStar) {
        Canvas.SetZIndex(newStar, -1000);
    }
}
```

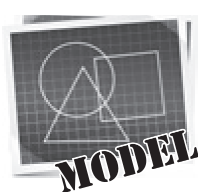
Контейнер Canvas обладает методами `SetLeft()` и `GetLeft()` для задания и считывания X-координаты элемента управления. Методы `SetTop()` и `GetTop()` задают и считывают Y-координату. Они работают и после добавления элемента на холст.

Мы добавили вспомогательный метод CreateDoubleAnimation(), создающий трехсекундный объект DoubleAnimation. Метод пользуется этим объектом для смещения элемента UI из текущего положения в новую точку через анимацию свойств Canvas.Left и Canvas.Top.

«Z Index» соответствует порядку слоев с элементами управления на панели. Элемент с более высоким индексом Z будет располагаться поверх элемента с низким индексом.

Добавим в папку Model классы Bee, Star и EventArgs.

Ваша модель должна фиксировать положение и размеры пчел, а также положения звезд и вызывать события, чтобы класс ViewModel знал об изменениях в состоянии звезд и пчел.




```
using Windows.Foundation;
class Bee {
    public Point Location { get; set; }
    public Size Size { get; set; }
    public Rect Position { get { return new Rect(Location, Size); } }
    public double Width { get { return Position.Width; } }
    public double Height { get { return Position.Height; } }

    public Bee(Point location, Size size) {
        Location = location;
        Size = size;
    }
}
```

```
using Windows.Foundation;
class Star {
    public Point Location {
        get; set;
    }

    public Star(Point location) {
        Location = location;
    }
}
```

Как только программа начнет работать, попробуйте добавить свойство Rotating к классу Star и с его помощью заставьте часть звезд медленно вращаться вокруг своей оси.

```
using Windows.Foundation;
class BeeMovedEventArgs : EventArgs {
    public Bee BeeThatMoved { get; private set; }
    public double X { get; private set; }
    public double Y { get; private set; }

    public BeeMovedEventArgs(Bee beeThatMoved, double x, double y) {
        BeeThatMoved = beeThatMoved;
        X = x;
        Y = y;
    }
}
```

Класс model будет вызывать использующие эти EventArgs события, сообщая классу ViewModel о возникающих изменениях.

```
using Windows.Foundation;
class StarChangedEventArgs : EventArgs {
    public Star StarThatChanged { get; private set; }
    public bool Removed { get; private set; }

    public StarChangedEventArgs(Star starThatChanged, bool removed) {
        StarThatChanged = starThatChanged;
        Removed = removed;
    }
}
```

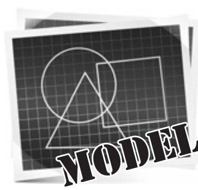
Структура Rect обладает набором перегруженных конструкторов и методов, позволяющих получать данные о высоте, ширине, размере и положении (или как объект Point, или в виде отдельных координат X и Y типа double).

Свойство Points элемента Polygon представляет собой коллекцию структур Point.

Структуры Point, Size и Rect

В пространстве имен Windows.Foundation находится несколько полезных структур. Структура Point использует свойства X и Y типа double для хранения набора координат. Size обладает двумя свойствами этого типа, Width и Height, а также специальным значением Empty. Rect хранит координаты верхнего левого и нижнего правого углов прямоугольника и обладает методами поиска ширины, высоты, пересечения с другими структурами Rect и пр.

Добавим в папку Model класс BeeStarModel.
Мы создали закрытые поля и пару полезных методов. Вам нужно закончить построение класса BeeStarModel.




```
using Windows.Foundation;

class BeeStarModel {
    public static readonly Size StarSize = new Size(150, 100);

    private readonly Dictionary<Bee, Point> _bees = new Dictionary<Bee, Point>();
    private readonly Dictionary<Star, Point> _stars = new Dictionary<Star, Point>();
    private Random _random = new Random();

    public BeeStarModel() {
        _playAreaSize = Size.Empty;
    }

    public void Update() {
        MoveOneBee();
        AddOrRemoveAStar();
    }

    private static bool RectsOverlap(Rect r1, Rect r2) {
        r1.Intersect(r2);
        if (r1.Width > 0 || r1.Height > 0)
            return true;
        return false;
    }

    public Size PlayAreaSize {
        // Добавьте вспомогательное поле и вызовите методом доступа CreateBees() и CreateStars()
    }
    private void CreateBees() {
        // Возврат управления при пустом игровом поле. При наличии пчел подвигайте каждую.
        // Создайте от 5 до 15 пчел разного размера (от 40 до 150 пикселей),
        // добавьте в коллекцию _bees collection и вызовите событие BeeMoved.
    }
    private void CreateStars() {
        // Возврат управления при пустом игровом поле. При наличии звезд
        // задайте новое положение каждой и вызовите событие StarChanged
        // в противном случае вызовите CreateAStar() от 5 до 10 раз.
    }
    private void CreateAStar() {
        // Найдите неперекрывающееся место, добавьте новый объект Star в
        // _stars collection и вызовите событие StarChanged.
    }
    private Point FindNonOverlappingPoint(Size size) {
        // Найдите верхний левый угол прямоугольника, не пересекающего ни пчелу, ни звезду
        // Можно создать случайный Rect, затем через запросы LINQ найти пчелу или звезду,
        // которые пересекают его (здесь пригодится метод RectsOverlap()).
    }
    private void MoveOneBee(Bee bee = null) {
        // При отсутствии пчел возврат управления. Если bee равен null, выберите случайную пчелу,
        // иначе используйте аргумент bee. Найдите новую точку без перекрытия, обновите положение
        // пчелы, обновите _bees collection, затем вызовите событие OnBeeMoved.
    }
    private void AddOrRemoveAStar() {
        // Подбросьте монетку (_random.Next(2) == 0) и создайте звезду методом CreateAStar()
        // или удалите ее и вызовите OnStarChanged. Создавайте звезду при <= 5, а удаляйте,
        // если >= 20. _stars.Keys.ToList()[_random.Next(_stars.Count)] ищет случайную звезду.
    }
    // Нужно добавить события BeeMoved и StarChanged и методы их вызова.
    // Они используют классы BeeMovedEventArgs и StarChangedEventArgs.
}
```

Ключевое слово readonly позволяет создать структуру с постоянным значением.

Size.Empty — это значение Size, зарезервированное для пустого размера. Оно используется для создания пчел и звезд при изменении размеров игрового поля.

Класс ViewModel для периодического вызова метода Update() будет использовать таймер.

Этот метод проверяет две структуры Rect и возвращает значение true, если они перекрываются при помощи метода Rect.Intersect().

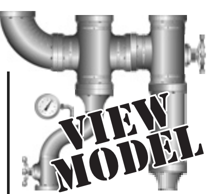
PlayAreaSize это свойство.

Если, проверив 1,000 случайных мест, метод не находит положения без перекрытия, значит, на игровом поле нет места, и следует вернуть управление.

Отладку приложения можно провести в симуляторе, чтобы убедиться в работоспособности при разных размерах и ориентации экрана.

Добавим класс BeeStarViewModel в папку ViewModel.

Заполните закоментированные методы. Нужно внимательно смотреть, как работает класс Model и чего ожидает класс View. Здесь пригодится вспомогательный метод.



Нужно гаратировать, что `DispatcherTimer` и `UIElement` — единственные классы из пространства имен `Windows.UI.Xaml`, используемые в классе ViewModel. Ключевое слово `using` позволяет использовать = для объявления одного члена в другом пространстве имен.

```
using View;
using Model;
using System.Collections.ObjectModel;
using System.Collections.Specialized;
using Windows.Foundation;
using DispatcherTimer = Windows.UI.Xaml.DispatcherTimer;
using UIElement = Windows.UI.Xaml.UIElement;

class BeeStarViewModel {
    private readonly ObservableCollection<UIElement>
                _sprites = new ObservableCollection<UIElement>();
    public INotifyCollectionChanged Sprites { get { return _sprites; } }

    private readonly Dictionary<Star, StarControl> _stars = new Dictionary<Star, StarControl>();
    private readonly List<StarControl> _fadedStars = new List<StarControl>();

    private BeeStarModel _model = new BeeStarModel();

    private readonly Dictionary<Bee, AnimatedImage> _bees = new Dictionary<Bee, AnimatedImage>();

    private DispatcherTimer _timer = new DispatcherTimer();

    public Size PlayAreaSize { /* возврат методов доступа и записи и _model.PlayAreaSize */ }

