

Эндрю Стилмен  
Дженнифер Грин

Включая  
C# .NET 4.0  
и Visual Studio 2010

# Изучаем C#

2-е издание

Управляй  
данными  
с LINQ



Познай секреты  
абстрагирования  
и наследования



Изучай C#  
на забавных  
примерах



Узнай,  
как методы  
расширения  
облегчают жизнь  
программиста



Научись  
эффективно  
использовать  
обобщенные  
коллекции

# Изучаем C#

2-е издание

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

Э. Стиллмен

Дж. Грин



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

2012

ББК 32.973.2-018.1

УДК 004.43

С80

**Стиллмен Э., Грин Дж.**

С80 Изучаем C#. 2-е изд. — СПб.: Питер, 2012. — 696 с.: ил.

ISBN 978-5-459-00422-9

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 примите участие в забавных и интересных программных проектах: от разработки карточной игры до создания серьезного бизнес-приложения. Второе издание книги включает последние версии C# .NET 4.0 и Visual Studio 2010 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.

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.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-4493-8034-2 англ.

© Authorized Russian translation of the English edition Head First C# © O'Reilly Media, Inc. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same

ISBN 978-5-459-00422-9

© Перевод на русский язык ООО Издательство «Питер», 2012  
© Издание на русском языке, оформление  
ООО Издательство «Питер», 2012

## авторы



**Эндрю Стилмен** родился в Нью-Йорке, дважды ему приходилось жить в Питтсбурге. Сначала он закончил там университет Карнеги-Меллон, затем именно в этом городе они с Дженнинг начали свой консультационный бизнес и работу над первой книгой для издательства O'Reilly.

Вернувшись после колледжа в родной город, Эндрю устроился программистом в фирму EMI-Capitol Records, что было не лишено смысла, ведь в школьные годы он научился играть на виолончели и джазовой бас-гитаре. И в той же компании, где Эндрю руководил командой программистов, работала Дженнинг. За прошедшие годы Эндрю сотрудничал с весьма квалифицированными программистами и уверен, что многому от них научился.

В свободное от написания книг время Эндрю создает бесполезные (но забавные) программы, пишет музыку (для видеоигр и не только), изучает тайцзи и айкидо, встречается со своей девушкой Лизой и выращивает шпинат.

С момента своей встречи в 1998-м Дженнинг и Эндрю вместе разрабатывают программное обеспечение и пишут о нем. Их первая книга *Applied Software Project Management* вышла в издательстве O'Reilly в 2005 году. А первая книга в серии *Head First Head First PMP* появилась в 2007-м.

В 2003 году ребята основали фирму *Stellman & Greene Consulting* по разработке программного обеспечения для ученых, исследующих воздействие гербицидов, примененных во времена войны во Вьетнаме. В свободное от написания программ и книг время Эндрю и Дженинг участвуют в конференциях разработчиков программного обеспечения и руководителей проектов.

Их личный блог расположен по адресу: <http://www.stellman-greene.com>

**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 Дженнинг переехала в Нью-Йорк и устроилась в фирму по разработке финансового программного обеспечения. Она управляла командой разработчиков, занимавшихся искусственным интеллектом и обработкой естественных языков.

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

Дженнифер обожает путешествия, индийское кино, комиксы, компьютерные игры и свою гончую.

## (содержание (сводка))

Введение	23
1 Эффективность с C#. Визуальные приложения за 10 минут	35
2 Это всего лишь код. Под покровом	73
3 Объекты, по порядку стройся! Приемы программирования	115
4 Типы и ссылки. 10:00 утра. Куда подевались наши данные?	153
5 Инкапсуляция. Пусть личное остается... личным	195
6 Наследование. Генеалогическое древо объектов	231
7 Интерфейсы и абстрактные классы. Пусть классы держат обещания	283
8 Перечисления и коллекции. Большие объемы данных	339
9 Чтение и запись файлов. Сохрани массив байтов и спаси мир	395
10 Обработка исключений. Борьба с огнем надоедает	449
11 События и делегаты. Что делает ваш код, когда вы на него не смотрите	491
12 Обзор и предварительные результаты. Знания, сила и построение приложений	525
13 Элементы управления и графические фрагменты. Наводим красоту	573
14 Капитан Великолепный. Смерть объекта	631
15 LINQ. Управляем данными	669

## (содержание (настоящее))

### **Введение**

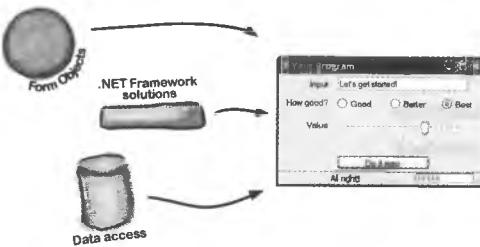
**Ваш мозг и C#.** Вы учитесь — готовитесь к экзамену. Или пытаетесь освоить сложную техническую тему. Ваш мозг пытается оказать вам услугу. Он старается сделать так, чтобы на эту очевидно несущественную информацию не тратились драгоценные ресурсы. Их лучше потратить на что-нибудь важное. Так как же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

Для кого написана эта книга	24
Мы знаем, о чем вы думаете	25
Метапознание: наука о мышлении	27
Заставьте свой мозг повиноваться	29
Что вам потребуется	30
Информация	31
Технические рецензенты	32
Благодарности	33

# 1 Эффективность с C#

## Визуальные приложения за 10 минут

**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#	36
C#, ИСР Visual Studio многое упрощают	37
Избавьте директора от бумаг	38
Перед началом работы выясните, что именно нужно пользователю	39
Что мы собираемся сделать	40
Это вы делаете в Visual Studio	42
А это Visual Studio делает за вас	42
Создаем пользовательский интерфейс	46
Visual Studio, за сценой	48
Редактирование кода	49
Первый запуск приложения	50
Где же мои файлы?	50
Вот что уже сделано	51
Для хранения информации нужна база данных	52
База данных, созданная ИСР	53
Язык SQL	53
Создание таблицы для списка контактов	54
Поля в контактной карте — это столбцы таблицы People	56
Завершение таблицы	59
Перенос в базу данных с карточек	60
Источник данных соединит форму с базой	62
Добавление элементов, связанных с базой данных	64
Хорошие программы понятны интуитивно	66
Тестирование	68
Как превратить ВАШЕ приложение в приложение для ВСЕХ	69
Передаем приложение пользователям	70
Работа НЕ окончена: проверим процесс установки	71
Управляющее данными приложение готово	72

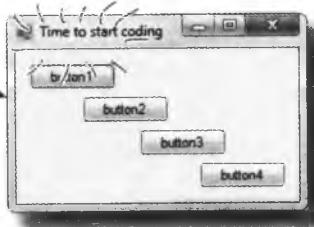
это Всё Го ЛиШЬ Код

# 2

## Под покровом

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

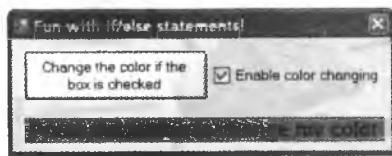
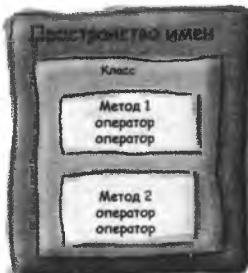
Постройте такую форму



Для каждой программы вы определяете свое пространство имен, отдавая код от классов .NET Framework.

Класс содержит фрагменты 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.



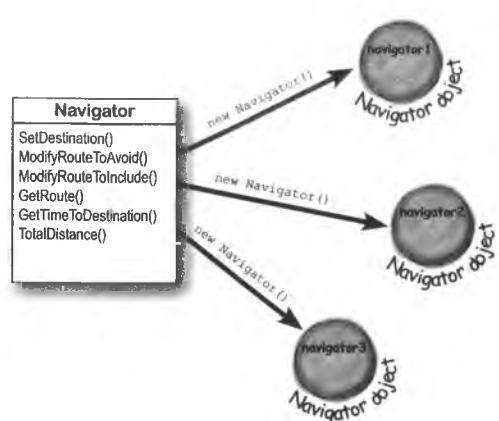
Когда вы делаете это...	74
...ИСР делает это.	75
Как рождаются программы	76
Писать код помогает ИСР	78
Любые действия ведут к изменению кода	80
Структура программы	82
Начало работы программы	84
Редактирование точки входа	86
Классы могут принадлежать одному пространству имен	91
Что такое переменные	92
Знакомые математические символы	94
Наблюдение за переменными в процессе отладки	95
Циклы	97
Перейдем к практике	98
Оператор выбора	99
Проверка условий	100

объекты, по порядку стройся!

# 3

## 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.



Определяя класс, вы определяете и его методы, точно также как чертеж определяет внешний вид дома.



Что думает Майк о своих проблемах	116
Проблема Майка с точки зрения навигационной системы	117
Методы прокладки и редактирования маршрутов	118
Построим программу с использованием классов	119
Идея Майка	121
Объекты как способ решения проблемы	122
Возьмите класс и постройте объект	123
Экземпляры	124
Простое решение!	125
Поля	130
Создаем экземпляры!	131
Спасибо за память	132
Что происходит в памяти программы	133
Значимые имена	134
Структура классов	136
Выбор структуры класса при помощи диаграммы	138
Помогите парням	142
Проект «Парни»	143
Форма для взаимодействия с кодом	144
Более простые способы присвоения начальных значений	147

трицы и ссылки

## 4

**10:00 утра. Куда подевались наши данные?**

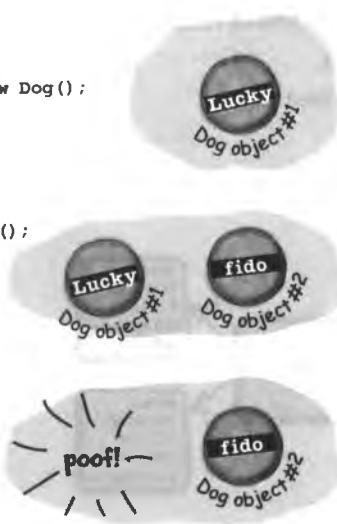
**Без данных программы бесполезны.** Взяв информацию от пользователей, вы производите новую информацию, чтобы вернуть ее им же. Практически все в программировании связано с обработкой данных тем или иным способом. В этой главе вы познакомитесь с используемыми в C# типами данных, узнаете методы работы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

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

Тип переменной определяет, какие данные она может сохранять	154
Наглядное представление переменных	156
10 литров в 5-литровой банке	157
Приведение типов	158
Автоматическая коррекция слишком больших значений	159
Иногда приведение типов происходит автоматически	160
Аргументы метода должны быть совместимы с типами параметров	161
Комбинация с оператором =	166
Объекты тоже используют переменные	167
Переменные ссылочного типа	168
Ссылки подобны маркерам	169
При отсутствии ссылок объект превращается в мусор	170
Побочные эффекты множественных ссылок	171
Две ссылки это ДВА способа редактировать данные объекта	176
Особый случай: массивы	177
Массив	177
Массив может состоять из ссылочных переменных	178
Добро пожаловать на распродажу сэндвичей от Джо!	179
Ссылки позволяют объектам обращаться друг к другу	181
Сюда объекты еще не отправлялись	182
Играем в печатную машинку	187

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

```
lucky = null;
```



## инкапсуляция

## 5

**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	196
Как происходит оценка	197
Кэтлин тестирует программу	202
Каждый вариант нужно было считать отдельно	204
Неправильное использование объектов	206
Форма	206
Инкапсуляция как управление доступом к данным	207
Доступ к методам и полям класса	208
НА САМОМ ЛИ ДЕЛЕ защищено поле realName?	209
Закрытые поля и методы доступны только изнутри класса	210
Инкапсуляция сохраняет данные нетронутыми	218
Инкапсуляция при помощи свойств	219
Приложение для проверки класса Farmer	220
Автоматические свойства	221
Редактируем множитель feed	222
Конструктор	223



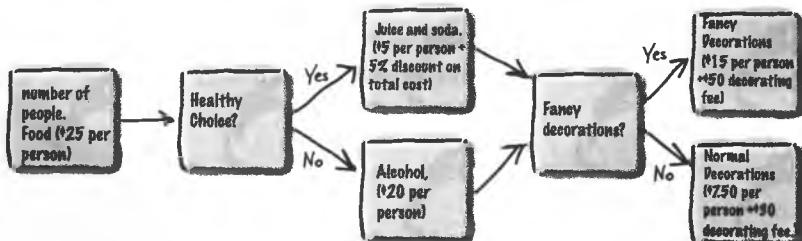
ciaAgent



kgbAgent



mi5Agent



## наследование

## 6

**Генеалогическое древо объектов**

**Иногда люди хотят быть похожим 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	232
Нам нужен класс BirthdayParty	233
Планировщик мероприятий, версия 2.0	234
Дополнительный взнос за мероприятия с большим количеством гостей	241
Наследование	242
Модель классов: от общего к частному	243
Симулятор зоопарка	244
Иерархия классов	248
Производные классы расширяют базовый	249
Синтаксис наследования	250
При наследовании поля свойства и методы базового класса добавляются к производному...	253
Перекрытие методов	254
Вместо базового класса можно взять один из производных	255
Производный класс умеет скрывать методы	262
Ключевые слова override и virtual	264
Ключевое слово base	266
Если в базовом классе присутствует конструктор, он должен остаться и в производном классе	267
Теперь мы готовы завершить программу для Кэтлин!	268
Система управления ульем	273
Построение основ	274
Совершенствуем систему управления улем при помощи наследования	282

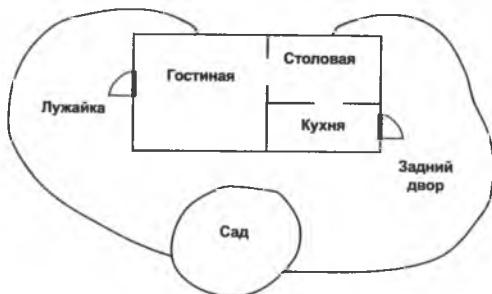
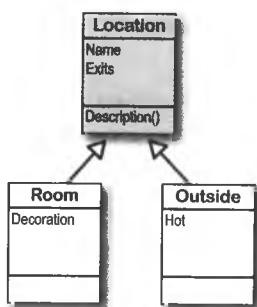


# 7 интерфейсы и абстрактные классы

## 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.

<b>Наследование</b>	
	Вернемся к нашим пчелам 284
	Классы для различных типов пчел 285
<b>Абстракция</b>	
	Интерфейсы 286
	Классы, реализующие интерфейсы, должны включать ВСЕ методы интерфейсов 288
<b>Инкапсуляция</b>	
	Учимся работать с интерфейсами 290
	Ссылки на интерфейс 292
<b>Полиморфизм</b>	
	Ссылка на интерфейс аналогична ссылке на объект 293
	Интерфейсы и наследование 295
	RoboBee 4000 функционирует без меда 296
	Кофеварка относится к Приборам 298
	Восходящее приведение 299
	Нисходящее приведение 300
	Нисходящее и восходящее приведение интерфейсов 301
	Модификаторы доступа 305
	Изменение видимости при помощи модификаторов доступа 306
	Классы, для которых недопустимо создание экземпляров 309
	Абстрактный класс. Перепутье между классом и интерфейсом 310
	Как уже было сказано, недопустимо создавать экземпляры некоторых классов 312
	Абстрактный метод не имеет тела 313
	Смертельный ромбом! 318
	Различные формы объекта 321



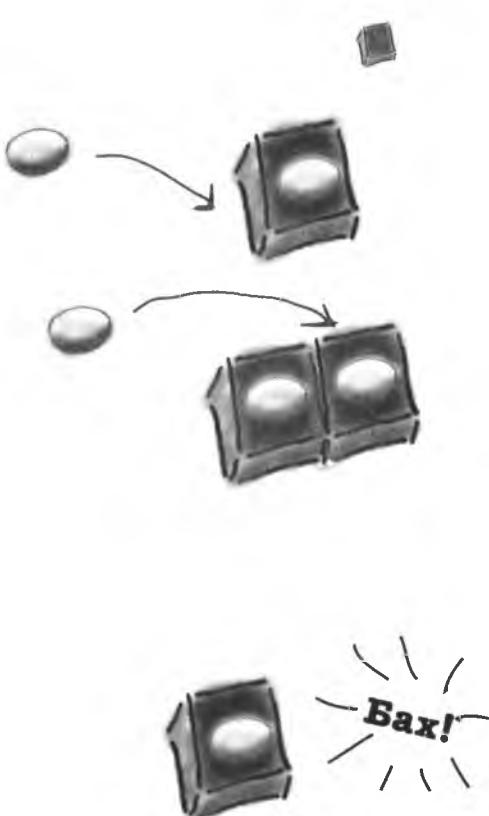
## перечисления и коллекции

## 8

**Большие объемы данных**

**Пришла беда — отворяй ворота.** В реальном мире данные, как правило, не хранятся маленькими кусочками. Данные приходят 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 идеи программирования, оставив задачу отслеживания данных коллекциям.

Категории данных не всегда можно сохранять в переменных типа string	340
Перечисления	341
Присвоим числам имена	342
Создать колоду карт можно было при помощи массива...	345
Проблемы работы с массивами	346
Коллекции	347
Коллекции List	348
Динамическое изменение размеров	351
Обобщенные коллекции	352
Инициализаторы коллекций	356
Коллекция уток	357
Сортировка элементов коллекции	358
Интерфейс IComparable<Duck>	359
Способы сортировки	360
Создадим экземпляр объекта-компаратора	361
Сложные схемы сравнения	362
Перекрытие метода ToString()	365
Обновим цикл foreach	366
Интерфейс IEnumerable<T>	367
Восходящее приведение с помощью IEnumerable	368
Создание перегруженных методов	369
Словари	375
Дополнительные типы коллекций...	389
Звенья следуют в порядке их поступления	390
Звенья следуют в порядке, обратном порядку их поступления	391

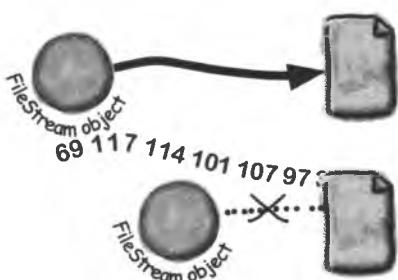
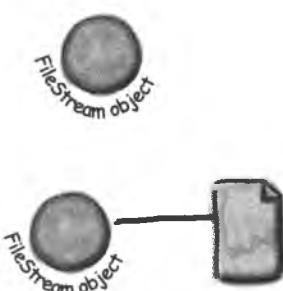


## 9

чтение и запись файлов

**Сохрани массив байтов и спаси мир****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	396
Различные потоки для различных данных	397
Объект <code>FileStream</code>	398
Трехшаговая процедура записи текста в файл	399
Чтение и запись при помощи двух объектов	403
Встроенные объекты для вызова стандартных окон диалога	407
Встроенные классы <code>File</code> и <code>Directory</code>	410
Открытие и сохранение файлов при помощи окон диалога	413
Интерфейс <code>IDisposable</code>	415
Операторы <code>using</code> как средство избежать системных ошибок	416
Запись файлов сопровождается принятием решений	422
Оператор <code>switch</code>	423
Чтение и запись информации о картах в файл	424
Чтение карт из файла	425
Сериализации подвергается не только сам объект...	429
Сериализация позволяет читать и записывать объект целиком	430
Сериализуем и десериализуем колоду карт	432
.NET использует Unicode для хранения символов и текста	435
Перемещение данных внутри массива байтов	436
Класс <code>BinaryWriter</code>	437
Чтение и запись сериализованных файлов вручную	439
Найдите отличия и отредактируйте файлы	440
Сложности работы с двоичными файлами	441
Программа для создания дампа	442
Достаточно <code>StreamReader</code> и <code>StreamWriter</code>	443
Чтение байтов из потока	444

# 10

## обработка исключений

### Борьба с огнем надоедает

**Программисты не должны уподобляться пожарным.**

Вы усердно работали, штудировали технические 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	450
Объект Exception	454
Код Брайана работает не так, как предполагалось	456
Исключения наследуют от объекта Exception	458
Работа с отладчиком	459
Поиск ошибки в приложении Excuse Manager с помощью отладчика	460
А код все равно не работает...	463
Ключевые слова try и catch	465
Вызов сомнительного метода	466
Результаты применения ключевых слов try/catch	468
Ключевое слово finally	470
Получения информации о проблеме	475
Обработка исключений разных типов	476
Один класс создает исключение, другой его обрабатывает	477
Исключение OutOfHoney для пчел	478
Оператор using как комбинация операторов try и finally	481
Избегаем исключений при помощи интерфейса IDisposable	482
Наихудший вариант блока catch	484
Временные решения	485
Краткие принципы обработки исключений	486
Наконец Брайан получил свой отдых...	489

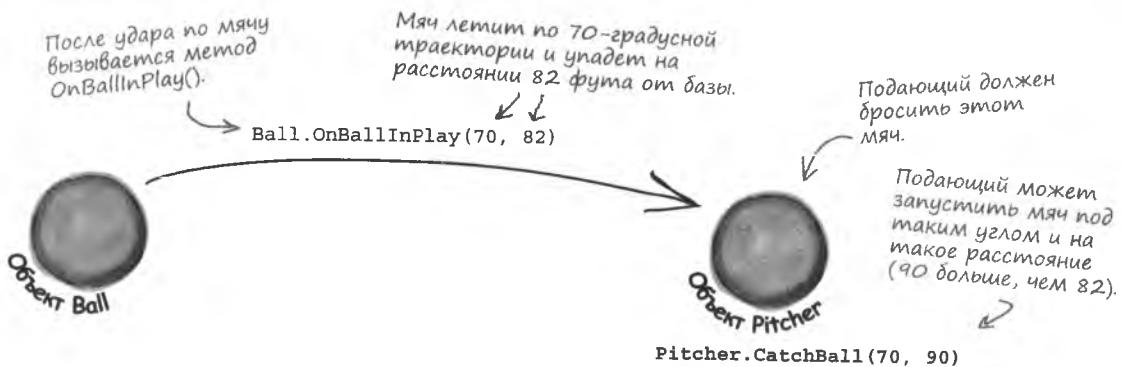
## события и Делегаты

## 11

**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?	492
Как объекты узнают, что произошло?	492
События	493
Один объект инициирует событие, другой реагирует на него	494
Обработка события	495
Соединим все вместе	496
Автоматическое создание обработчиков событий	500
Обобщенный EventHandler	506
Все формы используют события	507
Несколько обработчиков одного события	508
Связь между издателями и подписчиками	510
Делегат замещает методы	511
Делегаты в действии	512
Объект может подписаться на событие...	515
Обратный вызов	516
Обратный вызов как способ работы с делегатами	518



# 12

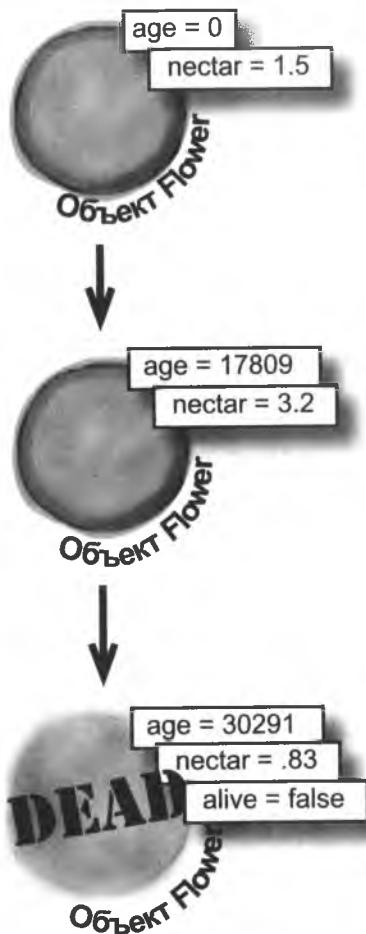
обзор и предварительные результаты

## Знания, сила и построение приложений

**Знания нужно применять на практике.**

Пока вы не начнете писать работающий код, вряд ли можно быть уверенным, что вы на самом деле усвоили сложные понятия C#. Сейчас мы займемся применением на практике полученных ранее знаний. Кроме того, вы найдете предварительные сведения о понятиях, которые будут рассматриваться в следующих главах. Мы начнем построение действительно сложного приложения, чтобы закрепить ваши навыки.

## Жизнь и смерть цветка



Вы прошли долгий путь	526
Вы стали пчеловодом	527
Архитектура симулятора улья	528
Построение симулятора улья	529
Жизнь и смерть цветка	533
Класс Bee	534
Программисты против бездомных пчел	538
Для жизни улья нужен мед	538
Построение класса Hive	542
Метод Go() для улья	543
Все готово для класса World	544
Система, основанная на кадрах	545
Код для класса World	546
Поведение пчел	552
Основная форма	554
Получение статистики	555
Таймеры	556
Работа с группами пчел	564
Коллекция коллекционирует... ДАННЫЕ	565
LINQ упрощает работу с коллекциями и базами данных	567
Тестирование (вторая попытка)	568
Последние штрихи: Open и Save	569

# 13

## Элементы управления и графические фрагменты

### Наводим красоту

**Иногда управление графикой приходится брать в свои руки.**

До этого момента визуальный аспект наших приложений был отдан на откуп элементам управления. Но иногда этого недостаточно, например, если вы хотите анимировать изображение. Вам предстоит научиться создавать свои элементы управления для .NET, применять двойную буферизацию и даже рисовать непосредственно на формах.

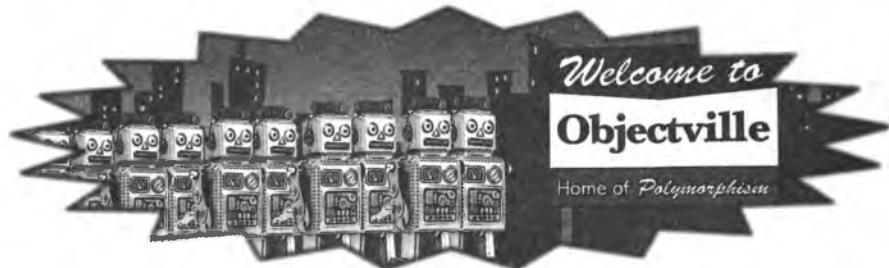


Элементы управления как средство взаимодействия с программой	574
Элементы управления – это тоже объекты	575
Анимируем симулятор улья	576
Визуализатор	578
Формы для визуализации	579
Первый анимированный элемент управления	583
Базовый класс PictureBox	584
Кнопка добавления элемента BeeControl	586
Удаление дочерних элементов управления	587
Класс UserControl	588
Поместим на форму анимированных пчел	590
Формы Hive и Field	592
Построение визуализатора	593
Соединим основную форму с формами HiveForm и FieldForm	596
Проблемы с производительностью	600
Объект Graphics	602
Объект Bitmap	603
Пространство имен System.Drawing	604
Знакомство с GDI+	605
Рисуем на форме	606
Решение проблемы с прозрачностью	611
Событие Paint	612
Перерисовка форм и элементов управления	615
Двойная буферизация	618
Вывод на печать	624
Окно предварительного просмотра и окно диалога Print	625

# 14

## КАПИТАН ВЕЛИКОЛЕПНЫЙ СМЕРТЬ ОБЪЕКТА

Метод завершения объекта	638
Когда запускает метод завершения объекта	639
Явное и неявное высвобождение ресурсов	640
Возможные проблемы	642
Сериализуем объект в методе Dispose()	643
Структура напоминает объект...	647
...но объектом не является	647
Значения копируются, а ссылки присваиваются	648
Структуры – это значимые, а объекты – ссылочные типы	649
Сравнение стека и кучи	651
Необязательные параметры	656
Типы, допускающие значение null	657
Типы, допускающие значение null, увеличивают robustность программы	658
Что осталось от Великолепного	661
Сравнение	661
Методы расширения	662
Расширяем фундаментальный тип: string	664



# 15 LINQ

## Управляем данными

**Этот мир управляет 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.

Простой проект...	670
...но сначала нужно собрать данные	671
List<StarbuzzData>	671
Сбор данных из разных источников	672
Коллекции .NET уже настроены под LINQ	673
Простой способ сделать запрос	674
Сложные запросы	675
Универсальность LINQ	678
Группировка результатов запроса	683
Сгруппируем результаты Джимми	684
Предложение join	687
List<Comic>	687
LINQ Sequence	687
List<Purchases>	687
Джимми изрядно сэкономил	688
LINQ Sequence	689
List<StarbuzzData>	689
Соединение LINQ с базой данных SQL	690
Соединим Starbuzz и Objectville	694



Как работать с этой книGой

## 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.

## Кому эта книга не подойдет?

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

- ① Навевает ли на вас скуку и нервозность мысль 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.



[Заметка 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?»

## 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 – сохранению того, что действительно *важно*. Мозг не считает нужным сохранять скучную информацию. Она не проходит фильтр, отсекающий «очевидно несущественное».

Но как же мозг *узнает*, что важно? Представьте, что вы выехали 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 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».

Ваш мозг считает, что ЭТО важно.



Ваш мозг полагает, что ЭТО можно не запоминать.

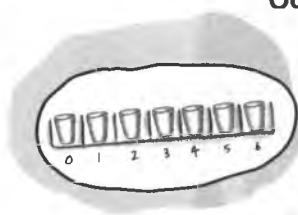


Замечательно.  
Еще 696 сухих,  
скучных страниц.

## 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 что-то большее, чем простой текст на странице. 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 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 сопротивления. 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 стоит тысячи слов. А когда текст комбинируется с графикой, мы внедряем текст прямо в рисунки, потому что мозг при этом работает эффективнее.

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

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

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

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

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

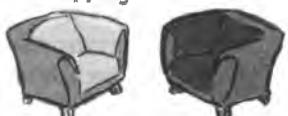
Определся класс, ви опре-  
делите его методы, тщно  
также как чертеж опреде-  
ляет внешний вид дома.

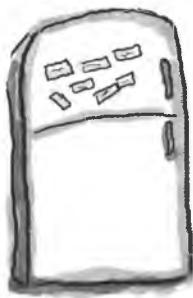


### КЛЮЧЕВЫЕ МОМЕНТЫ



### Беседа у камина





*Вырежьте и прикрепите  
на холодильник.*

## Что можете сделать Вы, чтобы заставить свой мозг поиноваться

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

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

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

### ② Выполняйте упражнения, делайте заметки.

Мы включили упражнения в книгу, но выполнять их за вас не собираемся. И не *разглядывайте* упражнения. **Берите карандаш и пишите.** Физические действия *во время* 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.

Мозг лучше всего работает в условиях высокой влажности. 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 получше запомнить, произнесите вслух. А еще лучше – попробуйте объяснить кому-нибудь другому. Вы будете быстрее усваивать материал и, возможно, откроете для себя что-то новое.

### ⑦ Прислушивайтесь к своему мозгу.

Следите за тем, когда ваш мозг начинает уставать. Если вы начинаете 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

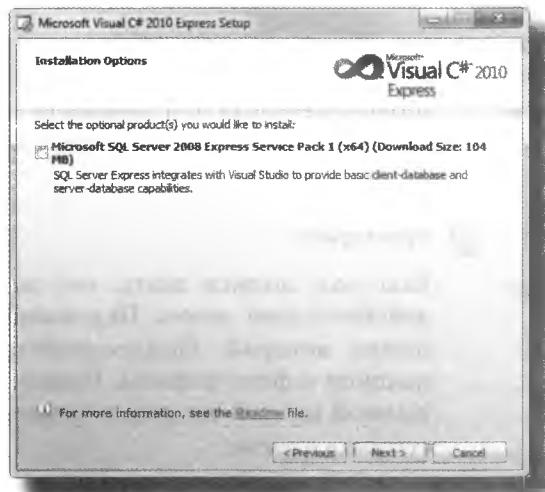
Эта книга написана при помощи Visual C# 2010 Express Edition, работающей с C# 4.0 и .NET Framework 4.0. Именно в этом приложении были сняты все скриншоты, поэтому мы рекомендуем установить себе именно его. Если у вас установлена Visual Studio 2010 Professional, Premium, Ultimate или Test Professional-версия, картинки будут отличаться. Где необходимо, это будет оговариваться. Бесплатно скачать версию Express можно с сайта Microsoft.

### Настройка VISUAL STUDIO 2010 EXPRESS EDITION

- Скачать и установить версию Visual C# 2010 Express довольно легко. Дистрибутив программы находится здесь:

<http://www.microsoft.com/express/downloads/>

Для выполнения упражнений из этой книги вам не потребуются никакие дополнительные параметры установки, но вы можете воспользоваться ими, если хотите.



Если у вас установлена более старая версия Visual Studio, C# или .NET Framework, некоторые примеры из книги могут не сработать.

- Скачайте и установите Visual C# 2010 Express Edition. Убедитесь, что установка завершена. На вашем компьютере окажутся интегрированная среда разработки (которую вам предстоит изучить), .NET Framework 4.0 и другие инструменты.
- После установки в меню Start появится новая строка: **Microsoft Visual C# 2010 Express Edition**. Это команда для вызова ИСР. Вы полностью готовы к работе.

# Информация

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

## Упражнения обязательны к выполнению.

Упражнения и задачи являются частью основного содержания книги, а не дополнительным материалом. Некоторые помогают запомнить новую информацию, некоторые – лучше понять ее, а некоторые – научиться применять ее на практике. Необязательными являются только «Ребусы в бассейне», но следует помнить, что они хорошо развивают логическое мышление.

## 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. В 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.

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

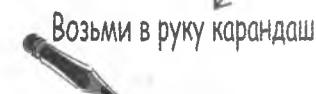
## Упражнения «Мозговой штурм» не имеют ответов.

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.

Для наглядного представления сложных понятий в книге используются диаграммы.



Выполняйте все упражнения этого раздела.



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



А этим значком помечены дополнительные упражнения для любителей логических задач.



## Технические рецензенты

Лиза Кельнер



Крис Барроус



Особая благодарность Крису за его озарения и удивительно полезные замечания.

Ник Паладино



Дэвид действительно нам помог, показав некоторые приемы работы с ИСР.

Дэвид Стерлинг



Здесь нет фотографий Джо Албахари, Джей Хиллера, Аяма Синга, Теодоры Петер Ричи, Билла Метельски, Энди Паркера, Вейна Бредни, Дэйва Мэрдока, Бриджит-Жули Ландерс. Особая благодарность Джону Скиту за его предложения по улучшению первого издания!

При работе над этой книгой мы столкнулись с массой неточностей, вопросов, проблем, опечаток и даже ужасных математических ошибок. Впрочем... все было не так ужасно, как можно подумать. Но мы все равно очень благодарны за ту работу, которую выполнили наши технические рецензенты. Книга вышла бы в свет с ошибками (парочка из которых довольно серьезны), если бы у нас не было самой лучшей на свете команды...

Прежде всего хотелось бы сказать большое спасибо Крису Барроусу и Дэвиду Стерлингу за техническое сопровождение. Также хотелось бы поблагодарить Лизу Кельнер — это уже шестой обзор, который она для нас делает, и именно она сделала конечный продукт настолько читабельным. Спасибо, Лиза! И особая благодарность Нику Паладино.

**Крис Барроус** — разработчик компании Microsoft из команды компиляторов C#. Его специализация — дизайн и реализация различных свойств языка C# 4.0.

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

**Николас Паладино** был одним из наиболее ценных специалистов Microsoft, занимавшихся .NET/C#. Он имеет 13-летний опыт в программировании, особенно в позиционировании технологий Microsoft.

# Благодарности

## Редакторы

Мы хотим поблагодарить наших редакторов **Бретта Маклафлина и Кортни Нэш** за работу над этой книгой. Бретт был автором большинства рассказов, а идея главы 14 принадлежит ему полностью. Спасибо!



Бретт Маклафлин



Кортни Нэш

## Команда издательства O'Reilly



Лу Бэрр



Сандерс Клейнфелд

Лу – замечательный дизайнер, постигший все секреты своей профессии. Она провела незабываемые часы и сделала для книги потрясающие графические фрагменты. Если вы видите в книге нечто фантастическое – благодарите ее (и ее умение работать с InDesign). Все комиксы являются ее творением. Спасибо огромное, Лу! С тобой потрясающе приятно работать!

В O'Reilly очень много сотрудников, которых мы хотели бы поблагодарить, надеемся, что никого не забыли. Особое спасибо выпускающему редактору **Рашель Монаган** и индексатору **Люси Хаскинс**, **Эмили Квил** за отменную корректуру, **Рону Билодо**, который не жалел своего времени на экспертные заключения, **Сандерсу Клейнфелду** за предложение проверить все еще один раз – всем, кто помог подготовить эту книгу к печати в рекордно короткие сроки. Мы любим **Мэри Треслер** и ждем случая поработать с ней снова! Жмем руку нашим друзьям и редакторам **Энди Ораму** и **Майку Хендриксону**. И за то, что вы сейчас читаете эту книгу, нужно поблагодарить лучшую команду рекламистов: **Марси Хенон**, **Сару Пейтон**, **Мэри Ротман**, **Джессику Байд**, **Кетрин Барет** и остальных сотрудников из города Севастополь, штат Калифорния.

1 Эффективность с C#

# Визуальные приложения за 10 минут

Не волнуйся, мама. На C# ты будешь программировать так быстро, что тушеное мясо больше никогда не подгорит.



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#

C# и ИСР Visual Studio облегчают и ускоряют процесс написания кода.

### Задачи, которые за Вас решает ИСР

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

ИСР, или Интегрированная Среда Разработки, — это программа для редактирования кода, управления файлами и публикации проектов.

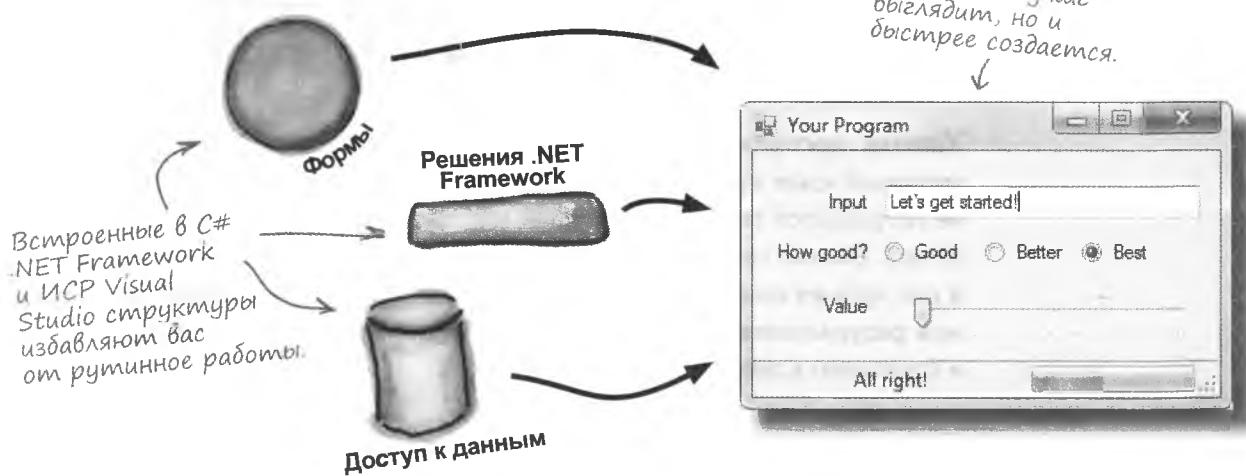
```
private void InitializeComponent()
{
    this.button1 = new System.Windows.Forms.Button();
    // ...
    this.button1.Location = new System.Drawing.Point(106, 56);
    this.button1.Name = "button1";
    this.button1.Size = new System.Drawing.Size(75, 23);
    this.button1.TabIndex = 0;
    this.button1.Text = "button1";
    this.button1.UseVisualStyleBackColor = true;
    this.button1.Click += new System.EventHandler(this.button1_Click);
}
// ...
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 добавляем 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.

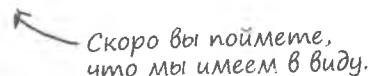


## C#, ИСР Visual Studio многое упрощают

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

- ➊ **Быстро создавать приложения.** Программировать на C# очень просто. Это мощный, легко осваиваемый язык, а Visual Studio позволяет автоматизировать большинство процессов.
- ➋ **Разрабатывать красивый пользовательский интерфейс.** Инструмент Form Designer в Visual Studio превращает создание великолепного пользовательского интерфейса в одну из самых увлекательных задач при разработке приложений на C#. Вам больше не потребуется тратить часы на написание графических элементов с нуля.
- ➌ **Создавать базы данных и взаимодействовать с ними.** ИСР снабжена простым интерфейсом для создания баз данных, которые затем легко интегрируются в SQL Server Compact Edition и другие популярные приложения.
- ➍ **Фокусироваться на решении РЕАЛЬНЫХ проблем.** За конечный результат работы, разумеется, отвечаете вы и только вы. Но ИСР позволяет концентрироваться на глобальных вещах, взяв на себя:
  - ★ слежение за всеми проектами;
  - ★ упрощенное редактирование кода;
  - ★ отслеживание графики, аудиофайлов, значков и прочих ресурсов;
  - ★ управление базами данных и взаимодействие с ними.

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



## Избавьте директора от бумаг

В объективильской фирме по производству бумаги появился новый исполнительный директор. Он любит пешие прогулки, кофе и природу... и с целью сохранения лесов решил перейти на безбумажный документооборот. На выходные он уехал кататься на лыжах в Аспен, а вам приказал к понедельнику написать программу для хранения контактной информации. Если этого не сделать... хм... вы составите компанию предыдущему директору вашей фирмы, который ищет новую работу.



## Перед началом работы выясните, что именно нужно пользователю

Перед началом работы над **любым** приложением нужно понять, кто будет его использовать и каким ожиданиям оно должно соответствовать.

- ① Директор хочет, чтобы программа работала не только на его офисном компьютере, но и на его ноутбуке. Значит, вам потребуется инсталлятор, который позволит поместить нужные файлы на любой компьютер.



Так как приложение должно работать не только в офисе, но и на ноутбуке директора, без установщика не обойтись.

- ② Другим сотрудникам фирмы также понадобится доступ к контактной информации. Как же в противном случае они сформируют списки рассылки, чтобы сделать фирму лидером по продаже бумаги.



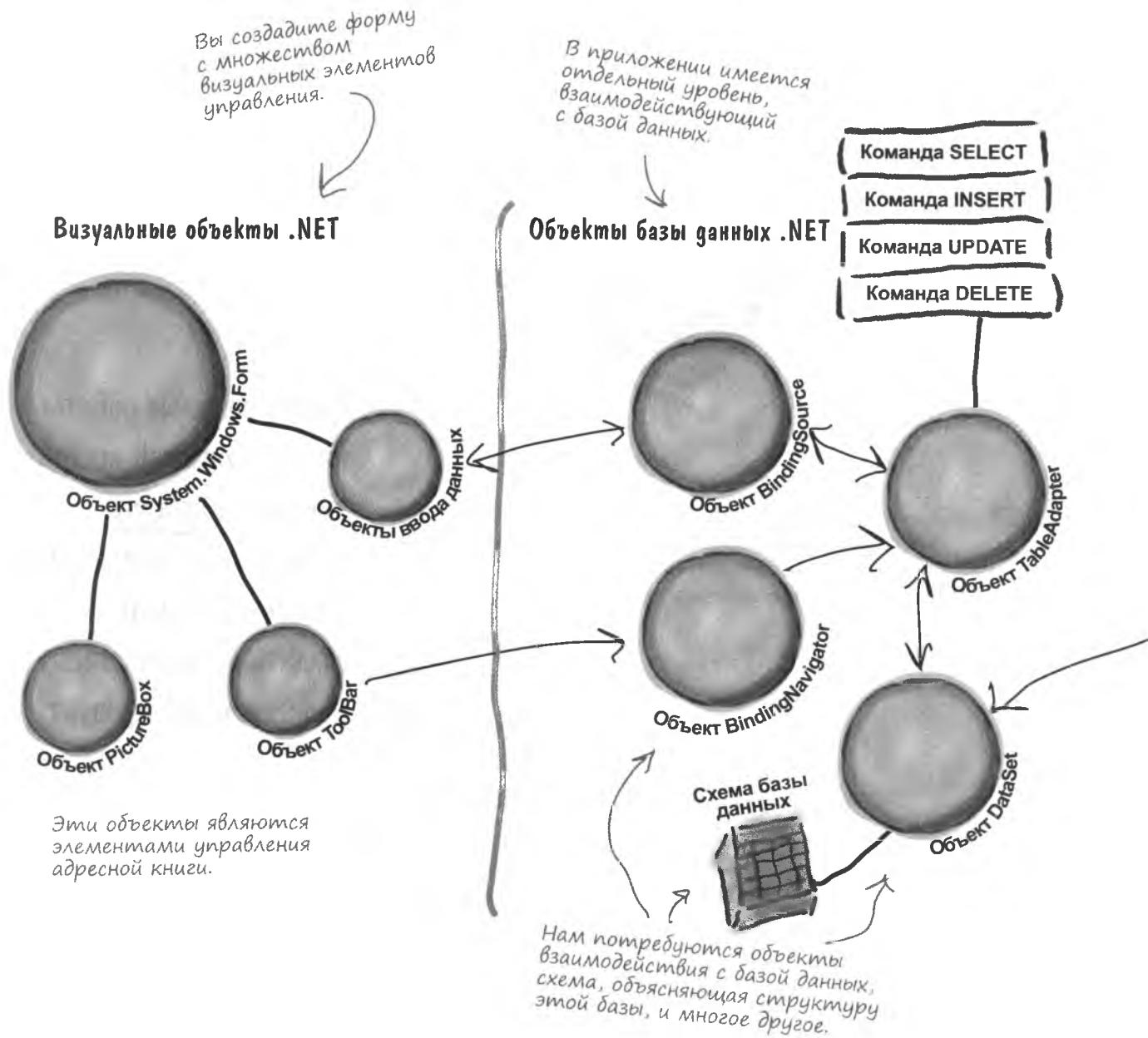
Мы уже знаем, что Visual C# упрощает работу с базами данных. Поместив контактную информацию в базу, вы обеспечите доступ к ней любому сотруднику.

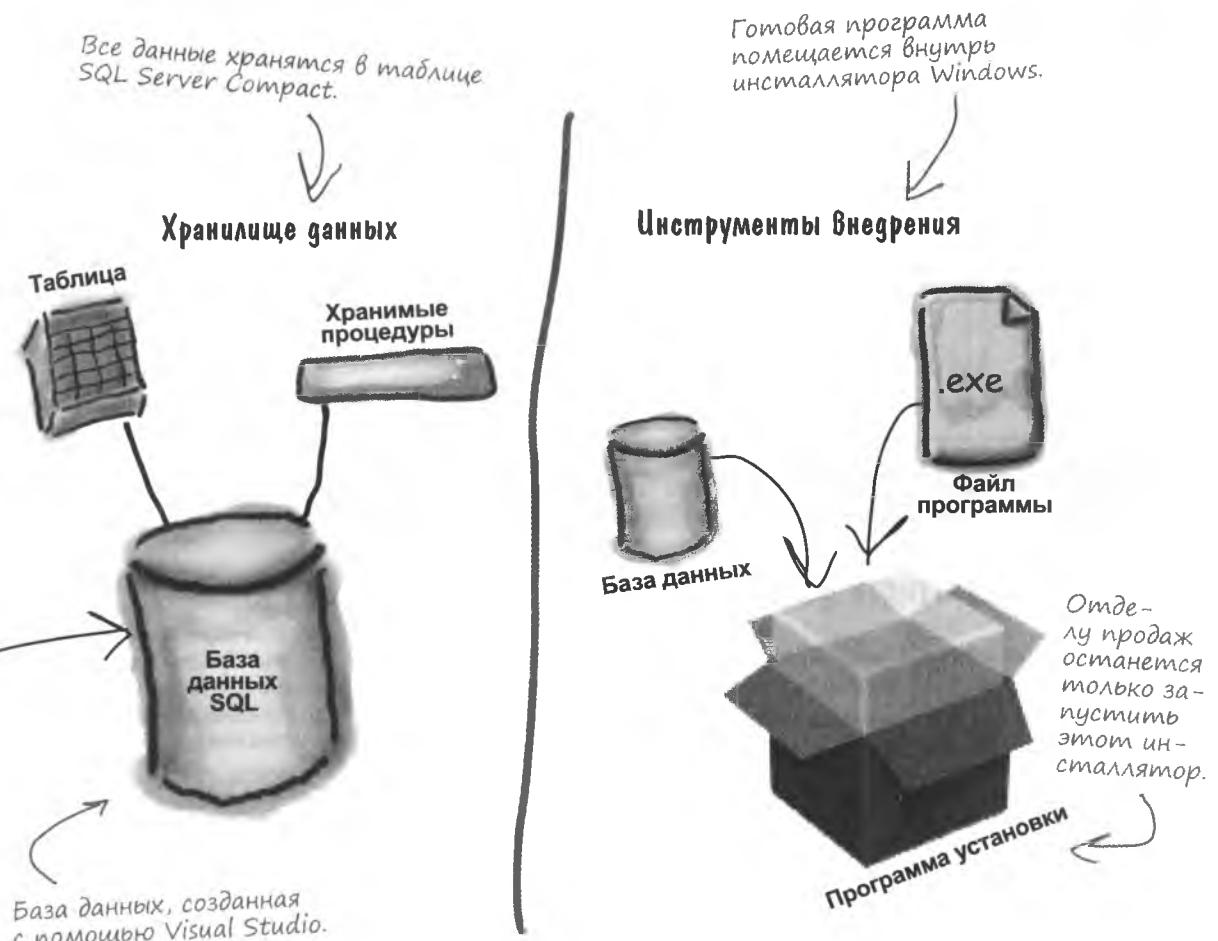
Перед началом работы над программой всегда думайте о нуждах конечных пользователей, только так можно создать по-настоящему качественный продукт!

## Что мы собираемся сделать

Нам потребуется приложение с графическим интерфейсом пользователя, объекты, взаимодействующие с базой данных, собственно база данных и программа установки. Вся эта огромная с виду работа будет проделана к концу этой главы.

Вот структура нашей будущей программы:

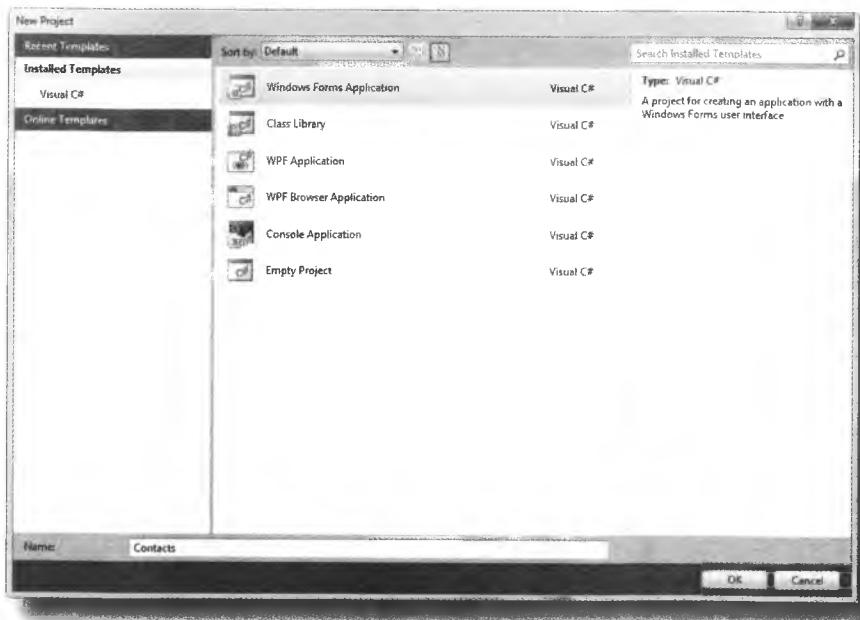




**приступим**

## Это вы делаете в Visual Studio

Запустите Visual Studio, если вы еще это не сделали. Пропустите начальную страницу и выберите в меню **File** команду **New Project**. В открывшемся окне диалога **New Project** выберите тип проекта **Windows Forms Application**, а в текстовое поле **Name** в нижней части окна введите название проекта **Contacts**.



В вашей ИСР все может выглядеть по-другому.

На рисунке показан вид окна **New Project** в **Visual Studio 2010 Express Edition**. В **Professional** или **Team Foundation edition** оно может выглядеть по-другому. Но суть его от этого не меняется.

## А это Visual Studio делает за Вас

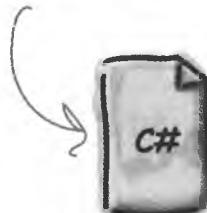
В момент сохранения проекта ИСР создает файлы **Form1.cs**, **Form1.Designer.cs** и **Program.cs**. Они добавляются в окно **Solution Explorer** и по умолчанию сохраняются в папке **My Documents\Visual Studio 2010\Projects\Contacts\**.

Этот файл содержит код, определяющий поведение формы.



**Form1.cs**

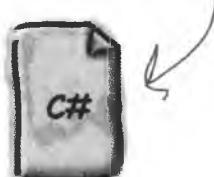
Этот файл содержит код, запускающий программу и показывающий форму.



**Program.cs**

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

Код, определяющий форму и ее объекты.



**Form1.Designer.cs**

Все эти файлы Visual Studio создает автоматически.

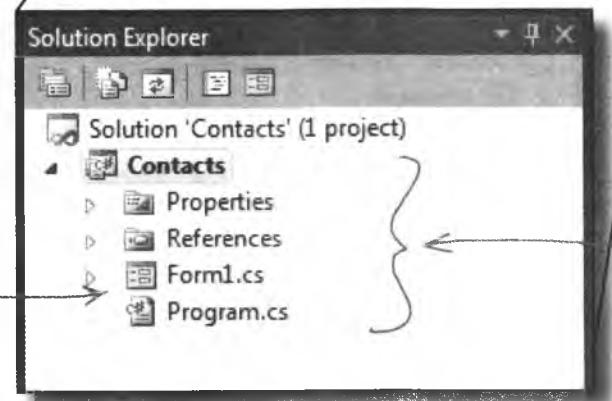
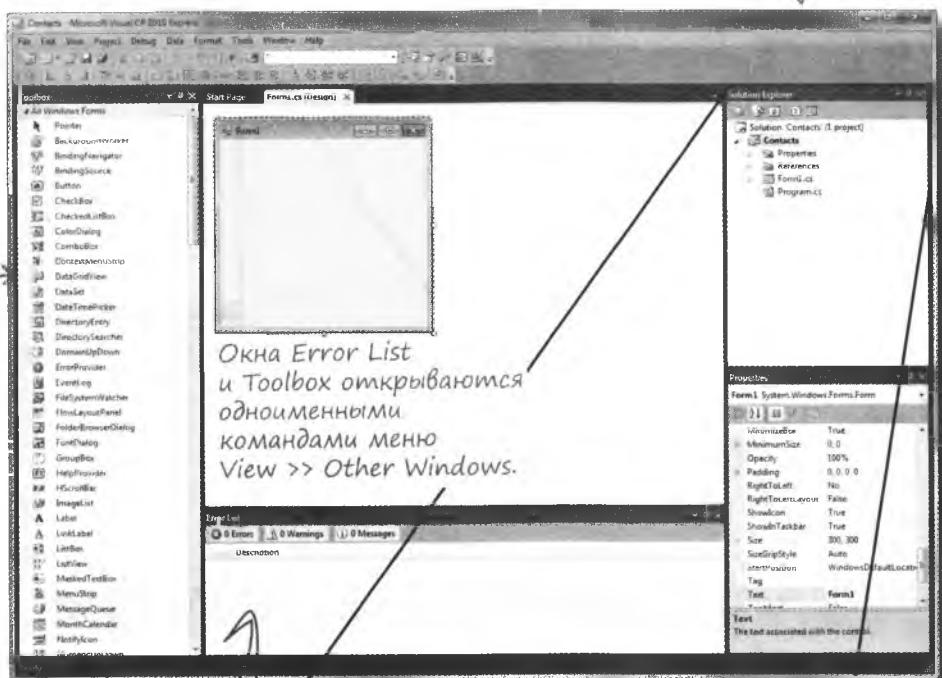
# Возьми в руку карандаш

Ниже показан возможный вид экрана. Вы должны понимать назначение большинства окон и файлов. Убедитесь в наличии панелей Toolbox и Error List, **вызываемых одноименными командами меню View >> Other Windows**. В пустые строки впишите назначение каждой части ИСР, как показано в примере.

Здесь выбираются инструменты для работы.

Если Ваша ИСР выглядит по-другому, воспользуйтесь командой Window>>Reset Window Layout.

Это окно было увеличено, чтобы дать вам больше пространства.



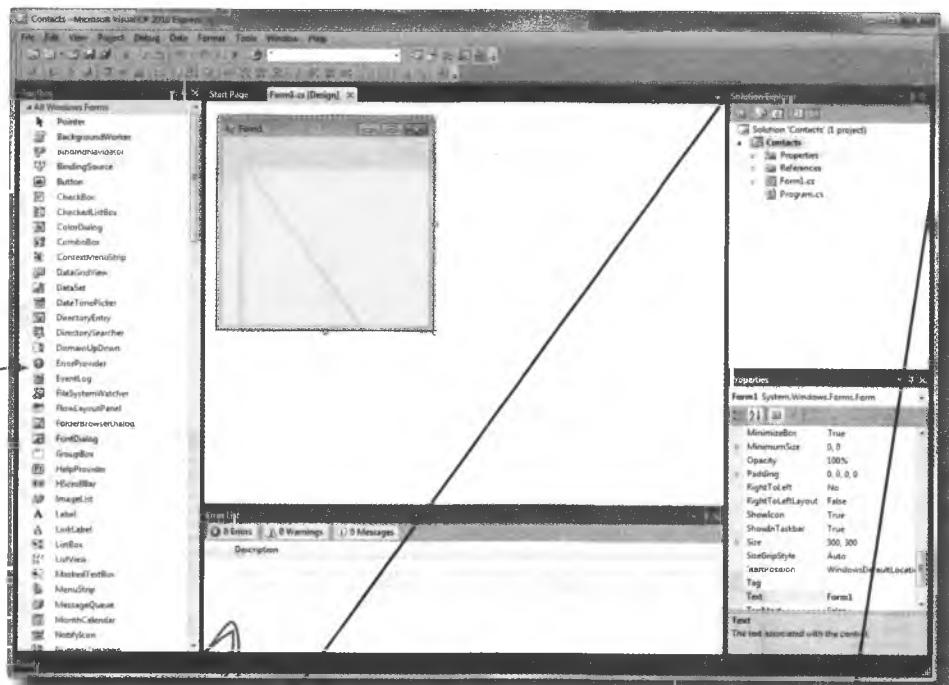
## Возьми в руку карандаш

### Решение

Здесь выбираются инструменты для работы.

Здесь собраны визуальные элементы управления, которые можно перетащить на форму.

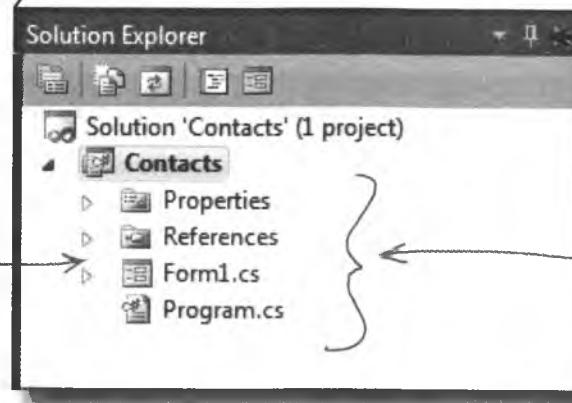
Итак, вы описали назначение различных частей ИСР Visual Studio C#. Здесь вы можете сравнить написанное с правильными вариантами ответа.



В окне Error List отображаются вкравшиеся в код ошибки.

Имена автоматически созданных файлов Form1.cs и Program.cs появляются в окне Solution Explorer.

Щелчок на этой кнопке включает и отключает функцию автоматического сворачивания окна. У панели Toolbox она включена по умолчанию.



часто  
Задаваемые  
Вопросы

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

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

**В:** Я создал новый проект, но не нашел его в папке Projects, вложенной в папку My Documents. Почему?

**О:** Новые проекты Visual Studio 2010 Express помещает в папку Local Settings\Application Data\Temporary Projects. При первом сохранении вам предлагается задать новое имя и поместить в папку My Documents\Visual Studio 2010\Projects. При открытии нового проекта или закрытии временного, вам предлагается сохранить или удалить последний. (**ПРИМЕЧАНИЕ:** Отличные от Express версии Visual Studio не используют папку временных проектов, а помещают файлы непосредственно в папку Projects!)

**В:** Что делать с ненужным кодом, автоматически созданным ИСР?

**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 ИСР.

**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:** Могу ли я переименовывать файлы, созданный ИСР?

**О:** Конечно. Новому проекту ИСР по умолчанию присваивает имя Form1 (Form1.cs, Form1.Designer.cs и Form1.resx). Но его можно поменять в окне Solution Explorer. По умолчанию имена файлов совпадают с именем формы. Как можно видеть в окне Properties, их изменение не влияет на имя формы. Последнее меняется изменением поля (Name) в окне Properties. Имя файла при этом остается тем же.

Для файлов, форм (и других частей программы) можно выбирать произвольные имена. Выбор значимых имен облегчает работу с программой, но об этом пока можно не думать.

**В:** Окно ИСР 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.**

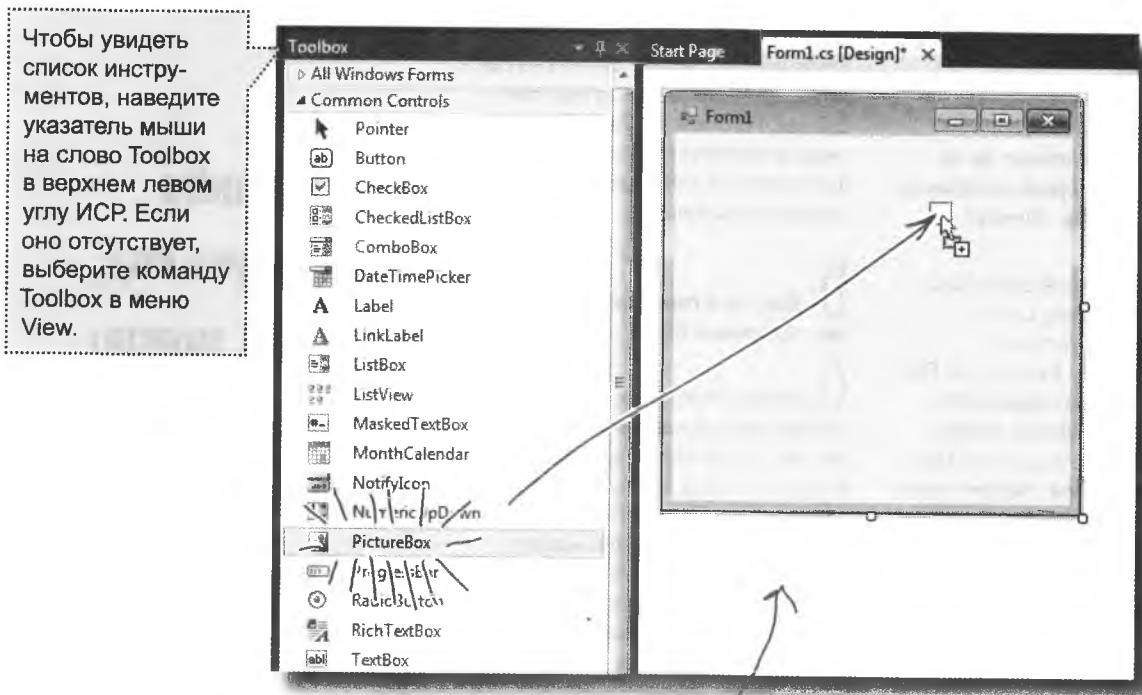
**Но только вы  
отвечаете за  
корректную работу  
этой программы.**

## Создаем пользовательский интерфейс

Добавление элементов управления в ИСР Visual Studio осуществляется методом Drag&Drop (перетащить и бросить). Рассмотрим процесс добавления логотипа к форме:

### 1 Элемент управления PictureBox.

Перетащите на форму элемент управления PictureBox из окна Toolbox. ИСР при этом добавит в файл Form1.Designer.cs код нового изображения.



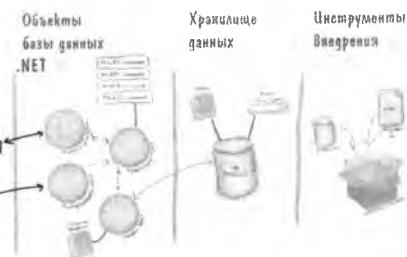
Редактирование свойств элементов управления формы меняет код в файле Form1.Designer.cs.



Form1.Designer.cs

Если вы не имеете представления о разработке пользовательских интерфейсов, ничего страшного.

О разработке интерфейсов мы поговорим позднее. На данном этапе сосредоточьтесь на *поведении* создаваемых элементов управления.

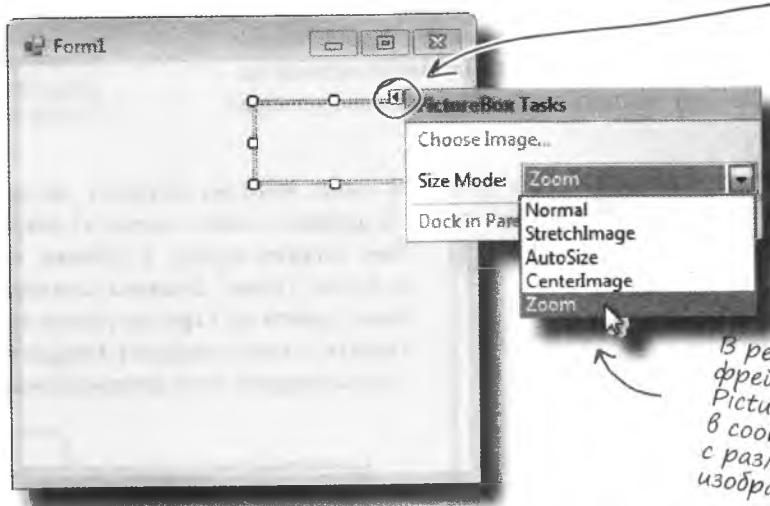


2

**Режим Zoom для элемента PictureBox.**

Доступ к свойствам элемента управления осуществляется щелчком на маленькой черной стрелке. Для свойства Size элемента PictureBox выберите тип Zoom:

Задать размер элемента можно в окне Properties. Щелчок на черной стрелке дает быстрый доступ к общим свойствам элементов.



Щелчок на маленькой черной стрелке дает доступ к свойствам элемента управления.

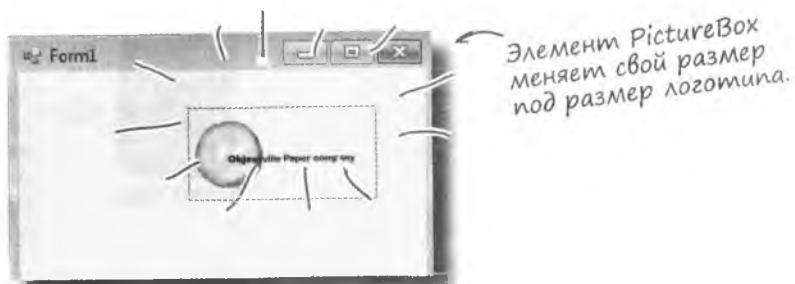
В режиме Zoom фрейм элемента PictureBox меняется в соответствии с размерами выбранного изображения.

Щелчок на строке Choose Image открывает окно диалога Select Resource, в котором можно выбрать картинку.

3

**Загрузка логотипа фирмы по производству бумаги.**

Архив с логотипом находится на сайте ([http://www.headfirstlabs.com/books/hfcsharp/HFCsharp\\_graphics\\_ch01.zip](http://www.headfirstlabs.com/books/hfcsharp/HFCsharp_graphics_ch01.zip)). Сохраните его на жестком диске своего компьютера. Щелкните на черной стрелке элемента PictureBox и выберите в появившемся меню команду Choose Image. Откроется окно диалога Select Resources. Установите переключатель в положение Local Resource и щелкните на кнопке Import... Остается только указать путь к файлу с картинкой.

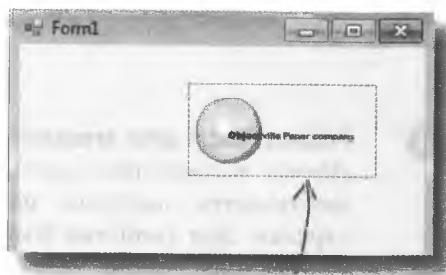


Элемент PictureBox меняет свой размер под размер логотипа.

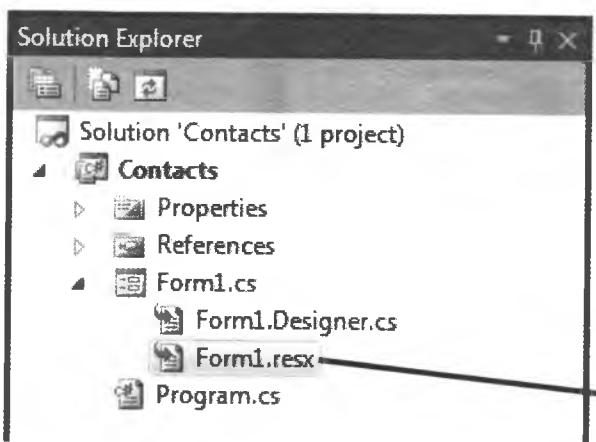
## Visual Studio, за сценой

Каждое ваше действие Visual Studio сопровождается *написанием кода*. В момент загрузки логотипа Visual Studio создала ресурс и связала его с вашим приложением. **Ресурсом** называется любой графический или звуковой файл, значок или другие данные, подгружаемые к приложению. Логотип интегрировался в программу, и при открытии на другом компьютере элемент управления PictureBox сможет его использовать.

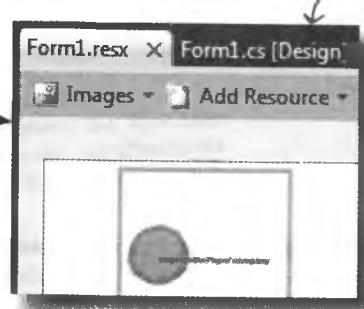
В момент перетаскивания элемента управления PictureBox на форму для хранения данного ресурса и его связи с проектом автоматически был создан файл Form1.resx. Двойной щелчок на имени этого файла позволит увидеть импортированное изображение.



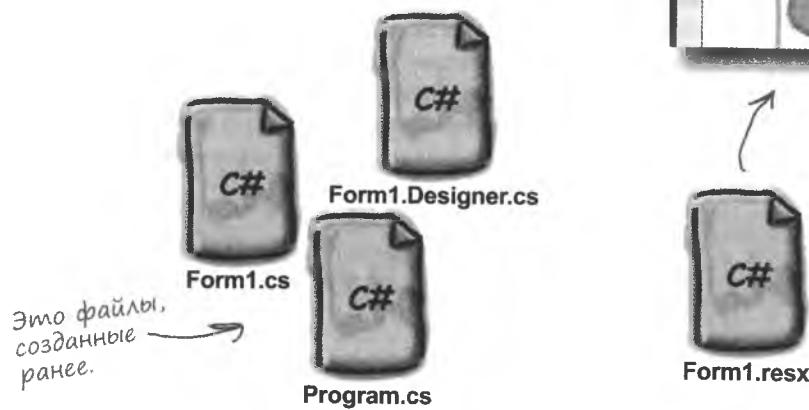
Логотип стал ресурсом приложения Contact List.



В окне Solution Explorer щелкните на квадратике со знаком «плюс» слева от имени файла Form1.cs. Это откроет доступ к файлам: Form1.Designer.cs и Form1.resx. Дважды щелкните на файле Form1.resx, затем на стрелке рядом со словом Strings и выберите в меню вариант Images (или нажмите Ctrl-2), чтобы увидеть импортированный логотип.



Вторая кнопка Import в окне Select Resource помещает логотип в папку Resources окна Solution Explorer. Чтобы исправить ситуацию, установите переключатель Local Resource и повторно импортируйте файл.



Этот файл ИСР создала при импорте изображения. Сюда помещаются любые ресурсы, связанные с формой.

## Редактирование кода

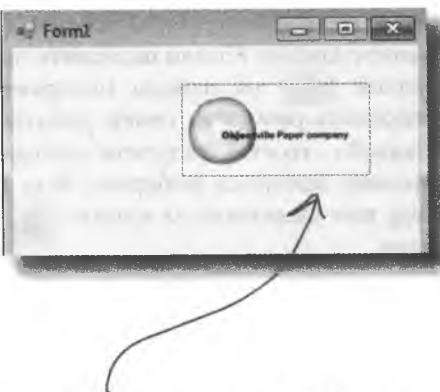
Иногда в автоматически созданный код требуется внести изменения. Рассмотрим эту процедуру на примере добавления к логотипу всплывающего текста.

При редактировании формы двойной щелчок на любом элементе управления автоматически добавляет в проект код. Дважды щелкните на фрейме PictureBox и вы увидите код, который выполняется при каждом щелчке на данном элементе управления. Этот код должен выглядеть примерно так:

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

    private void pictureBox1_Click(object sender, EventArgs e)
    {
        MessageBox.Show("Contact List 1.0.\nWritten by: Your Name", "About");
    }
}
```

После двойного щелчка на элементе PictureBox курсор окажется здесь. При вводе кода игнорируйте всплывающие окна.



Двойной щелчок на элементе PictureBox создает метод, который будет запускаться при каждом щелчке пользователя на логотипе.

Название метода показывает, что он запускается щелчком на элементе PictureBox.

Введите этот код. Он отвечает за появление всплывающего сообщения с текстом About.

Введя код, щелкните на кнопке Save панели инструментов ИСР или выберите команду Save в меню File.

### часто задаваемые вопросы

**В:** Что такое метод?

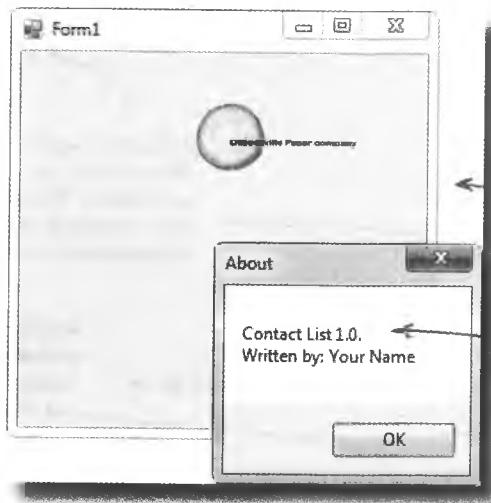
**О:** Методом называется именованный блок кода. Более подробную информацию вы получите в главе 2.

**В:** Зачем нужен символ \n?

**О:** Это знак переноса строки. Он показывает, что Contact List 1.0 будет написан на строке, а Written by: записывается с новой строки.

## Первый запуск приложения

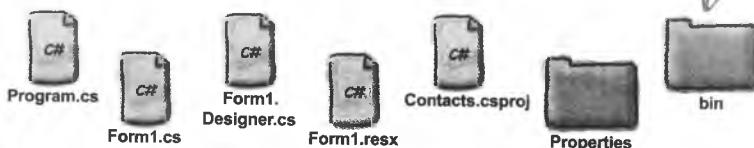
Нажмите кнопку F5 или щелкните на кнопке с зеленой стрелкой (▶) на панели инструментов, чтобы протестировать результат своей работы. (Это называется «отладкой», то есть запуском программы в ИСР.) Для остановки процесса выберите Stop Debugging в меню Debug или щелкните на кнопке [ ] на панели инструментов.



Все три кнопки работают, и вам не потребовалось писать для этого код.

## Где же мои файлы?

При запуске программы Visual Studio копирует файлы в папку My Documents\Visual Studio 2010\Projects\Contacts\Contacts\bin\debug. Дважды щелкните на созданном ИСР файле .exe, и программа начнет работать.



Это не ошибка. Существуют два уровня папок. Файлы с кодом C# находятся во внутренних папках.

C# превращает вашу программу в исполняемый файл, находящийся в папке debug.

## часто задаваемые вопросы

**В:** В моей ИСР кнопка с зеленой стрелкой называется Debug (Начать отладку). Она запустит программу?

**О:** Конечно. Щелчок на ней приведет к выполнению программы внутри ИСР, что в данный момент вам, собственно, и требуется. О процедуре отладки мы поговорим позднее.

**В:** Я не вижу на панели инструментов кнопку Stop Debugging. Что делать?

**О:** Эта кнопка появляется только при запущенной на выполнение программе. Запустите приложение, и вы увидите, как она появится.

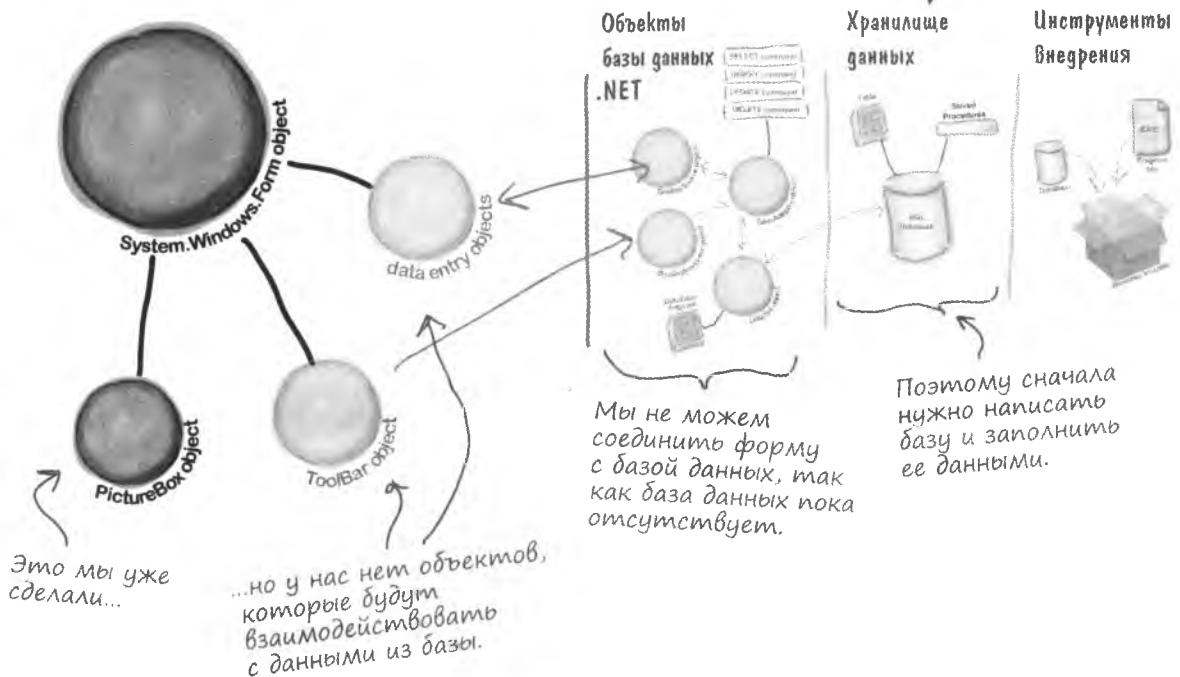
## Вот что уже сделано

Итак, мы создали форму и объект PictureBox, щелчок на котором вызывает диалоговое окно с текстом. Теперь нужно добавить поля с информацией: имена и телефоны из списка контактов.

Эту информацию мы сохраним в базе данных. Visual Studio соединит с ней поля, то есть нам не потребуется вручную писать код для доступа к этой базе. Сейчас мы займемся ее написанием, то есть перейдем от раздела *Визуальные объекты .NET* прямо к разделу *Запись данных*.



## Визуальные объекты .NET



**Visual Studio автоматически генерирует для вас код, соединяющий форму с базой данных, но СНАЧАЛА нужно создать саму базу.**

сохраните, чтобы потом использовать

## Для хранения информации нужна база данных

Код, связывающий форму с данными ИСР, генерируется автоматически, так что нам нужно всего лишь:

Убедитесь, что  
процесс отладки  
остановлен.

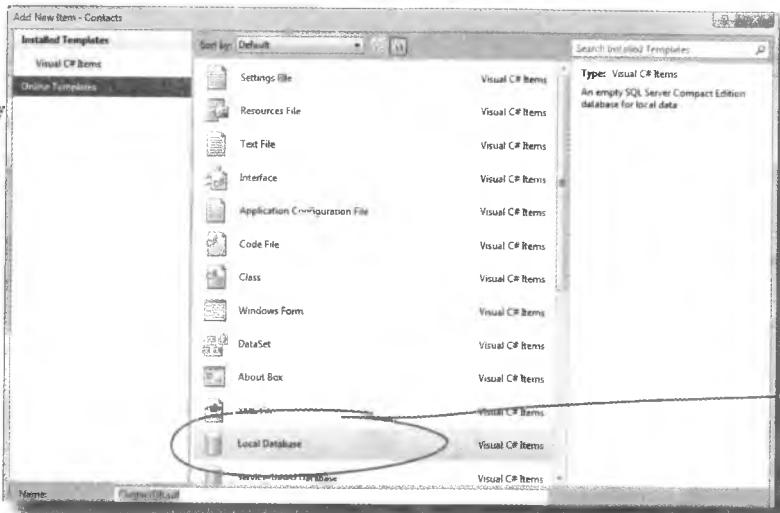
### 2 Добавить к проекту базу данных SQL.

В окне Solution Explorer щелкните правой кнопкой мыши на проекте Contacts, выберите команду Add>>New Item. В открывшемся окне выделите строчку Local database, а в поле Name укажите имя файла ContactDB.sdf.

Это файл  
нашей новой  
базы данных.



Выбрав Local Database, вы создадите файл SQL Server Compact Edition, в котором и будет храниться наша база.



Файлы базы  
данных SQL  
Server Compact  
Edition имеют  
расширение SDF.

### 2 Щелкните на кнопке Add в нижней части окна Add New Item.



Будьте  
осторожны!

### 3 Закройте мастер Data Source Configuration. Щелкните на кнопке Cancel, так как в данный момент мы не будем останавливаться на конфигурации источника данных.

В версиях приложения, отличных от Express, вместо окна Database Explorer фигурирует окно Server Explorer

Окно Server Explorer в версиях Professional и Team Foundation editions сохранив всю функциональность окна Database Explorer, позволяет просматривать сетевые данные.

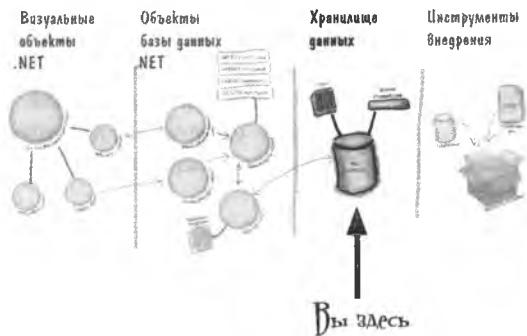
### 4 Посмотрите на полученный результат.

В списке файлов окна Solution Explorer должен появиться файл ContactDB.sdf. Дважды щелкните на его имени и посмотрите на левую часть экрана. Вместо окна Toolbox там окно Database Explorer.

## База данных, созданная ИСР

**База данных SQL** – это способ систематизированного хранения информации. ИСР дает вам все инструменты для управления как данными, так и самой базой.

Данные в базе хранятся в виде таблиц. Столбцы разделяют данные по категориям (например, «имя» или «номер телефона»), а строки содержат информацию о каждом контакте.