    public BeeStarViewModel() {
        // Свяжите обработчики с событиями BeeMoved и StarChanged класса BeeStarModel,
        // и запустите таймер на каждые две секунды.
    }
    void timer_Tick(object sender, object e) {
        // При каждом отсчете таймера ищите все ссылки на StarControl в коллекции
        // _fadedStars и удаляйте их из from _sprites, затем вызывайте метод Update()
        // класса BeeViewModel, чтобы обновить этот класс.
    }
    void BeeMovedHandler(object sender, BeeMovedEventArgs e) {
        // Словарь _bees сопоставляет объектам Bee в слое Model элементы управления AnimatedImage
        // в слое View. При смещении пчелы BeeViewModel вызывает свое событие BeeMoved, чтобы
        // сообщить, какая пчела куда сместилась. Если в словаре _bees отсутствует элемент
        // AnimatedImage для этой пчелы, его следует создать, задать положение холста
        // и обновить словари _bees и _sprites.
        // Если этот элемент присутствует в словаре _bees, его нужно найти и переместить
        // по холсту в новое положение с исопльзованием анимации.
    }
    void StarChangedHandler(object sender, StarChangedEventArgs e) {
       // Словарь _stars работает как _bees, сопоставляя объектам Star соответствующие
       // элементы управления StarControl. Объект EventArgs содержит ссылки на объект Star
       // (обладающий свойством Location) и логический параметр, сообщающий, была ли удалена
       // звезда. Удаленная звезда должна затухать, поэтому удалите ее из словаря _stars,
       // добавьте в _fadedStars и вызовите метод FadeOut() (из словаря _sprites ее удалят при
       // следующем вызове метода Update(), именно поэтому мы задали больший интервал
       // таймера, чем в анимации затухания элемента StarControl).
       //
       // Если звезда не удалена, проверьте ее наличие в словаре _stars - если она есть, получите
       // ссылку на StarControl; в противном случае нужно создать новый StarControl, осветить,
       // добавить в словарь _sprites и отправить на задний план, чтобы пчелы могли летать.
       // Затем нужно задать положение элемента StarControl на холсте.
    }
}
```

При задании нового положения на элементе Canvas элемент управления обновляется, даже если он присутствует в контейнере Canvas. Именно поэтому звезды смещаются при изменении игрового поля.

Решение длинных упражнений

Вот готовые методы класса BeeStarModel.

```
using Windows.Foundation;

class BeeStarModel {
    public static readonly Size StarSize = new Size(150, 100);

    private readonly Dictionary<Bee, Point> _bees = new Dictionary<Bee, Point>();
    private readonly Dictionary<Star, Point> _stars = new Dictionary<Star, Point>();
    private Random _random = new Random();

    public BeeStarModel() {
        _playAreaSize = Size.Empty;
    }

    public void Update() {
        MoveOneBee();
        AddOrRemoveAStar();
    }

    private static bool RectsOverlap(Rect r1, Rect r2) {
        r1.Intersect(r2);
        if (r1.Width > 0 || r1.Height > 0)
            return true;
        return false;
    }

    private Size _playAreaSize;
    public Size PlayAreaSize {
        get { return _playAreaSize; }
        set
        {
            _playAreaSize = value;
            CreateBees();
            CreateStars();
        }
    }

    private void CreateBees() {
        if (PlayAreaSize == Size.Empty) return;

        if (_bees.Count() > 0) {
            List<Bee> allBees = _bees.Keys.ToList();
            foreach (Bee bee in allBees)
                MoveOneBee(bee);
        } else {
            int beeCount = _random.Next(5, 10);
            for (int i = 0; i < beeCount; i++) {
                int s = _random.Next(50, 100);
                Size beeSize = new Size(s, s);
                Point newLocation = FindNonOverlappingPoint(beeSize);
                Bee newBee = new Bee(newLocation, beeSize);
                _bees[newBee] = new Point(newLocation.X, newLocation.Y);
                OnBeeMoved(newBee, newLocation.X, newLocation.Y);
            }
        }
    }
```

Это мы заполнили за вас.

Это методы программного кода StarControl:

```
public void FadeIn() {
    fadeInStoryboard.Begin();
}

public void FadeOut() {
    fadeOutStoryboard.Begin();
}
```

При любом изменении свойства PlayAreaSize класс Model обновляет вспомогательное поле _playAreaSize и после этого вызывает методы CreateBees() и CreateStars(). Это дает слою ViewModel возможность вызвать измнения слоя Model при изменении размера, возникающего, например, когда программа запущена на планшете и пользователь поменял его ориентацию.

Если пчелы уже есть, переместите каждую из них. Метод MoveOneBee() найдет для каждой пчелы положение без перекрытия и вызовет событие BeeMoved.

Если пчелы в слое model пока отсутствуют, здесь будут созданы новые объекты Bee и задано их местоположение. При каждом добавлении или изменении пчелы следует вызывать событие BeeMoved.

```
private void CreateStars() {
    if (PlayAreaSize == Size.Empty) return;

    if (_stars.Count > 0) {
        foreach (Star star in _stars.Keys) {
            star.Location = FindNonOverlappingPoint(StarSize);
            OnStarChanged(star, false);
        }
    } else {
        int starCount = _random.Next(5, 10);
        for (int i = 0; i < starCount; i++)
            CreateAStar();
    }
}

private void CreateAStar() {
    Point newLocation = FindNonOverlappingPoint(StarSize);
    Star newStar = new Star(newLocation);
    _stars[newStar] = new Point(newLocation.X, newLocation.Y);
    OnStarChanged(newStar, false);
}

private Point FindNonOverlappingPoint(Size size) {
    Rect newRect;
    bool noOverlap = false;
    int count = 0;
    while (!noOverlap) {
        newRect = new Rect(_random.Next((int)PlayAreaSize.Width - 150),
            _random.Next((int)PlayAreaSize.Height - 150),
            size.Width, size.Height);

        var overlappingBees =
            from bee in _bees.Keys
            where RectsOverlap(bee.Position, newRect)
            select bee;

        var overlappingStars =
            from star in _stars.Keys
            where RectsOverlap(
                new Rect(star.Location.X, star.Location.Y, StarSize.Width, StarSize.Height),
                newRect)
            select star;

        if ((overlappingBees.Count() + overlappingStars.Count() == 0) || (count++ > 1000))
            noOverlap = true;
    }
    return new Point(newRect.X, newRect.Y);
}

private void MoveOneBee(Bee bee = null) {
    if (_bees.Keys.Count() == 0) return;
    if (bee == null) {
        int beeCount = _stars.Count;
        List<Bee> bees = _bees.Keys.ToList();
        bee = bees[_random.Next(bees.Count)];
    }
    bee.Location = FindNonOverlappingPoint(bee.Size);
    _bees[bee] = bee.Location;
    OnBeeMoved(bee, bee.Location.X, bee.Location.Y);
}
```

Если звезды уже есть, мы меняем положение каждой из них в PlayArea и вызываем событие StarChanged. Обработка этого события и перемещение соответствующего элемента управления является делом слоя ViewModel.

Здесь создается случайный объект Rect и проверяется, пересекает ли он другие объекты. Справа добавлено поле 250 пикселей, а снизу — 150 пикселей, чтобы звезды и пчелы не выходили за границу игровой области.

Эти запросы LINQ вызывают метод RectsOverlap(), который ищет пересекающиеся с новым объектом Rect звезды и пчел. Если хотя бы одно из возвращаемых значений отлично от нуля, новый Rect пересекается с каким-то объектом.

Более 1000 итераций означает, что в игровой области не удается найти место, в котором объект не будет ни с чем пересекаться, и пора выходить из бесконечного цикла.

Последние несколько членов класса BeeStarModel.

Бросьте монетку, выбирая между 0 и 1, но всегда создавайте звезду при значениях менее 5 и удаляйте при значениях более 20.

```
    private void AddOrRemoveAStar() {
        if (((_random.Next(2) == 0) || (_stars.Count <= 5)) && (_stars.Count < 20 ))
            CreateAStar();
        else {
            Star starToRemove = _stars.Keys.ToList()[_random.Next(_stars.Count)];
            _stars.Remove(starToRemove);
            OnStarChanged(starToRemove, true);
        }
    }

    public event EventHandler<BeeMovedEventArgs> BeeMoved;

    private void OnBeeMoved(Bee beeThatMoved, double x, double y)
    {
        EventHandler<BeeMovedEventArgs> beeMoved = BeeMoved;
        if (beeMoved != null)
        {
            beeMoved(this, new BeeMovedEventArgs(beeThatMoved, x, y));
        }
    }

    public event EventHandler<StarChangedEventArgs> StarChanged;

    private void OnStarChanged(Star starThatChanged, bool removed)
    {
        EventHandler<StarChangedEventArgs> starChanged = StarChanged;
        if (starChanged != null)
        {
            starChanged(this, new StarChangedEventArgs(starThatChanged, removed));
        }
    }
}
```

Каждый вызов метода Update() означает, что мы хотим добавить или удалить звезду. CreateAStar() создает звезды. При удалении звезда убирается из коллекции _stars и вызывается событие StarChanged.

Это обычные обработчики событий и методы их вызова.

Это готовые методы класса BeeStarViewModel.