## Язык SQL

SQL расшифровывается как **Structured Query Language** – язык структурированных запросов. Он предназначен для доступа к данным в базах и имеет собственный синтаксис и структуру. Код SQL представлен в виде **операторов (statements)** и **запросов (queries)**, а также **хранимых процедур (stored procedures)**. ИСР генерирует за вас операторы SQL и хранимые процедуры, позволяя создаваемому вами приложению осуществлять доступ к данным базы.

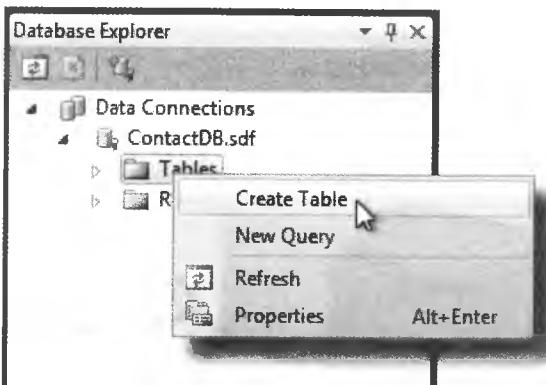


## Создание таблицы для списка контактов

Теперь базу нужно наполнить данными. Обычно используется представление в виде таблицы, поэтому сначала создадим таблицу **People**, в которую и поместим контактную информацию:

### 1 Добавление таблицы

Щелкните правой кнопкой мыши на строчке Tables в окне Database Explorer и выберите команду Create Table.



Теперь к таблице нужно добавить столбцы. Начнем со столбца ContactID, чтобы присвоить каждому контакту уникальный номер.

### 2 Создание столбца ContactID

Введите в поле Column Name название ContactID, а в раскрывающемся списке Data Type выберите вариант Int. В списке Allow Nulls должен быть выбран вариант No.

В списке Primary Key выберите вариант Yes. То есть именно это поле будет использоваться в качестве уникального идентификатора записи.

Column Name	Data Type	Length	Allow Nulls	Unique	Primary Key
ContactID	int	4	No	Yes	Yes

Добавьте столбец ContactID с типом данных int. В списке Allow Nulls выберите вариант No, а в списках Unique и Primary Key вариант Yes.

### часто задаваемые вопросы

В: Что же такое столбец?

О: Столбец — это одно из полей таблицы. В таблице People можно создать столбцы FirstName и LastName, относящиеся к типу String или Date или Bool.

В: Зачем нам столбец ContactID?

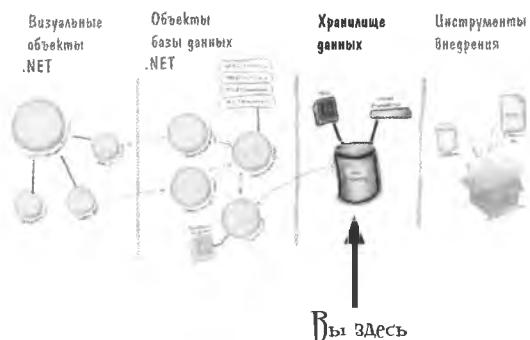
О: Он позволит присвоить уникальный номер каждой записи в таблицах базы. Так как мы сохраняем контактные данные отдельных людей, имеет смысл их пронумеровать.

В: Что такое тип данных Int?

О: Int — это сокращение от Integer (целое число). То есть в столбце ContactID могут содержаться только целые числа.

В: Информации слишком много, должен ли я во всем этом разбираться?

О: Ничего страшного, если на данном этапе что-то остается непонятным. Пока что ваша основная задача — получение навыков работы с ИСР Visual Studio. Достаточно научиться подготавливать формы и запускать программу (Если же вы хотите узнать больше о базах данных, почитайте, например, книгу «Изучаем SQL»).



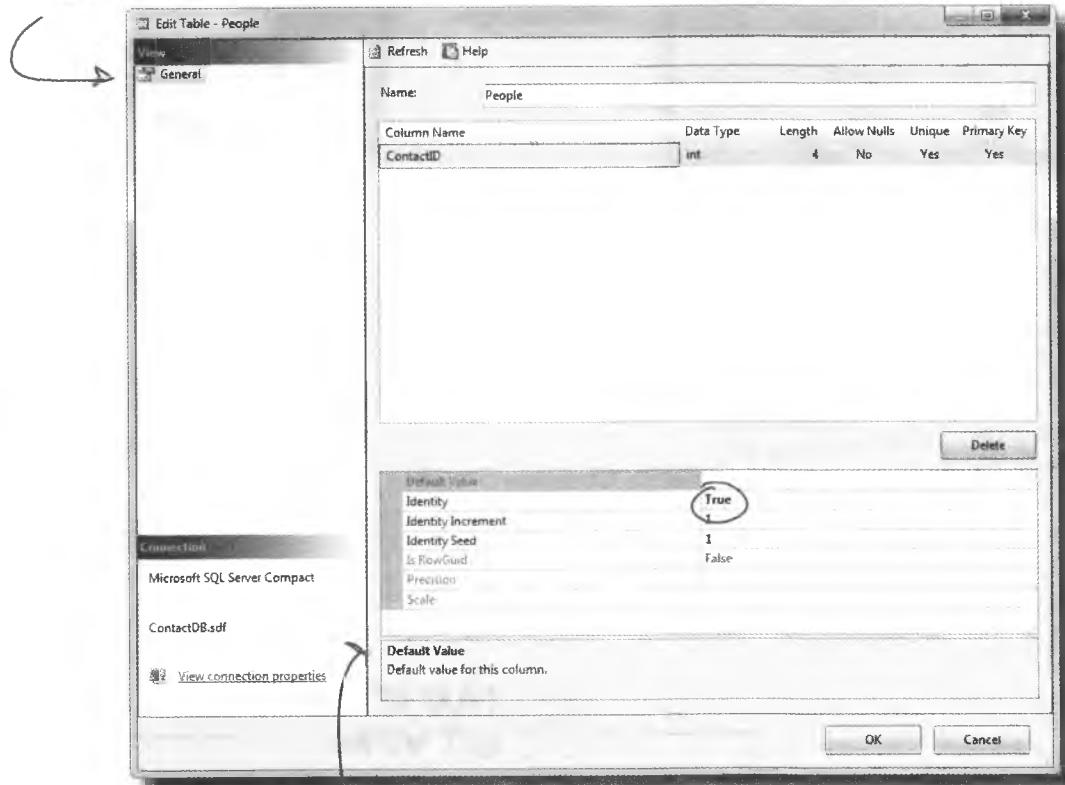
### 3 Генерация идентификационных номеров

Информация в столбце ContactID будет использоваться только базой данных, нам она не требуется, ее имеет смысл генерировать автоматически.

В окне свойств присвойте свойству Identity значение True, чтобы сделать столбец ContactID идентификатором для всей таблицы.

В поле Name в верхней части окна введите значение People.

В этом окне задается таблица и свойства сохраняемых в ней данных.



В раскрывающемся списке справа от поля Identity нужно указать значение True, чтобы сделать столбец ContactID идентификатором записей таблицы.

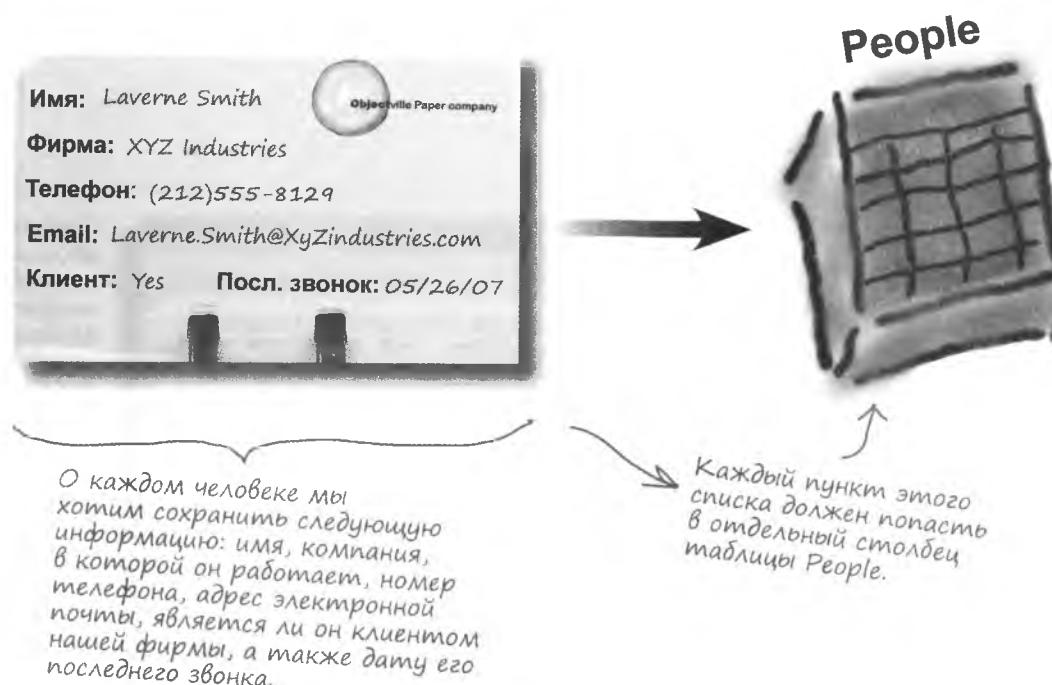
Первичный ключ является уникальным идентификатором записи, поэтому он обязательно должен быть задан.

При добавлении новой записи поле ContactID будет обновляться автоматически.

оформим в таблицу

## Поля в контактной карте — это столбцы таблицы People

Теперь, когда у нас есть первичный ключ таблицы, нужно определить все остальные поля. Каждое поле должно стать столбцом в таблице People.



### МОЗГОВОЙ ШТУРМ

Какие проблемы могут возникнуть, если одному человеку сопоставить несколько записей?

# Кто и что делает?

Таблица People готова и для нее уже задан первичный ключ, осталось добавить столбцы для всех полей с данными. Укажите, к какому типу должны принадлежать данные, и выберите для них верное описание.

Название столбца	Тип данных	Описание
Последний звонок	datetime	Дата и время
Имя	bit	Логический тип true/false
Идентификатор	nvarchar(100)	Строка из букв, цифр и других символов максимальной длиной 100 знаков
Является ли клиентом?	datetime	Целое число

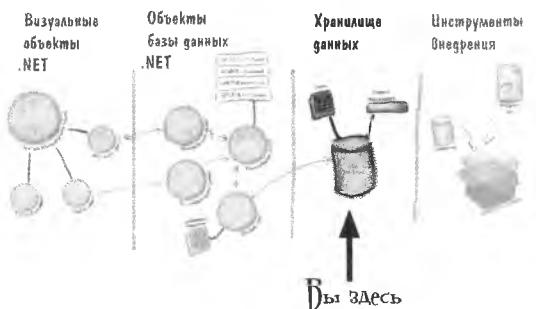
## \*КТО И ЧТО ДЕЛАЕТ?

Проверьте, правильно ли вы сопоставили типы данных и описания таблицы People.

Название столбца	Тип данных	Описание
Последний звонок	int	Целое число
Имя	bit	Логический тип true/false
Идентификатор	nVarchar(100)	Строка из букв, цифр и других символов максимальной длиной 100 знаков
Является ли клиентом?	datetime	Дата и время

## Завершение таблицы

По аналогии со столбцом ContactID добавьте еще пять столбцов. Вот как должно выглядеть окно Edit Table после завершения работы:



**Name:** People

Column Name	Data Type	Length	Allow Nulls	Unique	Primary Key
ContactID	int	4	No	Yes	Yes
Name	nvarchar	100	Yes	No	No
Company	nvarchar	100	Yes	No	No
Telephone	nvarchar	100	Yes	No	No
Email	nvarchar	100	Yes	No	No
Client	bit	1	Yes	No	No
LastCall	datetime	8	Yes	No	No

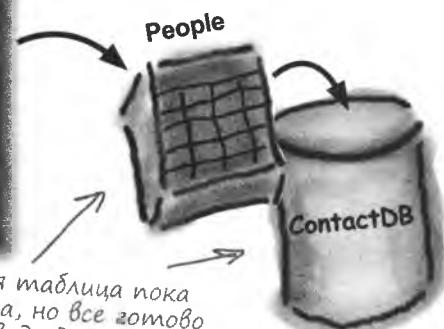
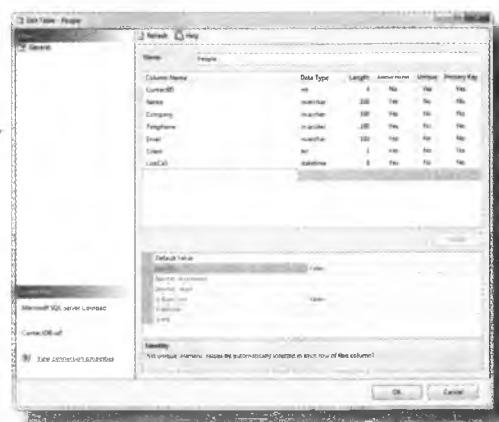
Поля типа Bit при-  
нимаю-  
т значения True  
или False и  
могут быт-  
и предст-  
авлены в виде  
переключа-  
теля.

Если па-  
раметр  
Allow Nulls  
имеет  
значение  
No, столбец  
не может  
оставляться  
пустым.

Информация о клиен-  
те может быть  
неполной,  
поэтому  
выбираем  
значение Yes.

Щелкните на кнопке OK для сохранения таблицы. Она будет добавлена к вашей базе данных.

Щелчок на кнопке  
OK добавит  
таблицу People  
к базе данных.

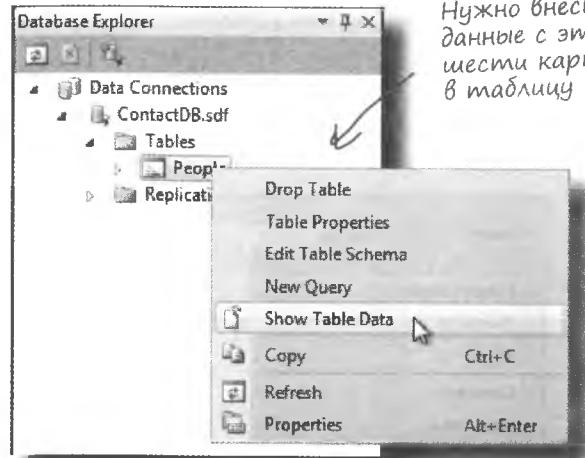


Новая таблица пока  
пуста, но все готово  
для ввода данных!

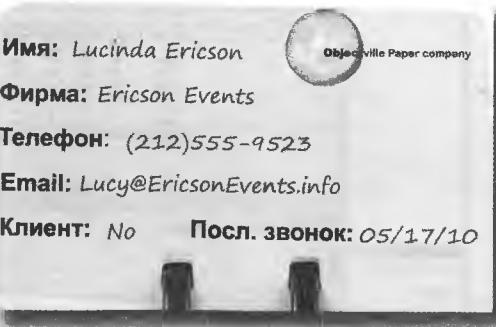
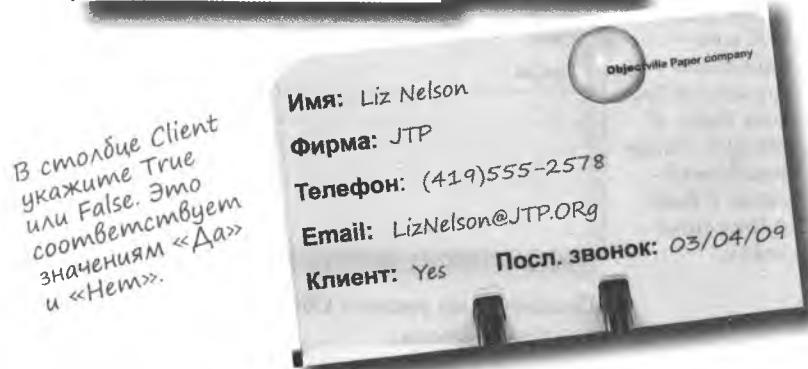
## Перенос в базу данных с карточек

Все готово для ввода данных в базу. Для начала возьмем шесть карточек с контактной информацией от вашего босса.

- ① Щелкните на квадратике со знаком «плюс» слева от строчки Tables в окне Database Explorer (или Server Explorer), затем щелкните правой кнопкой мыши на имени таблицы People и выберите команду Show Table Data в появившемся меню.
- ② В центральном окне появится сетка таблицы, куда нужно ввести данные с карточек. (Изначально во всех ячейках стоит значение null, введите вместо него свою информацию.) Столбец ContactID будет заполнен автоматически.



Нужно внести  
данные с этих  
шести карточек  
в таблицу People.



**Имя:** Matt Franks

Objectville Paper company

**Фирма:** XYZ Industries

**Телефон:** (212)555-8125

**Email:** Matt.Franks@XyZindustries.com

**Клиент:** Yes      **Посл. звонок:** 05/26/10

Фирма находится в США, поэтому директор записывает даты как 05/26/10, что означает 26 мая 2010. Вы можете вводить их в другом формате.

**Имя:** Sarah Kalter

Objectville Paper company

**Фирма:** Kalter, Riddle and Stoft

**Телефон:** (614)555-5641

**Email:** Sarah@KRS.org

**Клиент:** No      **Посл. звонок:** 12/10/08

**Имя:** Laverne Smith

Objectville Paper company

**Фирма:** XYZ Industries

**Телефон:** (212)555-8129

**Email:** Laverne.Smith@XyZindustries.com

**Клиент:** Yes      **Посл. звонок:** 04/11/10

- ③ После ввода всех данных снова выберите в меню File команду Save All. Записи будут сохранены в базе данных.

Команда Save All сохраняет все открытые файлы, а команда Save — только файл, над которым вы сейчас работаете.

### Часто задаваемые вопросы

**В:** Куда попадают введенные мной данные?

**О:** ИСР автоматически сохраняет данные в таблице People вашей базы данных. При этом таблица, ее столбцы, типы данных и сами данные хранятся в файле ContactDB.sdf, который является частью вашего проекта, поэтому ИСР автоматически обновляет его при внесении любых изменений.

**В:** Становятся ли введенные записи частью моей программы?

**О:** Да, данные — такая же часть программы, как код или созданные формы. Файл ContactDB.sdf копируется и хранится вместе с исполняемым файлом. Для доступа к данным приложение читает файл ContactDB.sdf и вносит в него новые записи.

Этот файл является базой данных SQL, и ваша программа может использовать его вместе с кодом, который был создан ИСР.



ContactDB.sdf

## Источник данных соединит форму с базой

Все готово для создания объекта базы данных .NET, который будет использовать для взаимодействия с базой. Нам потребуется **источник данных**. Он представляет собой всего лишь набор операторов SQL, при помощи которых ваша программа будет обращаться к базе ContactDB.

### 1 Возвращение к форме.

Закройте таблицу People и схему базы данных ContactDB для возвращения на вкладку Form1.cs [Design].

Завершив ввод данных, закройте окно ввода, чтобы вернуться к форме.

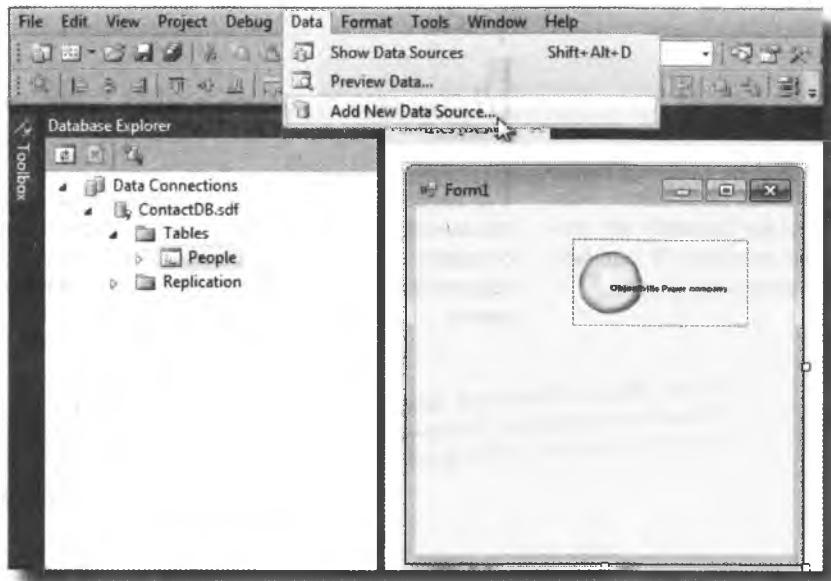
The screenshot shows the Microsoft Visual Studio interface. On the left, the Database Explorer window displays a tree structure with 'Data Connections' expanded, showing 'ContactDB.sdf' and its sub-items: 'Tables' (with 'People' selected), 'Replication', and 'Scripting'. On the right, the 'Form1.cs' window is visible, showing a single button labeled 'Objectville Paper Company'. A status bar at the bottom indicates 'Objectville Paper Company'.

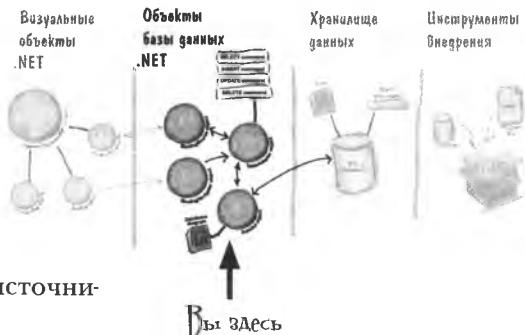
ContactID	Name	Company	Telephone	Email	Client	LastCall
1	Lloyd Jones	Black Box Inc.	(718)555-5638	LJones@xblack...	True	5/26/2010 12:00...
2	Lucinda Ericson	Ericson Events	(212)555-9523	lucy@ericsnev...	False	5/17/2010 12:00...
3	Liz Nelson	JTP	(419)555-2578	liznelson@JTP....	True	3/4/2009 12:00...
4	Matt Franks	XYZ Industries	(212)555-8125	Matt.Franks@x...	True	5/26/2010 12:00...
5	Sarah Kalter	Kalter, Riddle a...	(614)555-5641	sarah@krs.org	False	12/10/2008 12:0...
6	Laverne Smith	XYZ Industries	(212)555-8129	Laverne.Smith...	True	4/11/2010 12:00...
**	NULL	NULL	NULL	NULL	NULL	NULL

### 2 Добавление к приложению нового источника данных.

Выберите в меню Data команду Add New Data Source.

Создаваемый источник данных будет отвечать за все взаимодействия формы с базой.





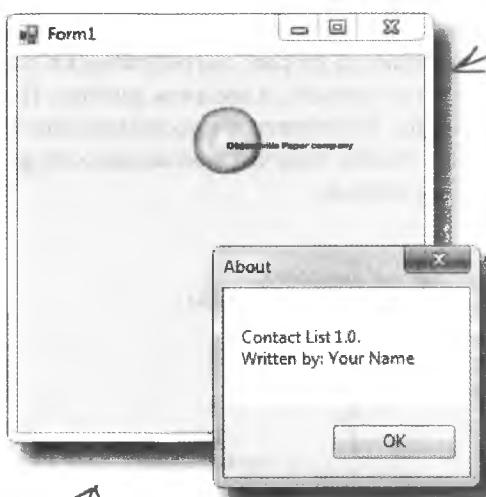
### 3 Конфигурация нового источника данных

Теперь нужно указать параметры взаимодействия источника данных с базой ContactDB. Вот как это сделать:

- ★ В поле Data Source Type выберите вариант **Database** и щелкните на кнопке Next.
- ★ В поле Database Model выберите вариант **Dataset** и щелкните на кнопке Next.
- ★ Раскрывающийся список Choose Your Data Connection содержит только один вариант, соответствующий вашей базе Contact. Щелкните на кнопке Next.
- ★ В поле Choose Your Database Objects установите флагок **Tables**.
- ★ Убедитесь, что в поле Dataset Name введено значение **ContactDBDataSet** и щелкните на кнопке **Finish**.

Таким способом вы соединяете новый источник данных с таблицей People в базе данных ContactDB.

В версиях, отличных от Express, предлагается сохранить строку подключения в файле конфигурации приложения. Ответьте «Yes».



Это ваша форма.

Теперь форма может использовать источник данных для взаимодействия с базой ContactDB.



ContactDBDataSet.xsd



ContactDB.sdf



ContactDBDataSet.  
Designer.cs

Файл с базой данных.

Эти файлы созданы источником данных, параметры которого вы только что выбрали.

## Добавление элементов, связанных с базой данных

Добавим на форму другие элементы управления. Все они должны быть *связаны* с базой данных и со столбцами таблицы People.

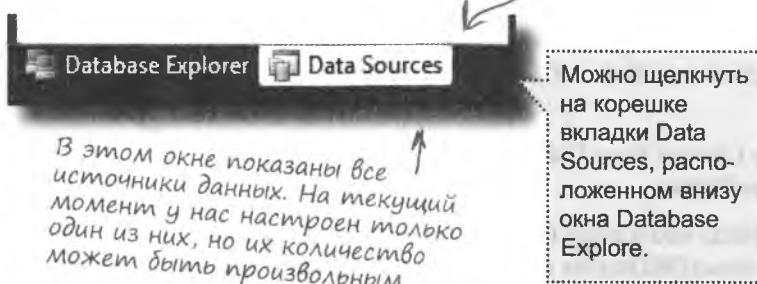
То есть изменение данных в любом элементе управления формы должно отображаться в соответствующем столбце таблицы.

Мы возвращаемся к созданию объектов формы, которые будут взаимодействовать с хранилищем данных.

### 1 Выбор источника данных.

В меню Data выберите команду Show Data Sources. Откроется окно Data Sources с перечнем настроенных источников данных.

Если вы не видите эту вкладку, в меню Data выберите команду Show Data Sources.

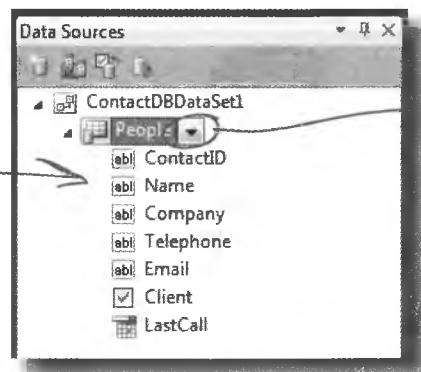


### 2 Выделение таблицы People.

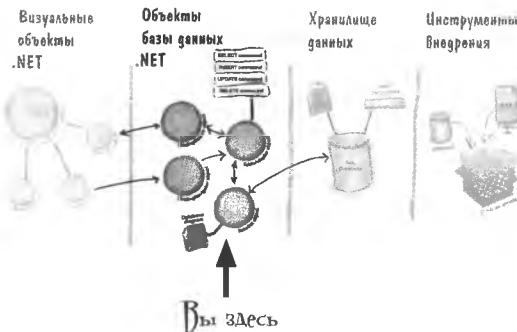
Под пунктом ContactDBDataSet вы увидите таблицу People со всеми столбцами. Если выделить строчку People в окне Data Sources и затем перетащить мышью на форму, автоматически появится еще один элемент, при помощи которого можно просматривать и вводить данные. По умолчанию это будет DataGridView, то есть еще одна таблица. Щелкните на расположенной слева от имени таблицы People и выберите команду Details, чтобы получить возможность добавлять индивидуальные элементы управления для каждого столбца.

Щелкните на стрелке и выберите в меню команду Details, чтобы добавить к форме несколько разных элементов управления.

Здесь должны быть показаны все созданные столбцы.



Это раскрывающееся меню видно только в режиме конструктора форм, когда у вас есть возможность перетаскивать элементы с источника данных на форму.



### 3 Создание элементов управления связанных с таблицей People

Перетащите таблицу People на форму. Появятся элементы управления, соответствующие столбцам базы данных. Волноваться о том, как именно они выглядят, пока не нужно, важно, чтобы они просто появились.

Для перевода формы в активное состояние, достаточно перейти на вкладку Form1.cs [Design] или открыть файл Form1.cs из окна Solution Explorer.

Эту панель управления для навигации по таблице создала ИСР.

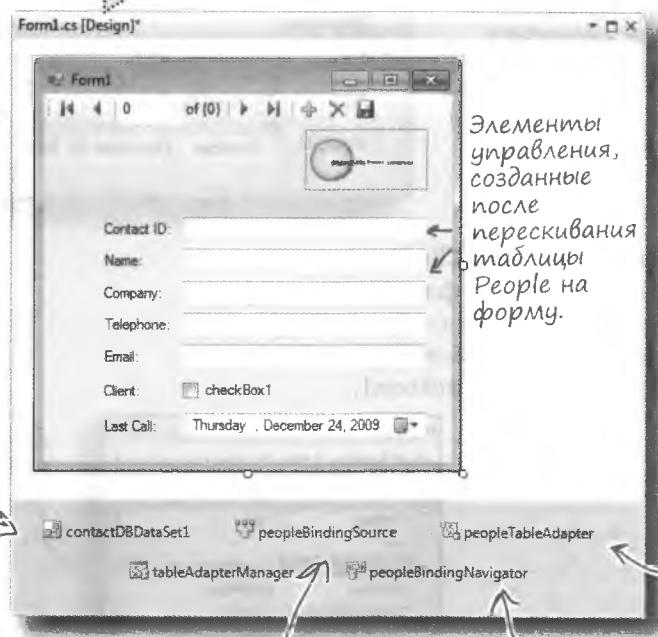
Эти компоненты не видны на самой форме, они лишь указывают на код, который был создан ИСР для взаимодействия с таблицей People и базой данных ContactDB.

Элементы управления, созданные после пересквивания таблицы People на форму.

Этот адаптер отвечает за взаимодействие с командами SQL.

Этот объект соединяет форму с таблицей People.

Этот компонент соединяет ваши элементы управления с таблицей инструментов.



## Хорошие программы понятны интуитивно

Теперь наша форма работает. Но при этом она имеет не самый презентабельный вид, а ведь приложение желательно делать не только функциональным. Им должно быть приятно пользоваться. Сделаем его похожим на карточки, с которых мы переписывали контактную информацию.

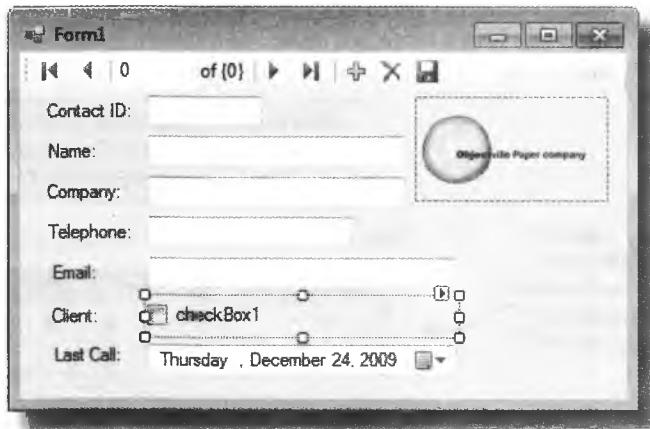
Форма станет более понятной, если придать ей вот такой вид.

Имя: Laverne Smith  
Фирма: XYZ Industries  
Телефон: (212) 555-8129  
Email: Laverne.Smith@XyZindustries.com  
Клиент: Yes    Посл. звонок: 05/26/07

### ① Выравнивание полей и их названий.

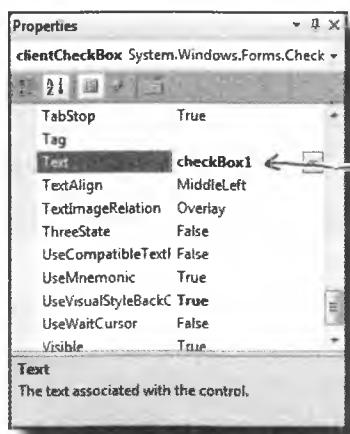
Выровняем поля и их названия по левому краю формы. В таком виде ею будет удобней пользоваться.

При перетаскивании элементов управления будут появляться линии синего цвета, упрощающие процедуру выравнивания.

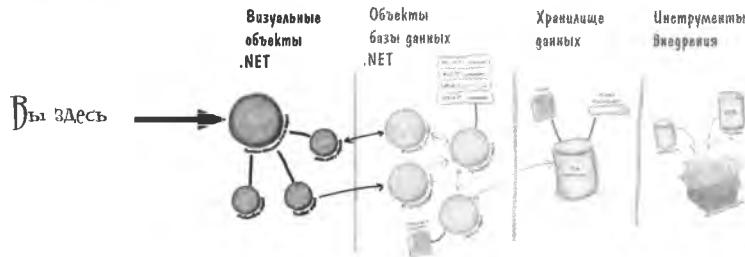


### ② Удаление подписи к флагажку Client.

После перетаскивания на форму флагажку Client справа от него оказалась ненужная подпись. В окне Properties, расположенном под окном Solution Explorer найдите свойство Text и удалите слово checkBox1.



Для удаления метки очистите это поле.



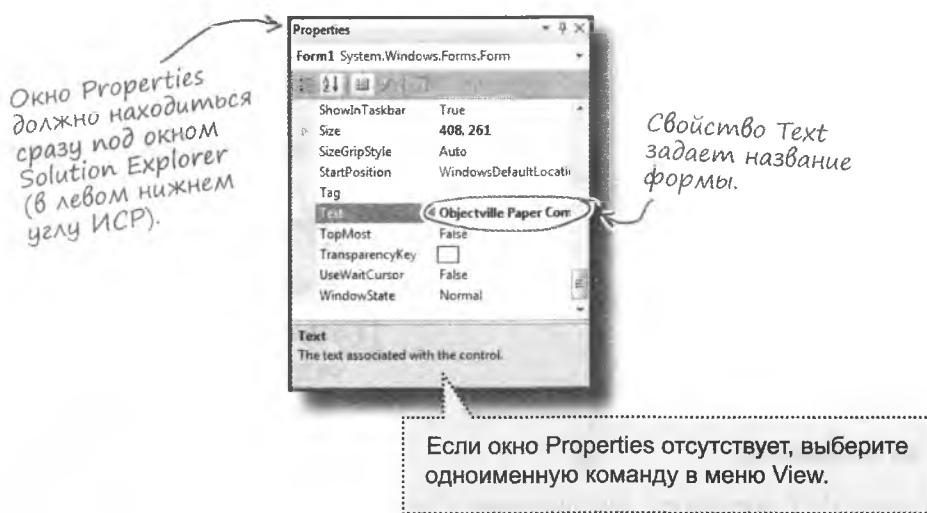
③

### Придание профессионального вида.

Щелкните в произвольном месте формы в окне Properties и найдите свойство Text. Введите в это поле название *Objectville Paper Company Contact List*.

Присвоив свойствам MaximizeBox и MinimizeBox значение False, можно убрать возможность разворачивать и сворачивать форму.

При раскрытии формы на полный экран положение элементов управления не меняется, форма выглядит некрасиво. Поэтому мы просто уберем эту возможность.



**Хорошим считается приложение, с которым легко работать. Его функциональность должна соответствовать ожиданиям среднестатистического пользователя.**

## Тестирование

Осталось запустить программу и убедиться, что она работает правильно! Вы уже знаете, как это сделать, поэтому просто нажмите F5 или щелкните на кнопке с зеленой стрелкой на панели инструментов (также можно выбрать команду Run из меню Debug).

Запускать можно даже не полностью готовые программы. Если в коде ИСР есть ошибки, программа не запустится.

Для остановки программы достаточно щелкнуть на красной кнопке с крестиком в правом верхнем углу.

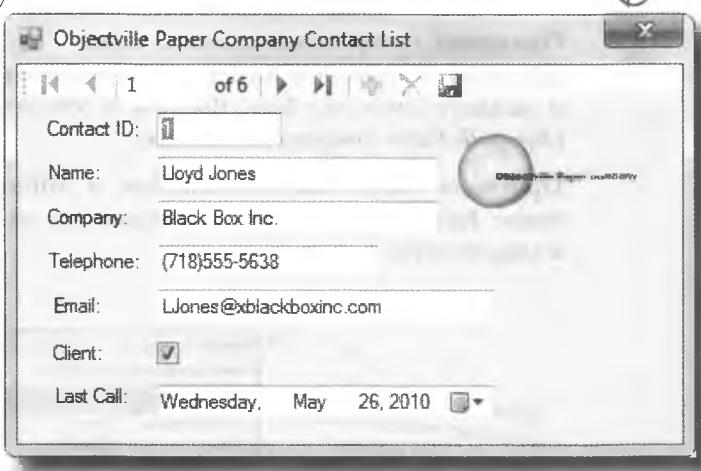
Компиляция  
программы  
переписывает  
данные в базе.

Эти элементы  
управления позволяют  
переходить от одной  
записи базы к другой.

ИСР сначала компилирует,  
потом запускает на выполнение

При запуске программы ИСР выполняет две операции. Сначала она строит программу, затем выполняет ее. Эта работа состоит из нескольких этапов. Код компилируется, то есть превращается в исполняемый файл. Этот файл вместе с другими ресурсами помещается в подпапку папки bin.

Свой исполняемый файл вместе с базой SQL вы найдете в папке bin/debug. Его запуск позволяет сохранить все внесенные в процессе пользования приложением изменения. В то время как при запуске из ИСР база заменяется копией с данными из окна database Explorer.



Этот аспект  
будет подробно  
рассмотрен  
в следующей главе.



Будьте  
осторожны!

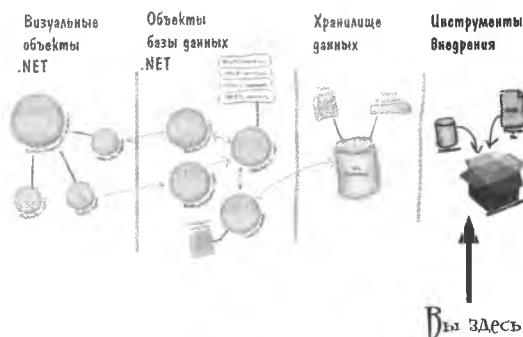
При каждой компиляции программы в папку bin помещается свежая копия базы данных. Все данные, накопленные с момента последней отладки, при этом уничтожаются.

В процессе отладки, если код был изменен, ИСР перестраивает программу — это означает, что база данных может оказаться переписанной. Этой проблемы можно избежать, запуская программу из папки bin/debug или bin/release или предварительно воспользовавшись установщиком.

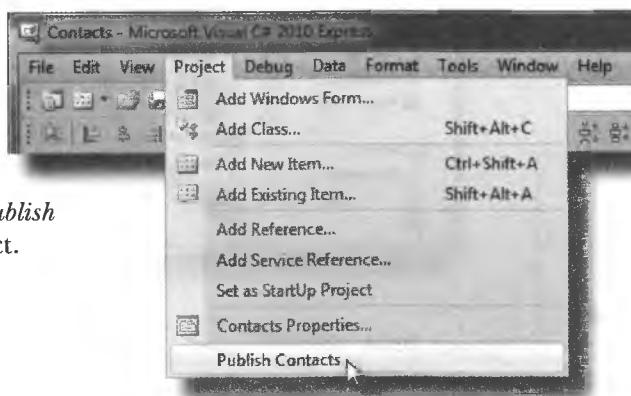
# Как превратить ВАШЕ приложение в приложение для ВСЕХ

Пока что приложение работает только на вашем компьютере. Никто не может им воспользоваться, купить его у вас, увидеть, какой вы замечательный программист и нанять на работу... даже начальник и клиенты не видят сообщения из вашей базы данных.

C# позволяет легко **внедрить** созданное вами приложение. Внедрением называется установка программы на другие компьютеры. В ИСР Visual C# эта процедура осуществляется в два этапа.

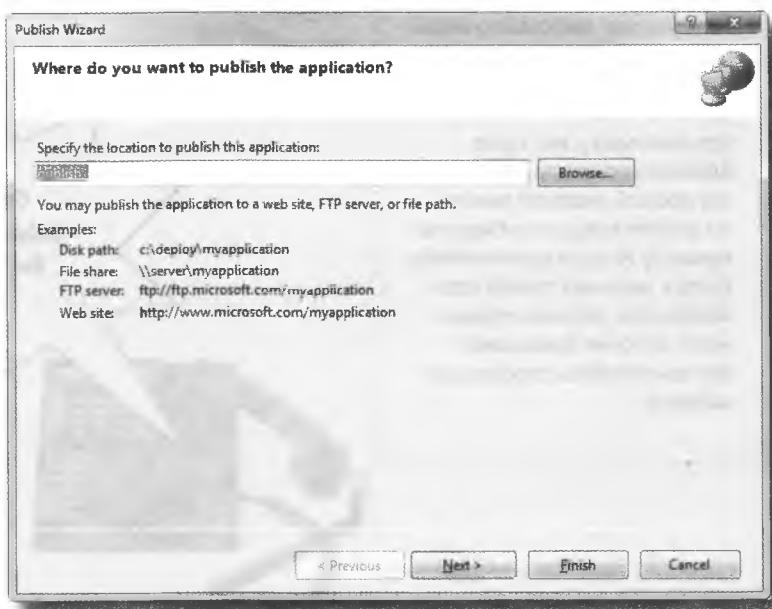


- Выберите команду *Publish Contacts* в меню Project.



При построении решения файлы копируются на ваш компьютер. Процедура публикации создает выполняемый конфигурационный файл, при помощи которого программу можно установить на любой компьютер.

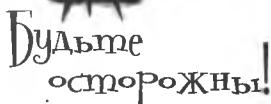
- Щелкните на кнопке *Finish* в окне Publish Wizard. Ваше приложение будет обработано, и вы увидите папку в которую сохраняется файл setup.exe.



В Visual Studio Express команда Publish находится в меню Project, в то время как в других версиях Visual Studio ее можно обнаружить в меню Build.