```
using View;
using Model;
using System.Collections.ObjectModel;
using System.Collections.Specialized;
using Windows.Foundation;
using DispatcherTimer = Windows.UI.Xaml.DispatcherTimer;
using UIElement = Windows.UI.Xaml.UIElement;

class BeeStarViewModel {
    private readonly ObservableCollection<UIElement>
                _sprites = new ObservableCollection<UIElement>();
    public INotifyCollectionChanged Sprites { get { return _sprites; } }

    private readonly Dictionary<Star, StarControl> _stars = new Dictionary<Star, StarControl>();
    private readonly List<StarControl> _fadedStars = new List<StarControl>();

    private BeeStarModel _model = new BeeStarModel();

    private readonly Dictionary<Bee, AnimatedImage> _bees
                                          = new Dictionary<Bee, AnimatedImage>();

    private DispatcherTimer _timer = new DispatcherTimer();
```

Этот код мы написали за вас.

Если вы хорошо поработали над разделением ответственности, готовый проект естественным образом будет состоять из слабо связанных частей.

Свойство `PlayAreaSize` класса ViewModel является простым переходом к свойству класса Model, а здесь метод доступа `PlayAreaSize` вызывает методы, которые становятся причиной событий `BeeMoved` и `StarChanged`. И при изменении разрешения экрана (1) элемент Canvas вызывает свое событие `SizeChanged`, которое (2) обновляет свойство `PlayAreaSize` класса ViewModel, которое (3) обновляет свойство класса Model, которое (4) вызывает методы обновления пчел и звезд, которые (5) вызывают события `BeeMoved` и `StarChanged`, которые (6) запускают обработчики событий в ViewModel, которые (7) обновляют коллекцию `Sprites`, которая (8) обновляет элементы в Canvas. Это пример слабого связывания, с отсутствием центрального координирующего объекта. Это надежный способ построения программ, так как каждый объект может обойтись без информации о способе работы других объектов. Он выполняет только свою работу: обрабатывает, вызывает событие, вызывает метод, задает свойство и пр.

```
    public Size PlayAreaSize {
        get { return _model.PlayAreaSize; }
        set { _model.PlayAreaSize = value; }
    }

    public BeeStarViewModel() {
        _model.BeeMoved += BeeMovedHandler;
        _model.StarChanged += StarChangedHandler;

        _timer.Interval = TimeSpan.FromSeconds(2);
        _timer.Tick += timer_Tick;
        _timer.Start();
    }

    void timer_Tick(object sender, object e) {
        foreach (StarControl starControl in _fadedStars)
            _sprites.Remove(starControl);

        _model.Update();
    }

    void BeeMovedHandler(object sender, BeeMovedEventArgs e) {
        if (!_bees.ContainsKey(e.BeeThatMoved)) {
            AnimatedImage beeControl = BeeStarHelper.BeeFactory(
                        e.BeeThatMoved.Width, e.BeeThatMoved.Height, TimeSpan.FromMilliseconds(20));
            BeeStarHelper.SetCanvasLocation(beeControl, e.X, e.Y);
            _bees[e.BeeThatMoved] = beeControl;
            _sprites.Add(beeControl);
        } else {
            AnimatedImage beeControl = _bees[e.BeeThatMoved];
            BeeStarHelper.MoveElementOnCanvas(beeControl, e.X, e.Y);
        }
    }

    void StarChangedHandler(object sender, StarChangedEventArgs e) {
        if (e.Removed) {
            StarControl starControl = _stars[e.StarThatChanged];
            _stars.Remove(e.StarThatChanged);
            _fadedStars.Add(starControl);
            starControl.FadeOut();
        } else {
            StarControl newStar;
            if (_stars.ContainsKey(e.StarThatChanged))
                newStar = _stars[e.StarThatChanged];
            else {
                newStar = new StarControl();
                _stars[e.StarThatChanged] = newStar;
                newStar.FadeIn();
                BeeStarHelper.SendToBack(newStar);
                _sprites.Add(newStar);
            }
            BeeStarHelper.SetCanvasLocation(
                        newStar, e.StarThatChanged.Location.X, e.StarThatChanged.Location.Y);
        }
    }
}
```

В коллекцию _fadedStars входят элементы управления, яркость которых в данный момент падает и которые при следующем вызове метода Update() класса ViewModel будут удалены.

После добавления звезды следует вызвать ее метод FadeIn(). Уже существующие звезды перемещаются из-за изменения размера игрового поля. В любом случае мы хотим установить их в новое положение на элементе Canvas.

Наши поздравления! (Но нам еще рано почивать на лаврах...)

Вы закончили последнее упражнение? Поняли все, что происходило в коде? Если да, **наши поздравления**: вы узнали о C# много и, вероятно, быстрее, чем ожидали! Вас ждет мир программирования.

Осталось несколько вещей, которые хорошо бы проделать, если вы хотите убедиться, что все сведения, которые вы попытались запихнуть в свой мозг, все еще там.

Еще раз вернитесь к проекту Save the Humans.

Если вы следовали нашим советам, то проект *Save the Humans* был построен дважды: один раз в начале книги и второй — перед главой 10. Но даже во втором случае оставались загадочные аспекты, создающие впечатление магии. Но в программировании **магии нет**. Поэтому еще раз внимательно посмотрите на написанный вами код. Вы удивитесь, сколько всего вы поймете! А ничто не закрепляет уроки лучше, чем положительное подкрепление.

Обсудите это с друзьями.

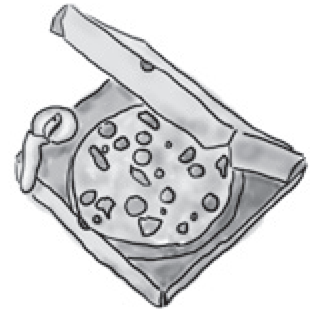
Человек — социальное животное; и рассказывая другим о том, что изучили, вы это лучше запоминаете. В наши дни «обсуждение» подразумевает и общение в социальных сетях! Кроме того, вы закончили несколько проектов. Не стесняйтесь **хвастаться этим**!

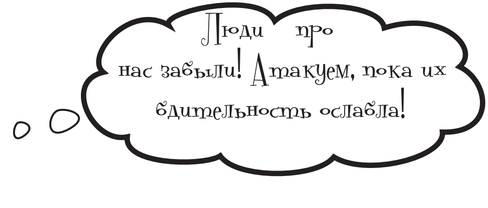
Программирование — это не магия. Программы работают потому, что они так построены, а код понятен.

...но понять код намного проще, если программист пользовался хорошими шаблонами проектирования и придерживался принципов объектно-ориентированного программирования.

Сделайте перерыв. А лучше поспите.

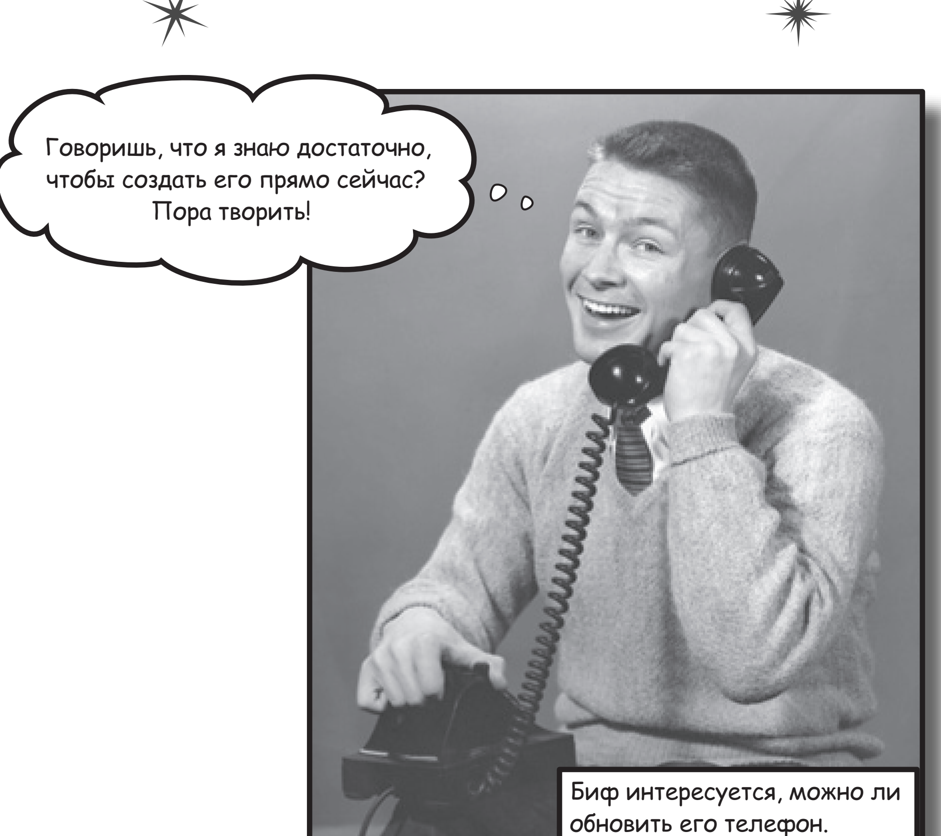
Ваш мозг получил много информации, и иногда лучшее, что можно сделать для «фиксации» нового знания, — это отдохнуть. Исследования показывают, что усвоение материала значительно улучшается **после хорошего ночного сна**. Дайте своему мозгу заслуженный отдых!





17 бонусный проект!

Приложение Windows Phone



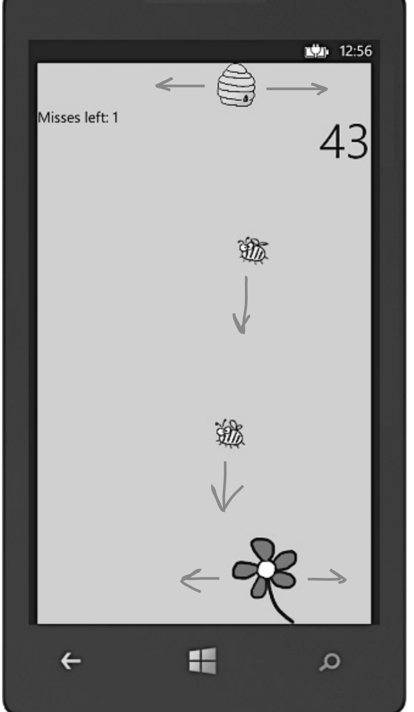
Вы уже можете создавать приложения Windows Phone.
Классы, объекты, XAML, инкапсуляция, наследование, полиморфизм, LINQ, MVVM… у вас есть все, что нужно превосходным приложениям для магазина Windows и для рабочего стола. Но знали ли вы, что эти же инструменты позволяют создавать приложения Windows Phone? Это действительно так! И наш бонусный проект посвящен созданию игры для Windows Phone. Не волнуйтесь, если у вас отсутствует Windows Phone, — вам поможет эмулятор. Так что начинаем!

Пчелы атакуют!

Вы создадите для Windows Phone 8 игру **Пчелы атакуют**. Имеется улей сильно рассерженных пчел, успокоить которых может только прекрасный цветок. Чем больше пчел вы поймаете на цветок, тем выше будет ваш счет.

Ловите пчел, летящих по экрану вниз.

Улей двигается в верхней части экрана, и из него вылетают пчелы. Чем больше пчел вы поймаете, тем чаще они будут вылетать из улья, который начнет активнее перемещаться вправо и влево.



Управление через касание.

Вы должны двигать цветок влево и вправо. Он будет следовать за вашим пальцем, позволяя ловить летящих вниз пчел. После пяти промахов игра заканчивается, а по мере увеличения количества пойманных пчел игра становится сложнее.

Перед тем как начать...

Для этого проекта вам потребуется **установить Visual Studio 2012 для Windows Phone**. Бесплатную версию Express можно скачать на сайте:

http://www.microsoft.com/visualstudio/eng/products/visual-studio-express-for-windows-phone

Существуют два варианта запуска игры. Проще всего использовать **Windows Phone Emulator**, который входит в пакет Visual Studio.

Запуск игры в эмуляторе.

По умолчанию Visual Studio для Windows Phone запускает приложения в эмуляторе, воспроизводящем Windows Phone 8 (с выходом в Internet, фальшивой сетью GSM и пр.)

▶ Emulator WVGA 512MB ▾

Кнопка Run в IDE запускает эмулятор.

Эмулятор позволяет запускать приложения Windows Phone на вашем компьютере.

Windows Phone Emulator требует Hyper-V

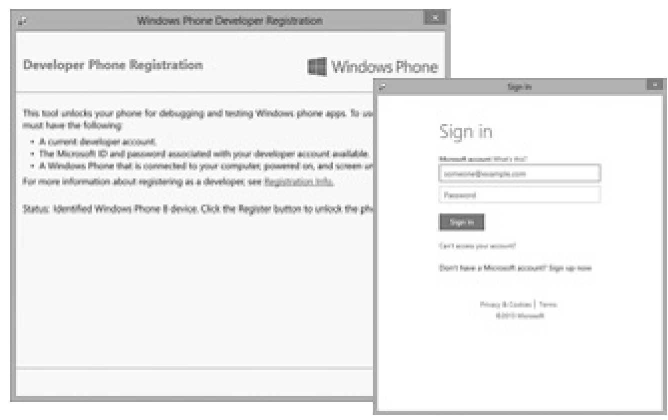
Hyper-V — это технология виртуализации для Windows 8, но запускается она далеко не на любом процессоре и не в любой версии Windows 8. Руководство по запуску и настройке:

http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj863509.aspx

*Hyper-V **работает** на VMWare, VirtualBox, Parallels и других поддерживаемых процессором средствах виртуализации. Подробности можно узнать в документации к VM (могут потребоваться функции «nested virtual machines», «virtualize VT-x или AMD-V»).*

Запуск игры на Windows Phone.

Присоединив Windows Phone 8 к компьютеру через кабель USB, вы получите возможность отладить приложение на устройстве.

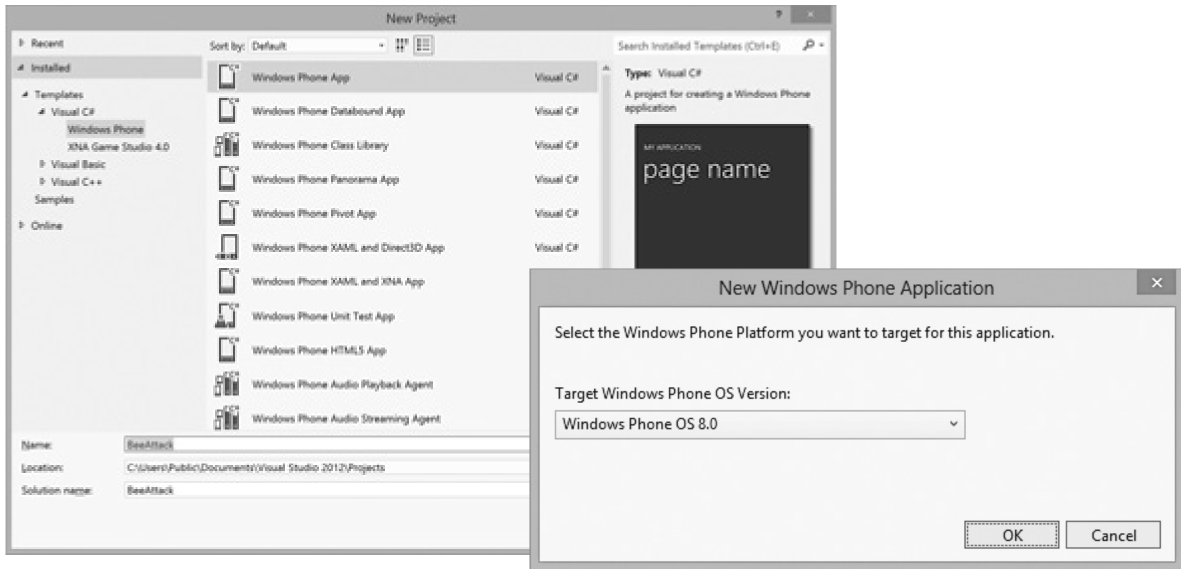


При этом у телефона должна быть учетная запись в Windows Phone Dev Center (это платная услуга): **https://dev.windowsphone.com/**.

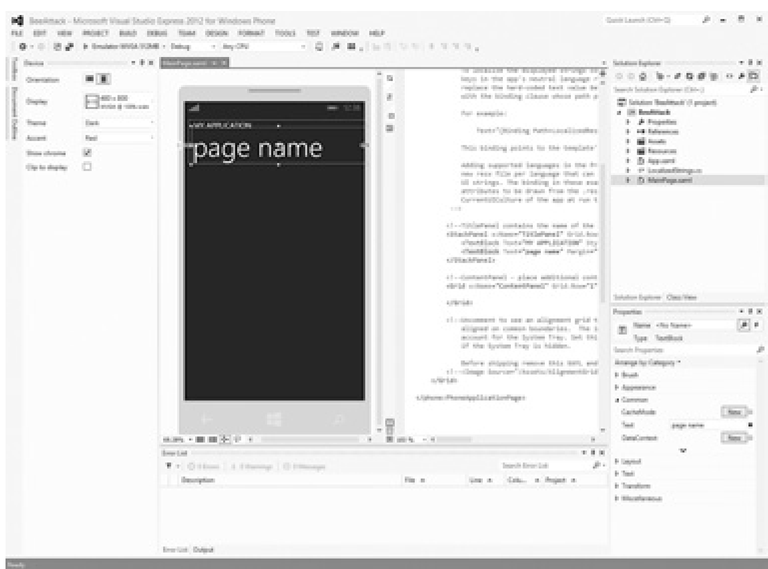
После настройки учетной записи вы сможете зарегистровать телефон:

http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff769508.aspx

1. **Создадим новое приложение Windows Phone.**
Откройте Visual Studio 2012 для Windows Phone и **создайте новый проект Windows Phone App с именем BeeAttack**. (Можете назвать его по-другому, но тогда пространство имен будет отличаться от примеров в книге.)



2. **Посмотрим на созданные IDE файлы.**
IDE Visual Studio для Windows Phone практически совпадает с IDE для редакций Windows 8 и Desktop. При создании проекта IDE добавляет файлы:



★ XAML и C# код главной страницы (*MainPage.xaml* и *MainPage.xaml.cs*)

★ Главные файлы приложения (*App.xaml* и *App.xaml.cs*)

★ Папку *Assets*

★ Файл C# для статических ресурсов и локализации (*LocalizedStrings.cs*)

★ Манифест приложения, ресурсы и ряд файлов и папок (но сейчас это не потребуется).

3 **Скроем панель заголовка на главной странице.**

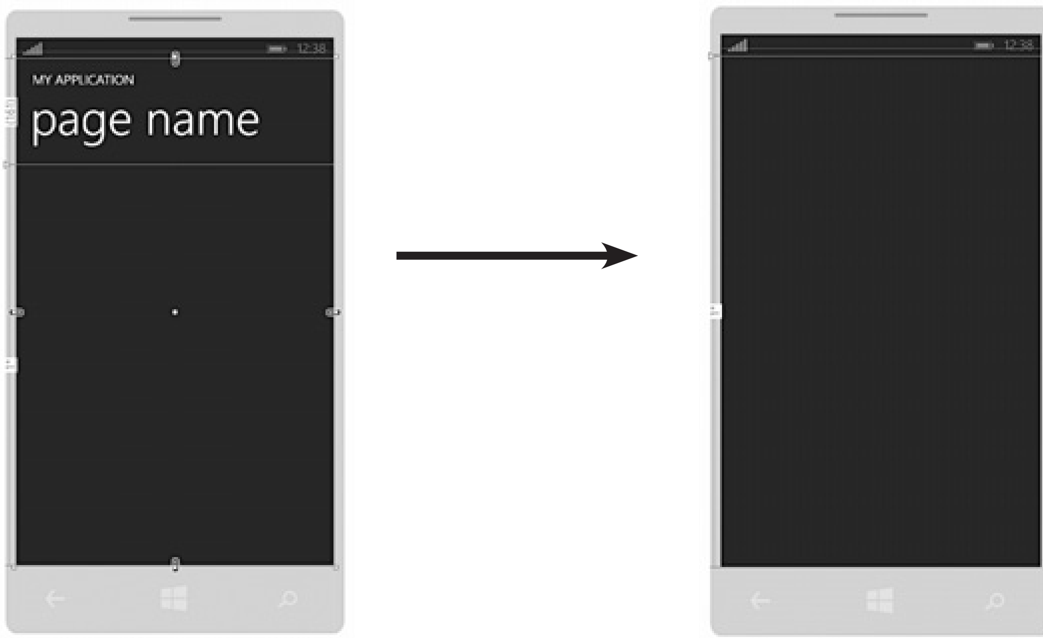
В IDE уже должна быть открыта страница *MainPage.xaml* (если это не так, откройте ее). Это главная страница вашего приложения. Найдите в коде XAML элемент StackPanel с именем TitlePanel.