## Передаем приложение пользователям

После внедрения у вас появится папка с именем `publish/`. В ней находятся все компоненты, необходимые для установки. Самым важным из них является файл `setup`. Именно он позволяет установить приложение на любой компьютер.

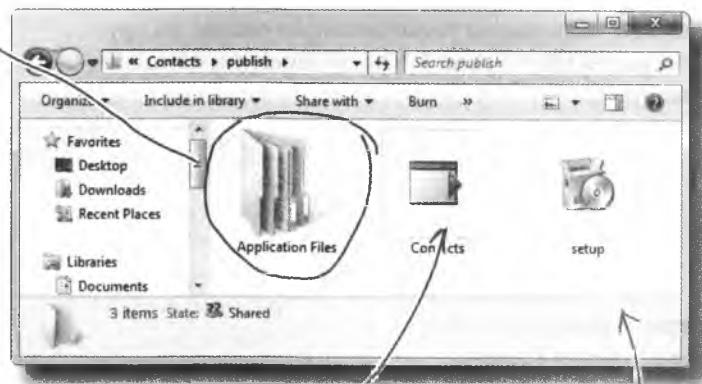


Будьте  
осторожны!

Для инсталляции вам  
могут понадобиться права  
администратора.

Если у вас еще нет базы  
*SQL Server Compact*, ин-  
сталлятор автоматически  
скачет и установит ее.  
На некоторых компью-  
терах это возможно только  
при наличии у вас прав  
администратора. Щелкни-  
те правой кнопкой мыши  
на файле `setup` и выберите  
команду *Run as administrator*.  
Если у вас нет такой воз-  
можности, ничего страш-  
ного. Это не помешает  
вам выполнять следующие  
задания.

Здесь находятся все  
файлы, необходимые  
для инсталляции.



Этот файл говорит  
установщику, что  
следует включить  
в инсталлируемую  
программу.

С помощью этого  
файла осуществляется  
инсталляция.

Секретарь сказала, что у нас уже  
работает новая база данных с контактами.  
Вы хорошо поработали и заслужили  
небольшой отдых...

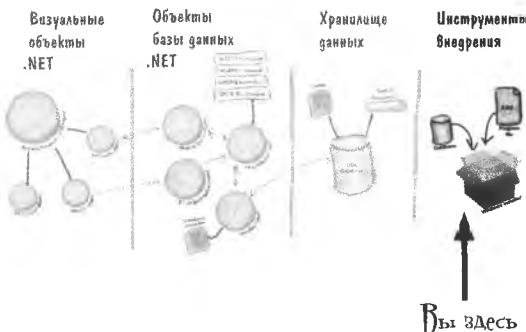


Кажется, начальник доволен.  
Но перед тем как собирать  
чемоданы, промесстрируем  
результаты своего труда.

## Работа НЕ окончена: проверим процесс установки

Перед тем как открыть бутылку шампанского и отпраздновать победу, нужно протестировать внедрение и установку.

Закройте ИСР Visual Studio. Запустите файл setup и укажите, куда следует установить ваше приложение. Убедитесь, что все работает так, как вы ожидали, — можно добавлять и редактировать записи, и все изменения сохраняются в базе данных.



Стрелки и текстовое поле позволяют переходить от одной записи к другой.

Попробуйте внести изменения в данные.

Objectville Paper Company Contact List

Contact ID:	1	of 6
Name:	Lloyd Jones	
Company:	Black Box Inc.	
Telephone:	(718)555-5638	
Email:	LJones@xblackboxinc.com	
Client:	<input checked="" type="checkbox"/>	
Last Call:	Wednesday, May 26, 2010	

Внесенные изменения сохраняются в базе.

## ТЕСТИРУЙТЕ ВСЁ!

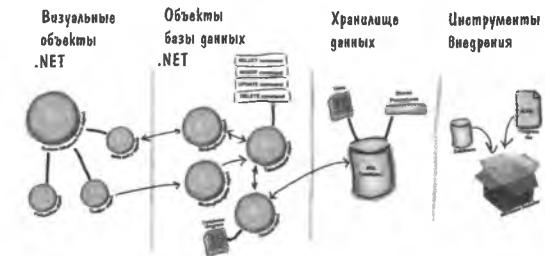
Тестируйте вашу программу, процедуру внедрения, данные в вашем приложении.

Все контакты на месте. Ведь они являются частью базы данных ContactDB.sdf, которая была установлена вместе с приложением.

*быстрее не бывает*

## Управляющее данными приложение готово

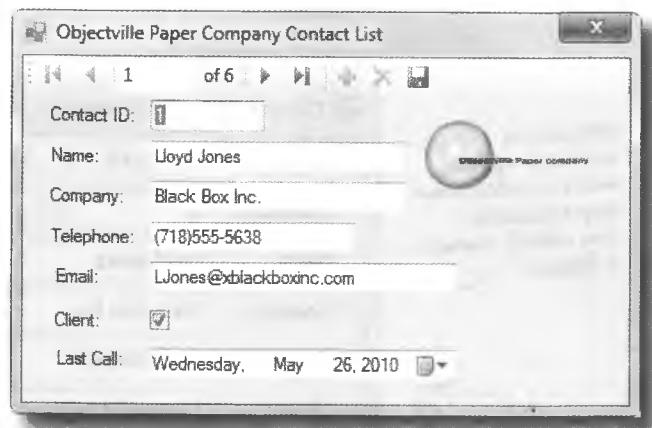
Visual Studio позволяет с легкостью разработать Windows-приложение, написать базу данных и связать эти два объекта вместе. Вы даже смогли построить программу-инсталлятор всего несколькими дополнительным щелчками мыши.



### От бумажной карточки



До элeктронной



на много быстрее,  
чем казалось.

Средства Visual C# позволяют сфокусироваться на  
выполнении задания, ведь вам уже не требуется  
писать код для окон, кнопок и доступа к базе SQL.

**2** это ВсёГо ЛиШь Код



## Под покровом



**Вы — программист, а не просто пользователь ИСР.**

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

## Когда Вы делаете это...

ИСР – это мощный инструмент. 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 ИСР при разработке типичного приложения.

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

1

### Создание формы в Windows

ИСР позволяет строить различные приложения, но сейчас мы рассмотрим вариант Windows Forms – программу, имеющую визуальные элементы, например, формы и кнопки.

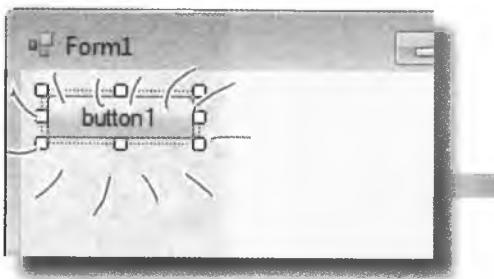
При выборе варианта  
*Windows Forms Application*  
ИСР создает пустую форму и добавляет ее к новому проекту.



2

### Перетаскивание кнопки с панели инструментов на форму и двойной щелчок на ней.

Именно кнопки заставляют форму работать. На их примере мы будем рассматривать различные аспекты языка C#. Кроме того, они являются частью практически любого создаваемого приложения.



3

### Задание свойств формы

Окно Properties позволяет легко поменять атрибуты практически любого компонента программы.

Окно Properties – это самый простой способ автоматически отредактировать любую часть кода формы *Form1.Designer.cs*. Если это окно закрыто, нажмите клавишу F4.



## ...ИСР делает это.

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

Эти файлы были созданы на базе шаблона, содержащего код добавления и отображения формы.

- 1 ...ИСР создает файлы и папки.



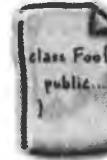
WindowsApplication1.csproj



Form1.cs



Form1.Designer.cs



Program.cs



Properties

- 2 ...ИСР добавляет код в файл Form1.Designer.cs, отвечающий за появление кнопки на форме. А затем добавляет в файл Form1.cs – код, определяющий результат щелчка на кнопке.

```
private void button1_Click(object sender, EventArgs e)
{
}
```

ИСР может добавить пустой метод для обработки щелчка на кнопке. Но выбор параметров этого метода – ваша работа.



Этот код добавляется в файл Form1.cs.



- 3 ...ИСР открывает файл Form1.Designer.cs и обновляет строчку кода.

```
partial class Form1
{
    ...
    this.Text = "Objectville Paper Company Contact List";
    ...
}
```

...и обновляет эту строчку.

ИСР открывает файл...



## 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, ИСР сохраняет вашу программу в файлы. Эти файлы можно открыть и найти все добавленные к проекту элементы: формы, ресурсы, код и прочее.

ИСР 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 ИСР редактирует файлы, содержащие вашу программу.

ИСР 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.

### Инструменты от .NET Framework

C# – это всего лишь язык, и сам по себе он не может ничего **делать**. Но здесь вам на помощь приходит технология **.NET Framework**. При щелчке на кнопке развертки окна на весь экран срабатывает код, указывающий, как именно должна выполняться эта операция. Этот код является частью .NET Framework, как и другие кнопки, флагки, списки и даже механизмы связи с базой данных. Это хороший инструмент для создания графики, чтения и записи файлов, управления наборами объектов и прочих рутинных операций.

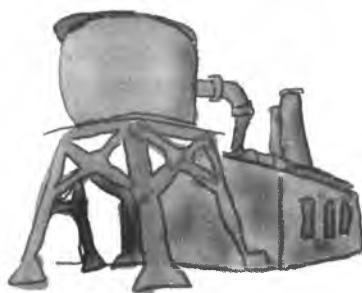


Инструменты .NET Framework находятся в **пространствах имен**. Вы уже видели их ранее, в верхней части кода. Это было пространство `System`. `Windows.Forms` – именно здесь содержатся кнопки, флагки и формы. При создании любого проекта Windows Forms в верхней части кода вы увидите строчку `using System.Windows.Forms`.

## 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 сохраняется на диск компьютера с расширением .exe. Для запуска программы достаточно дважды щелкнуть на его имени. По умолчанию этот файл помещается в папку bin,ложенную в папку project. При публикации решения он (вместе с другими нужными файлами) копируется в выбранную вами папку.

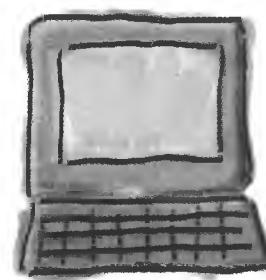
В ответ на команду Start Debugging (меню Debug) ИСР компилирует программу и запускает полученный файл на выполнение. Существуют и более совершенные инструменты **отладки**, то есть запуска программы с возможностью прерывания, чтобы вы могли понять механизм происходящего.



## Программы работают в CLR

Двойной щелчок на имени исполняемого файла запускает программы. Но между операционной системой и программой существует дополнительный «слой», называемый **Common Language Runtime**, или CLR (общезыковая исполняющая среда). Раньше (еще до изобретения C#) писать программы было намного сложнее, так как приходилось иметь дело с аппаратным обеспечением, заниматься низкоуровневым программированием. CLR – часто называемая также **виртуальной машиной** – является своего рода «переводчиком» между вашей программой и компьютером, на котором она работает.

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.



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

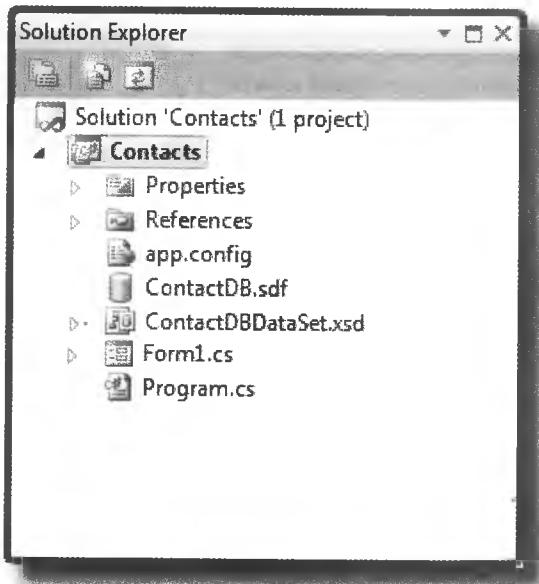
## Писать код помогает ИСР

С частью возможностей ИСР 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 знаком (\*).

Здесь находится файл с ресурсами формы, к которому вы добавляли логотип компании.

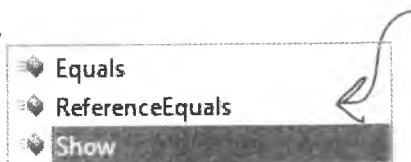


При работе над формой одновременно открыты две вкладки: одна с конструктором форм, а вторая с кодом. Для быстрого перехода между вкладками пользуйтесь комбинацией клавиш Ctrl-Tab.

### ИСР помогает писать код

Замечали ли вы в процессе набора кода маленькое всплывающее окошко? Это крайне полезная функция IntelliSense. Она показывает возможные способы завершения строчки кода. Например, если вы набрали слово MessageBox и поставили точку, дальше может следовать только:

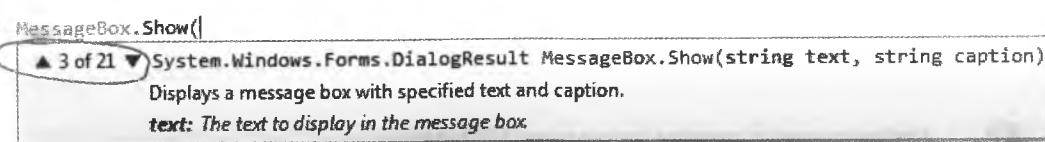
MessageBox.



ИСР знает, что MessageBox имеет три метода: Equals, ReferenceEquals и Show. При вводе буквы S будет выбран вариант Show. Вам останется нажать Enter. Иногда эта функция очень экономит время.

Если после выбора варианта Show напечатать (), функция IntelliSense предложит вам варианты заполнения строки:

Существует  
21 способ  
вызыва метода  
Show для  
MessageBox.



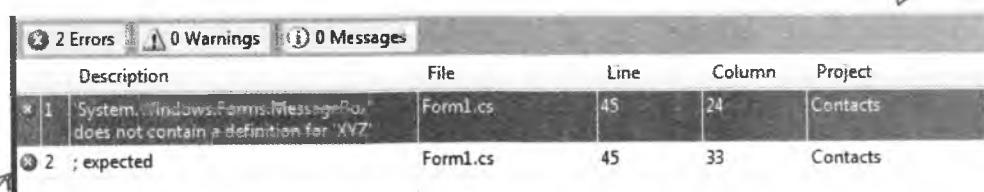
В ИСР существуют сокращения, называемые **фрагментами кода**. Для их вставки достаточно напечатать нужную аббревиатуру. К примеру, если напечатать mBox и дважды нажать клавишу Tab, появится строка для метода MessageBox.Show:

MessageBox.Show("Test");

### Окно Error List показывает ошибки компиляции

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

Отсутствие  
точки с запятой  
после оператора  
является одной  
из самых частых  
ошибок.



В окне Error List вы найдете список всех ошибок, мешающих компиляции программы.

Для исправления некорректного кода дважды щелкните на ошибке:

```
private void pictureBox1_Click(object sender, EventArgs e)
{
    MessageBox.XYZ("hi")
}
```

Некорректный код  
подчеркивается  
красным цветом.

## Любые действия ведут к изменению кода

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

Значок «Упражнение!» является сигналом к запуску Visual Studio.



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**.

3

### Просмотр созданного конструктором для элемента PictureBox кода

Вернитесь на вкладку `Form1.Designer.cs`. Найдите вот эту строчку:

Щелкните на квадратике со знаком «илюс».  
+ Windows Form Designer generated code

Щелчок на этом квадратике раскроет код. Найдите в нем следующие строчки:

```
//  
// pictureBox1  
//  
  
this.pictureBox1.Location = new System.Drawing.Point(276, 28);  
this.pictureBox1.Name = "pictureBox1";  
this.pictureBox1.Size = new System.Drawing.Size(100, 50);  
this.pictureBox1.TabIndex = 0;  
  
this.pictureBox1.TabStop = false;
```

Ваше значение параметров `Location` и `Size` будут отличаться от указанных в книге.

## Что же все 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 двумя косыми слешами (`//`). Но ИСР иногда может поставить три слеша.

Это комментарии XML, о которых вы узнаете в разделе #1 Приложения.

Для детей нет ничего более притягательного, чем табличка с надписью «Не трогать!» Не отрицайте, вы же знаете, что вам тоже хочется... поэтому отредактируем содержимое данного метода, воспользовавшись редактором кода! Добавьте к форме кнопку 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 поменяйте 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]`.

## Поменять код проще всего средствами ИСР.

**программа делает заявление**

## Структура программы

Код любой программы на C# структурируется одинаково. Все программы используют **пространства имен, классы и методы**.

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.

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



### Внимательно рассмотрим код

Откройте код проекта Contacts Form1.cs, чтобы рассмотреть его фрагменты.

#### ➊ Файл кода начинается с перечисления инструментов .NET Framework

В верхней части любого файла программы вы обнаружите набор строк, начинающихся с оператора `using`. Они указывают, с какой частью .NET Framework будет работать программа. Чтобы использовать классы из других пространств имен, нужно указать их при помощи директивы `using`. Для построения форм применяется множество инструментов .NET Framework, поэтому в начале программы так много строк с директивой `using`.

```
using System;  
using System.Collections.Generic;  
using System.ComponentModel;  
using System.Data;  
using System.Drawing;  
using System.Linq;  
using System.Text;  
using System.Windows.Forms;
```

Такие строчки находятся в верхней части любого кода. Каждая из них показывает, какое пространство имен из .NET Framework использует том или иной файл .cs.

В принципе без оператора `using` можно обойтись, если пользоваться полными именами. Если строка `using System.Windows.Forms` отсутствует, окно сообщений можно создать, написав `System.Windows.Forms.MessageBox.Show()`, и компилятор все равно поймет, какое пространство имен вы имеете в виду.

## 2 Программы на C# используют классы

Программы на C# используют **классы**. Каждый класс выполняет свою задачу. Когда вы создавали свою первую программу, ИСР добавила класс **Form1**, отображающий формы.

namespace Contacts

{

public partial class Form1 : Form

{

Для программы *Contacts*, ИСР создает одноименное пространство имен. Именно в этом пространстве находится все содержимое скобок.

↗

Класс **Form1** содержит код создания как самой формы, так и ее элементов управления. ИСР добавляет этот класс при создании проекта Windows Form Application.

## 3 Классы содержат методы

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

Для выполнения различных действий классы используют **метод**. Метод берет входные данные и производит некоторое действие. Данные передаются при помощи **параметров**. Именно от них зависит поведение метода. Слово **void** перед названием метода означает, что он не возвращает никаких параметров.

public Form1()

{

    InitializeComponent();

Здесь вызывается метод InitializeComponent().

}

## 4 Каждый оператор выполняет только одно действие

Строчкой **MessageBox.Show()** вы добавляете **оператор**. Именно из операторов составлены методы. При вызове метода выполняется сначала первый оператор, потом следующий и т. д. Достигнув конца списка операторов или оператора **return**, метод завершает работу.

Метод **pictureBox1\_Click()**  
вызывается щелчком на  
графическом фрагменте.

Этот метод имеет  
два параметра:  
**sender** и **e**.

private void pictureBox1\_Click(object sender, EventArgs e)

{

    MessageBox.Show("Contact List 1.0", "About");

}

Это **оператор**, вызывающий  
диалоговое окно.

Оператор вызывает метод **Show()**,  
принадлежащий классу **MessageBox** в  
пространстве имен **System.Windows.Forms**.

}

Оператор передает методу **Show()** два па-  
раметра: строку, которая будет отображаться  
в окне диалога и заголовок этого окна.

далее ▶

83

еще более подробно

## Начало работы программы

При создании нового решения ИСР добавляет файл **Program.cs**. Найдите его в окне Solution Explorer и дважды щелкните на имени. Внутри класса Program вы обнаружите метод Main(). Этот метод является точкой входа, то есть именно отсюда программа начинает работу.

В программе на C# возможна только одна точка входа. Это метод Main().



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

Код под увеличительным стеклом

```
1 using System;
using System.Linq;
using System.Collections.Generic;
using System.Windows.Forms;

2 namespace Contacts
{
    3 static class Program
    {
        4 /// <summary>
        /// Точка входа в приложение.
        /// </summary>

        [STAThread]
        5 static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1()); ←
        }
    }
}
```

Код находится в пространстве имен Contacts.

Две и более косые черты в начале строки обозначают комментарии.

Программа начинает работу с точки входа.

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

Первая часть имени класса или метода называется объвлением.

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

## 1 Встроенные функции C# и .NET.

Подобные строчки находятся в верхней части почти всех файлов классов C#. System.Windows.Forms – это **пространство имен**. Стока `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.

## 2 Выбор пространства имен для кода.

ИСР назвала созданное пространство имен `Contacts` (в соответствии с именем проекта). Именно этому пространству относится весь код.

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

## 3 Код принадлежит к конкретному классу.

В вашей программе этот класс называется `Program`. Он содержит код запуска программы и код вызова формы `Contacts`.

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

В одном пространстве имен может находиться 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 `Contacts`.

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

**Любая программа на C# должна иметь единственный метод `Main`. Он является точкой входа для вашего кода.**

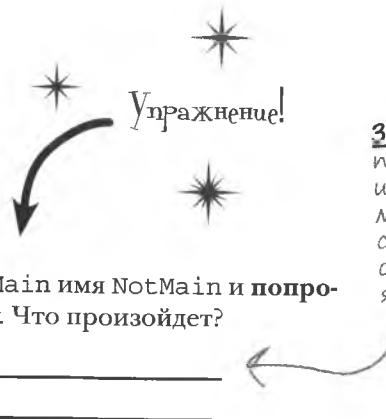
## 5 Точка входа.

Каждая программа на C# должна иметь 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.

**При запуске кода метод `Main()` выполняется ПЕРВЫМ.**

## Реактирование точки входа

В программе главное – наличие точки входа. При этом не имеет значение, к какому классу принадлежит содержащий ее метод и какие действия производит. **Откройте программу из главы 1** и удалите метод Main, чтобы создать его заново.



Запишите, что произошло при изменении имени метода, и ваши соображения о причинах этого явления.

- 1 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... ИСР добавит в программу класс AnotherClass. После этого код примет вид:

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

namespace Contacts
{
    class AnotherClass
    {
    }
}
```

В файл были добавлены четыре стандартные строчки с оператором using.

Этот класс тоже находится в пространстве имен Contacts.

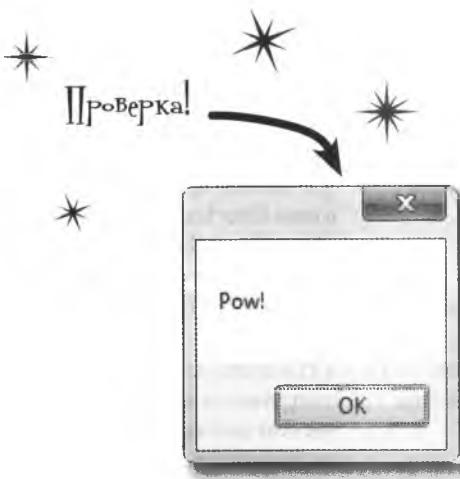
Имя присваивается классу автоматически (на основе имени файла).

- 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:

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

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



### Что же произошло?

Теперь вместо приложения Contacts 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



Опишите назначение различных строк кода, как показано в примере.

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

3 Оператор `using` добавляет  
в класс методы из других  
пространств имен.

```
namespace SomeNamespace
```

```
{
```

```
    class MyClass {
```

```
        public static void DoSomething() {
```

```
            MessageBox.Show("Здесь будет сообщение");
```

```
}
```

```
}
```

```
}
```

## часто Задаваемые Вопросы

**В:** Какую роль играют фигурные скобки?

**О:** Скобки группируют операторы в блоки и используются только попарно. Открывающей скобке всегда должна соответствовать закрывающая.

Для выделения пары скобок достаточно щелкнуть на любой из них.

**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 называется точкой входа. Первый оператор этого метода и будет выполняться самым первым.

**В:** Почему при запуске программы в окне Error List появляется сообщение об ошибках? Я думал, что подобное возможно только при выполнении команды Build Solution.

**О:** Первое, что происходит при запуске программы на выполнение, это сохранение всех файлов в виде решения и попытка их компиляции. А при компиляции кода — неважно, запускаете вы при этом программу или строите ее решение — все имеющиеся ошибки отображаются в окне Error List.

В тексте кода ошибки выделяются красным цветом.

Возьми в руку карандаш

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

Вот правильные варианты ответа на вопрос о строках.

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

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.

# \* КТО И ЧТО ДЕЛАЕТ?

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

```
partial class Form1
{
    :
    this.BackColor = Color.DarkViolet;
}
```

// Цикл выполняется три раза

```
partial class Form1
{
    private void InitializeComponent()
    {
        :
    }
}
```

```
number_of_pit_stopsLabel.Name
    = "number_of_pit_stopsLabel";
number_of_pit_stopsLabel.Size
    = new System.Drawing.Size(135, 17);
number_of_pit_stopsLabel.Text
    = "Number of pit stops:";
```

```
/// <summary>
/// При щелчке на кнопке появляется
/// графический фрагмент Rover
/// </summary>
```

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

Задает свойства объекта label.

Это всего лишь комментарий, добавленный программистом, чтобы объяснить назначение кода тем, кто будет читать его программу.

Убирает кнопку сворачивания окна (最小化) окна Form1 из строки заголовка.

Подсказки системы, объясняющие назначение различных блоков кода.

Меняет фоновый цвет окна Form1.

Блок кода, выполняемый при открытии программой окна Form1.

# КТО И ЧТО ДЕЛАЕТ?

Вот правильные ответы на вопрос о назначении различных блоков кода. Сравните со своими вариантами!

```
partial class Form1
{
    ...
    this.BackColor = Color.DarkViolet;
}

// Цикл выполняется три раза
```

```
partial class Form1
{
    private void InitializeComponent()
    {
        ...
    }
}
```

```
number_of_pit_stopsLabel.Name
    = "number_of_pit_stopsLabel";
number_of_pit_stopsLabel.Size
    = new System.Drawing.Size(135, 17);
number_of_pit_stopsLabel.Text
    = "Number of pit stops:";
```

```
/// <summary>
/// При щелчке на кнопке появляется
/// графический фрагмент Rover
/// </summary>
```

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

Задает свойства объекта label.

Это всего лишь комментарий, добавленный программистом, чтобы объяснить назначение кода тем, кто будет читать его программу.

Убирает кнопку сворачивания окна (最小化) окна Form1 из строки заголовка.

Подсказки системы, объясняющие назначение различных блоков кода.

Меняет фоновый цвет окна Form1.

Блок кода, выполняемый при открытии программой окна Form1.

## 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.

MoreClasses.cs

```
namespace PetFiler2 {

    class Fish {
        public void Swim() {
            // операторы
        }
    }

    partial class Cat {
        public void Meow() {
            // операторы
        }
    }
}
```

SomeClasses.cs

```
namespace PetFiler2 {

    class Dog {
        public void Bark() {
            // Здесь операторы
        }
    }

    partial class Cat {
        public void Meow() {
            // Еще операторы
        }
    }
}
```

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

Для помещения класса в разные файлы используйте ключевое слово `partial`. Именно его использует ИСР, создавая файлы `Form1.cs` и `Form1.Designer.cs`.

*параметры могут варьироваться*

## 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 *тип* и ее *имя*. Благодаря указанию типов невозможно скомпилировать программы в случае, если вы сделали ошибку и пытаетесь сделать нечто, лишенное смысла, например вычесть Fido из 48353.

Это типы переменных.  
Это имена переменных.

```
int maxWeight;  
string message;  
bool boxChecked;
```

Тип переменной определяет, какие именно данные в ней могут храниться.

Лучше выбирать значимые имена, отражающие суть переменных.

### 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.



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.

Для работы  
с числами, тек-  
стом, булевы-  
ми значениями  
и любым другим  
видом данных  
используйте  
переменные.

## Присвоение значений

Поместите в программу эти операторы:

```
int z;
MessageBox.Show("Ответ " + z);
```

При попытке запустить программу, ИСР 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:

Присвоенные  
переменным  
значения.

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

В объявлении переменной, как и раньше, указывается ее тип.

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

## 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`.

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

пе-ре-мен-ный, прил.

умеющий меняться или приспосабливаться.  
*Переменная скорость дрели позволяет Бобу сверлить то быстро, то медленно, в зависимости от текущих задач.*

## Знакомые математические символы

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:

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

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

Класс MessageBox вызывает окно с текстом «hello again hello».

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.

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

```
int count = 0;
```

```
count ++;
```

```
count --;
```

```
string result = "hello";
```

```
result += " again " + result; ←
```

```
MessageBox.Show(result);
```

```
result = "the value is: " + count;
```

```
result = "";
```

```
bool yesNo = false;
```

```
bool anotherBool = true;
```

```
yesNo = !anotherBool;
```

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

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.

Если 71 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.

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



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 Новый проект Windows Forms Application

Перетащите на форму кнопку и щелкните на ней два раза. Введите код с предыдущей страницы.

```

Form1.cs
Chapter_2_Code.Form1
private void button1_Click(object sender, EventArgs e)
{
    // Here's a great way to use the IDE to see how this code works!
    //
    // First, create a new project in the IDE add a button. Next, double-click on the
    // button so the IDE adds a button1_Click method to your program. Fill in that
    // method with all of the code, starting with "int number = 15;".
    //
    // Now put a breakpoint on the first line by right-clicking on it and choosing
    // "Breakpoint >> Insert Breakpoint". The line should turn red.
    //
    // Next, start debugging your program. You'll see it break on the line where you
    // inserted the breakpoint. Your program's just paused! If you click the Run
    // toolbar button (or hit F5), it will continue. Right-click on "number" and
    // choose "Expression: 'number' >> Add Watch" from the menu. The bottom panel in
    // the IDE should change to the Watches window, and there should be a line in that
    // window for "number". Step through the program line by line using Step Over (F10).
    // You can see the value of the "number" variable change as you go!
    //
    // Do the same for the count, result, yesNo, and anotherBool variables.

    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;
    MessageBox.Show(result);
    result = "the value is: " + count;
    result = "";

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

```

ИСР откроет новый проект с пустой формой и точкой входа. Вы можете дать ему любое имя.

Комментарии выделяются зеленым цветом и могут содержать любой текст, компилятор их не замечает.

Строка с точкой останова выделяется красным, а на полях редактора слева от нее появляется красная точка.

Достигнув точки останова программа прекращает работу и дает возможность проверить и отредактировать значения переменных.

## 2 Вставка точки останова в первую строчку кода

Щелкните правой кнопкой мыши на строке (`int number = 15;`) и выберите команду Insert Breakpoint из меню Breakpoint. (Можно воспользоваться Toggle Breakpoint из меню Debug или нажать клавишу F9.)

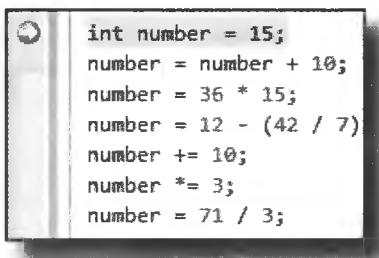
Переверните страницу и продолжим!

**3 Отладка программы**

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

**4 Переход к точке останова**

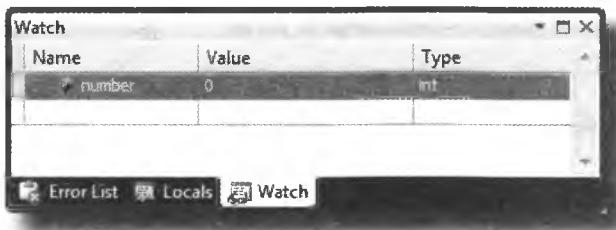
Как только программа дойдет до точки останова, ИСР автоматически вызовет редактор кода и выделит нужную строчку желтым цветом.



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

**5 Контрольное значение переменной number**

Щелкните правой кнопкой мыши на переменной number и выберите в появившемся меню команду Expression: 'number' >> Add Watch. Внизу экрана появится окно Watch:

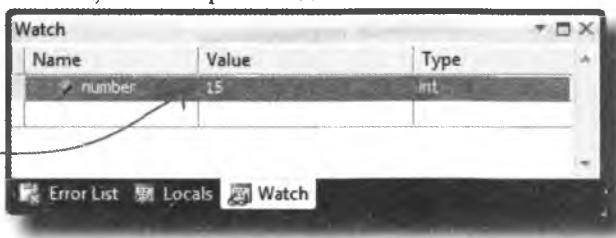


**Функция Watch помогает отслеживать значения переменных на каждом этапе выполнения кода. Вы оцените ее удобство во мере усложнения ваших программ.**

**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.

Присвоенное  
переменной  
значение тут  
же появилось  
в окне Watch.



**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`.

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)
{
    MessageBox.Show("Я появлюсь 4 раза");
}
```

### Написание простого цикла при помощи фрагментов кода

Через минуту вам предстоит написать программу. Эту задачу можно ускорить средствами ИСР. Если набрав ключевое слово `for`, дважды нажать кнопку `Tab`, нужный код наберется сам. 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.

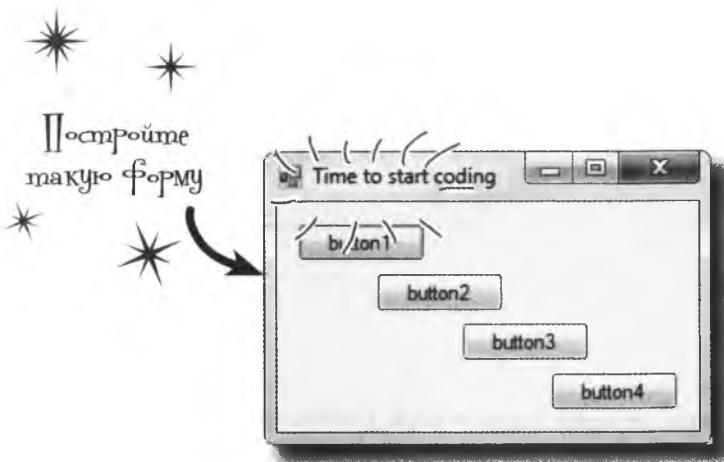
### СОВЕТ: Скобки

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

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

## Перейдем к практике

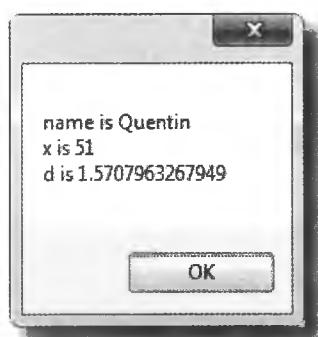
Работа программы определяется операторами. Но операторы существуют не в вакууме. Впрочем, проще всего это понять на примере. **Создайте проект Windows Forms Application.**



### Оператор, Выводящий сообщение

Дважды щелкните на первой кнопке и добавьте к методу button1\_Click() эти операторы. Внимательно изучите код и результат его выполнения.

*x — это переменная. int говорит о том, что это — целое число, остальная же часть оператора присваивает переменной значение 3.*



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

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

*\n — это esc последовательность, добавляющая в окно сообщения символ переноса строки.*

## Несколько советов

★ Не забывайте, что в конце оператора должна стоять точка с запятой:

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

★ За двумя косыми чертами идут комментарии:

```
// этот текст игнорируется
```

★ Объявляя переменную, укажите ее имя и тип (типы будут подробно рассматриваться в главе 4):

```
int weight;
// weight — целое число
```

★ Код для классов и методов заключается в фигурные скобки:

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

★ В большинстве случаев количество пробелов не имеет значения:

```
int j      =      1234 ;
```

это то же самое, что и:

```
int j = 1234;
```

## 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.

```

if (someValue == 24)
{
    MessageBox.Show("Значение 24.");
}

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

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

```

Оператор `if` начинается с проверки условия.

Оператор в фигурных скобках выполняется только при соблюдении условия.



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.

## 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)`.

### Задание переменной и проверка ее значения

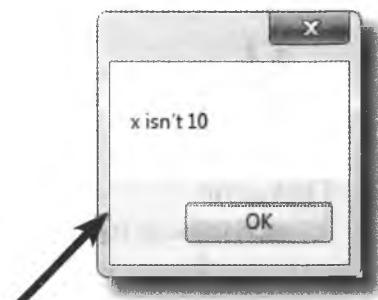
Ниже вы видите код для второй кнопки. Он содержит оператор `if/else`, проверяющий, равна ли целочисленная переменная `x` десяти.

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

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

Использование условного оператора для сравнения двух чисел называется проверкой условий.

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

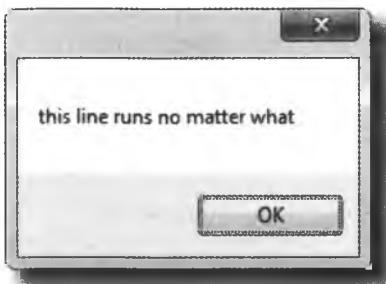


Это результат работы программы. Отредактируйте код таким образом, чтобы сообщение изменилось на `x must be 10`.

## Проверка еще одного условия

Щелчок на третьей кнопке должен приводить к появлению окна. Самостоятельно отредактируйте код, чтобы в окне появлялся этот текст.

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



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

## Добавление цикла

Ниже показан код для последней кнопки. Он содержит два цикла. Цикл `while`, который реализует операторы в скобках, пока выполняется заданное условие. И цикл `for`. Посмотрим, как это работает.

```
private void button4_Click(object sender, EventArgs e)
{
    int count = 0;

    Цикл работает,
    пока значение
    переменной
    count меньше 10.

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

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

    здесь переменной,
    используемой
    для подсчета
    проходов цикла,
    присваивается
    начальное значение.
    for (int i = 0; i < 5; i++)
    {
        count = count - 1;
    }

    Оператор, при
    каждом проходе
    цикла увеличива-
    ющий значение i на 1.
```

Message Box code:

```
    MessageBox.Show("The answer is " + count);
```

Перед тем как щелкнуть на кнопке, попробуйте по виду кода понять, какое значение будет в окне. Проверьте правильность своих выводов!

выше и выше и выше и...

## Возьми в руку карандаш



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

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

//

MessageBox.Show("Результат равен " + result);

Заполните  
остальные  
строки по  
аналогии.

## 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;
}
```

Сколько раз будет выполнен этот оператор?

### 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 оператор.

### 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?

### 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;
    }
}
```

При повторении оператора  $q = q * 2$  помните, что начальное значение  $q$  равно 2.

## МОЗГОВОЙ ШТУРМ

Подумайте, в какой ситуации может понадобиться бесконечный цикл? (Один из вариантов ответа вы найдете в главе 13...)

если только, но только если

## 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
MessageBox.Show("Результат равен" + result);
```

## 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 #5

Будет выполнен восемь раз.

#### Цикл #4

Еще один бесконечный цикл.

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 какому-нибудь классу?

**О:** Да. Все операторы принадлежат к определенным классам, точно также как все классы, в свою очередь, принадлежат пространствам имен. Когда вы используете конструктор для задания свойств объектов формы, 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, которые я не могу использовать? А как насчет тех, которые я обязан использовать?

**О:** Да, некоторые пространства имен использовать не рекомендуется, например, пространство имен `System`. Именно там находятся `System.Data`, позволяющий работать с таблицами и базами данных, и `System.IO`, обеспечивающий работу с файлами и потоками данных. Но по большей части вы можете называть пространства имен, как вам нравится. Или отдать это на откуп ИСР, которая будет автоматически создавать имена, взяв за основу имя программы.

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

**О:** Оно позволяет распределять код одного класса между разными файлами. При создании формы, ИСР сохраняет редактируемый вами код в файл (к примеру, `Form1.cs`), а автоматически модифицируемый — в файл (`Form1.Designer.cs`). Но оба они находятся в одном пространстве имен. Достаточно объявить пространство имен в верхней части файла, и ему будет принадлежать все, что попадает в фигурные скобки, расположенные ниже. При этом в одном файле могут находиться объекты из разных пространств имен и разных классов. Но об этом мы поговорим в следующей главе.

**В:** Что происходит с кодом, который был автоматически создан ИСР, 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 ИСР просто возвращает вас в предыдущую точку. Но при попытке отменить вставку в базу данных SQL появится окно с предупреждением, что после отмены операции вы не сможете отменить свое решение командой `Redo`.

**В:** Насколько внимательно нужно относится 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 ИСР.

### 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.

ваш код... теперь в виде магнитов



## 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 пары фрагментов!)

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

```
string Result = "";
```

Этот магнит не упал на пол...

```
MessageBox.Show(Result);
```

```
if (x == 1) {  
    Result = Result + "d";  
    x = x - 1;  
}
```

```
if (x == 2) {  
  
    Result = Result + "b,c";  
  
}
```

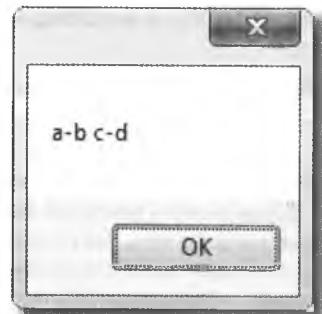
```
if (x > 2) {  
  
    Result = Result + "a";  
  
}
```

```
int x = 3;
```

```
x = x - 1;  
Result = Result + "-";
```

```
while (x > 0) {
```

Результат:



→ Ответы на с. 113.

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

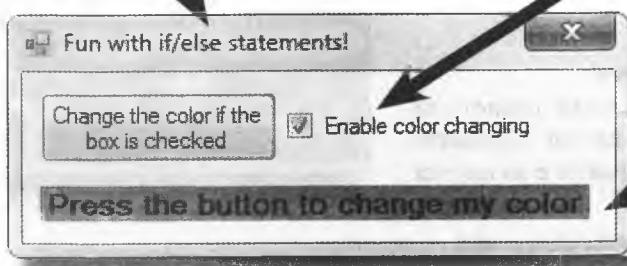
В процессе чтения книги вам предстоит создать много разных приложений, и все их нужно будет как-то именовать. Мы советуем присвоить этому приложению имя «*«2 Fun with if-else statements»*» (2 Задача с операторами if-else), полученное в результате комбинации номера главы и текста в строке заголовка формы.



## упражнение

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

**Форма, которую нужно получить.**



**Добавьте флажок.**

Перетащите флажок с панели инструментов на форму и отредактируйте свойство **Text**. Аналогичным способом нужно будет отредактировать текст кнопки и метки.

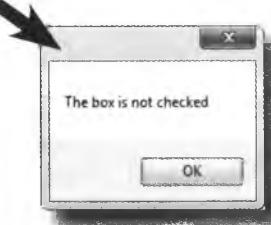
**Это метка.**

В окне Properties поменяйте размер шрифта и сделайте его полужирным. С помощью свойства **BackColor** сделайте фон красным. Выберите вариант **Red** на вкладке web colors.

**Если пользователь забыл поставить флажок, должно выводиться окно, информирующее его об этом.**

Имя флажка **checkBox1**. Если флажок установлен, должно соблюдаться условие:

```
checkBox1.Checked == true
```



**При установленном флажке щелчок на кнопке должен менять фоновый цвет метки.**

Красный фоновый цвет метки должен меняться на синий. И наоборот. Вот оператор, задающий фоновый цвет метки с именем **label1**:

```
label1.BackColor = Color.Red;
```

(Подсказка: Проверка условия, является ли фоновый цвет метки красным, выглядит практически так же, но с одним маленьким отличием!)



Построим что-то яркое! Начните с создания нового проекта Windows Forms Application.

### 1 Вот форма, которую нужно получить

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 с уже объявлена вне циклов, в них ее использовать нельзя.



### 2 Сделайте фоновый цвет броским!

Пусть результатом щелчка на кнопке станет циклическое 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 существует 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 числа: процент красного, процент зеленого и процент синего цветов.

### 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`.

**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-е! 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 с задержкой в несколько миллисекунд.

Проверку видимости объекта можно не писать в форме — `Visible == true`. Достаточно первой половины выражения.

`Visible` имеет значение `true`, пока отображается 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

Вот как должен выглядеть текст программы, циклично изменяющей фоновый цвет формы.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

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

        private void button1_Click(object sender, EventArgs e)
        {
            if (checkBox1.Checked == true)
            {
                if (label1.BackColor == Color.Red)
                {
                    label1.BackColor = Color.Blue;
                }
                else
                {
                    label1.BackColor = Color.Red;
                }
            }
            else
            {
                MessageBox.Show("The box is not checked");
            }
        }
    }
}
```

Мы присвоили решению имя Fun with If Else, поэтому ИСР назвала пространство имен таким образом.

После двойного щелчка на кнопке ИСР добавила к форме метод button1\_Click(). Этот метод запускается при каждом щелчке на кнопке.

Внешний оператор if проверяет, установлен ли флажок.

Внутренний оператор if проверяет цвет метки. Если метка красная, выполняется оператор, изменяющий цвет на синий.

Если метка не красная, оператор изменяет цвет на красный.

Это окно диалога появляется при отсутствии флажка.

Скачать код с решениями всех упражнений из книги  
можно здесь [www.headfirstlabs.com/books/hfcsharp/](http://www.headfirstlabs.com/books/hfcsharp/)



## Решение

Построим что-то яркое!

Иногда в разделе «Решение» приводится не весь код программы, а только те фрагменты, которые требовалось отредактировать.

```
private void button1_Click(object sender, EventArgs e)
{
    while (Visible) {
```

Внешний цикл выполняется, пока открыта форма.

Добавляя этот метод, ИСР 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.

Крайне важно, чтобы ваш код могли легко читать другие пользователи. Но мы намеренно показываем разные варианты, так как вы должны привыкнуть читать код, написанный в различном стиле.

```
    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);
```

```
}
```

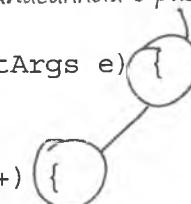
```
}
```

**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`.**



Оба цикла меняют цвет формы, но циклы работают в разные стороны.

Оператор `&&` позволяет прервать цикл `for`, как только параметр `Visible` примет значение `false`.

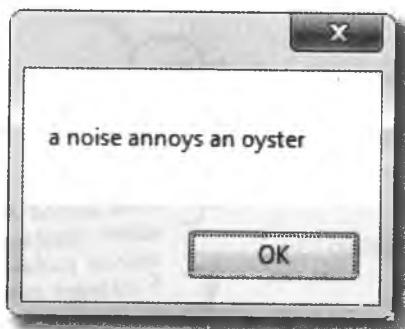
*это не такой простой ребус, как кажется*

## Ребус в бассейне



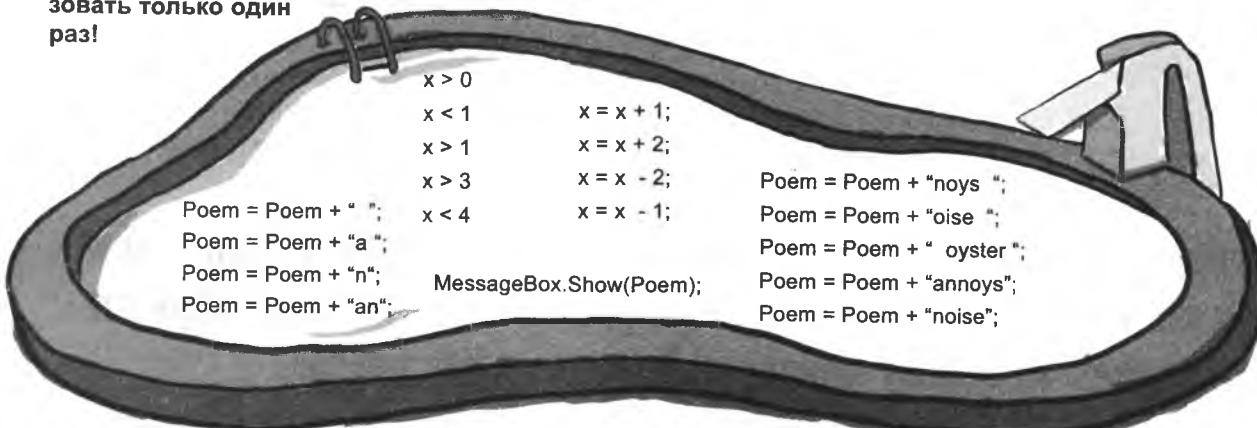
Возьмите фрагменты кода из бассейна и поместите их на пустые строчки. Каждый фрагмент можно использовать один раз. В бассейне есть и лишние фрагменты. Нужно получить программу, которая может быть скомпилирована и запущена. Имейте в виду, что задача не так проста, как кажется на первый взгляд!

**Результат:**



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!**



```
int x = 0;  
String Poem = "";
```

```
while ( _____ ) {
```

```
    if ( x < 1 ) {
```

```
    }
```

```
    if ( _____ ) {
```

```
    }
```

```
    if ( x == 1 ) {
```

```
    }
```

```
    if ( _____ ) {
```

```
    }
```

```
}
```



## Магниты с кодами

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

```

string Result = "";
int x = 3;
while (x > 0) {
{
    if (x > 2) {
        Result = Result + "a";
    }
    x = x - 1;
    Result = Result + "-";
}
if (x == 2) {
    Result = Result + "b c";
}
if (x == 1) {
    Result = Result + "d";
}
x = x - 1;
}
MessageBox.Show(Result);

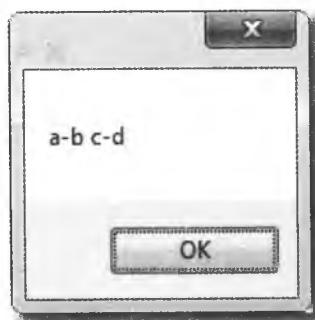
```

Этот магнит не упал на пол...

В начале цикла  $x$  равен 3, поэтому проверка условия даст результат true.

Сначала переменной  $x$  присваивается значение 2, а при втором выполнении цикла оно становится равным 1.

Результат:





## Решение ребуса в бассейне

Требовалось расположить представленные в бассейне фрагменты кода на пустых строчках таким образом, чтобы в итоге получилась работающая программа.

```
int x = 0;
String Poem = "";

while ( x < 4 ) {

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

    if ( x > 1 ) {

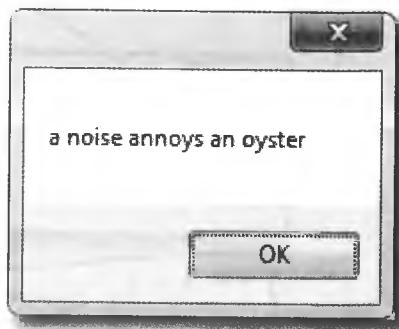
        Poem = Poem + " oyster";

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

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

        Poem = Poem + "oise ";
    }
    x = x + 1;
}
MessageBox.Show(Poem);
```

Результат:



Видите другое решение?  
Проверьте его в ИСР  
и посмотрите, как оно  
работает!



Подсказка: Существует еще  
одно решение.

**3** объекты, по порядку стройся!

## Приемы программирования

...так вот почему в классе  
Муж отсутствуют методы  
ПомогиПодому() или  
СледиЗаСвоимВесом().



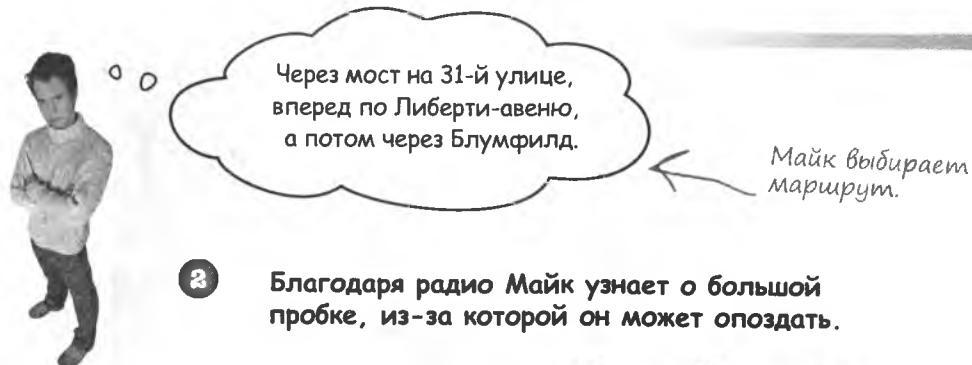
**Каждая программа решает какую-либо проблему.**

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!

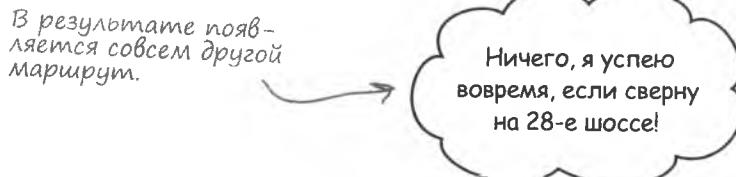
- 1 Майк обдумывает, каким маршрутом лучше поехать.



- 2 Благодаря радио Майк узнает о большой пробке, из-за которой он может опоздать.



- 3 Майк придумывает, как объехать пробку и попасть на собеседование вовремя.



## Проблема Майка с точки зрения навигационной системы

Майк использует навигационную систему GPS, которая помогает ему в перемещениях по городу.

Диаграмма класса в программе Майка. Вверху имя класса, внизу перечислены методы.

```
SetDestination("Fifth Ave & Penn Ave");
string route;
route = GetRoute();
```

Результатом работы ме-  
тода GetRoute() — строка  
с маршрутом.

### **Navigator**

SetCurrentLocation()
SetDestination()
ModifyRouteToAvoid()
ModifyRouteToInclude()
GetRoute()
GetTimeToDestination()
TotalDistance()

Навигационная  
система генерирует  
маршрут, исходя из  
пункта назначения.

"Take 31st Street Bridge to Liberty Avenue to Bloomfield"

Навигационная система  
узнает, каких улиц,  
следует избегать.

```
ModifyRouteToAvoid("Liberty Ave");
```

Теперь система может  
сгенерировать новый способ  
попасть в пункт назначения.



```
string route;
route = GetRoute();
```

"Take Route 28 to the Highland Park Bridge to Washington Blvd"

Метод GetRoute() прокладывает  
маршрут, исключив улицы,  
которые Майк хочет обходить.



Навигационная система Майка решает про-  
блему 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 значения.
```

```
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 значения, 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,  
а затем при помощи  
оператора return  
передает значение  
вызывавшему его  
оператору.

```
public int MultiplyTwoNumbers(int firstNumber, int secondNumber) {  
    int result = firstNumber * secondNumber;  
    return result;  
}
```

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 значения. Но оператор `return` может использоваться для прерывания такого метода: `if (finishedEarly) { return; }`.

## Построим программу 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 ИСР появится новая вкладка с именем `Talker.cs`.

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 `объявляем`  
переменную  
`finalString`  
и присваива-  
ем ей нулевое  
значение (ну-  
стую строку).

Метод `BlahBlahBlah()` возвращает  
значения типа `int`. Свойство `Length`  
можно добавить к любой строке  
и узнать ее длину.

К переменной `finalString`  
добавляется значение  
переменной `thingToSay` и знак  
переноса строки (`\n`).

Свойством `Length`  
обладают все строки.  
Знак переноса (`\n`)  
считается за один  
символ.

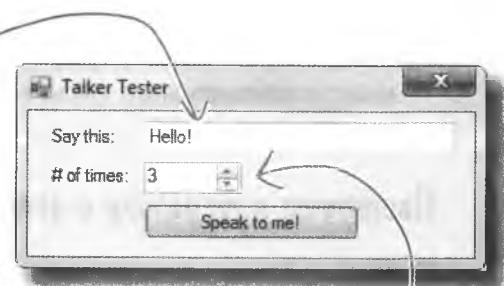
→ Переверните страницу и продолжим!

## 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,рабатывающую с новым классом!

Отредактируйте свойство `Text` таким образом, чтобы в текстовом поле по умолчанию появлялся текст «Hello!»



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);
}
```

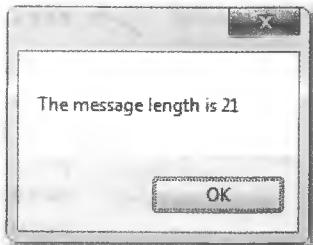
Свойству `Minimum` присвойте значение 1, свойству `Maximum` – 10, а свойству `Value` – 3.

Запустите программу! После щелчка на кнопке вы увидите два окна с текстом. Класс вызывает первое окно, форма – второе.

Метод `BlahBlahBlah()` вызывает окно с сообщением, сформированным на основе введенных пользователем параметров.



Возвращенное методом значение форма показывает в этом окне диалога.



# 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.

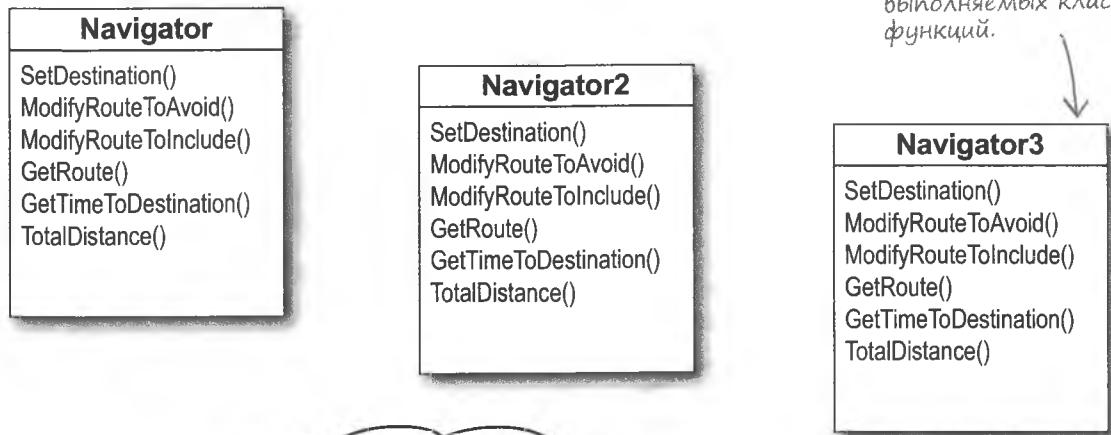


Вот если бы сравнить  
несколько маршрутов и выбрать  
из них самый быстрый...

### 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.



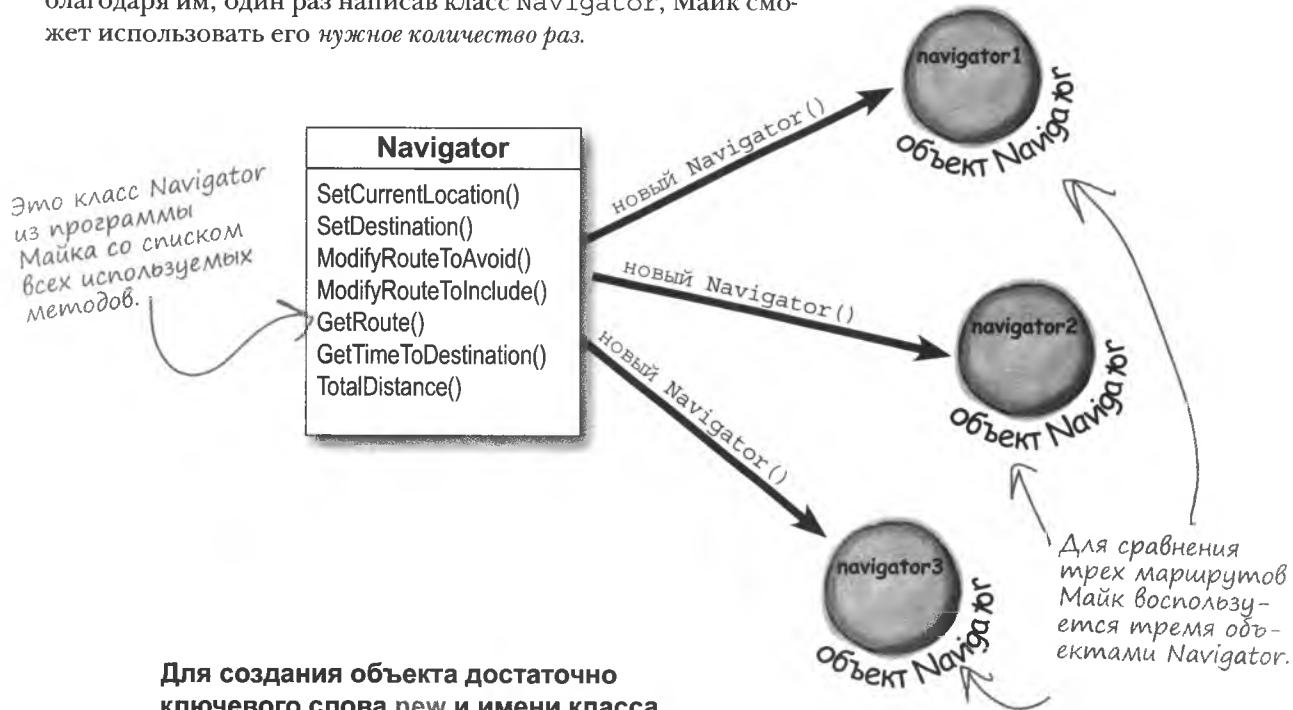
Получается, что редактировать  
метод теперь придется три раза  
вместо одного?



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*.

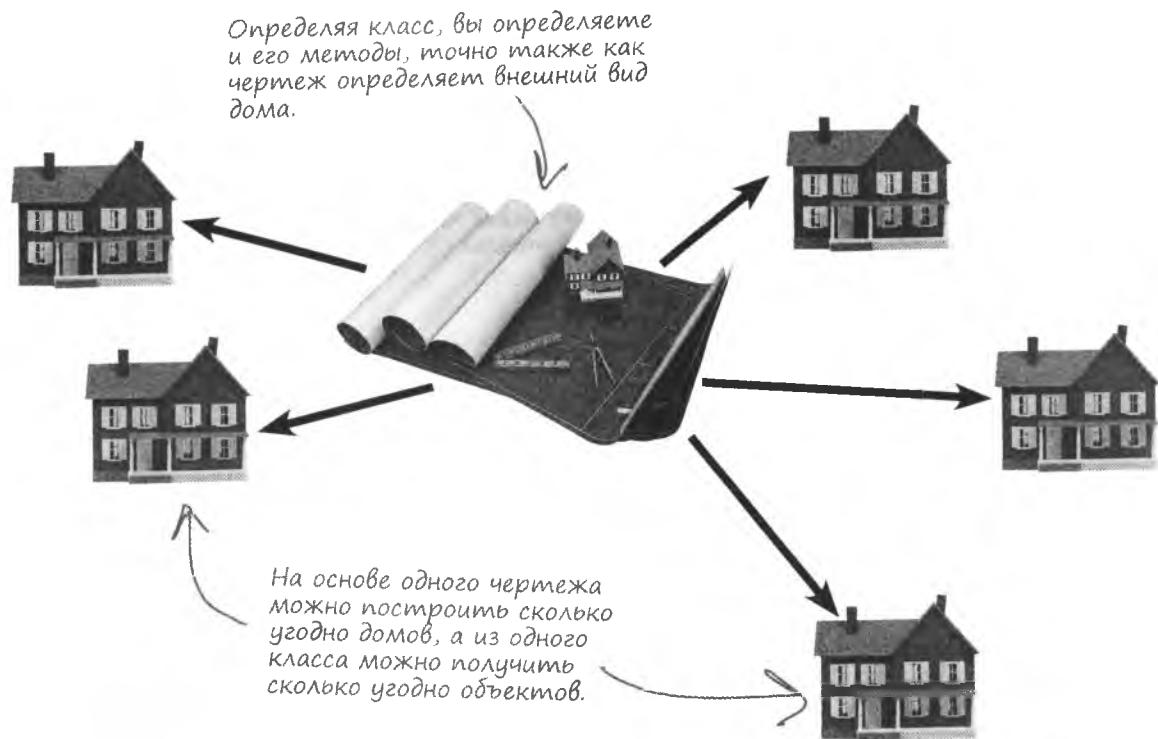


```
Navigator navigator1 = new Navigator();  
navigator1.SetDestination("Fifth Ave & Penn Ave");  
string route;  
route = navigator1.GetRoute();
```

Объект уже можно использовать! Так как он получен из класса, то содержит все входившие в класс методы.

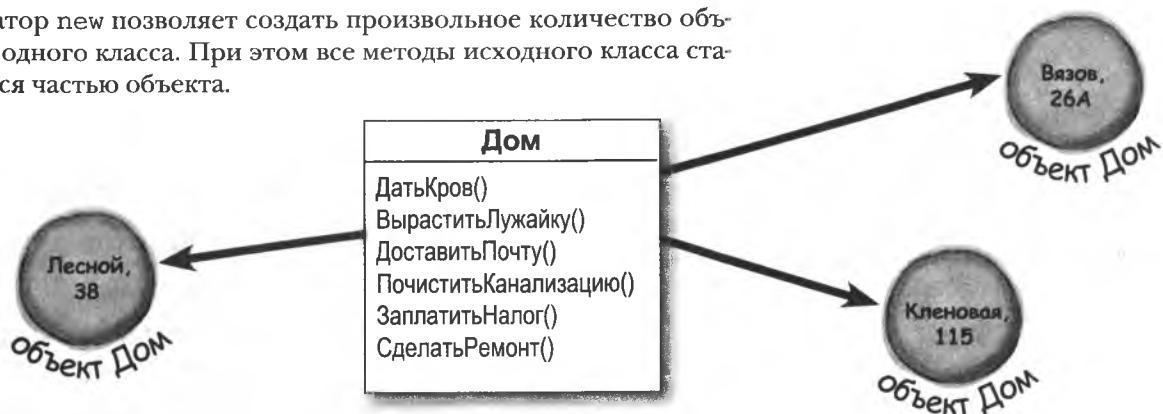
## Возьмите класс и постройте объект

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 ИСР добавила в конструктор форм при создании экземпляра класса Button.

\* ↗ Упражнение!

После: В памяти появился экземпляр класса House.



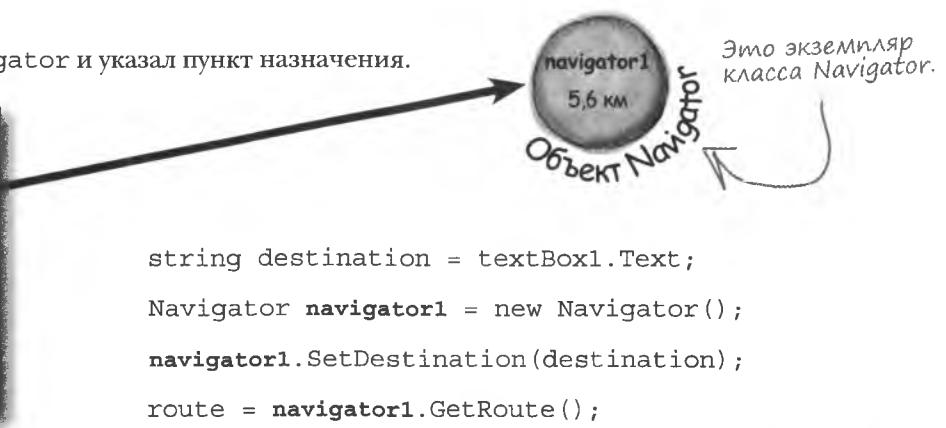
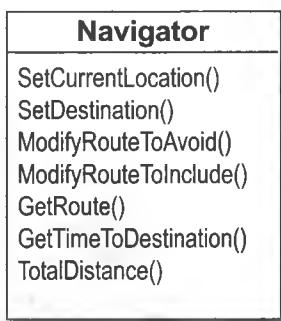
Экземпляр, образец, вещь, подобная другим. Функция поиска и замены находит все экземпляры слова и заменяет их.

## 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** обехать.
- 2 Он создал объект Navigator и указал пункт назначения.



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).

Параметрами методов SetDestination(), ModifyRouteToAvoid() и ModifyRouteToInclude() являются переменные типа string.

- 3 Третий объект Navigator называется navigator3. Майк указал пункт назначения и вызвал метод ModifyRouteToInclude() (Редактировать включаемые в маршрут улицы)



- 4 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!

**Создание нового объекта на основе класса называется созданием экземпляра класса.**



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 попытать 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.



У вас будет достаточно возможностей применить полученные знания на практике. Иногда это выполнение письменных упражнений, один 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.

## 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 2010! Но от опечаток никто не застрахован (можно перепутать 1 и букву L нижнего регистра).

**Если у вас сложности с выполнением упражнения, можно сразу посмотреть решение. Его можно даже скачать с сайта Head First Labs.**

## 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;

Navigator navigator1 = new Navigator();
navigator1.SetDestination(destination);
int distance1 = navigator1.TotalDistance();
```

Здесь Майк задавал  
пункт назначения  
и улицы, которых  
следует избегать.

А тут 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 объектов?

**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 невозможен 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.

## часто Задаваемые Вопросы

**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!

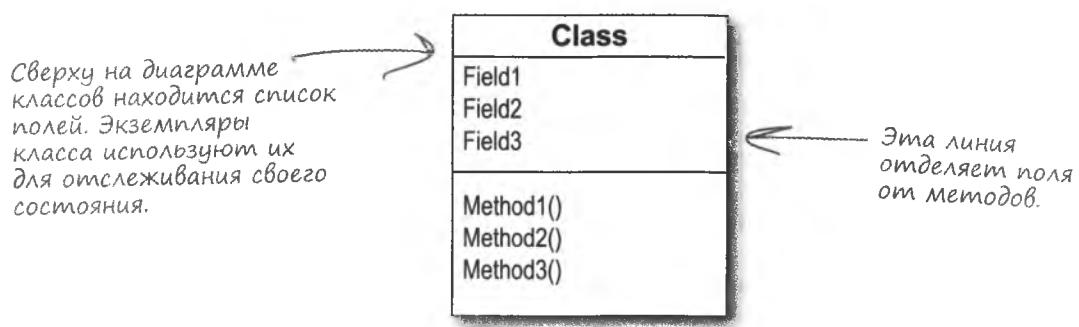
## Поля

Редактирование расположенного текста на кнопке осуществляется при помощи свойства `Text`. При внесении подобных изменений в конструктор добавляется следующий код:

```
button1.Text = "Текст для кнопки";
```

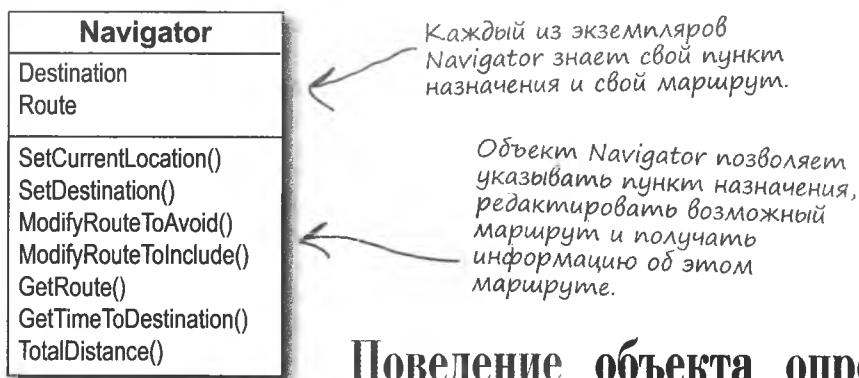
Как вы уже знаете, `button1` – это экземпляр класса `Button`. Код же редактирует поле этого экземпляра. На диаграмме классов список полей находится сверху, а список методов – снизу.

Технически вы задаете свойство. Свойства очень похожи на поля, но об этом мы поговорим чуть позже.



**Метод — это то, что объект делает. Поле — это то, что объект знает.**

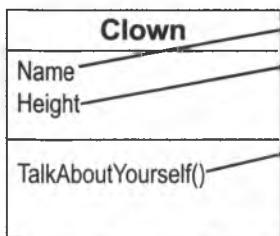
В результате создания Майком трех экземпляров класса `Navigator` его программа создала три объекта, каждый из которых отслеживает свой маршрут. При вызове метода `SetDestination()` для экземпляра `navigator2` пункт назначения указывается только для этого экземпляра. Он никак не влияет на экземпляры `navigator1` и `navigator3`.



**Поведение объекта определяется его методами, поля используются для отслеживания его состояния.**

## Создаем экземпляры!

Для добавления полей достаточно объявить переменные вне методов. Так, все экземпляры будут иметь свои копии этих переменных.



При создании экземпляра, не используйте ключевое слово static ни в объявлении класса, ни в объявлении метода.

```
class Clown {
    public string Name;
    public int Height;

    public void TalkAboutYourself() {
        MessageBox.Show("My name is "
            + Name + " and I'm "
            + Height + " inches tall.");
    }
}
```

Модификатор void указывает на отсутствие возвращаемых методом значений.

### 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.



Внимательно посмотрим на происходящее здесь

Возьми в руку карандаш



Решение

```
Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;
oneClown.TalkAboutYourself();
```

```
Clown anotherClown = new Clown();
anotherClown.Name = "Biff";
anotherClown.Height = 16;
anotherClown.TalkAboutYourself();
```

```
Clown clown3 = new Clown();
clown3.Name = anotherClown.Name;
clown3.Height = oneClown.Height - 3;
clown3.TalkAboutYourself();
```

```
anotherClown.Height *= 2;
anotherClown.TalkAboutYourself();
```

Вот как нужно было заполнить пробелы в тексте.

Операторы new создают экземпляры класса *Clown*, резервируя участки памяти и заполняя их данными об объекте.

«Меня зовут Boffo, мой рост 14 дюймов»

«Меня зовут Biff, мой рост 16 дюймов»

«Меня зовут Biff, мой рост 11 дюймов»

«Меня зовут Biff, мой рост 32 дюйма».

Новые объекты добавляются в кучу — динамически распределляемую 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.*

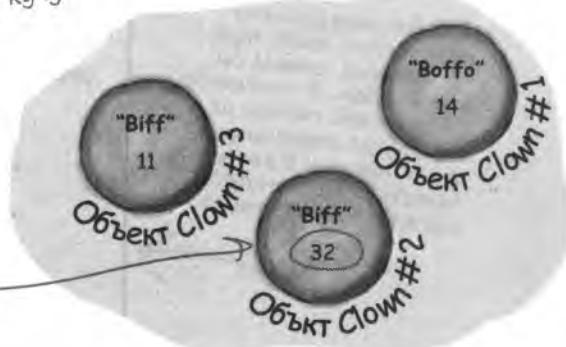
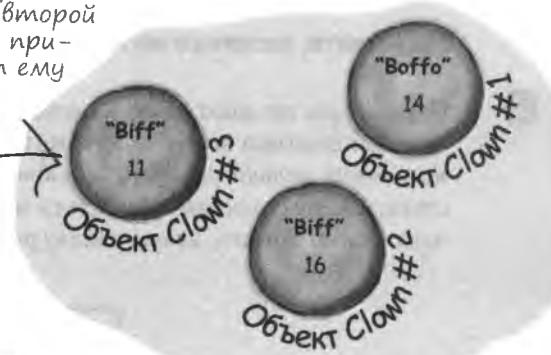
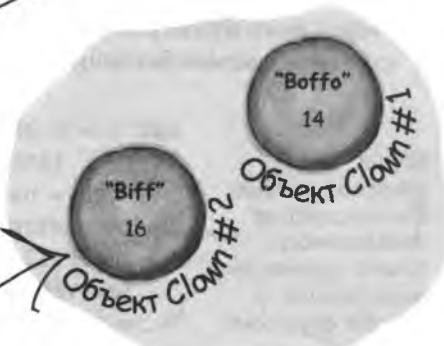
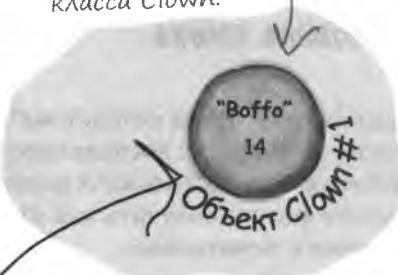
3 Clown clown3 = new Clown();  
 clown3.Name = anotherClown.Name;  
 clown3.Height = oneClown.Height - 3;  
 clown3.TalkAboutYourself();

*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.*

Это экземпляр класса Clown.



## Значимые имена

При написании кода методов выбирается структура программы. Вы ищете ответы на вопросы: воспользоваться ли 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.

- 1 Перед вами пример компактного кода. Это программа, управляющая автоматом по производству шоколадных батончиков.

```
tb, ics, m —  
ужасные имена!  
Вы никогда не  
догадаетесь,  
зачем нужны эти  
переменные и  
какова функция  
класса T.  
  
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?

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, нуж-  
но понять, зачем он  
был создан. В качестве  
отправной точки ис-  
пользуем страницу из  
инструкции, в соот-  
ветствии 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
- Запомните водой систему охлаждения
- Выпустите 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!**

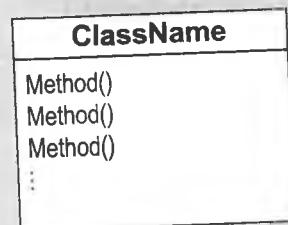
## Структура классов

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 не было предназначение программы, чем больше код напоминает о том, какая именно проблема с его помощью решается, тем проще такую программу писать, читать и редактировать.

### Диаграмма класса

Это наглядное представление класса на листе бумаги.

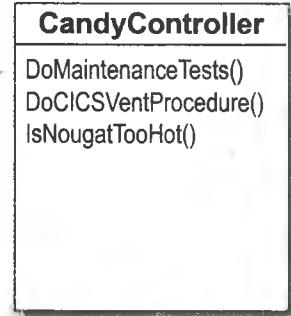
Вверху пишется имя класса, а под ним все входящие в класс методы!



### 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 относятся друг с другом код и диаграмма классов.

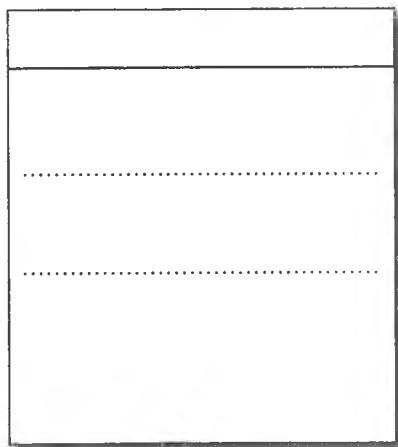
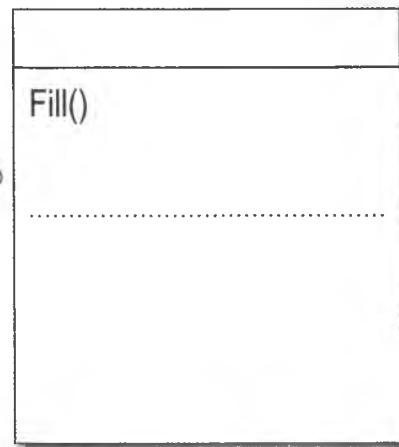
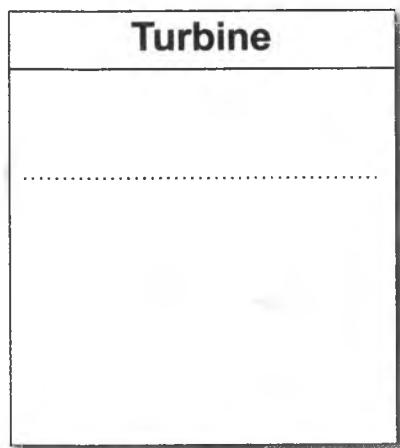
```
class CandyController {  
  
    public void DoMaintenanceTests() {  
        ...  
        if (IsNougatTooHot() == true) {  
            DoCICSVentProcedure();  
        }  
        ...  
    }  
  
    public void DoCICSVentProcedure() ...  
  
    public boolean IsNougatTooHot() ...  
}
```



Возьми в руку карандаш

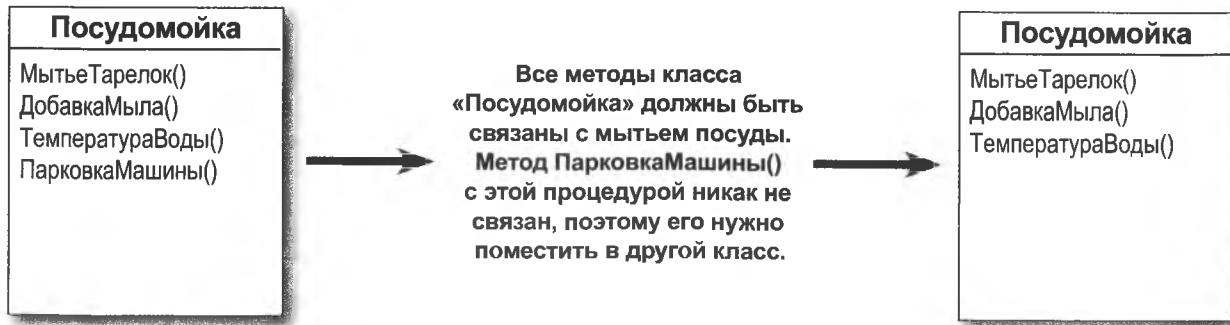


Код для управления машиной по производству шоколадных батончиков содержит еще три класса. Изучите его внимательно и заполните эти диаграммы.



## 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.



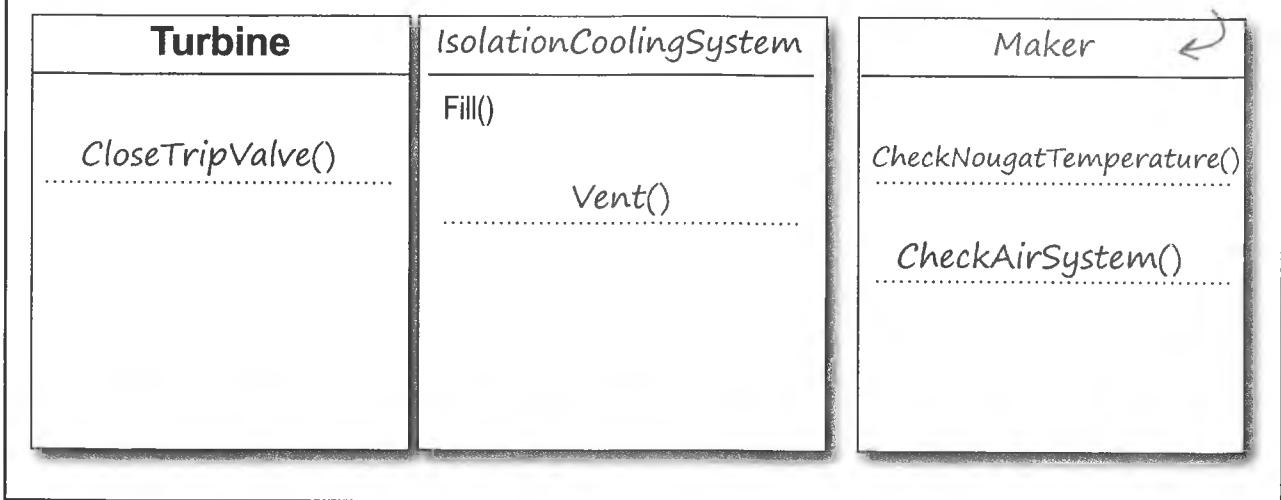
Возьми в руку карандаш



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()*.



## 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.

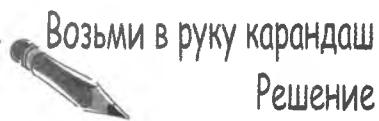
.....  
.....  
.....  
.....

### КассовыйАппарат

Продажи()  
НетПродаж()  
ЗакачатьГаз()  
ВозвратДенег()  
ВсегоВКассе()  
СписокТранзакций()  
ДобавитьСумму()  
ВычестьСумму()

Класс КассовыйАппарат является частью программы автоматизированной системы контроля в круглосуточном магазине.

.....  
.....  
.....  
.....



## 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()* не понятно 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
{
    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...";
    }
}

```

**Каждый фрагмент можно использовать несколько раз!**

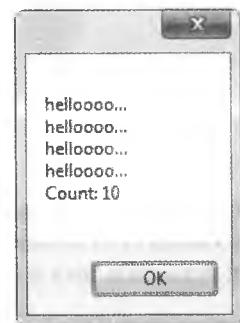
e1 = e1 + 1;	x < 4	Echo	e2 = e1;	x == 3
e1 = count + 1;	x < 5	Tester	Echo e2;	
e1.count = count + 1;	x > 0	Echo( )	Echo e2 = e1;	
e1.count = e1.count + 1;	x > 1	Count( )	Echo e2 = new Echo( );	x == 4
		Hello( )		

## Ребус в бассейне



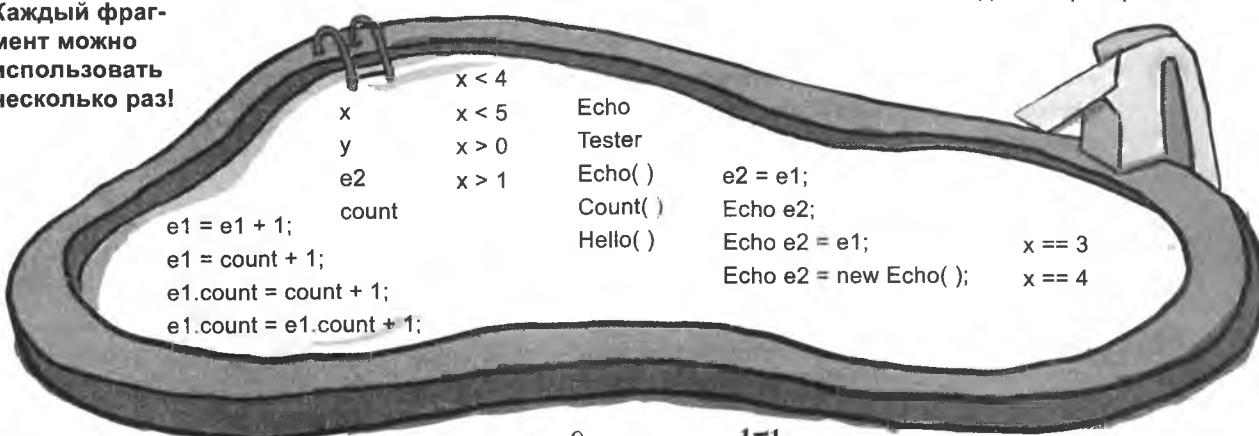
Возьмите фрагменты кода из бассейна и поместите их на пустые строчки. Фрагменты можно использовать несколько раз. В бассейне есть и лишние фрагменты. Полученная в итоге программа должна выводить окно с показанным ниже текстом.