```
<!--TitlePanel contains the name of the application and page title-->
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
    <TextBlock Text="MY APPLICATION" Style="{StaticResource PhoneTextNormalStyle}" Margin="12,0"/>
    <TextBlock Text="page name" Margin="9,-7,0,0" Style="{StaticResource PhoneTextTitle1Style}"/>
</StackPanel>
```

Добавьте **к элементу StackPanel код** **`Visibility="Collapsed"`**, чтобы убрать заголовок TextBlocks. Это развернет ваше приложение на весь экран.

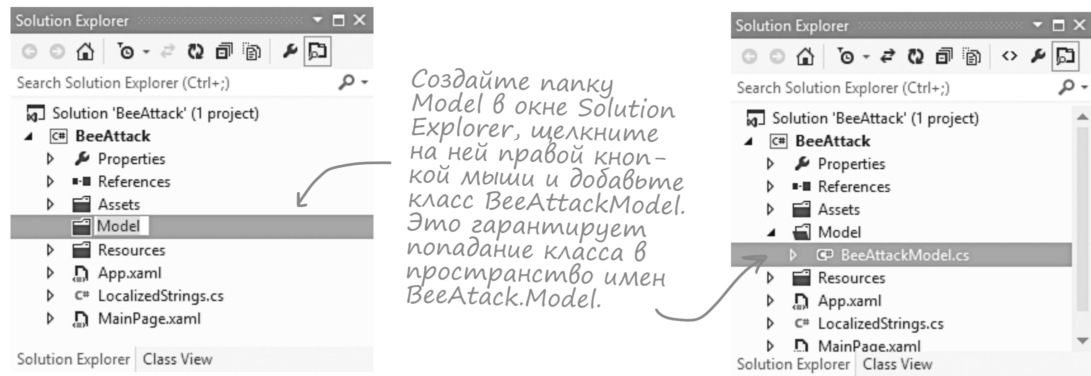
IDE создала файл MainPage.xaml с элементом StackPanel, который называется TitlePanel и содержит два элемента TextBlock. Спрячьте их с помощью свойства Visibility.

```
<!--TitlePanel contains the name of the application and page title-->
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28" Visibility="Collapsed">
    <TextBlock Text="MY APPLICATION" Style="{StaticResource PhoneTextNormalStyle}" Margin="12,0"/>
    <TextBlock Text="page name" Margin="9,-7,0,0" Style="{StaticResource PhoneTextTitle1Style}"/>
</StackPanel>
```



Создадим папку Model и добавим класс BeeAttackModel.

Основой приложения BeeAttack является MVVM, и нам понадобятся слои Model, View и ViewModel. Model представлен классом `BeeAttackModel`, находящимся в папке и в пространстве имен *Model*. Создайте папку *Model* и добавьте в нее класс `BeeAttackModel`:



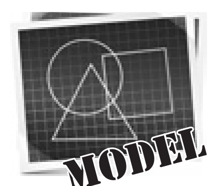
Вот код класса `BeeAttackModel`. В нем присутствуют свойства, описывающие количество оставшихся промахов, общий счет и время между вылетами пчел. Кроме того, есть метод запуска игры, метод, отвечающий за перемещение цветка, метод, управляющий падением пчелы, и метод, определяющий положение улья при следующем вылете.

```
using Windows.Foundation;

class BeeAttackModel {
    public int MissesLeft { get; private set; }
    public int Score { get; private set; }
    public TimeSpan TimeBetweenBees  {
        get {
            double milliseconds = 500;
            milliseconds = Math.Max(milliseconds - Score * 2.5, 100);
            return TimeSpan.FromMilliseconds(milliseconds);
        }
    }

    private double _flowerWidth;
    private double _beeWidth;
    private double _flowerLeft;
    private double _playAreaWidth;
    private double _hiveWidth;
    private double _lastHiveLocation;
    private bool _gameOver;
    private readonly Random _random = new Random();

    public void StartGame(double flowerWidth, double beeWidth, double playAreaWidth, double hiveWidth) {
        _flowerWidth = flowerWidth;
        _beeWidth = beeWidth;
        _playAreaWidth = playAreaWidth;
        _hiveWidth = hiveWidth;
        _lastHiveLocation = playAreaWidth / 2;
        MissesLeft = 5;
        Score = 0;
        _gameOver = false;
        OnPlayerScored();
    }
```

Эти свойства класс ViewModel использует для обновления счета и количества оставшихся попыток.

По мере ловли пчел игра ускоряется. Это свойство рассчитывает время между вылетами пчел, беря за основу счет игры.

Метод StartGame() перезагружает игру. Класс ViewModel передает в него данные о ширине цветка, пчелы, игровой области и улья. Все это используется во внутренних методах.



Этот метод вызывается каждый раз, когда игрок перемещает цветок. Он обновляет положение цветка.

```
    public void MoveFlower(double flowerLeft) {
        _flowerLeft = flowerLeft;
    }

    public void BeeLanded(double beeLeft) {
        if ((beeLeft < _flowerLeft) || (beeLeft > _flowerLeft + _flowerWidth)) {
            if (MissesLeft > 0) {
                MissesLeft--;
                OnMissed();
            } else {
                _gameOver = true;
                OnGameOver();
            }
        }
        else if (!_gameOver) {
            Score++;
            OnPlayerScored();
        }
    }
```

При приземлении пчелы Model проверяет, попадает ли левый угол пчелы в границы цветка. Отрицательный результат — промах; в случае положительного результата счет игрока увеличивается.

```
    public double NextHiveLocation() {
        double delta = 10 + Math.Max(1, Score * .5);

        if (_lastHiveLocation <= _hiveWidth * 2)
            _lastHiveLocation += delta;
        else if (_lastHiveLocation >= _playAreaWidth - _hiveWidth * 2)
            _lastHiveLocation -= delta;
        else
            _lastHiveLocation += delta * (_random.Next(2) == 0 ? 1 : -1);


        return _lastHiveLocation;
    }
```

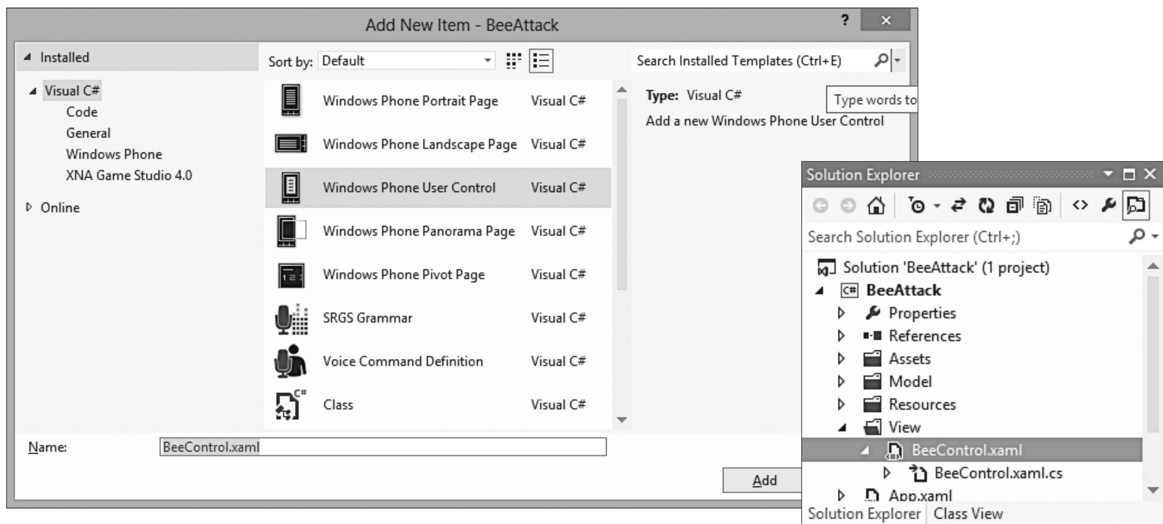
После каждого приземления пчелы ViewModel использует этот метод для определения следующего положения улья (по горизонтали) в момент вылета пчелы. Метод гарантирует, что улей не будет смещаться слишком далеко — по мере роста счетчика точки сброса пчел становятся все дальше и дальше друг от друга.