### Результат:



### Дополнительный вопрос!

Как решить задачу, чтобы вместо **10** в последней строке оказалось **24**? Для этого достаточно заменить всего один оператор.



→ Ответ на с. 151.

далее ▶

141

## Помогите парням

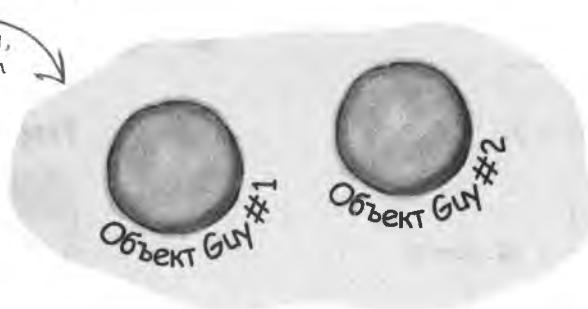
Джо и Боб все время одолживают друг другу деньги. Напишем программу отслеживания истории займов. Для начала нужно понять структуру создаваемого класса.

1

### Создадим класс *Guu* и добавим в форму два его экземпляра

Первое поле формы будет называться *joe* (для сложения за первым объектом), а второе *bob* (для сложения за вторым объектом).

Оператор *new*, создающий экземпляры, запускается в момент загрузки формы. На рисунке представлен вид кучи после этой операции.

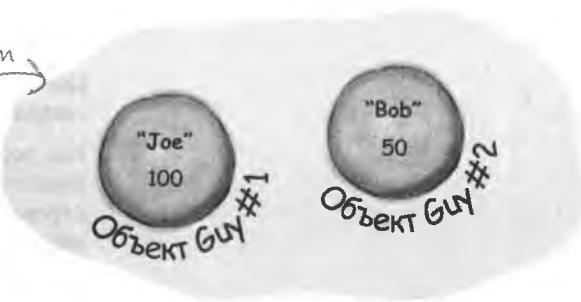


2

### Сопоставим каждому объекту *Guu* 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.



3

### Парни дают деньги и получают их назад

Метод *ReceiveCash()* увеличивает количество денег у объекта, а метод *GiveCash()* уменьшает этот параметр.



форма вызывает метод *ReceiveCash()*.

`joe.ReceiveCash(25);`

Метод возвращает сумму, которую объект добавил к полю *Cash*.

Параметром метода *ReceiveCash()* является количество получаемых денег. Поэтому запись *joe.ReceiveCash(25)* означает, что Джо получит 25 долларов.



## Guu

Name
Cash
GiveCash()
ReceiveCash()

Итак, наш класс называется *Guu* (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.

## 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 строку `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.

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 открывающихся скобок совпадало с количеством закрывающихся. ИСП поможет вам в этой задаче.



## 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 из них, а также кнопки, управляющие процессом взятия и возврата денег.

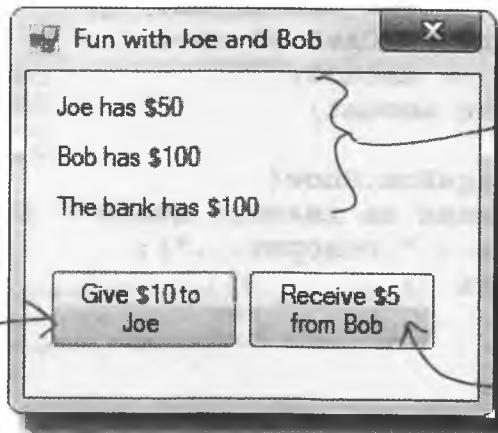


1

### 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 метками имена joesCashLabel и bobsCashLabel вместо имен label1 и label2, вы сделаете код более читабельным.

Эта кнопка вызывает метод ReceiveCash() объекта Joe, передает значение 10 и вычитает из поля bank сумму, которую получает Джо.



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.

```
Guy joe;
Guy bob;
int bank = 100;
```

```
public Form1() {
    InitializeComponent();
}
```

Значение поля bank то возрастает, то уменьшается в зависимости от того, сколько денег форма отдала объектам Guy и сколько взяла от них.

## 3

**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:

```
Метки обновляются при помощи полей Name и Cash метода Guy.
public void UpdateForm() {
    joesCashLabel.Text = joe.Name + " имеет $" + joe.Cash;
    bobsCashLabel.Text = bob.Name + " имеет $" + bob.Cash;
    bankCashLabel.Text = "В банке сейчас $" + bank;
}
```

Этот метод обновляет метки, изменяя их свойство `Text`.

## 4

**Код взаимодействия кнопок с объектами**

Убедитесь, что 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:

```
private void button1_Click(object sender, EventArgs e) {
    if (bank >= 10) {
        bank -= joe.ReceiveCash(10);
        UpdateForm();
    } else {
        MessageBox.Show("В банке нет денег.");
    }
}
```

По щелчку на кнопке `Give $10 to Joe`, форма вызывает метод `ReceiveCash()` объекта `Joe`, но только если в банке достаточно денег.

Чтобы дать деньги Джо, в банке должно быть не меньше \$10. Если их нет, появляется окно с сообщением.

```
private void button2_Click(object sender, EventArgs e) {
    bank += bob.GiveCash(5);
    UpdateForm();
}
```

Кнопка `Receive $5 from Bob` добавляет в банк деньги, полученные от Боба.

При отсутствии денег у Боба метод `GiveCash()` возвращает 0.

## 5

**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

## 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.

- 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" };
```

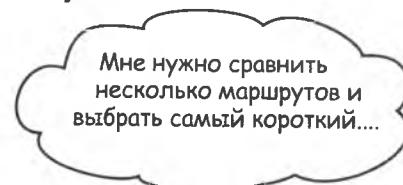
Новое объявление выполняет ту же функцию, что и приведенные вначале три строчки кода. Но оно короче и проще для чтения.

**Инициализаторы объектов экономят ваше время и увеличивают компактность и читабельность кода.**

## 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?



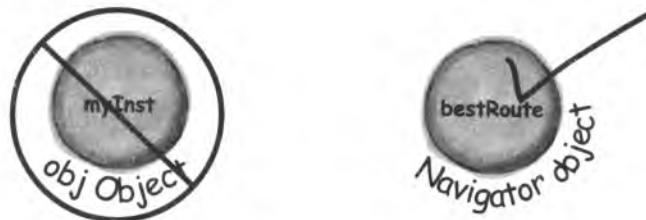
### ★ С какими реальными объектами работает программа?

Программе кормления животных в зоопарке наверняка потребуются классы для различных видов корма и различных видов животных.



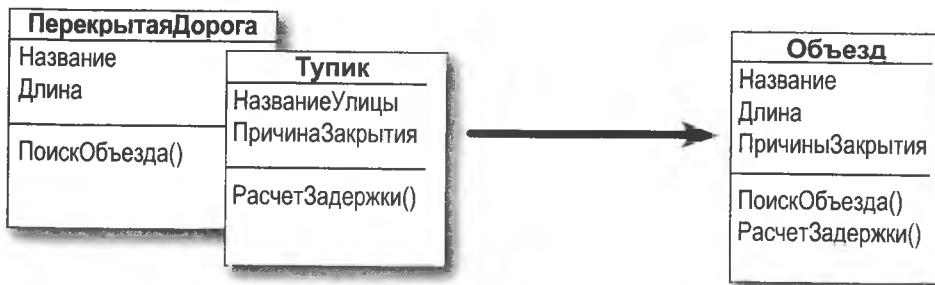
### ★ Присваивайте классам и методам значимые имена.

Другие пользователи должны понимать назначение классов и методов по виду их имен.



### ★ 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.





## Упражнение

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.

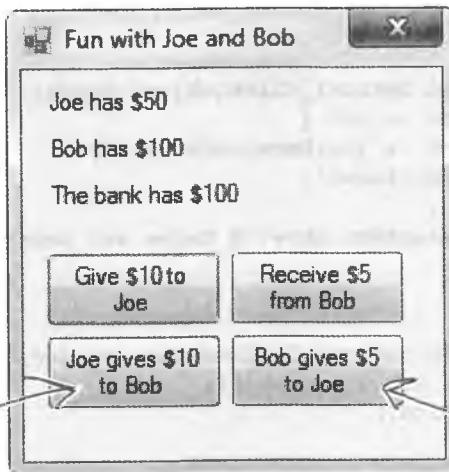
Если вы поспешили и уже щелкнули на кнопке, удалите ее, добавьте заново и присвойте новое имя. Затем удалите старый метод button3\_Click() и добавьте новый.

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.

Эта кнопка заставляет Джо отдать 10 долларов Бобу, поэтому воспользуйтесь свойством «(Name)» в окне Properties, чтобы присвоить ей имя joeGivesToBob.



Эта кнопка заставляет Боба отдать Джо 5 долларов. Присвойте ей имя bobGivesToJoe.

3

### Задавим 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.

## решение упражнения



### упражнение

### Решение

```
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();
    }
}
```

Важно помнить, кто дает деньги, а кто их получает.

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().

Результатом работы метода GiveCash() используются в качестве параметра метода ReceiveCash().

## Решение Ребуса в бассейне



Требовалось расположить представленные в бассейне фрагменты кода на пустых строках таким образом, чтобы в итоге получилась работающая программа. Вот как это нужно было сделать:

```

public partial class Form1 : Form
{
    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;

## 4 типы и ссылки

10:00 утра.

Куда подевались наши данные?

Они только что отправились  
в мусорную корзину.



**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

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 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  $\pm 5.0 \cdot 10^{324}$  до  $\pm 1.7 \cdot 10^{308}$  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

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.

Часто вы меняете тип переменной, а проблема может быть решена и при помощи «циклического присваивания», о котором мы говорим через пару страниц.

Буква «`s`» означает «со знаком». То есть число может быть отрицательным.

## Типы для хранения очень **БОЛЬШИХ** и очень маленьких чисел

Иногда семи значащих цифр оказывается недостаточно. Бывает так, что  $10^{38}$  – недостаточно большое число, а  $10^{-45}$  – недостаточно малое. С такими проблемами сталкиваются программы финансового учета и научных исследований, 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;

Тип данных  
`decimal` часто  
встречается  
в программах  
финансового  
учета.

- ★ `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.

«Константа» – это число, которое вы вводите в код. В выражении `<int i = 5;> 5` – это константа.

Свойство `Value` элемента управления `numericUpDown` принадлежит типу данных `decimal`.

### 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 'double' to 'int'. An explicit conversion exists (are you missing a cast?)

ИСР 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 «тысячу» (деньги). Я не шучу!

Попытавшись присвоить константу типа `float` переменной типа `int`, 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', 'z'). 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.

## МОЗГОВОЙ ШТУРМ

В Windows 7 у Калькулятора 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

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 не-численные 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`.

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.



## 10 литров в 5-литровой банке



Компилятор «видит», что вы пытаетесь положить стакан типа `int` в стакан типа `short`. Содержимое стакана `int` при этом не имеет значения.

**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";
```

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.



## Приведение типов

Посмотрим, что произойдет при попытке назначить значение типа decimal переменной типа int.

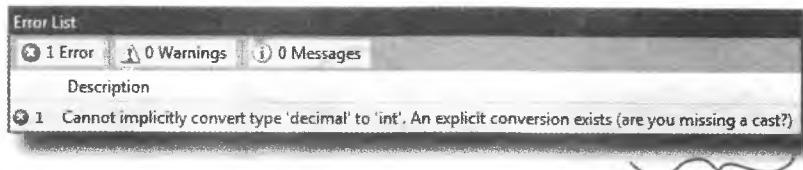


- Создайте новый проект и добавьте кнопку. Для метода `Click()` этой кнопки напишите:

```
decimal myDecimalValue = 10;
int myIntValue = myDecimalValue;

MessageBox.Show("The myIntValue is " + myIntValue);
```

- Попытавшись запустить программу, вы получите сообщение об ошибке:



- Устранитесь ошибку, преобразовав тип decimal в int. Внесите во вторую строку следующие изменения:

```
int myIntValue = (int) myDecimalValue;
```

Этот оператор делает  
приведение типа decimal  
к типу int.

### 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.

ИСР предполагает, что вы забыли осуществить приведение типов (casting).

Вернитесь в начало предыдущей главы и посмотрите, как приведение типов использовалось для передачи значения `NumericUpDown` форме `Talker Tester`.

**Упражнение**

**Решение**

Вот операторы, которые не будут компилироваться из-за несовпадения типов данных или из-за того, что им пытаются присвоить слишком большое или слишком маленькое значение.

`short y = 78000;`

Слишком большое число. Тип `short` хранит значения от -32,767 до 32,768.

`byte days = 365;`

Для этой переменной вам нужен тип `short`, так как переменные типа `byte` хранят числа до 256.

`bool isDone = yes;`

Переменной типа `bool` можно присвоить только значения `true` или `false`.

## 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 сохранение **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.



### 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;
double myFloat = 16.4D;
myFloat = myInt + myFloat;
```

После числа, которое присваивается переменной типа `double`, нужно добавлять `D`, 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`.

Фактически «зацикливание» – это операция деления по модулю. Переключите калькулятор в Инженерный режим, наберите `365 Mod 256`, и вы получите 109.

### 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 в 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. ИСР 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.

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:

```
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.

← Параметр — это то, что вы определяете внутри метода, а аргумент — что в него передается. Метод с параметром типа `int` может принять аргумент типа `byte`.

**Ошибка «неверный аргумент» означает, что вы попытались вызвать метод с переменными, тип которых не совпадает с его параметрами.**

Все работает, пока вы передаете методу то, что он ожидает получить (логическую переменную), вызов `MyMethod(true)` или `MyMethod(false)` позволяет легко скомпилировать код.

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()`.

← Переменной, параметру или полю можно назначить произвольное значение при помощи типа `object`.

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.



## **таблица зарезервирована**

Вы можете использовать для имен переменных даже зарезервированные слова. Достаточно поставить в начале знак @.

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



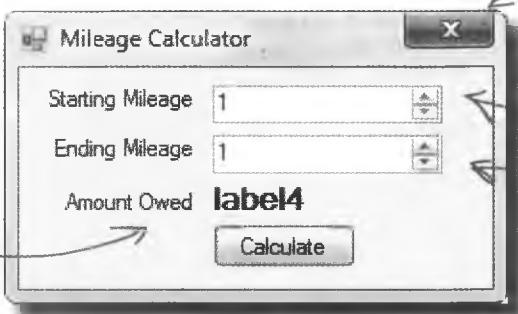
## Упражнение

Создадим калькулятор расходов для деловой поездки. Вы будете вводить данные с одометра в начале и конце путешествия, а счетчик 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:

Для этой метки используйте полужирный шрифт и кегль 12 pt.



Избавьтесь от кнопок управления размером окна.

Свойству Minimum этих двух элементов NumericUpDown присвойте значение 1, а свойству Maximum – значение 999999.

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 выводиться 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.



## Упражнение Решение

Вот как выглядит код для первой части упражнения по расчету компенсации.

```
public partial class Form1 : Form
{
    int startingMileage; ← Так как в данном
    int endingMileage; ← случае число может
    double milesTraveled; ← превысить 999999,
    double reimburseRate = .39; ← типы short или byte не
    double amountOwed; ← подходят.
    public Form1()
    {
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        startingMileage = (int) numericUpDown1.Value; ← Помните, что для
        endingMileage = (int) numericUpDown2.Value; ← элемента управления
        if (startingMileage <= endingMileage) { ← numericUpDown вам
            milesTraveled = endingMileage - startingMileage; ← нужно поменять тип
            amountOwed = milesTraveled *= reimburseRate; ← decimal на тип int?
            label4.Text = "$" + amountOwed; } ←
        } else { ← Здесь рассчитыва-
            MessageBox.Show( ← вается количество
                "Начальный пробег не может превышать конечный", ← пройденных миль,
                "Cannot Calculate Mileage"); ← которое затем
            } ← умножается на
        } ← тариф возмеще-  
ния расходов.
    }
}

Кнопка, кажется, работает, но есть одна проблема. Вы можете указать ее?
```

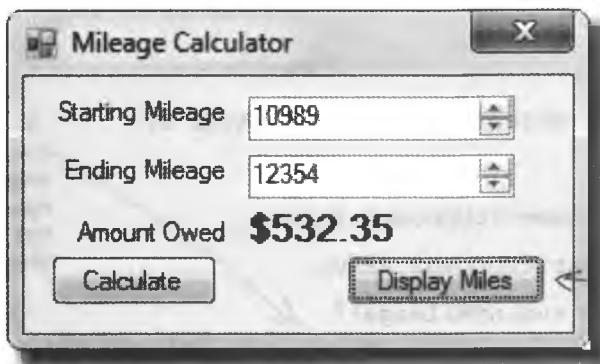
Помните, что для элемента управления numericUpDown вам нужно поменять тип decimal на тип int?

Здесь рассчитыва-  
вается количество  
пройденных миль,  
которое затем  
умножается на  
тариф возмеще-  
ния расходов.

Здесь использует-  
ся альтернатив-  
ный способ вызова  
Метода MessageBox.  
Show(). Первый па-  
метр — это ото-  
бражаемое сообщение,  
а второй — заголовок.

**1****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.

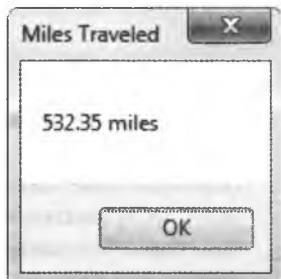
**2****Понадобится всего одна строка**

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**

Ведите 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.

**4****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 также используют переменные:

### Работа с типом int

- Пишем оператор объявления типа.

```
int myInt;
```

- Присваиваем новой переменной значение.

```
myInt = 3761;
```

- Используем переменную в коде.

```
while (i < myInt) {
```

### Работа с объектом

- Пишем оператор объявления типа.

```
Dog spot;
```

При объявлении переменной название класса используется в качестве типа.

- Присваиваем новой переменной значение:

```
spot = new Dog();
```

- 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.

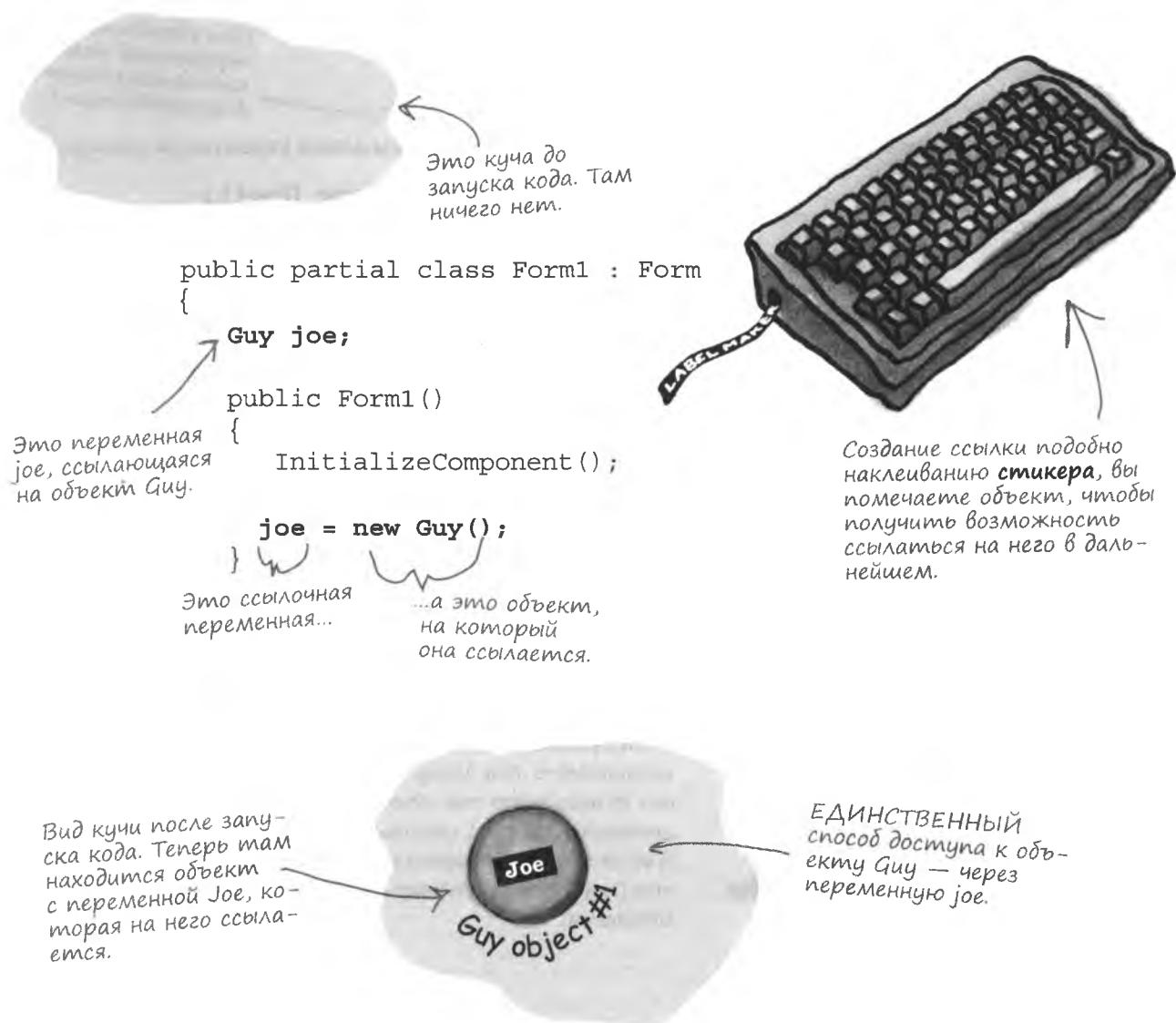
получи ссылку

## 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:



## 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.

Это объект типа Guy. Он ОДИН, но на него МНОГО ссылок.



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 экземпляров Guy. Вам нужна ссылочная переменная, например, `joe`, которой будет присвоен конкретный экземпляр Guy `joe = new Guy()`.

Теперь можно вызывать 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!

**Для работы с хранящимися в памяти объектами используются ссылки, которые являются переменными. Тип этих переменных определяется классом объекта, на который они ссылаются.**

Различные методы могут использовать экземпляр Guy в различных целях. Поэтому ссылочным переменным разумно присвоить разные имена в зависимости от контекста.

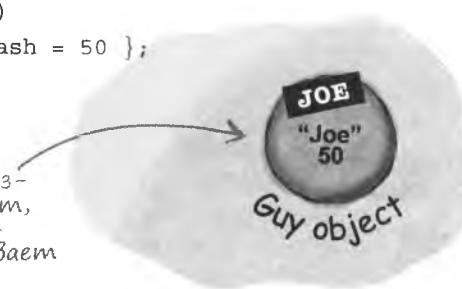
## При отсутствии ссылок объект превращается в мусор

Если все ссылки на объект исчезают, программа теряет к нему доступ. Теперь объект предназначен для **сборки мусора (garbage collection)**. C# избавляется ото всех объектов, на которые нет ссылок, освобождая место в памяти.

### 1 Вот код создания объекта.

```
Guy joe = new Guy()  
{ Name = "Joe", Cash = 50 };
```

Оператор new создает новый объект, а ссылочная переменная Joe указывает на него.



### 2 Был создан второй объект.

```
Guy bob = new Guy()  
{ Name = "Bob", Cash = 75 };
```

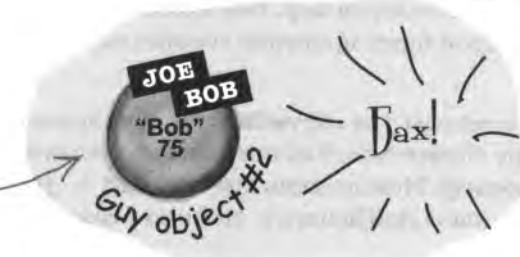
Теперь у вас два экземпляра Guy и две ссылочные переменных.



### 3 Пусть ссылка на первый экземпляр начнет указывать на второй.

```
joe = bob;
```

Теперь переменные joe и bob ссылкаются на один и тот же объект.



Ссылка на первый экземпляр не осталась...

... поэтому объект удаляется!

Объект остается в куче, пока на него есть хотя бы одна ссылка. Как только последняя ссылка исчезает, объект удаляется.

## 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



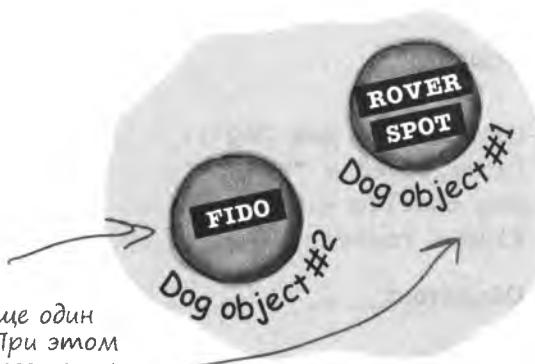
Ссылок: 1

2 Dog fido = new Dog();  
fido.Breed = "Beagle";  
Dog spot = rover;

Объектов: 2

Ссылок: 3

Fido — это еще один объект Dog. При этом Spot — это всего лишь дополнительная ссылка на первый объект.

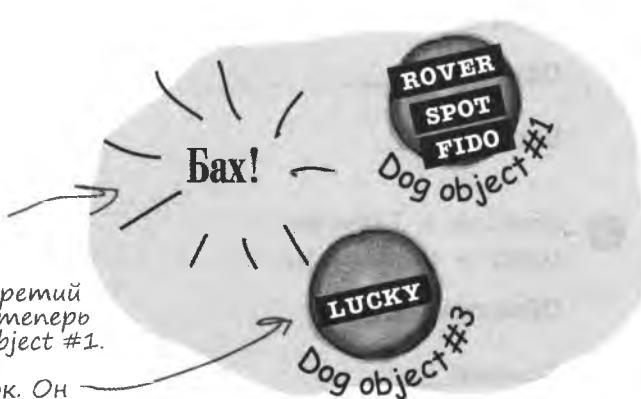


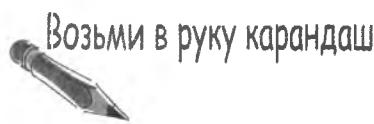
3 Dog lucky = new Dog();  
lucky.Breed = "Dachshund";  
fido = rover;

Объектов: 2

Ссылок: 4

Lucky — это третий объект. А Fido теперь ссылается на Object #1. На Object #2 не остается ссылок. Он удаляется.





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: \_\_\_\_\_

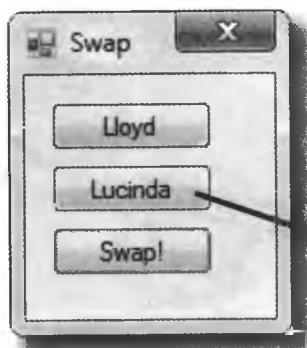
Ссылок: \_\_\_\_\_



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:



Кнопка Lucinda вызывает метод lucinda.WhoAmI(), отображающий вот такое окно.

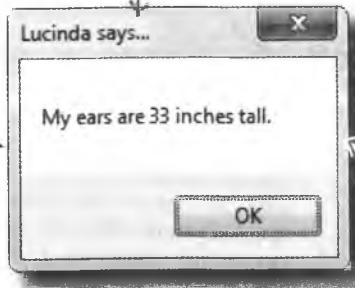
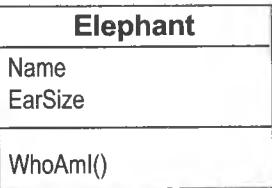


Диаграмма для класса Elephant. Name, EarSize, WhoAmI.



убедитесь, что в тексте содержится информация о размере уха, а в строке заголовка фигурирует имя слона.

### 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 вызывает 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;

**Объектов:** 4

**Ссылок:** 7

Charlie, как и Fido, начал указывать на объект #3. Затем Fido начал ссылаться на object #1.

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.





## Упражнение

### Решение

Посмотрим, как же поменять между собой ссылки на два экземпляра, чтобы ни один из них при этом не оказался удаленным.

```
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...");
    }
}
```

Это определение класса Elephant. Не забудьте строку «`using System.Windows.Forms;`», иначе оператор `MessageBox` не будет работать.

```
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");
    }
}
```

Это код класса `Form1` из файла `Form1.cs`.

Ссылка `Holder` нужна для того чтобы объект `Lloyd` не исчез, после того как ссылка будет перенаправлена на объект `Lucinda`.

Типы данных `string` и `array` отличаются от всех остальных отсутствием фиксированной длины.

Оператор `new` не используется, так как нам не нужен еще один экземпляр `Elephant`.

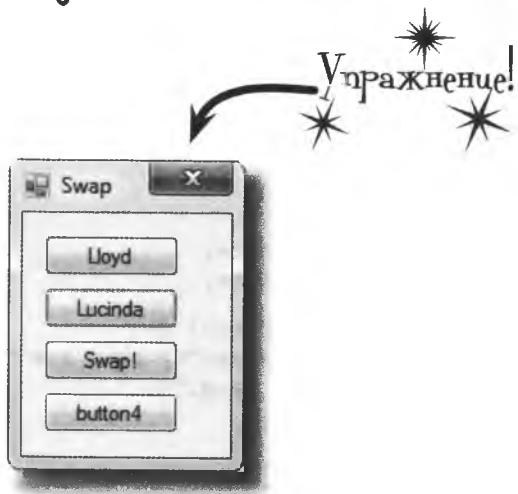


Как вы считаете, почему к классу `Elephant` не был добавлен метод `Swap()`?

## 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)
{
```

Этот оператор присваивает переменной EarSize значение 4321 тому объекту, на который покажет ссылка lloyd.

```
    lloyd = lucinda;
    lloyd.EarSize = 4321;
    lloyd.WhoAmI();
```

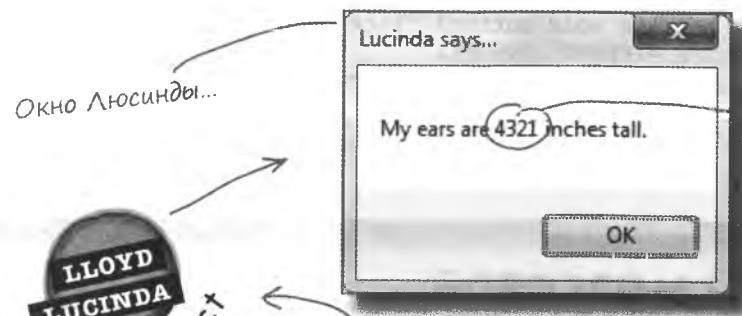
Метод WhoAmI() вызывается для объекта lloyd.

После запуска кода переменные lloyd и lucinda будут ссылаться на ОДИН объект Elephant.

Но lloyd указывает туда же, куда и lucinda.



- 3 Щелкните на кнопке, чтобы проверить свои догадки. Смотрите, это окно Lucinda. При том что вы вызывали метод WhoAmI () для Ллойда.



Но это значение переменной EarSize было определено для Ллойда. Чемо происходит?

Исчезла разница между переменными lloyd и lucinda. Какую бы из них вы ни редактировали, будет меняться объект, на который они ОБЕ указывают.

Данные при этом не переписывались, изменились только ссылки.

## Особый случай: массивы

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:

Для объявления массива  
нужно указать его тип,  
поставив следом квадратные скобки.

Новый массив создается командой new, как и любой другой объект.

```
bool[] myArray;
myArray = new bool[15];
```

`bool[] myArray = new bool[15];`

Этот массив состоит из 15 элементов.

**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**.

Нумерация элементов массива начинается с 0. Эта строка присваивает пятому элементу массива значение true.

В памяти массив всегда хранится одним блоком, сколько бы переменных в нем ни было.

Тип элементов массива → int[] heights;

```
heights = new int[7];
heights[0] = 68;
heights[1] = 70;
heights[2] = 63;
heights[3] = 60;
heights[4] = 58;
heights[5] = 72;
heights[6] = 74;
```

Вы ссылаетесь на элементы массива по их индексу



Массив – это единий объект, хотя и состоящий из семи обычных переменных.

## 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 Dog.

Рассмотрим код создания массива из 7 переменных типа Dog. Стока инициализации создает только ссылочные переменные. За ней следуют две строки new Dog(), то есть создаются два экземпляра класса Dog.

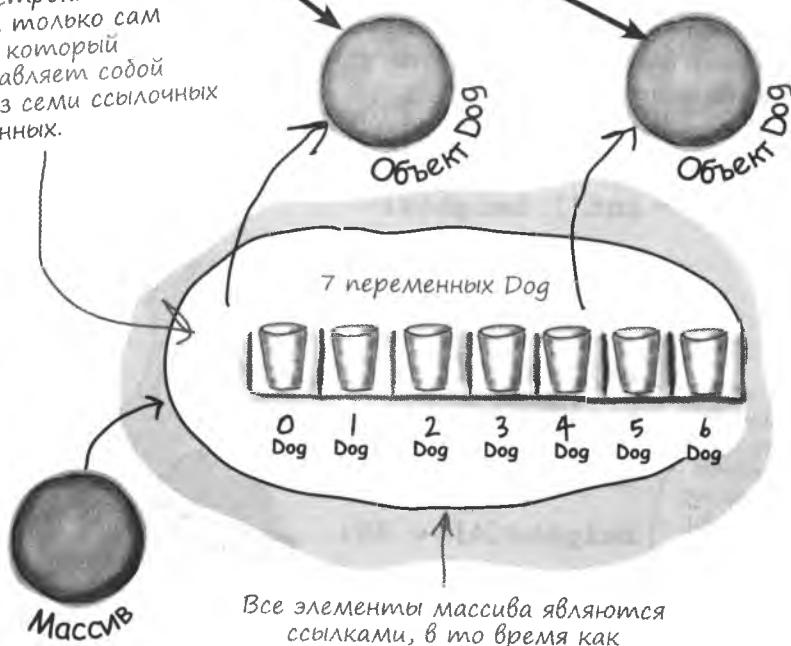
```
Dog[] dogs = new Dog[7];  
dogs[5] = new Dog();  
dogs[0] = new Dog();
```

Эта строка  
объявляет  
переменные,  
в которых будем  
храниться  
массив ссылок  
на объекты Dog.



Создаются экземпляры  
Dog(), которые поме-  
щаются в нулевой и пя-  
тый элементы массива.

Первая строка кода  
создает только сам  
массив, который  
представляет собой  
набор из семи ссылочных  
переменных.



Число в квадрат-  
ных скобках на-  
зывается индексом  
(index) и исполь-  
зуется для обраще-  
ния к элементам  
массива. Индекс  
первого элемента  
всегда равен нулю.

# 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?

1

## 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.

Упражнение!

MenuMaker
Randomizer
Meats
Condiments
Breads
GetMenuItem()

Из элементов этих трех массивов будет случайным образом создаваться сэндвич.

```
class MenuMaker {
    public Random Randomizer;
    Поле Randomizer
    содержит ссылку
    на объект Random.
    Метод Next() ген-
    рирует случайные
    числа.
    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" };
}
```

Для доступа к элементу укажите его номер  
в скобках. Например, элементом Breads[2] имеет  
значение white.

2

## Метод GetMenuItem()

Это инициализатор коллекций, с которым вы подробно познакомитесь в главе 8.

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 меньше значения 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) вернет 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 меньше, 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).

Я заказываю еду только у Джо!



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();  
}
```

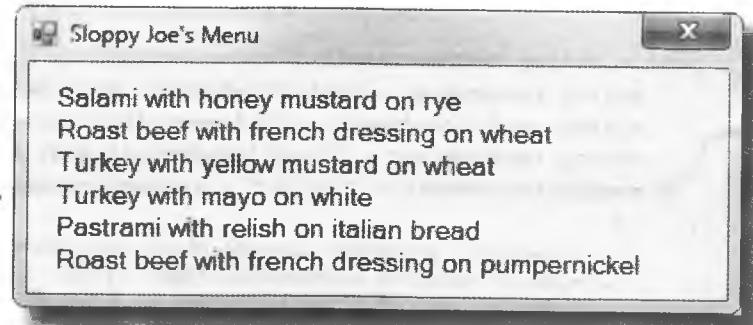
```
    MenuMaker menu = new MenuMaker() { Randomizer = new Random() };  
  
    label1.Text = menu.GetMenuItem();  
    label2.Text = menu.GetMenuItem();  
    label3.Text = menu.GetMenuItem();  
    label4.Text = menu.GetMenuItem();  
    label5.Text = menu.GetMenuItem();  
    label6.Text = menu.GetMenuItem();  
}
```

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.



# Ссылки позволяют объектам обращаться друг к другу

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.

Elephant	
Name	
EarSize	
WhoAmI()	
TellMe()	
SpeakTo()	

## 1 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.

## 2 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);
}
```

Посмотрим, как это работает.

```
lloyd.SpeakTo(lucinda, "Hello");
```

Этот метод класса Elephant вызывает метод `TalkTo()` (Поговори с), позволяющий одному слону разговаривать с другим.

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;`

Пока существует один объект, ссылка `fido` имеет значение `null`.

`Dog lucky = new Dog();`



`fido = new Dog();`

Когда ссылка `fido` указывает на объект, она больше не равна `null`.

`lucky = null;`

Если присвоить `lucky` значение `null`, объект будет уничтожен, так как на него не останется ссылок.



**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`?

### часто задаваемые вопросы

**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 первой главы мы говорили об **Общязыковой 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 объявили 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 };
```

Вы создаете массив из 7 ссылок Elephant().

Первый индекс любого массива равен 0, поэтому первым элементом будет Elephants[0].

```
    Elephant biggestEars = elephants[0];
```

Итерация #1 biggestEars.EarSize = \_\_\_\_\_

```
    for (int i = 1; i < elephants.Length; i++)
```

```
{
```

```
    if (elephants[i].EarSize > biggestEars.EarSize)
```

```
{
```

```
        biggestEars = elephants[i];
```

Итерация #2 biggestEars.EarSize = \_\_\_\_\_

```
}
```

Точка останова в этой строке поможет отследить значения elephants[i] всех элементов массива.

Итерация #3 biggestEars.EarSize = \_\_\_\_\_

```
    MessageBox.Show(biggestEars.EarSize.ToString());
```

Итерация #4 biggestEars.EarSize = \_\_\_\_\_

```
}
```

Будьте внимательны, этот цикл начинается со **второго элемента** массива (с индексом 1) и выполняется шесть раз, пока i не становится равной длине массива.

Итерация #5 biggestEars.EarSize = \_\_\_\_\_

Итерация #6 biggestEars.EarSize = \_\_\_\_\_



## 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];
```

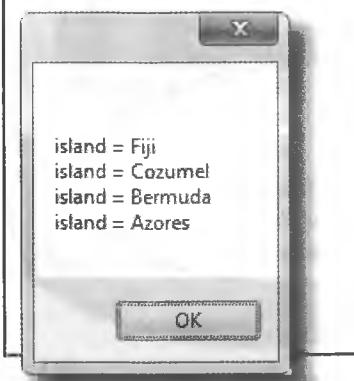
```
result += "\nisland = ";
```

```
int[] index = new int[4];
```

```
y = y + 1;
```

```
private void button1_Click (object sender, EventArgs e)
{
```

```
    string result = "";
```



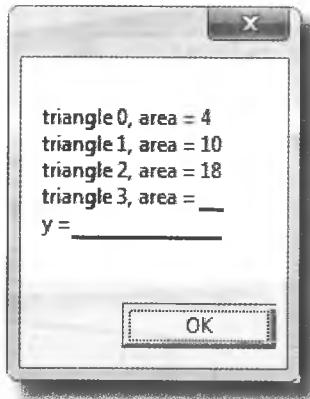
→ ОмБем на с. 193.