```
    public EventHandler Missed;
    private void OnMissed() {
        EventHandler missed = Missed;
        if (missed != null)
            missed(this, new EventArgs());
    }

    public EventHandler GameOver;
    private void OnGameOver() {
        EventHandler gameOver = GameOver;
        if (gameOver != null)
            gameOver(this, new EventArgs());
    }

    public EventHandler PlayerScored;
    private void OnPlayerScored() {
        EventHandler playerScored = PlayerScored;
        if (playerScored != null)
            playerScored(this, new EventArgs());
    }
}
```

Класс ViewModel ожидает эти события, чтобы обновить класс View при промахе игрока, удачной поимке пчелы и завершении игры.

5 **Создадим папку View и построим BeeControl.**

Класс View состоит из двух пользовательских элементов управления Windows Phone. Во-первых, `BeeControl` — анимированный рисунок пчелы, падающей вниз по игровой области. Начните с создания папки *View*. Щелкните на ней правой кнопкой мыши в окне Solution Explorer и добавьте новый элемент, затем в эту же папку **добавьте новый** Windows Phone User Control **с именем** ***BeeControl.xaml***:



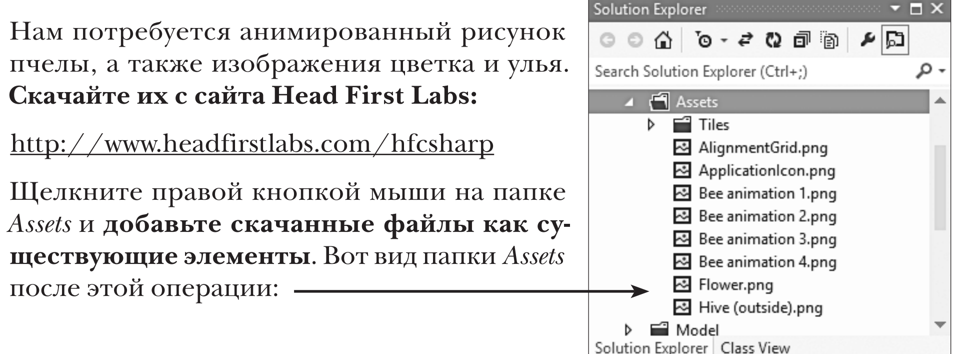
Отредактируйте код XAML в файле *BeeControl.xaml*. Найдите элемент Grid с именем LayoutRoot, сделайте его фон прозрачным и добавьте элемент управления Image с именем `image`. Вот как станет выглядеть итоговый XAML-код:

```
<Grid x:Name="LayoutRoot" Background="Transparent">

    <Image x:Name="image" Stretch="Fill"/>

</Grid>
```

Нам потребуется анимированный рисунок пчелы, а также изображения цветка и улья. **Скачайте их с сайта Head First Labs:**

http://www.headfirstlabs.com/hfcsharp

Щелкните правой кнопкой мыши на папке *Assets* и **добавьте скачанные файлы как существующие элементы**. Вот вид папки *Assets* после этой операции:



Теперь можно **добавить программный код C# для *BeeControl***. Вот что следует ввести в файл *BeeControl.xaml.cs*:

```
using Microsoft.Phone.Media;
using System.Windows.Media.Animation;
using System.Windows.Media.Imaging;

public sealed partial class BeeControl : UserControl {
    public readonly Storyboard FallingStoryboard;

    public BeeControl() {
        this.InitializeComponent();
        StartFlapping(TimeSpan.FromMilliseconds(30));
    }

    public BeeControl(double X, double fromY, double toY, EventHandler completed) : this() {
        FallingStoryboard = new Storyboard();
        DoubleAnimation animation = new DoubleAnimation();

        Storyboard.SetTarget(animation, this);
        Canvas.SetLeft(this, X);
        Storyboard.SetTargetProperty(animation, new PropertyPath("(Canvas.Top)"));
        animation.From = fromY;
        animation.To = toY;
        animation.Duration = TimeSpan.FromSeconds(1);

        if (completed != null) FallingStoryboard.Completed += completed;

        FallingStoryboard.Children.Add(animation);
        FallingStoryboard.Begin();
    }

    public void StartFlapping(TimeSpan interval) {
        List<string> imageNames = new List<string>() {
            "Bee animation 1.png", "Bee animation 2.png", "Bee animation 3.png", "Bee animation 4.png"
        };

        Storyboard storyboard = new Storyboard();
        ObjectAnimationUsingKeyFrames animation = new ObjectAnimationUsingKeyFrames();
        Storyboard.SetTarget(animation, image);
        Storyboard.SetTargetProperty(animation, new PropertyPath("Source"));

        TimeSpan currentInteval = TimeSpan.FromMilliseconds(0);
        foreach (string imageName in imageNames) {
            ObjectKeyFrame keyFrame = new DiscreteObjectKeyFrame();
            keyFrame.Value = CreateImageFromAssets(imageName);
            keyFrame.KeyTime = currentInteval;
            animation.KeyFrames.Add(keyFrame);
            currentInteval = currentInteval.Add(interval);
        }

        storyboard.RepeatBehavior = RepeatBehavior.Forever;
        storyboard.AutoReverse = true;
        storyboard.Children.Add(animation);
        storyboard.Begin();
    }

    private static BitmapImage CreateImageFromAssets(string imageFilename) {
        return new BitmapImage(new Uri("/Assets/" + imageFilename, UriKind.RelativeOrAbsolute));
    }
}
```

После завершения анимации объект storyboard вызывает свое событие `Completed`. Перегруженный конструктор `BeeControl` принимает в качестве параметра делегат `EventHandler`, на основе которого ViewModel принимает решение о моменте приземления пчелы.

Эта анимация смещает пчелу вниз по игровой области Canvas. Путь начинается в улье и заканчивается цветком.

Переданный в качестве аргумента делегат EventHandler связан с событием Completed объекта Storyboard.

Эта анимация последовательно показывает кадры, заставляя пчелу двигать крыльями.

6 **Создадим слой ViewModel.**

В слой ViewModel входит класс `BeeAttackViewModel`, который через методы, свойства и события класса Model обновляет элементы управления в слое View. **Создайте папку *ViewModel* и добавьте класс `BeeAttackViewModel`.**

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'BeeAttack' (1 project)
BeeAttack
Properties
References
Assets
Model
Resources
View
ViewModel
BeeAttackViewModel.cs
App.xaml
Solution Explorer | Class View

Через ItemsPanel класс View связывает коллекцию объектов BeeControl с объектами Children элемента Canvas, давая слою ViewModel возможность добавлять пчел, обновляя поле _beeControls.

```
using View;
using System.Collections.ObjectModel;
using System.Collections.Specialized;
using System.ComponentModel;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Threading;

class BeeAttackViewModel : INotifyPropertyChanged {
    public INotifyCollectionChanged BeeControls { get { return _beeControls; } }
    private readonly ObservableCollection<BeeControl> _beeControls
                                                         = new ObservableCollection<BeeControl>();

    public Thickness FlowerMargin { get; private set; }
    public Thickness HiveMargin { get; private set; }
    public int MissesLeft { get { return _model.MissesLeft; } }
    public int Score { get { return _model.Score; } }
    public Visibility GameOver { get; private set; }

    private Size _beeSize;
    private readonly Model.BeeAttackModel _model = new Model.BeeAttackModel();
    private readonly DispatcherTimer _timer = new DispatcherTimer();
    private double _lastX;
    private Size _playAreaSize { get; set; }
    private Size _hiveSize { get; set; }
    private Size _flowerSize { get; set; }

    public BeeAttackViewModel() {
        _model.Missed += MissedEventHandler;
        _model.GameOver += GameOverEventHandler;
        _model.PlayerScored += PlayerScoredEventHandler;

        _timer.Tick += HiveTimerTick;

        GameOver = Visibility.Visible;
        OnPropertyChanged("GameOver");
    }

    public void StartGame(Size flowerSize, Size hiveSize, Size playAreaSize) {
        _flowerSize = flowerSize;
        _hiveSize = hiveSize;
        _playAreaSize = playAreaSize;
        _beeSize = new Size(playAreaSize.Width / 10, playAreaSize.Width / 10);
        _model.StartGame(flowerSize.Width, _beeSize.Width, playAreaSize.Width, hiveSize.Width);
        OnPropertyChanged("MissesLeft");

        _timer.Interval = _model.TimeBetweenBees;
        _timer.Start();

        GameOver = Visibility.Collapsed;
        OnPropertyChanged("GameOver");
    }
```

Эти свойства привязаны к параметру Margin изображений цветка и улья, поэтому класс ViewModel может перемещать их влево и вправо, обновляя левое поле и вызывая событие PropertyChanged.

Ссылку на экземпляр Model класс ViewModel хранит в закрытом поле. И именно здесь его обработчики событий привязываются к событиям слоя Model.

Класс ViewModel создает каждый объект BeeControl и задает его высоту и ширину, поэтому нам нужен объект size.

Класс View запускает игру, вызывая метод StartGame() класса ViewModel и передавая в него размеры цветка, улья и игровой области. Класс ViewModel задает свои закрытые поля, запускает таймер и обновляет свое свойство GameOver.

Когда пользователь проводит пальцем, запускается событие ManipulationDelta в слое view и его обработчик вызывает данный метод для обновления местоположения цветка.

```
    public void ManipulationDelta(double newX) {
        newX = _lastX + newX * 1.5;
        if (newX >= 0 && newX < (_playAreaSize.Width - _flowerSize.Width)) {
            _model.MoveFlower(newX);
            FlowerMargin = new Thickness(newX, 0, 0, 0);
            OnPropertyChanged("FlowerMargin");
            _lastX = newX;
        }
    }

    private void GameOverEventHandler(object sender, EventArgs e) {
        _timer.Stop();
        GameOver = Visibility.Visible;
        OnPropertyChanged("GameOver");
    }

    private void MissedEventHandler(object sender, EventArgs e) {
        OnPropertyChanged("MissesLeft");
    }

    void HiveTimerTick(object sender, EventArgs e) {
        if (_playAreaSize.Width <= 0) return;

        double x = _model.NextHiveLocation();

        HiveMargin = new Thickness(x, 0, 0, 0);
        OnPropertyChanged("HiveMargin");

        BeeControl bee = new BeeControl(x + _hiveSize.Width / 2, 0,
                                   _playAreaSize.Height + _flowerSize.Height / 3, BeeLanded);
        bee.Width = _beeSize.Width;
        bee.Height = _beeSize.Height;
        _beeControls.Add(bee);
    }

    private void BeeLanded(object sender, EventArgs e) {
        BeeControl landedBee = null;
        foreach (BeeControl sprite in _beeControls) {
            if (sprite.FallingStoryboard == sender)
                landedBee = sprite;
        }
        _model.BeeLanded(Canvas.GetLeft(landedBee));
        if (landedBee != null) _beeControls.Remove(landedBee);
    }

    public event PropertyChangedEventHandler PropertyChanged;
    private void OnPropertyChanged(string propertyName) {
        PropertyChangedEventHandler propertyChanged = PropertyChanged;
        if (propertyChanged != null)
            propertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }

    private void PlayerScoredEventHandler(object sender, EventArgs e) {
        OnPropertyChanged("Score");
        _timer.Interval = _model.TimeBetweenBees;
    }
}
```

В момент завершения игры класс ViewModel останавливает таймер и обновляет свое свойство GameOver, связанное со свойством Visibility элемента StackPanel, содержащего элемент TextBlock с текстом Game Over TextBlock, кнопку и ссылку.

При каждом отсчете таймера класс ViewModel запрашивает у класса Model новое положение улья и выпускает новую пчелу, создавая элемент BeeControl и добавляя его в коллекцию _beeControls, в результате чего он включается в коллекцию Children элемента Canvas.

Делегат метода BeeLanded передается в `BeeControl`, который добавляет его к событию `Completed` анимации storyboard — при возникновении этого события параметру `sender` присваивается объект `Storyboard`. Свойство `FallingStoryboard` объекта `BeeControl` возвращает ссылку на объект `Storyboard`, и класс ViewModel использует ее для поиска нужного объекта `BeeControl` в коллекции `_beeControls`.

При получении игроком очередного очка таймер обновляется через свойство TimeBetweenBees класса Model, и игра ускоряется.

Создадим BeeAttackGameControl для управления игрой.

Класс View содержит еще кое-что. В `BeeAttackGameControl` находятся элементы управления Image для улья и цветка, Canvas, по которому падают пчелы, и элементы, отображающиеся при завершении игры (в том числе кнопка Button для начала новой игры). **Добавьте новый** Windows Phone User Control **с именем *BeeAttackGameControl.xaml*** в папку *View*. Вот его код XAML:

```
<Grid x:Name="LayoutRoot" Background="SkyBlue">
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="10*"/>
        <RowDefinition Height="2*"/>
    </Grid.RowDefinitions>

    <Image x:Name="hive"
            Source="/Assets/Hive (outside).png"
            HorizontalAlignment="Left"
            Margin="{Binding HiveMargin}"/>

    <ItemsControl Grid.Row="1" x:Name="playArea">
        <ItemsControl ItemsSource="{Binding BeeControls}">
            <ItemsControl.ItemsPanel>
                <ItemsPanelTemplate>
                    <Canvas/>
                </ItemsPanelTemplate>
            </ItemsControl.ItemsPanel>
        </ItemsControl>
    </ItemsControl>

    <TextBlock Grid.Row="1" Foreground="Black" VerticalAlignment="Top">
        <Run>Misses left: </Run>
        <Run Text="{Binding MissesLeft}"/>
    </TextBlock>

    <TextBlock Grid.Row="1" Foreground="Black" VerticalAlignment="Top"
                HorizontalAlignment="Right" Text="{Binding Score}"
                Style="{StaticResource PanoramaItemHeaderTextStyle}"/>

    <Image x:Name="flower"
            Source="/Assets/Flower.png"
            Grid.Row="2"
            HorizontalAlignment="Left"
            Margin="{Binding FlowerMargin}"/>

    <StackPanel Grid.Row="1" VerticalAlignment="Center"
                HorizontalAlignment="Center" Visibility="{Binding GameOver}">
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
            <TextBlock Foreground="Yellow"
                        Style="{StaticResource JumpListAlphabetSmallStyle}">Bee</TextBlock>
            <view:BeeControl Width="75" Height="75"/>
            <TextBlock Foreground="Black"
                        Style="{StaticResource JumpListAlphabetSmallStyle}">Attack</TextBlock>
        </StackPanel>

        <Button Click="Button_Click">Start a new game</Button>

        <HyperlinkButton Content="Learn how to build this game"
                            NavigateUri="http://www.headfirstlabs.com/hfcsharp"
                            TargetName="_blank" />
    </StackPanel>
</Grid>
```

IDE добавляет пустой элемент Grid с именем LayoutRoot в новый пользовательский элемент Windows Phone. Измените фон на SkyBlue и поделите сетку на три строки: верхняя для изображения улья, средняя для игровой области, нижняя для изображения цветка.

Свойство Margin изображения улья связано со свойством HiveMargin класса ViewModel.

Свойство ItemsSource элемента ItemsControl связано с коллекцией объектов BeeControl в классе ViewModel, поэтому добавление на холст происходит внутри ItemsPanelTemplate.

Эти элементы TextBlocks показывают счет игрока и количество оставшихся до конца игры промахов.

Свойство GameOver класса ViewModel возвращает перечисление Visibility, поэтому его можно напрямую связать со свойством XAML Visibility.

Для этой строки вам понадобится пространство имен XML xmlns:view, добавленное на следующей странице. Она рисует пчелу, машущую крыльями.

Обработчик щелчка на этой кнопке есть в коде на следующей странице.

Добавим к тегу <UserControl> в верхней части файла XAML свойство **`xmlns:view`** и обработчик событий **`ManipulationDelta`**. Поместите курсор перед скобкой **>** в строке 10 и нажмите Enter, затем **начните вводить `xmlns:view=""`**. Сразу после кавычек появится окно IntelliSense для вставки пространства имен:

```
xmlns:view=""
    http://schemas.microsoft.com/client/2007
    http://schemas.microsoft.com/expression/blend/2008
    http://schemas.microsoft.com/winfx/2006/xaml
    http://schemas.openxmlformats.org/markup-compatibility/2006
    BeeAttack (BeeAttack)
    BeeAttack.Model (BeeAttack)
    BeeAttack.Resources (BeeAttack)
    BeeAttack.View (BeeAttack)
    BeeAttack.ViewModel (BeeAttack)
```

Выберите из списка пространство имен View. Если вы назвали проект по-другому, вместо BeeAttack вы увидите свой вариант имени.

Нажмите Enter и начните вводить **`ManipulationDelta=""`**, чтобы добавить обработчик событий. Вот предлагаемый окном IntelliSense вариант:

```
xmlns:view="clr-namespace:BeeAttack.View"
ManipulationDelta=""
                 <New Event Handler>
```

После этого в открывающем теге <UserControl> появятся две строки:

```
xmlns:view="clr-namespace:BeeAttack.View"
ManipulationDelta="UserControl_ManipulationDelta"
```

Если по какой-то причине окно IntelliSense не появилось, введите эти строки вручную.

Напоследок **откройте файл *BeeAttackGameControl.xaml.cs* и добавьте к нему следующий код**:

```
using ViewModel;

public partial class BeeAttackGameControl : UserControl {
    private readonly ViewModel.BeeAttackViewModel _viewModel = new ViewModel.BeeAttackViewModel();

    public BeeAttackGameControl() {
        InitializeComponent();