## 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!

Вспользуйтесь фрагментами из бассейна, чтобы составить код, заполняющий даже пустые поля в нижней части окна диалога.

Каждый фрагмент кода можно использовать несколько раз!

area	4, t5 area = 18
ta.area	4, t5 area = 343
x	27, t5 area = 18
y	27, t5 area = 343

```

Triangle [ ] ta = new Triangle(4);
Triangle ta = new [ ] Triangle[4];
Triangle [ ] ta = new Triangle[4];

```

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;  
 }  
}

Подсказка: Метод SetArea() НЕ является статическим.

int x;	x = x + 1;      ta.x
int y;	x = x + 2;      ta(x)
int x = 0;	x = x - 1;      ta[x]
int x = 1;	ta = new Triangle();
int y = x;	ta[x] = new Triangle();
28	ta.x = new Triangle();
30.0	x < 4
	x < 5

## 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 растет. Если же буква введена неправильно, уровень уменьшается. 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**:





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

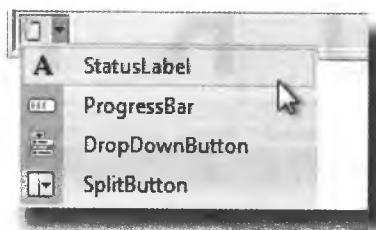
Щелкните на стрелке справа от элемента управления StatusStrip и выберите в появившемся меню вариант StatusLabel:



РАССЛАБЬТЕСЬ

Вы познакомились с тремя новыми элементами управления!

Задать свойства для ListBox, StatusStrip и Timer вы сможете знакомым способом.



- ★ Свойству SizingGrip элемента StatusStrip присвойте значение False.
- ★ В окне Properties присвойте свойству (Name) значение correctLabel, а свойству Text значение Correct: 0 (Правильных: 0). Добавьте три аналогичные метки: missedLabel, totalLabel и accuracyLabel (Пропущенных, Всего и Точность).
- ★ Добавьте StatusLabel. Укажите для Spring значение True, для TextAlign значение MiddleRight и для Text – Difficulty (Сложность). Напоследок добавьте элемент ProgressBar и присвойте ему имя difficultyProgressBar.

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 вида формы. Его функция **снова и снова вызывать некий метод**. Присвойте свойству Interval значение 800, чтобы вызов метода проходил 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?





4

## 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 результатом 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 это не сделать?



6

**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:

```

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;

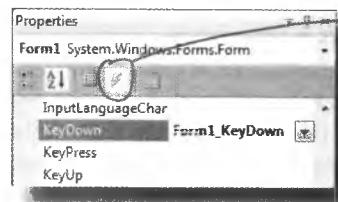
        // При правильном нажатии клавиши обновляем объект Stats,
        // вызывая метод Update() с аргументом true
        stats.Update(true);
    }
    else
    {
        // При неправильном нажатии клавиши обновляем объект Stats,
        // вызывая метод Update() с аргументом false
        stats.Update(false);
    }

    // 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 буквы в `ListBox` на- жатой игро- ком*

*Уменьшая числа, вычитаемые из параметра `timer1.Interval`, вы облегчаете игру, а увеличивая — усложняете ее.*



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.

7

**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()`.

**namespace**

Еще раз напомним, что в C# существует примерно 77 зарезервированных слов. Вам было предложено объяснить назначение некоторых из них. Вот правильные ответы.

**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 };
```

Помните, что цикл начинается со второго элемента массива? Как вы думаете, почему?

```
Elephant biggestEars = elephants[0];
for (int i = 1; i < elephants.Length; i++)
{
    if (elephants[i].EarSize > biggestEars.EarSize)
    {
        biggestEars = elephants[i];
    }
}
```



Поместите сюда точку останова для проверки результата.

```
MessageBox.Show(biggestEars.EarSize.ToString());
```

Цикл for начинается со второго слона и сравнивает размеры ушей. Если уши следующего слона большие, ссылка biggestEars начинает указывать на него. Таким образом определяется слон с самыми большими ушами.

Ссылка  
biggestEars  
отслеживает  
самый боль-  
шой элемен-  
т в цикле.

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 ходильнике 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;
}
```

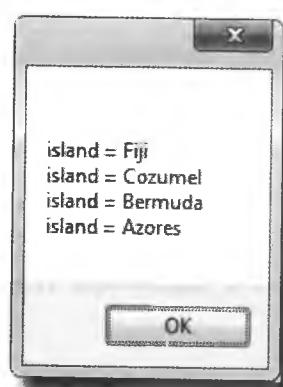
```
MessageBox.Show(result);
```

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[]`.

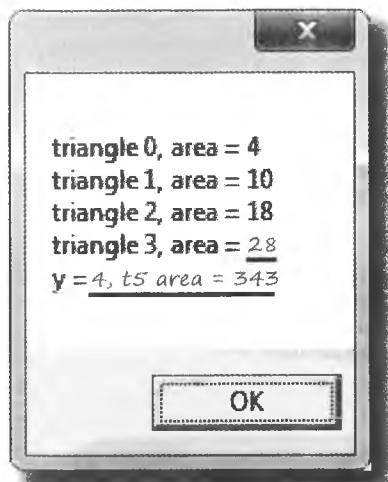


## Решение ребуса в бассейне



После этой строки  
вы получаете массив  
из четырех ссылок  
на класс Triangle, но пока нет  
ни одного объекта  
Triangle!

### 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.

```

    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;
        }
    }

```

Вы заметили, что класс имеет точку входа и при этом создает экземпляр себя? В C# такое вполне допустимо.

Цикл while создает четыре экземпляра Triangle, четыре раза вызывая оператор new.

## 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 записи 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.

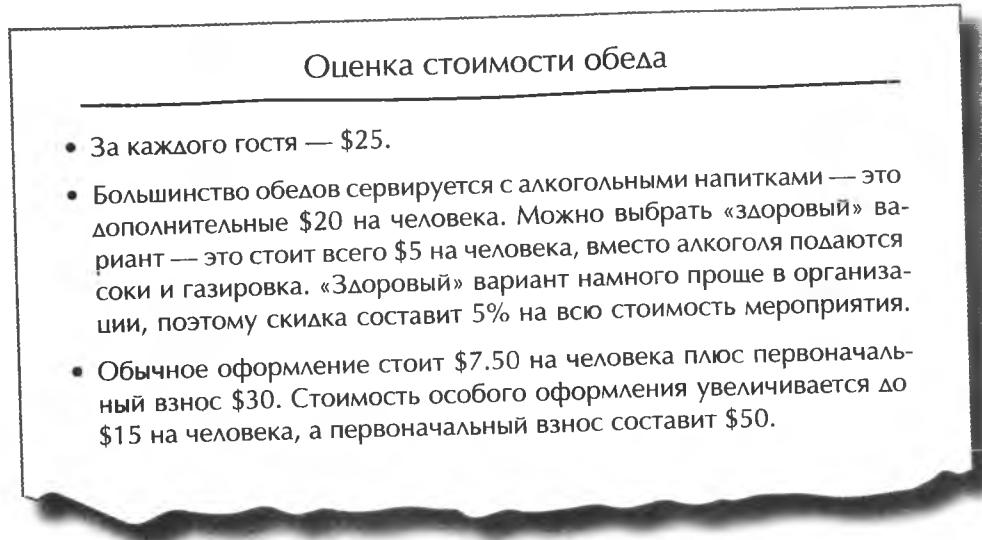


После получения от нового клиента информации о количестве участников, желаемых напитках и оформлении она с помощью специальной диаграммы выполняет довольно сложные вычисления, чтобы определить конечную стоимость. К сожалению, это медленный процесс, и пока Кэтлин считает, клиенты звонят ее конкурентам.

Вы можете написать программу, автоматизирующую расчеты, и спасти бизнес Кэтлин. Только представьте, какой обед она устроит вам в качестве благодарности!

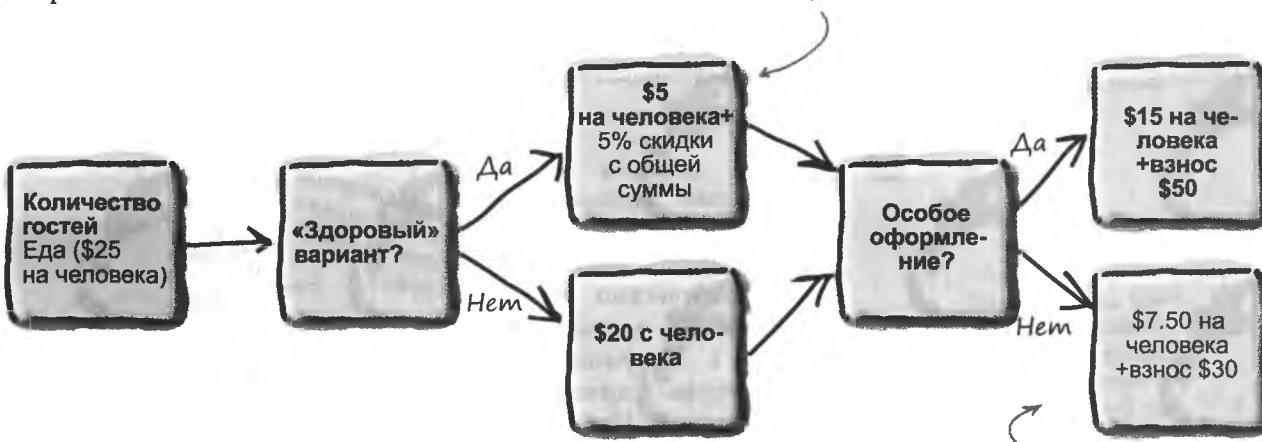
## Как происходит оценка

Вот фрагмент записей Кэтлин, касающийся расчетов стоимости мероприятия:



Для наглядности представим эти правила в виде диаграммы:

В некоторых случаях меняется не только стоимость на одного гостя, но и общая цена мероприятия.



Выбор этого варианта не только влияет на стоимость за одного человека, но и включает единовременный взнос.



Создадим программу, оценивающую стоимость обедов.

1

Создайте проект Windows Application, добавьте класс DinnerParty.cs и оформите его в соответствии с показанной слева диаграммой. Методы затрачивают расчеты стоимости «здорового» варианта, стоимости оформления, а также общей стоимости мероприятия. Два последних поля должны принадлежать типу decimal, а первое – 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

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.

5

Форма должна выглядеть так. Используйте элемент NumericUpDown, чтобы задать максимальное количество людей равным 20, а минимальное – 1. По умолчанию количество приглашенных равно пятью.

Код using System.Windows.Forms; для класса DinnerParty не требуется, так как вы не собираетесь использовать методы из пространства имен .NET Framework.

Свойство Checked флашка Fancy Decorations должно иметь значение True.



Имя этой метки labelCost. 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 Form1(). Он будет вызываться при щелчке на элементе NumericUpDown:

```
private void DisplayDinnerPartyCost()
{
    decimal Cost = dinnerParty.CalculateCost(checkBox2.Checked);
    costLabel.Text = Cost.ToString("c");
```

Этот метод вызывается всеми методами, связанными с формой. Именно так обновляется значение метки Cost.

Присвойте метке, отображающей цену, имя costLabel.

Аргумент "c" метода ToString() отображает валюту с использованием принятого по соглашению символа.

Метод рассчитывает стоимость обеда и передает результат методу Cost.

Значение true появляется при установке флагка 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:

```
private void numericUpDown1_ValueChanged(
    object sender, EventArgs e)
{
    dinnerParty.NumberOfPeople = (int) numericUpDown1.Value;
}
```

При двойном щелчке на кнопке ИСР добавляем обработчик события Click.

numericUpDown1\_ValueChanged(

object sender, EventArgs e)

(int) numericUpDown1.Value;

DisplayDinnerPartyCost();

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 флагке Fancy Decorations и убедитесь, что сначала он вызывает метод CalculateCostOfDecorations(), а потом метод DisplayDinnerPartyCost(). Затем дважды щелкните на флагке 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.

```
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;
        }
    }
}
```

Использование констант гарантирует, что данные параметры останутся неизменными на протяжении всей программы.

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%-я скидка при условии, что обед подается без алкоголя.

Для переменных, содержащих цены, выбирается тип `decimal`. Всегда помещайте букву М после цифры: чтобы присвоить переменной значение \$35.26, нужно написать 35.26M. Это легко запомнить, потому что М – 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 флагкам 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



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 погодить у нас пару недель. Сколько будет стоить 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.

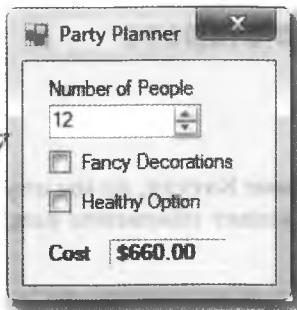


Изменив параметр *Number of People* с 10 на 12, в итоге получаем \$665. Сумма кажется ей слишком маленькой...

**Кэтлин:** Кажется, общая стоимость обеда возрастет с \$575 до \$665.

**Роб:** Всего на \$90? Соблазнительно! А какая цена получится, если убрать особое оформление?

Снятие флашка Fancy Decorations уменьшает сумму всего на \$5. Быть такого не может!

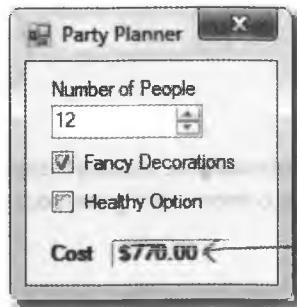


**Кэтлин:** Ээээ... получается сумма в \$660.

**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.

Повторная установка флашка Fancy Decorations увеличивает конечную сумму до \$770. Так быть не должно!



**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 центом больше!

## МОЗГОВОЙ ШТУРМ

Как вы думаете, почему каждое изменение условий давало ошибку в результате?

неожиданно оказалось, что...

## Каждый Вариант нужно было считать отдельно

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:

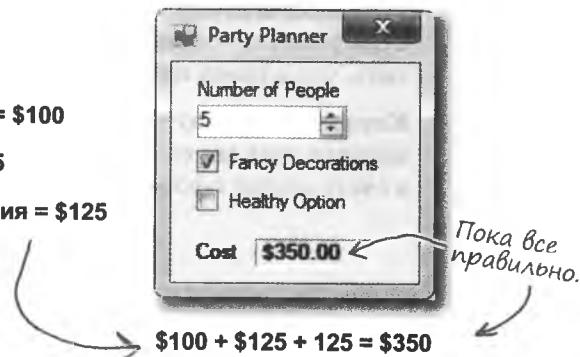
5 человек

\$20 с человека за напитки → Стоимость напитков = \$100  
\$25 с человека за еду → Стоимость еды = \$125  
\$15 с человека за оформление и взнос \$50 → Стоимость оформления = \$125



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.



При изменении количества гостей приложение должно пересчитывать сумму аналогичным образом. Но этого не происходит:

10 человек

\$20 с человека за напитки → Стоимость напитков = \$200  
\$25 с человека за еду → Стоимость еды = \$250  
\$15 с человека за оформление и взнос \$50 → Стоимость оформления = \$200

$$\$200 + \$250 + 200 = \$650$$

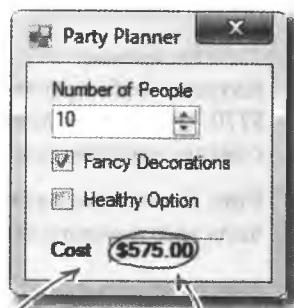
Сумма, которую мы должны получить.

Программа складывает старую цену оформления с новыми ценами еды и напитков.

$$\$200 + \$250 + \$125 = \$575.$$

Новая цена еды и напитков.

Старая цена оформления.



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();  
}
```

Данный метод вызывает метод CalculateCost(), но забывает вызывать метод CalculateCostofDecorations().

Эта строка задает значение параметра NumberofPeople для экземпляра DinnerParty на основе введенных в форму данных.

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 флагка Fancy Decorations снова запускает метод CalculateCostOfDecorations(), что приводит к коррекции данных.



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 этими «пользователями» являются вы сами!

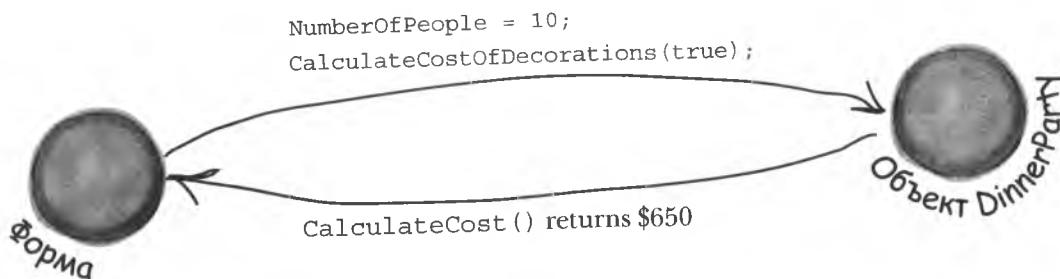
## Неправильное использование объектов

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 вызывать класс `DinnerParty`

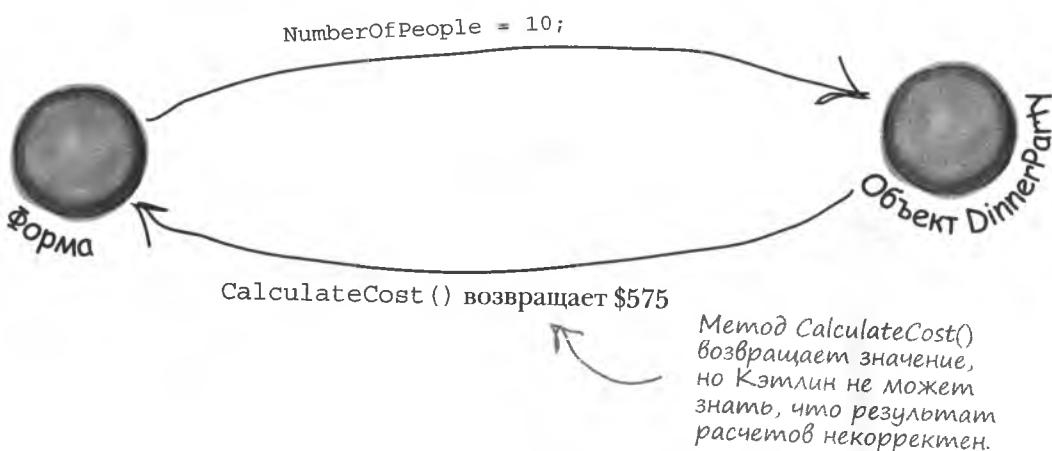
Класс `DinnerParty` предоставляет форме прекрасный метод расчета конечной стоимости оформления. Достаточно указать количество гостей и вызвать метод `CalculateCostOfDecorations()`, после чего метод `CalculateCost()` покажет правильную сумму.



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;
```

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».

```
public void SetPartyOptions(int people, bool fancy) {  
    numberOfPeople = people;  
    CalculateCostOfDecorations(fancy);  
}  
  
public int GetNumberOfPeople() {  
    return numberOfPeople;  
}
```

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 недочета 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. Подвод-  
ники полностью инкапсулиро-  
ваны 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 не-предсказуемых 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).



Реальное имя (RealName) : "Херб Джонс"  
Псевдоним (Alias) : "Даш Мартин"  
Пароль (Password) : "ворона летает в полночь"

<b>SecretAgent</b>
Alias
RealName
Password
<code>AgentGreeting()</code>

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.

3

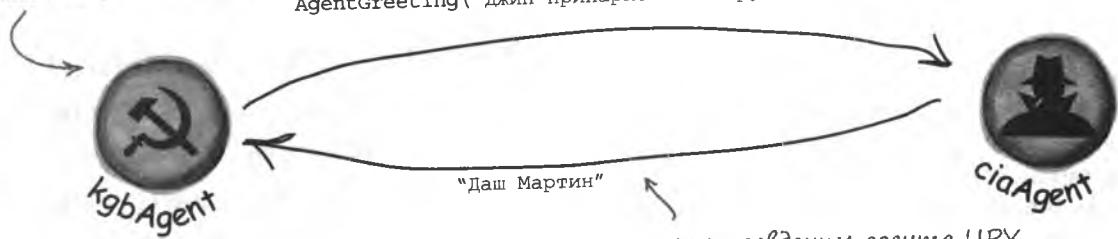
Кажется, что это вполне надежная защита, не так ли? Если объект, вызывающий метод, не «знает» правильного пароля, он не «узнает» настоящее имя агента Джонса.

<b>EnemyAgent</b>
Borsch
Vodka
<code>ContactComrades()</code>
<code>OverthrowCapitalists()</code>

Объект `ciaAgent` является экземпляром класса `SecretAgent`, в то время как `kgbAgent` – это экземпляр класса `EnemyAgent`.

`AgentGreeting("джип припаркован снаружи")`

Агент КГБ использует в качестве приветствия неверный пароль.



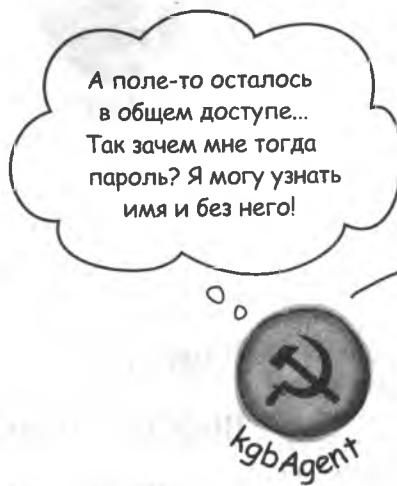
Поэтому он узнает только псевдоним агента ЦРУ.

## 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`:

Модификатор  
`public` означает,  
что переменная  
доступна для  
редактирования  
извне класса.

→ `public string RealName;`



Ограничений доступа к общим переменным не существует.

`string name = ciaAgent.RealName;`



Нам не потребуются методы.  
Ведь поле `realName` открыто  
для просмотра!

Для сохранения тайны агенту Джонсу нужно использовать поля **закрытого доступа**. Стоит объявить поле `realName` с модификатором `private`, единственным способом узнать информацию станет вызов метода имеющих доступ к закрытым элементам класса. И агент КГБ останется с носом!

Объект `kgbAgent` не может «видеть» закрытые поля объекта `ciaAgent`, так как принадлежат **другому** классу.

Заменив `public` на `private`, вы скроете поле от внешнего мира.

→ `private string realName;`

Убедитесь, что закрытым является поле, в котором хранится пароль.

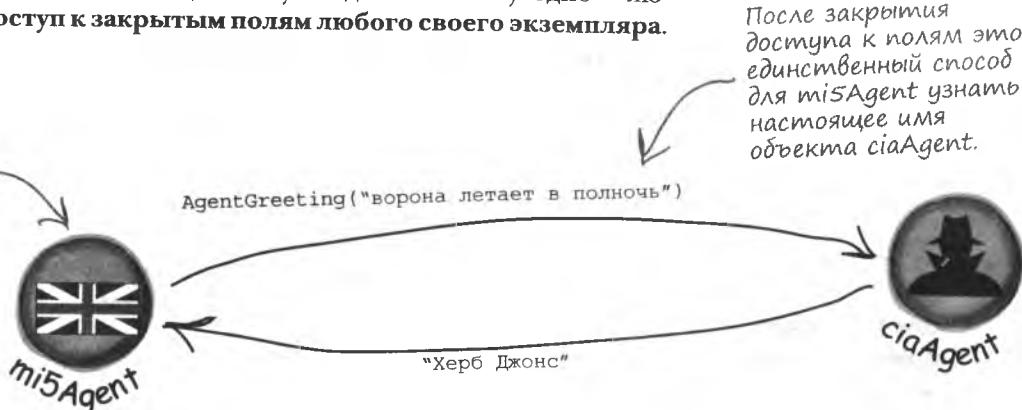
Модификатор `private` гарантирует, что внешний код не сможет отредактировать значения полей без вашего ведома.

## 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.

Объект `mi5Agent` принадлежит классу `BritishAgent`, поэтому он тоже не имеет доступа к закрытым полям объекта `ciaAgent`.

Их может видеть только другой `ciaAgent`.



### 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 постоянными формами или объектами. Но нет правила, это предписывает. Вы можете щелкнуть два раза на кнопке и указать в объявлении обработчика событий модификатор `public`. И код все равно будет компилироваться и запускаться.

**Единственным способом получения информации из закрытых полей являются методы общего доступа, которые возвращают данные.**

## 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 = 94;`
  4. `mySuperChef.secretIngredient = "кардамон";`
  5. `mySuperChef.cookieRecipe = "3 яйца, 2 1/2 чашки муки, 1 ст. л. соли, 1 ст. л. ванили и 1.5 чашки сахара смешать. Выпекать 10 минут при температуре 190 °С. 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;`

Строки 2 и 4 не будут компилироваться из-за закрытых полей `ingredientSupplier` и `secretIngredient`.

3. `int loyalCustomerOrderAmount = 54;`

4. `mySuperChef.secretIngredient = "кардамон";`

Созданная вами локальная переменная `loyalCustomerAmount`, которой было присвоено значение `54`, не меняет значение переменной объекта `loyalCustomerAmount`, поэтому секретный компонент не выводится.

5. `mySuperChef.cookieRecipe = "3 яйца, 2 1/2 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?

3 яйца, 2 1/2 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!



Ничего не понимаю. Закрытое поле не позволяет другому классу использовать себя. Но если поменять `private` на `public` программа все равно будет построена! А вот добавление модификатора `private` делает компиляцию невозможной. Зачем мне тогда вообще его использовать?

**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 некоторые 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 применяет 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

Вспомните программу Кэтлин? Проблема возникла 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.**



## МОЗГОВОЙ ШТУРМ

Как плохо инкапсулированные классы могут помешать редактированию программы в будущем?

## Программе Майка не помешала бы инкапсуляция

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.

Геокэшингом называется игра, в которой игроки ищут тайники, с помощью GPS определяют их координаты и сообщают в Интернете.

Не могу вспомнить, мне требовалось поле `StartPoint` или метод `SetStartPoint()`. Но раньше все работало!



### Возьми в руку карандаш



Route
StartPoint
EndPoint
Length
<code>GetRouteLength()</code>
<code>GetStartPoint()</code>
<code>GetEndPoint()</code>
<code>SetStartPoint()</code>
<code>SetEndPoint()</code>
<code>ChangeStartPoint()</code>
<code>ChangeEndPoint()</code>

Вот объект `Route` из программы Майка. Какие свойства и методы вы бы пометили как `private`, чтобы облегчить их использование?

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!

**Правильно  
инкапсулировав  
классы сегодня,  
вы облегчите себе  
работу с ними  
завтра.**



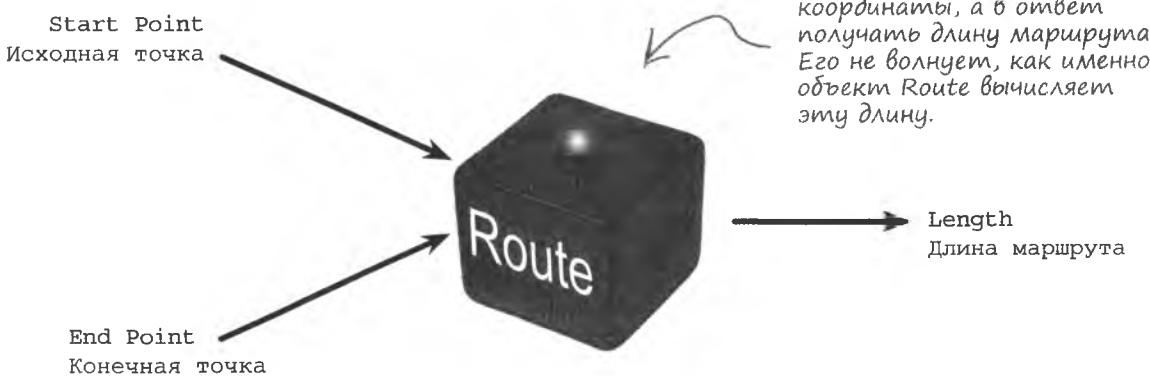
Мой объект Route  
работает! Но вот как его  
сейчас приспособить  
его для геокэшинга?

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!

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.

### Пример инкапсуляции

Класс Farmer (Фермер) использует поле для хранения информации о количестве 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 поменять 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:

```
Одной ко-  
рове тре-  
буется  
30 мешков  
коров.  
public const int FeedMultiplier = 30;  
public int GetNumberOfCows()  
{  
    return numberOfCows;  
}  
  
public void SetNumberOfCows(int newNumberOfCows)  
{  
    numberOfCows = newNumberOfCows;  
    BagsOfFeed = numberOfCows * FeedMultiplier;  
}
```

Этот метод сообщает другим классам количество коров.

Они выполняют одну и ту же функцию!

Имена полей ограниченного доступа начинаются со строчной буквы, в то время как у полей общего доступа имена начинаются с прописной буквы.

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 `value`, содержащий значение, записываемое в поле.

```
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;
}
```

Поле `NumberOfCows` запускает метод записи, передавая значение 20. Запрос к полю `BagsOfFeed` запускает метод чтения, возвращающий значение 300.

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.

## 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()`.

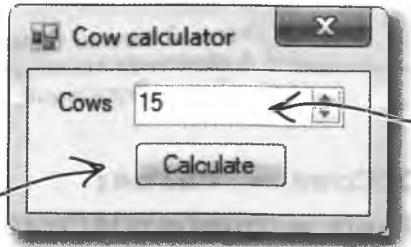


- 1 Добавьте к проекту класс **Farmer**:

```
class Farmer {  
    public int BagsOfFeed;  
    public const int FeedMultiplier = 30;  
  
    private int number_of_cows;  
    public int Number_of_Cows {  
        (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() { Number_of_Cows = 15 };  
    }  
    private void numericUpDown1_ValueChanged(object sender, EventArgs e) {  
        farmer.Number_of_Cows = (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.Number_of_Cows);  
    }  
}
```



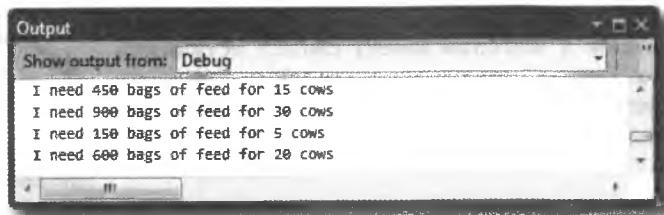
Метод `Console.WriteLine()` отправляет строчку с текстом в окно **Output**.

Требуется **{0}** мешков  
корова для **{1}** коров.

Метод `WriteLine()` замещает `{0}` –  
значением первого параметра, а `{1}` –  
значением второго параметра.

## 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:



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 для любого количества коров! После редактирования параметра NumericUpDown кнопка Calculate снова начнет работать корректно.

### Полностью инкапсулируем класс Farmer

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. Вот как это сделать:

Вы понимаете,  
почему это  
стало причиной  
ошибки?

Напечатав `prop`  
и дважды нажав  
`tab`, вы добавите  
к коду автома-  
тическое свой-  
ство.

- Удалите поле BagsOfFeed из класса Farmer. Вместо него введите `prop` и дважды нажмите `tab`. Появится следующая строка кода:

```
public int MyProperty { get; set; }
```

- 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.

- 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; } }
```

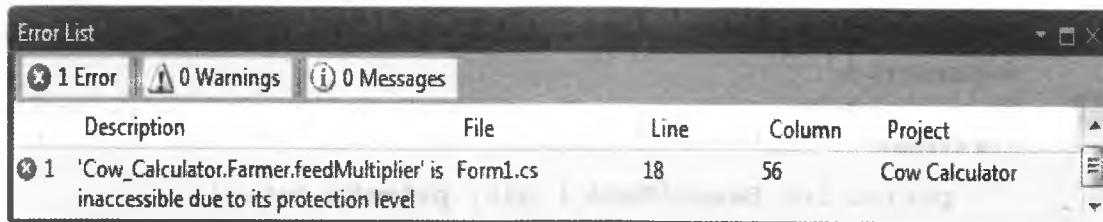
Так как вместо константы общего доступа у нас закрытое поле типа `int`, его имя теперь начинается со строчной буквы `f`.

### 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:



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.**

## 1 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.

Ключевое слово  
this в кон-  
струкции this.  
feedMultiplier  
указывает,  
что вы имеете  
в виду поле,  
а не одноимен-  
ный параметр.

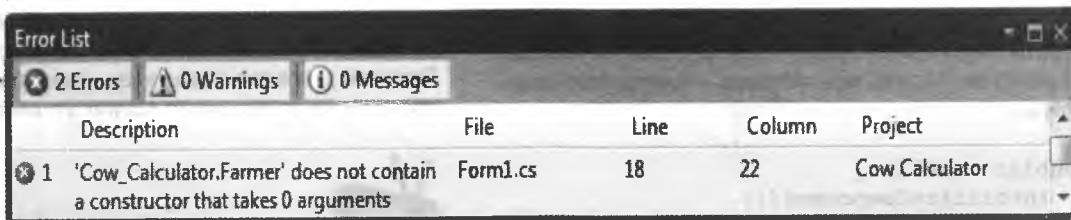
**public Farmer(int numberOfCows, int feedMultiplier) {**

**this.feedMultiplier = feedMultiplier;**  
**numberOfCows = numberOfCows;**

Запись в закрытое поле numberOfCows исключает дальнейший вызов метода записи numberOfCows. Данная же строка гарантирует его вызов.

Перед вызовом  
метода запи-  
си NumberOfCows  
нужно за-  
дать параметр  
feedMultiplier.

Это сообже-  
ние об ошибке  
означает, что  
оператор new  
не имеет па-  
раметров.



## 2 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!

**public Form1() {**  
    **InitializeComponent();**  
    **farmer = new Farmer(15, 30);**

Так как форма — это тоже  
объект, для нее определен  
конструктор с именем Form1.  
Обратите внимание, что он не  
возвращает значение.

Метод new, вызывающий конструктор,  
отличается наличием передаваемых кон-  
структурой параметров.



## Конструкторы под Микроскопом

Конструкторы не возвращают значения.

Внимательно рассмотрим конструктор Farmer, чтобы составить представление о том, как он работает.

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().



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 полностью 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 имеет смысла 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:** С# 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. С# 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 С# 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 другого класса прочитать свойство или сделять запись в него.**

## 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());`
2. `CableBill february = new CableBill(7);  
february.payPerViewDiscount = 1;  
MessageBox.Show(february.CalculateAmount(3).ToString());`
3. `CableBill march = new CableBill(9);  
march.Discount = true;  
MessageBox.Show(march.CalculateAmount(6).ToString());`

Значение  
переменной  
`amountOwed`?

Значение  
переменной  
`amountOwed`?

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.

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.

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.

### — 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";
        }
    }
}
```

## 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());
2. CableBill february = new CableBill(7);  
february.payPerViewDiscount = 1;  
MessageBox.Show(february.CalculateAmount(3).ToString());
3. CableBill march = new CableBill(9);  
march.Discount = true;  
MessageBox.Show(march.CalculateAmount(6).ToString());

Значение  
переменной  
amountOwed?

28

Значение  
переменной  
amountOwed?

не компилируется

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, который еще даже не задан! Строчка станет корректной, если заменить имя параметра конструктора на Price (с прописной буквы).

Ключевое слово this у не-верной 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";
    }
}
```

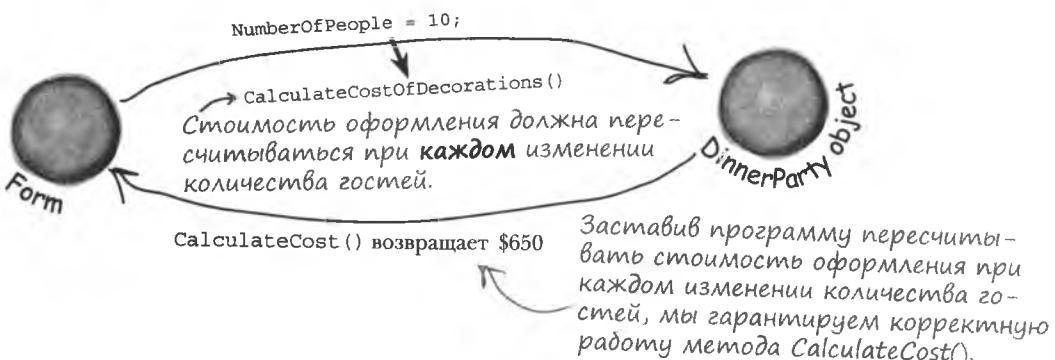
Ключевое слово this не имеет отношения к данному параметру. Оно должно присутствовать рядом с параметром price.

**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`.



### 2 Добавление свойств и конструктора

Нужно инкапсулировать класс `DinnerParty`. Для начала **сделаем параметр `NumberOfPeople` свойством**, вызывающим метод `CalculateCostOfDecorations()`. Затем нужно **добавить конструктор**, гарантировав корректную инициализацию экземпляра. И наконец останется **заставить форму использовать этот конструктор**.

- ★ Создайте свойство для переменной `NumberOfPeople`, снабженное методом записи,зывающим метод `CalculateCostOfDecorations()`. Вам потребуется вспомогательное поле `numberOfPeople`.
- ★ Методу записи `NumberOfPeople` потребуется переменная, которую он будет передавать в качестве параметра методу `CalculateCostOfDecorations()`. Добавьте закрытое поле типа `bool` с именем `fancyDecorations`, запись в которое осуществляется при каждом вызове метода `CalculateCostOfDecorations()`.
- ★ Чтобы добавить конструктор, вам нужны три параметра: количество гостей, `Healthy Option` и `Fancy Decorations`. В момент инициализации объекта `DinnerParty` форма вызывает два метода, поместите их в конструктор:

```
dinnerParty.CalculateCostOfDecorations(fancyBox.Checked);
dinnerParty.SetHealthyOption(healthyBox.Checked);
```

- ★ Вот как выглядит конструктор для формы. Все остальное остается без изменений:

```
public Form1() {
    InitializeComponent();
    dinnerParty = new DinnerParty((int)numericUpDown1.Value,
        healthyBox.Checked, fancyBox.Checked);
    DisplayDinnerPartyCost();
}
```



## упражнение Решение

Итак, вот как нужно было написать программу для расчета стоимости мероприятий:

```

class DinnerParty {
    const int CostOfFoodPerPerson = 25;

    private int numberOfPeople;
    public int NumberOfPeople {
        get { return numberOfPeople; }
        set {
            numberOfPeople = value;
            CalculateCostOfDecorations(fancyDecorations);
        }
    }
    private bool fancyDecorations;

    public decimal CostOfBeveragesPerPerson;
    public decimal CostOfDecorations = 0;

    public DinnerParty(int numberOfPeople, bool healthyOption, bool fancyDecorations) {
        NumberOfPeople = numberOfPeople;
        this.fancyDecorations = fancyDecorations;
        SetHealthyOption(healthyOption);
        CalculateCostOfDecorations(fancyDecorations);
    }

    public void SetHealthyOption(bool healthyOption) {
        if (healthyOption) {
            CostOfBeveragesPerPerson = 5.00M;
        } else {
            CostOfBeveragesPerPerson = 20.00M;
        }
    }

    public void CalculateCostOfDecorations(bool fancy) {
        fancyDecorations = 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;
        }
    }
}

```

После закрытия поля `numberOfPeople` форма не может вносить в него изменения до пересчета стоимости оформления. Ошибка, которая чуть не стоила Кэйтлин ее лучшего клиента, исправлена!

При помощи свойства вы гарантируете, что стоимость оформления пересчитывается при каждом изменении количества гостей.

Будьте внимательны с ключевым словом `this`. Именно оно поможет различить параметр и закрытое поле `numberOfPeople`.

Перед `fancyDecorations` нужно поместить ключевое слово `this`, чтобы отличить название закрытого поля от названия параметра.

Информация об особым оформлении должна храниться в поле, чтобы его смог использовать метод записи `NumberOfPeople`.

## 6 наследование

# Генеалогическое древо объектов

Входя в крутой поворот, я вдруг понял, что унаследовал свой велосипед от ДвухКолесного, но забыл добавить метод Тормоза()...  
В итоге двадцать шесть швов и лишение прогулок на целый месяц.