        DataContext = _viewModel;
    }

    private void Button_Click(object sender, RoutedEventArgs e) {
        _viewModel.StartGame(flower.RenderSize, hive.RenderSize, playArea.RenderSize);
    }

    private void UserControl_ManipulationDelta(object sender,
                                               System.Windows.Input.ManipulationDeltaEventArgs e) {
        _viewModel.ManipulationDelta(e.DeltaManipulation.Translation.X);
    }
}
```

Элемент управления игрой обладает экземпляром ViewModel, который используется как контекст данных.

Кнопка Start вызывает метод StartGame() класса ViewModel, передавая нужные этому классу размеры в качестве параметров.

Обработчик события ManipulationDelta вызывает непосредственно метод ViewModel.

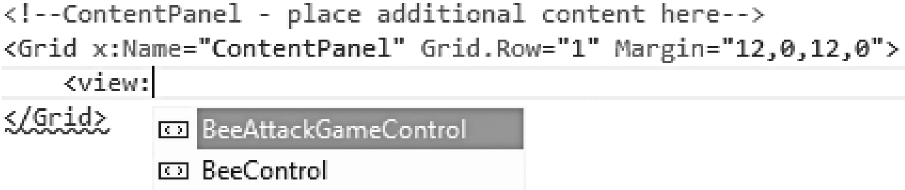
8 **Добавим управление игрой на главную страницу.**

Теперь, когда вся игра инкапсулирована в классе `BeeAttackGameControl`, осталось добавить ее на главную страницу. Откройте файл *MainPage.xaml* и **добавьте к открывающему тегу `<phone:PhoneApplicationPage>` пространство имен `xmlns:view`**, как вы это делали с открывающим тегом в файле *BeeAttackGameControl.xaml*.

```
xmlns:view="clr-namespace:BeeAttack.View"
```

← Если вы назвали проект по-другому, вместо BeeAttack будет фигурировать другое имя.

Найдите элемент Grid с именем ContentPanel и добавьте `BeeAttackGameControl`. Поместите курсор между открывающим и закрывающим тегами, **начните вводить `<view:`** и в окне IntelliSense выберите вариант `BeeAttackGameControl`:



Вот как после этого должен выглядеть код XAML:

```
<!--ContentPanel - place additional content here-->
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <view:BeeAttackGameControl/>
</Grid>
```

Поздравляем, игра работает!

Сразу после обновления файла *MainPage.xaml* вы увидите в окне конструктора главную страницу игры. Ее можно запустить в эмуляторе или на устройстве... и проявите фантазию!

- Добавьте приносящих больше очков пчел или злых пчел, которых нужно избегать.
- Заставьте пчел менять направление движения, добавив анимацию (`Canvas.Left`).
- Мы добавили в проект изображение «внутри улья». Есть идеи, как его использовать?
- **Похвастайтесь своими результатами**, опубликовав код на сайте CodePlex, GitHub или другом аналогичном сайте... и поделитесь успехами с читателями форума *Head First C#*: http://www.headfirstlabs.com/hfcsharp.

Э. Стиллмен, Дж. Грин

Изучаем C#

3-е издание

Перевела с английского И. Рузмайкина

Заведующий редакцией *П. Щеголев*
Руководитель проекта *А. Юрченко*
Ведущий редактор *Ю. Сергиенко*
Корректор *Н. Викторова*
Верстка *Н. Лукьянова*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Подписано в печать 23.05.14. Формат 84×108/16. Усл. п. л. 85,680. Тираж 1500. Заказ 0000.

Отпечатано в соответствии с предоставленными материалами в ООО «ИПК Парето-Принт».
170546, Тверская область, Промышленная зона Боровлево-1, комплекс № 3А, www.pareto-print.

Программируем для iPhone и iPad. 3-е изд.

Т. Пайлон, Д. Пайлон

ISBN 978-5-496-01083-2

Объем: 336 с.

С появлением iPhone мир изменился. Потом с появлением iPhone 4 он изменился снова. А теперь к iPhone добавился еще и революционный планшет iPad. Современные устройства на базе iOS используются в бизнесе и учебе, для работы и развлечений, и на App Store уже сейчас успешно работают десятки тысяч программистов и известных софтверных компаний. Представим, что у вас появилась гениальная идея приложения для iPhone и iPad. С чего начать? Эта книга поможет вам разработать свое первое приложение в самые кратчайшие сроки. Вы не только узнаете, как спроектировать приложение для устройств Apple и сделать его уникальным, но и в совершенстве овладеете принципами программирования на Objective-C и инструментами iPhone SDK, в том числе Interface Builder и Xcode. Apple предоставляет программное обеспечение, эта книга дает знания — от вас потребуются лишь энтузиазм и желание научиться разрабатывать оригинальные и коммерчески успешные приложения для iPhone и iPad. Новое издание книги охватывает версии iOS7 и Xcode5. Особенностью данного издания является уникальный способ подачи материала, выделяющий серию «Head First» издательства O'Reilly в ряду множества скучных книг, посвященных программированию.

Rails 4. Гибкая разработка веб-приложений

С. Руби, Д. Томас, Д. Хэнссон

ISBN 978-5-496-00898-3
Объем: 448 с.

Перед вами новое издание бестселлера «Agile web development with Rails», написанного Сэмом Руби — руководителем Apache Software Foundation и разработчиком формата Atom, Дэйвом Томасом — автором книги «Programming Ruby» и Дэвидом Хэнссоном — создателем технологии Rails. Rails представляет собой среду, облегчающую разработку, развертывание и обслуживание веб-приложений. За время, прошедшее с момента ее первого релиза, Rails прошла путь от малоизвестной технологии до феномена мирового масштаба и стала именно той средой, которую выбирают, чтобы создавать так называемые «приложения Web 2.0». Эта книга, уже давно ставшая настольной по изучению Ruby on Rails, предназначена для всех программистов, собирающихся создавать и развертывать современные веб-приложения. Из первой части книги вы получите начальное представление о языке Ruby и общие сведения о самой среде Rails. Далее на примере создания интернет-магазина вы изучите концепции, положенные в основу Rails. В третьей части рассматривается вся экосистема Rails: ее функции, возможности и дополнительные модули. Обновленное издание книги описывает работу с Rails поколения 4 и Ruby 1.9 и 2.0.

Идеальная IT-компания. Как из гиков собрать команду программистов

Б. Фитцпатрик, Б. Коллинз-Сассмэн

ISBN 978-5-496-00949-2

Объем: 208 с.

В современном мире разработки ПО успех программиста во многом зависит не только от качества кода, но и от его взаимодействия с другими людьми. В этой занимательной и ироничной книге раскрываются основные закономерности и шаблоны поведения, возникающие в команде разработчиков ПО. Рассматриваются основные роли каждого из участников коллектива, паттерны их поведения и примеры организации наиболее эффективного взаимодействия внутри команды программистов. Эта книга поможет вам оценить важность человеческого фактора в процессе разработки ПО и научиться выстраивать эффективно работающую команду для IT-проекта любой сложности..

Как программировать на Visual C# 2012. 5-е изд.

П. Дейтел, Х. Дейтел

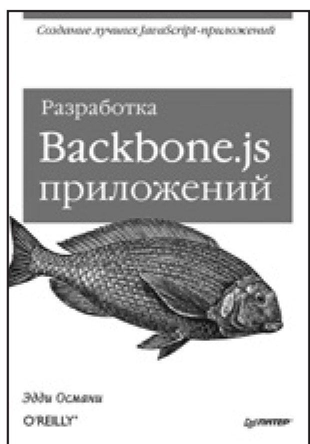
ISBN 978-5-496-00897-6

Объем: 864 с.

Эта книга, выходящая уже в пятом издании, является одним из самых популярных в мире учебников по программированию на платформе Microsoft .NET на языке Visual C# 2012. Здесь рассматриваются основы синтаксиса Visual C# и работа с программой Visual C# Express 2012. По ходу работы с книгой читатели изучат структуры управления, классы, объекты, методы, переменные, массивы C# и основные методы объектно-ориентированного программирования. Также рассматриваются и более сложные методы, в том числе поиск, сортировка, структуры данных, коллекции. Каждая глава содержит множество практических примеров. Пятое издание было полностью обновлено под новейшую версию Visual C# 2012. Книга может служить учебником по Visual C#, также она будет полезна широкому кругу начинающих программистов, которые хотят научиться программировать на C#.

Разработка Backbone.js приложений

Э. Османи



ISBN 978-5-496-00962-1

Объем: 352 с.

Backbone — это javascript-библиотека для тяжелых фронтэнд javascript-приложений, таких, например, как gmail или twitter. В таких приложениях вся логика интерфейса ложится на браузер, что дает очень значительное преимущество в скорости интерфейса. Цель этой книги — стать удобным источником информации в помощь тем, кто разрабатывает реальные приложения с использованием Backbone. Издание охватывает теорию MVC и методы создания приложений с помощью моделей, представлений, коллекций и маршрутов библиотеки Backbone; модульную разработку ПО с помощью Backbone.js и AMD (посредством библиотеки RequireJS), решение таких типовых задач, как использование вложенных представлений, устранение проблем с маршрутизацией средствами Backbone и jQuery Mobile, а также многие другие вопросы.