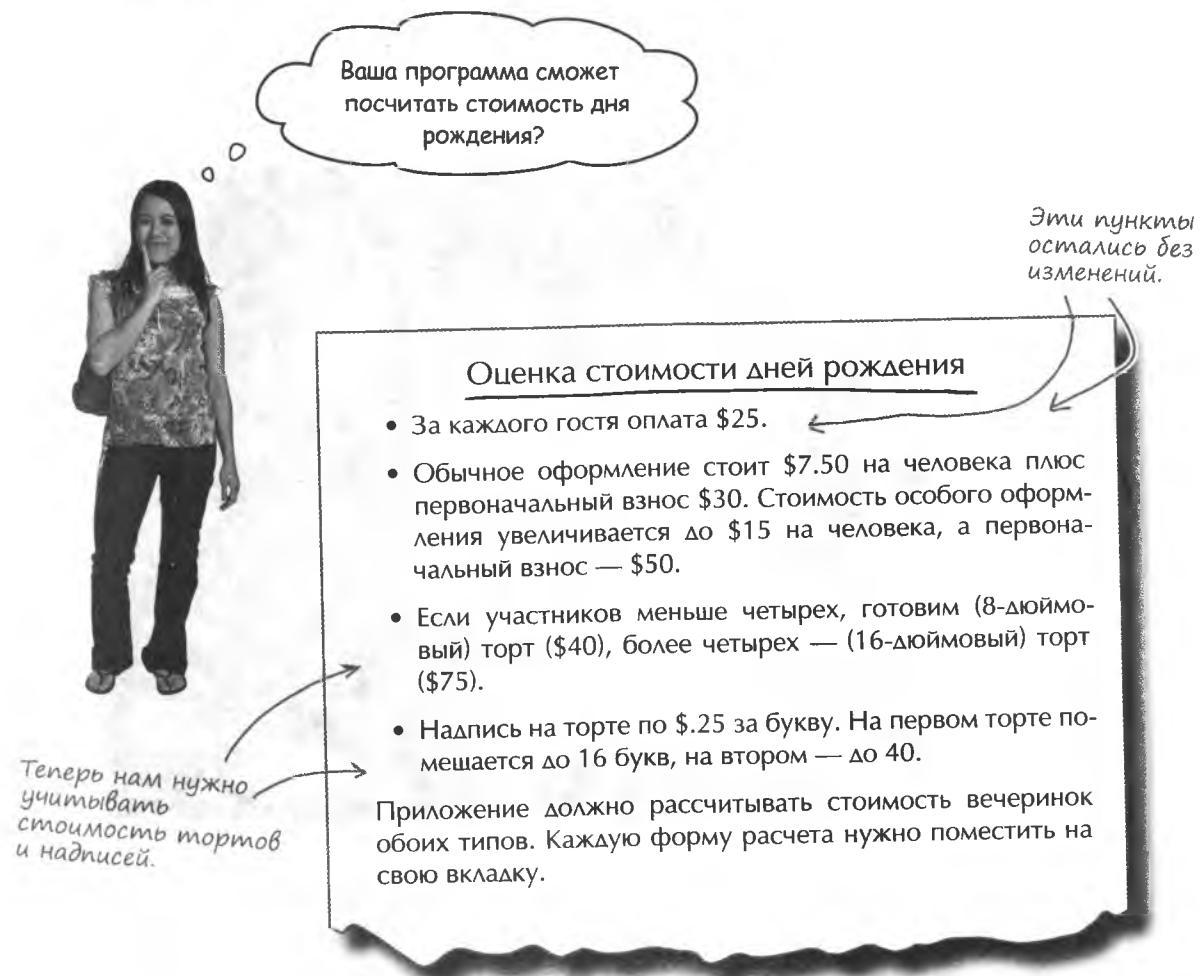
Иногда люди хотят быть похожим на своих родителей.

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.

с днем рождения, крошка

## 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. Этую схему хотелось бы добавить в программу.



## Нам нужен класс BirthdayParty

Чтобы программа получила возможность рассчитывать стоимость вечеринок другого типа, нам нужно поменять форму.

**Вот как мы это будем делать:**

Вы займетесь этим через минуту, а пока представим общую картину.

BirthdayParty
NumberOfPeople
CostOfDecorations
CakeSize
CakeWriting
CalculateCostOfDecorations()
CalculateCost()

### 1 Создание класса BirthdayParty

Новый класс будет подсчитывать стоимость, исходя из выбранного варианта оформления, а также количества букв на торте.

### 4 Добавление к форме элемента TabControl

Работать с вкладками просто. Выберите нужную и перетащите на нее элементы управления.

### 5 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.

## 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

Начните новый проект, мы сделаем для Кэтлин версию программы, которая сможет рассчитать стоимость дней рождения и званных гостей. Начнем с инкапсулированного класса `BirthdayParty`, который будет выполнять все расчеты.



### 1 Новый класс `BirthdayParty`

Вы уже знаете, что нужно делать со свойством `NumberOfPeople` и методом `CostOfDecorations`, вы с ними встречались в классе `DinnerParty`. Добавим в новый класс сначала их.

- ★ Добавьте поле общего доступа типа `int` с именем `CakeSize` (Размер торта). Потом вы добавите закрытый метод `CalculateCakeSize()` (Вычислить размер торта), присваивающий полю `CakeSize` значение 8 или 16 в зависимости от количества гостей. Но начнем мы с конструктора и метода записи `NumberOfPeople`.

`using System.Windows.Forms;` ← Убедитесь в наличии оператора `using` в верхней части класса, так как вам потребуется метод `MessageBox.Show()`.

```
class BirthdayParty {
    public const int CostOfFoodPerPerson = 25;

    public decimal CostOfDecorations = 0;
    private bool fancyDecorations;
    public int CakeSize;

    public BirthdayParty(int numberofPeople,
                         bool fancyDecorations, string cakeWriting)
    {
        this.numberofPeople = numberofPeople;
        this.fancyDecorations = fancyDecorations;
        CalculateCakeSize();
        this.CakeWriting = cakeWriting;
        CalculateCostOfDecorations(fancyDecorations);
    }
}
```

Для инициализации объекта `BirthdayParty` необходимы данные о количестве гостей, виде оформления и надписи на торте. В этом случае при вызове метода `CalculateCost()` стоимость будет правильно рассчитана.

Для подсчета стоимости надписи на торте конструктор вызывает метод записи. Чтобы понять, не является ли этот параметр слишком большим, сначала нужно узнать размер торта.

Конструктор задает свойства, а затем начинает вычисления.

Поля и свойства, содержащие информацию о денежных суммах, должны принадлежать типу `decimal`.

<b>BirthdayParty</b>
NumberOfPeople
CostOfDecorations
CakeSize
CakeWriting
CalculateCostOfDecorations()
CalculateCost()

- ★ Для хранения информации о надписи на торте потребуется свойство CakeWriting типа string. Этот метод записи проверяет параметр CakeSize, так как максимальное количество букв зависит от размера торта. Затем при помощи метода value.Length проверяется длина строки. Если строка оказывается слишком длинной, метод записи вызывает окно с текстом, «Слишком много букв для 16-дюймового торта» (или 8-дюймового).

- ★ Кроме того, вам потребуется метод CalculateCakeSize():

```
private void CalculateCakeSize() {
    if (NumberOfPeople <= 4)
        CakeSize = 8;
    else
        CakeSize = 16;
}
```

Метод CalculateCakeSize()  
задает поле CakeSize. Он  
вызывается методом записи  
NumberOfPeople и методом  
CalculateCost().

```
private string cakeWriting = "";
public string CakeWriting {
```

```
    get { return this.cakeWriting; }
    set {
```

```
        int maxLength;
```

```
        if (CakeSize == 8)
            maxLength = 16;
```

```
        else
```

```
            maxLength = 40;
```

```
        if (value.Length > maxLength) {
```

```
            MessageBox.Show("Слишком много букв для" + CakeSize + " дюймового торта");
```

```
            if (maxLength > this.cakeWriting.Length)
```

```
                maxLength = this.cakeWriting.Length;
```

```
                this.cakeWriting = cakeWriting.Substring(0, maxLength);
```

```
}
```

```
    else
```

```
        this.cakeWriting = value;
```

```
}
```

Здесь свойство CakeWriting определяет,  
поместится ли строка на торте.  
Его метод записи проверяет размер  
торта. Затем используется свойство  
Length вспомогательного поля, чтобы  
проверить длину. Слишком длинные  
строки обрезаются.

Метод Substring() обрезает строку до ука-  
занной длины. Вам понадобится перезагру-  
зить надпись в текстовом поле при измене-  
нии текста или размера торта.

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 обя-  
зательно 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 метод CalculateCost(). Но если в классе DinnerParty вы брали стоимость оформления и прибавляли к нему стоимость напитков, теперь нужно будет добавить еще и стоимость торта.

Тип decimal выбран, так как мы работаем с деньгами.

```
public decimal CalculateCost() {
    decimal TotalCost = CostOfDecorations + (CostOfFoodPerPerson * NumberOfPeople);
    decimal CakeCost;
    if (CakeSize == 8)
        CakeCost = 40M + CakeWriting.Length * .25M;
    else
        CakeCost = 75M + CakeWriting.Length * .25M;
    return TotalCost + CakeCost;
}
```

В данном случае метод CalculateCost() вместо стоимости напитков прибавляет стоимость торта.

```
private int numberOfPeople;
public int NumberOfPeople {
    get { return numberOfPeople; }
    set {
        numberOfPeople = value;
        CalculateCostOfDecorations(fancyDecorations);
        CalculateCakeSize();
        this.CakeWriting = cakeWriting;
    }
}
Подобный метод вы уже
использовали в классе
DinnerParty.
```

```
public void CalculateCostOfDecorations(bool fancy) {
    fancyDecorations = fancy;
    if (fancy)
        CostOfDecorations = (NumberOfPeople * 15.00M) + 50M;
    else
        CostOfDecorations = (NumberOfPeople * 7.50M) + 30M;
}
```

Метод CakeWriting должен не только обрезать строку, но и запускать метод записи при изменении количества гостей.

При изменении количества гостей класс сначала пересчитывает размер торта и, используя метод записи, заставляет метод CakeWriting обрезать текст.

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.

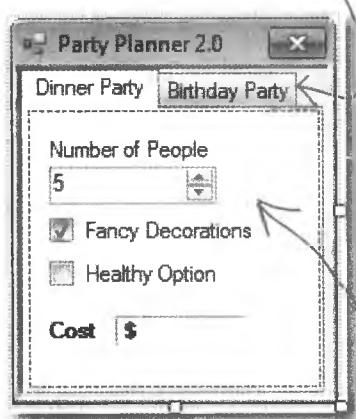
Дайте вкладкам названия при помощи свойства **TabCollection**.

3

## Вставка элементов управления на вкладку Dinner Party

Откройте программу **Party Planner** (из главы 5) в отдельном окне ИСР. 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**).

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.



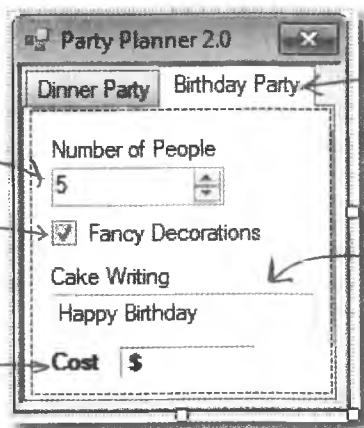
Эти элементы управления доступны только на вкладке Dinner Party.

4

## Пользовательский интерфейс для вкладки Birthday Party

На вкладке **Birthday Party** должны 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.

Присвойте элементам управления **NumericUpDown**, **CheckBox** и **Label** имена **numberBirthday**, **fancyBirthday** и **birthdayCost** соответственно.



Перейдите на вкладку **Birthday Party** и добавьте новые элементы управления.

Добавьте элемент управления **TextBox** с именем **cakeWriting** для ввода надписи на торте. Свойству **Text** присвойте значение **Happy Birthday**. Эта надпись будет выводиться по умолчанию.

## 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 с 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 при-  
сваивает экземпляру `BirthdayParty` на-  
чальные значения подобно тому, как это  
было сделано для экземпляра `DinnerParty`.

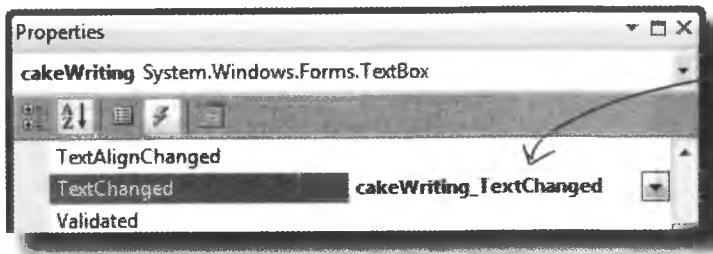
- ★ Добавьте код к обработчику событий элемента `NumericUpDown`, чтобы задать свойство `NumberOfPeople` и заставить функционировать флагок `Fancy Decorations`.

```
private void numberBirthday_ValueChanged(object sender, EventArgs e) {  
    birthdayParty.NumberOfPeople = (int)numberBirthday.Value;  
    DisplayBirthdayPartyCost();  
}  
  
private void fancyBirthday_CheckedChanged(object sender, EventArgs e) {  
    birthdayParty.CalculateCostOfDecorations(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.



Когда вы выделите поле cakeWriting и дважды щелкните на строчке TextChanged в списке Events окна Properties, ИСР 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.

```
private void DisplayBirthdayPartyCost() {
    cakeWriting.Text = birthdayParty.CakeWriting;
    decimal cost = birthdayParty.CalculateCost();
    birthdayCost.Text = cost.ToString("c");
}
```

Способ, которым форма обрабатывает надпись на торте, очень прост, потому что класс BirthdayParty **инкапсулирован**. Форма должна всего лишь задавать свойства объекта. Все остальное сделает самостоятельное.

Способ обработки надписи, количества гостей и размера торта встроен в методы записи NumberOfPeople и CakeWriting, поэтому форма должна всего лишь передать им нужные значения.

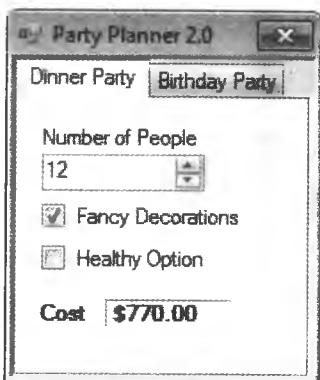


**...ВОТ ВЫ И ЗАКОНЧИЛИ СОЗДАНИЕ ФОРМЫ!**

6

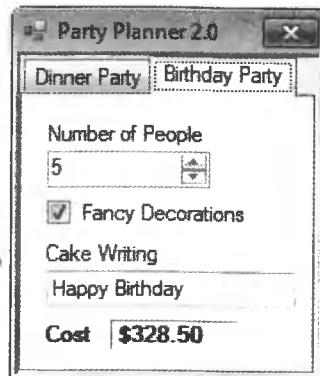
### Программа готова!

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!



Запустите программу и откройте вкладку Dinner Party. Убедитесь, что она работает точно так же, как и старая программа Party Planner.

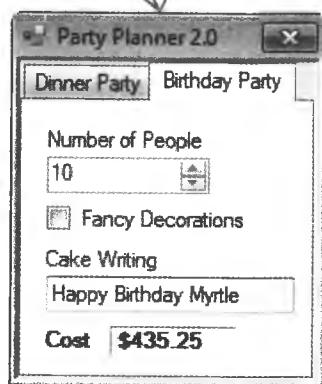
Перейдите на вкладку Birthday Party. Убедитесь, что значение поля Cost меняется при изменении количества гостей и установке флашка Fancy Decorations.



Проверим правильность расчетов для 10 гостей.  
 $\$25 \times 10 = \$250$ ,  
 16-дюймовый торт стоит  
 $\$75$ , обычное оформление:  
 $\$7.50 \times 10 = \$75$ ,  
 единовременный взнос:  $\$30$ ,  
 21 буква на торте по цене  
 $\$.25$  за букву =  $\$5.25$ .

Итого  $\$250 + \$75 + \$75 + \$30 + \$5.25 = \$435.25$ .  
 Все правильно!

При вводе информации в поле Cake Writing обработчик событий TextChanged должен обновлять значение поля Cost.



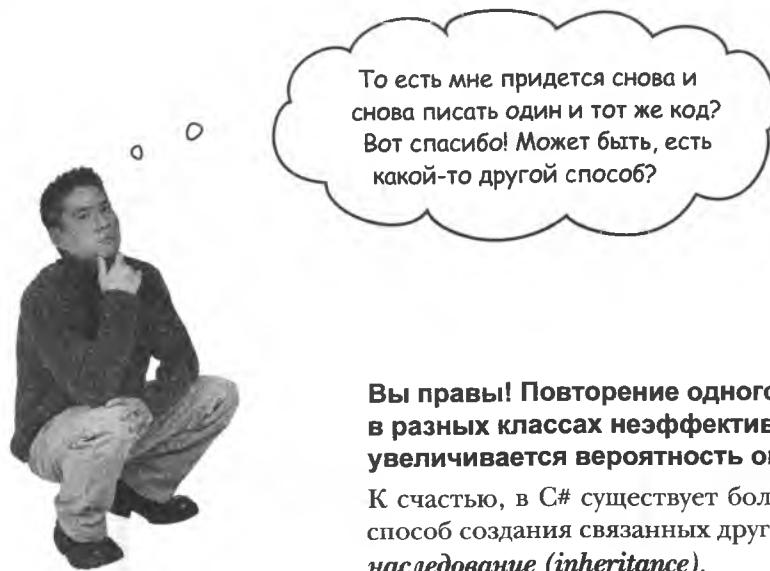
## Дополнительный взнос за мероприятия с большим количеством гостей

Благодаря программе дела Кэтлин пошли в гору, и теперь она может себе позволить брать дополнительную плату за мероприятия с очень большим количеством гостей (более 12 человек). Как же добавить к программе еще один платеж?

- ★ Метод `DinnerParty.CalculateCost()` должен проверять значение переменной `NumberOfPeople`, и если возвращаемое значение превышает 12, добавлять \$100.
- ★ Аналогично для метода `BirthdayParty.CalculateCost()`.

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 существовании 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*?

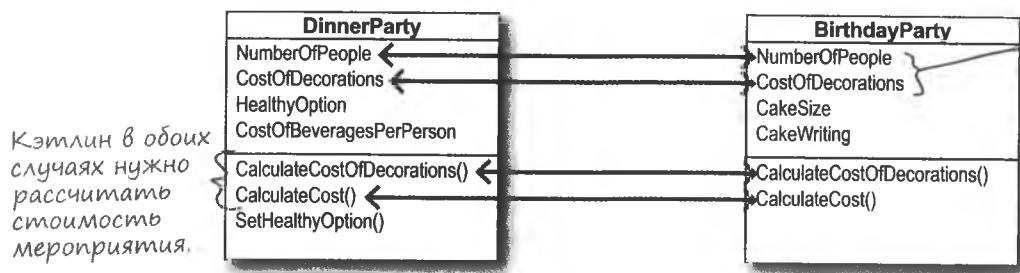


**Вы правы! Повторение одного и того же кода в разных классах неэффективно. И к тому же увеличивается вероятность ошибок.**

К счастью, в C# существует более продуктивный способ создания связанных друг с другом классов: **наследование (inheritance)**.

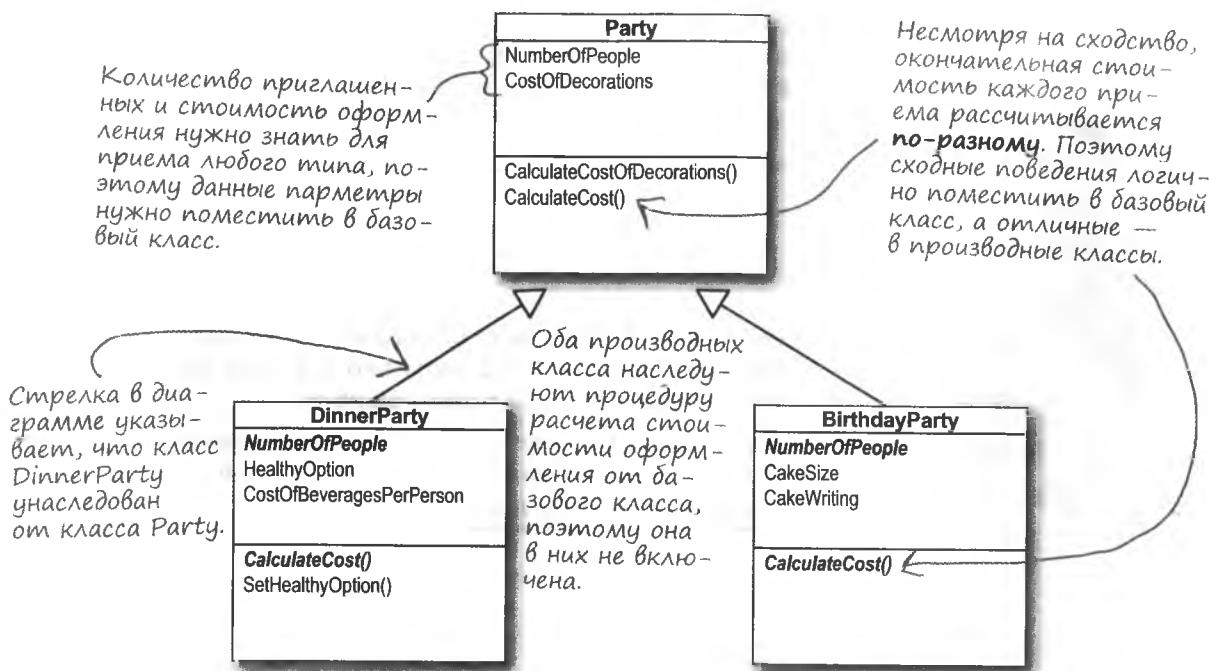
## Наследование

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**.



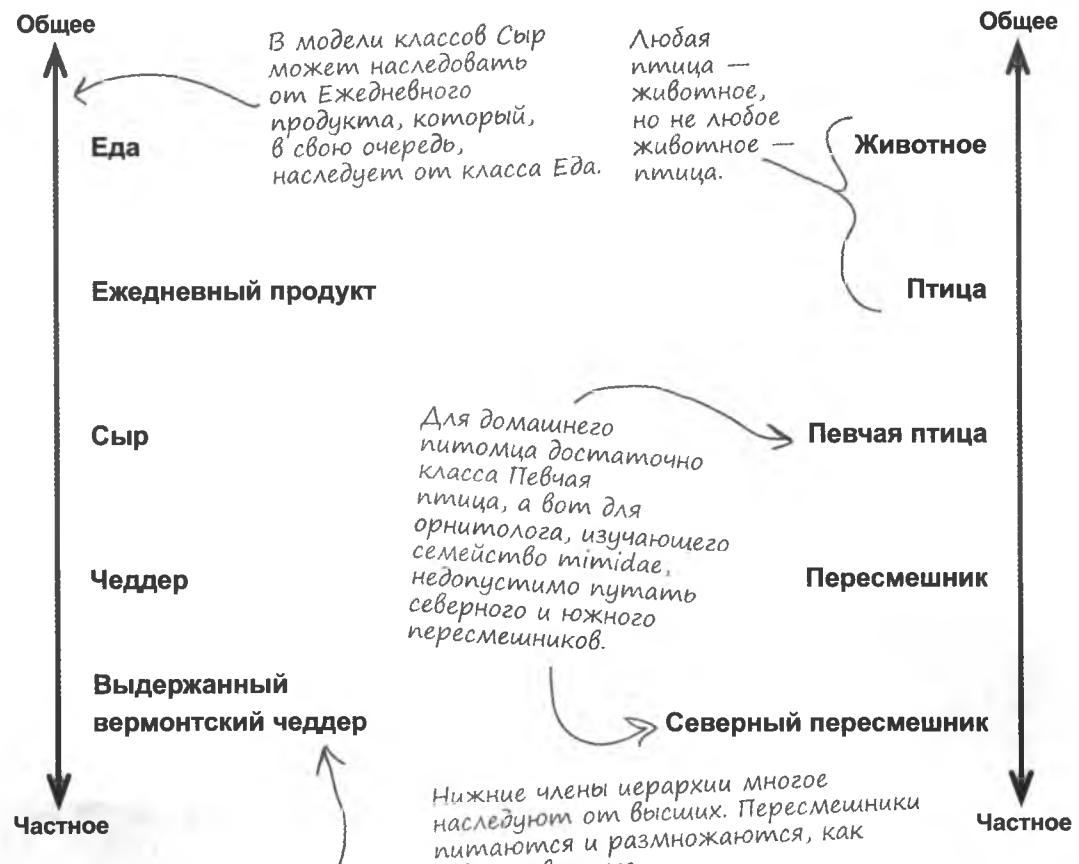
### Званые обеды и дни рождения относятся к приемам

Если существуют 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.



**Нас-ле-до-вать, глаг.**  
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.

### 1

#### Что у животных общего

Посмотрите на шесть животных. Как связаны лев, бегемот, тигр, кошка, волк и далматин? Именно эти общие для всех свойства лягут в основу базового класса.



# Наследование позволяет избежать дублирования кода в производных классах

2

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:

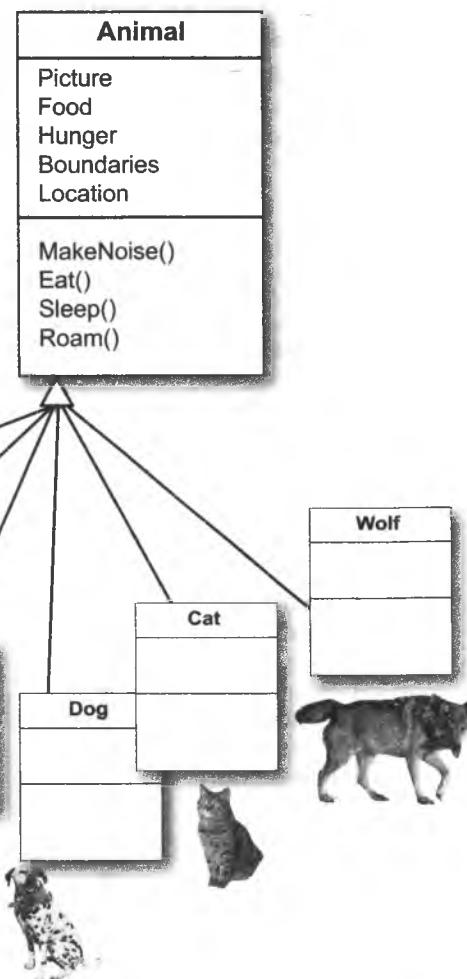
- ★ Picture: картинка, которую можно поместить в PictureBox.
- ★ Food: тип пищи. Пока у этого поля только два значения: meat (мясо) и grass (трава).
- ★ Hunger: переменная типа int, показывающая, насколько животное хочет есть. Она меняется в зависимости от количества выданного корма.
- ★ Boundaries: ссылка на класс, в котором хранится информация о высоте, ширине 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 предпочтаемого корма.
- ★ Sleep(): метод, заставляющий животное спать.
- ★ Roam(): метод, учитывающий перемещения по вольеру.

## 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)**.

### 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 заставит 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 **переопределить**.



Свойства и методы из базового класса `Animal` не обязательно использовать в производных классах в неизменном виде. Вы можете вообще их не использовать!

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.



Animal
Picture
Food
Hunger
Boundaries
Location
MakeNoise()
Eat()
Sleep()
Roam()

## МОЗГОВОЙ ШТУРМ

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

«Выдержаный 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.

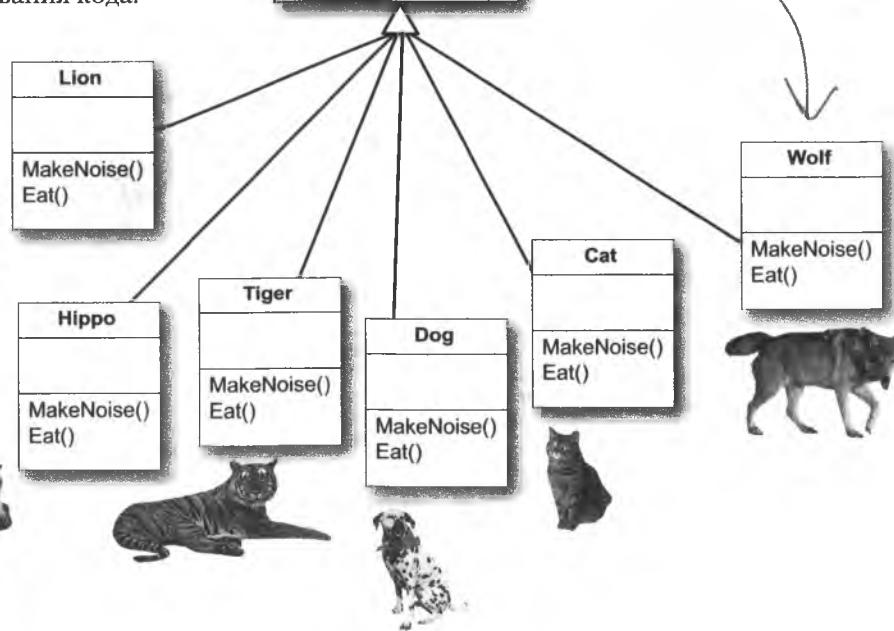
4

### 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 перемещаются по месту своего обитания. Наверное, имеет смысл создать для них базовый класс Feline (Кошачьи), который будет производным от класса Animal. Это позволит избежать дублирования кода.

Animal
Picture
Food
Hunger
Boundaries
Location
MakeNoise()
Eat()
Sleep()
Roam()

Не добавить ли нам класс Canine (Псовые), от которого будут наследовать как собаки, так и волки?



От класса Animal наследуются все четыре метода, но перекрываются только методы MakeNoise() и Eat().

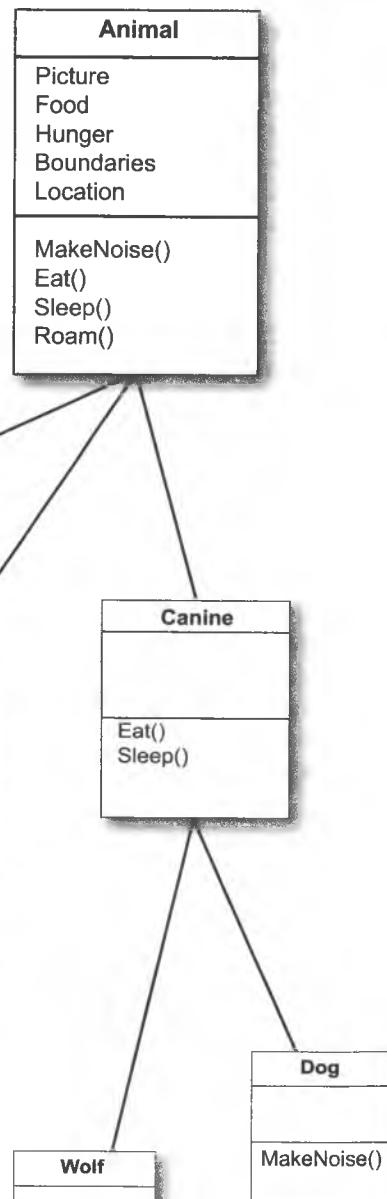
Вот почему на диаграмме классов показаны только эти два метода.

## Иерархия классов

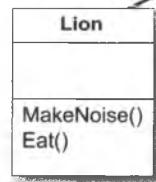
Конструкция, в которой под базовым классом располагаются производные классы, которые, в свою очередь, являются базовыми для других классов, называется **иерархией** (*class hierarchy*). Подобный подход не только позволяет избежать многократного дублирования кода, но и делает код намного более читабельным. 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

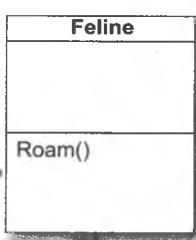
Вам осталось добавить классы Feline и Canine, и иерархия будет готова.



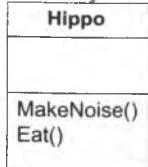
Класс Feline перекрывает метод Roam(), поэтому все унаследованные свойства будут иметь дело с новым методом, а не с тем, который был определен в классе Animal.



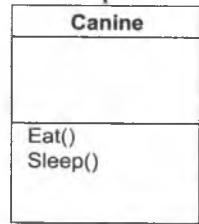
Все кошки двигаются одинаково, поэтому метод Roam() для них общий. Но они по-разному пытаются и издают разные звуки, поэтому методы Eat() и MakeNoise(), унаследованные от класса Animal, перекрываются.



**Roam()**

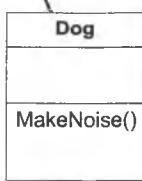


**MakeNoise()**  
**Eat()**



**Canine**

**Eat()**  
**Sleep()**

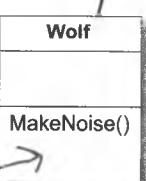


**Dog**

**MakeNoise()**



Волки и собаки питаются одинаково, поэтому их общий метод Eat() помещен в класс Canine.



## 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 псовых и никак не повлияет на классы Wolf, Canine, Animal, Hippo и любые другие.

создает новый объект Dog

```
Dog spot = new Dog();
```

вызывает метод класса Dog

```
spot.MakeNoise();
```

вызывает метод класса Animal

```
spot.Roam();
```

вызывает метод класса Canine

```
spot.Eat();
```

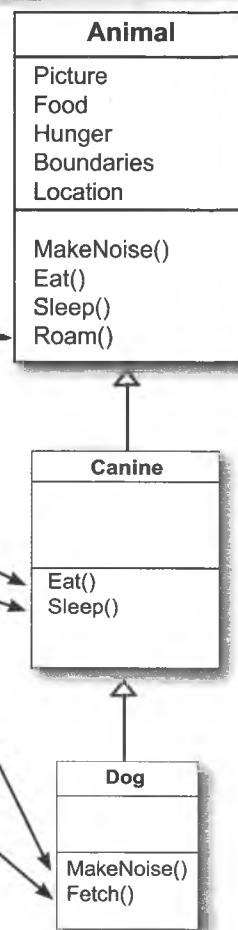
вызывает метод класса Canine

```
spot.Sleep();
```

вызывает метод класса Dog

```
spot.Fetch();
```

**и-е-пар-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

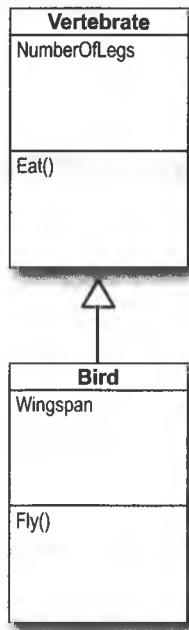
Перемещаться собаку заставляет всего один метод из класса Animal. А вот какой из методов MakeNoise() нужно вызвать, чтобы собака залаяла?

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.

как низко вы можете пасть?

## Синтаксис наследования

При наследовании имена производного и базового классов разделяются двоеточием (:). Производный класс получает все поля, свойства и методы базового класса.



```
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
    }
}
```

При наследовании все поля, свойства и методы базового класса автоматически добавляются в производный класс.

Вы расширяете класс, добавив в конец его объявления двоеточие и имя базового класса.

Так как tweety — это экземпляр объекта Bird, он имеет все методы и поля этого объекта.

```
public button1_Click(object sender, EventArgs e) {
    Bird tweety = new Bird();
    tweety.Wingspan = 7.5;
    tweety.Fly();
    tweety.NumberOfLegs = 2;
    tweety.Eat();
}
```

Так как класс Bird — производный по отношению к классу Vertebrate, все экземпляры Bird имеют поля и методы, определенные в классе Vertebrate.

## Задаваемые вопросы

В: Почему стрелка указывает от производного класса к базовому? Не логичней было бы рисовать ее наоборот?

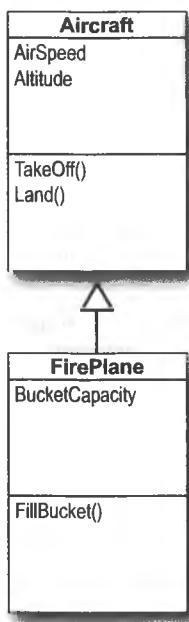
О: Это было бы не совсем точно. Заставляя один класс наследовать от другого, вы встраиваете это отношение в производный класс, а базовый остается без изменений. Это имеет смысл, если подумать о происходящем с точки зрения базового класса.

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.

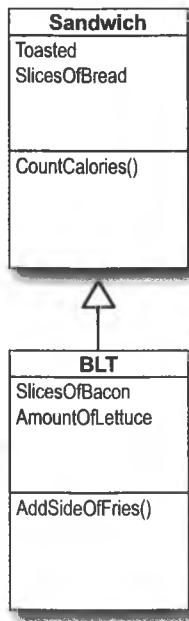


**Aircraft**

- AirSpeed
- Altitude
- TakeOff()
- Land()

**FirePlane**

- BucketCapacity
- FillBucket()



**Sandwich**

- Toasted
- SlicesOfBread
- CountCalories()

**BLT**

- SlicesOfBacon
- AmountOfLettuce
- AddSideOfFries()

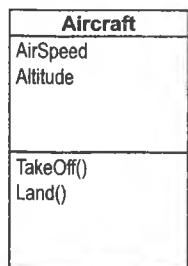
Я знаю, как заставить летать пингвина...

Возьми в руку карандаш

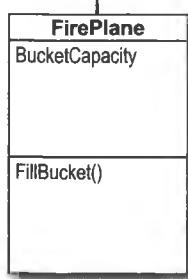


Решение

Вот какие операторы следовало обвести как неработающие.



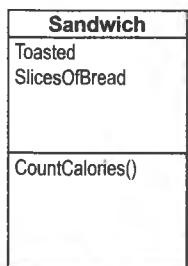
```
class Aircraft {  
    public double AirSpeed;  
    public double Altitude;  
    public void TakeOff() { ... };  
    public void Land() { ... };  
}  
  
class FirePlane : Aircraft {  
    public double BucketCapacity;  
    public void FillBucket() { ... };  
}
```



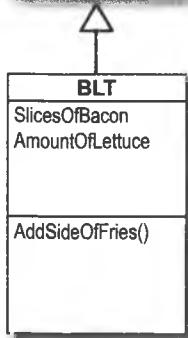
```
public void FireFightingMission() {  
    FirePlane myFirePlane = new FirePlane();  
    new FirePlane.BucketCapacity = 500;  
    Aircraft.Altitude = 0;  
    myFirePlane.TakeOff();  
    myFirePlane.AirSpeed = 192.5;  
    myFirePlane.FillBucket();  
    Aircraft.Land();  
}
```

Ключевое слово **new**  
так использовать  
нельзя.

Эти операторы  
используют имя  
класса вместо  
имени экземпляра  
myFirePlane.



```
class Sandwich {  
    public boolean Toasted;  
    public int SlicesOfBread;  
    public int CountCalories() { ... }  
}
```



```
class BLT : Sandwich {  
    public int SlicesOfBacon;  
    public int AmountOfLettuce;  
    public int AddSideOfFries() { ... }  
}
```

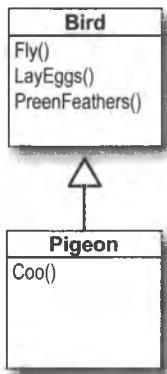
```
public BLT OrderMyBLT() {  
    BLT mySandwich = new BLT();  
    BLT.Toasted = true;  
    Sandwich.SlicesOfBread = 3;  
    mySandwich.AddSideOfFries();  
    mySandwich.SlicesOfBacon += 5;  
    MessageBox.Show("В моем сэндвиче "  
        + mySandwich.CountCalories + " калорий.");  
    return mySandwich;  
}
```

Эти свойства при-  
надлежат экземпляру,  
в то время как опе-  
раторы пытаются  
вызывать их по имени  
классов.

После имени метода  
CountCalories отсут-  
ствуют скобки () .

## При наследовании поля свойства и методы базового класса добавляются к производному...

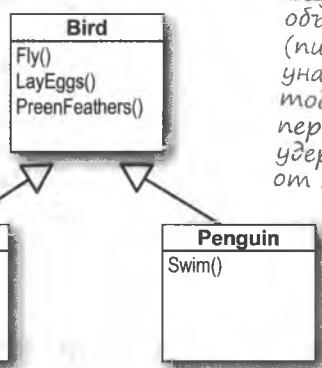
Наследование является простым, если производному классу нужны **все** методы, свойства и поля базового класса.



Здесь **Pigeon** (голубь) производный класс от **Bird**, поэтому ему принадлежат все методы этого класса — **Fly()** (летать), **LayEggs()** (откладывать яйца), **PreenFeathers()** (чищить перья), — а также его собственный метод **Coo()** (ворковать).

## ...но не Все птицы летают!

Что делать, если базовый класс имеет метод, который в производном классе требуется **отредактировать**?



Izzy — экземпляр объекта **Penguin** (пингвин). Он унаследовал метод **Fly()**, и теперь ничто не удерживает его от полета!

```

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();
}
  
```

Как класс **Pigeon**, так и класс **Penguin** наследуют у класса **Bird**, так что оба они получают методы **Fly()**, **LayEggs()** и **PreenFeathers()**.

Пингвины не должны летать! Но так как класс **Penguin** наследует от класса **Bird**, у бедняг просто нет выбора!



Если бы этот код писали вы, как бы вы избавили пингвина от необходимости летать?

Голуби летают, откладывают яйца и чистят перья, поэтому проблем с наследованием от класса **Bird** не возникает.

## 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.

1

### 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!")  
    }  
}
```

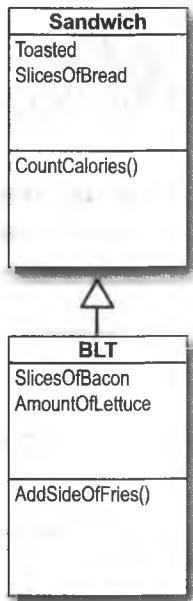
Чтобы перекрыть метод `Fly()`, добавьте в производный класс идентичный метод и воспользуйтесь ключевым словом `override`.

При перекрытии сигнатуры нового метода должна совпадать с сигатурой 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`.**

## 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 с беконом, салатом и помидорами `BLT` (сэндвич). Свойства этого специального 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.



# сМесь пражнение сОбщениЙ

## 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, ";
    }
}
```

**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();

```

a = 6; → 56  
b = 5; → 11  
a = 5; → 65

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.

```
class C : B {
    public _____ string m3() {
        return "C's m3, " + (ivar + 6);
    }
} Вот точка входа в программу, она не показывает форму, а только вызывает окно с сообщением.
class Mixed5 {
    public static void Main(string[] args) {
        A a = new A();
        B b = new B();
        C c = new C();
        A a2 = new C(); Подсказка: Как следует подумайте, что именно это означает.
        string q = "";
        _____
        System.Windows.Forms.MessageBox.Show(q);
    }
}
```

Вставьте сюда  
фрагмент (три  
строчки)

## 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, A's m3,

B's m1, A's m2, C's m3, 13

B's m1, C'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 ИСР, 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 сложней, чем кажется на первый взгляд!

```
class Rowboat ..... {
    public ..... rowTheBoat() {
        return "stroke natasha";
    }
}

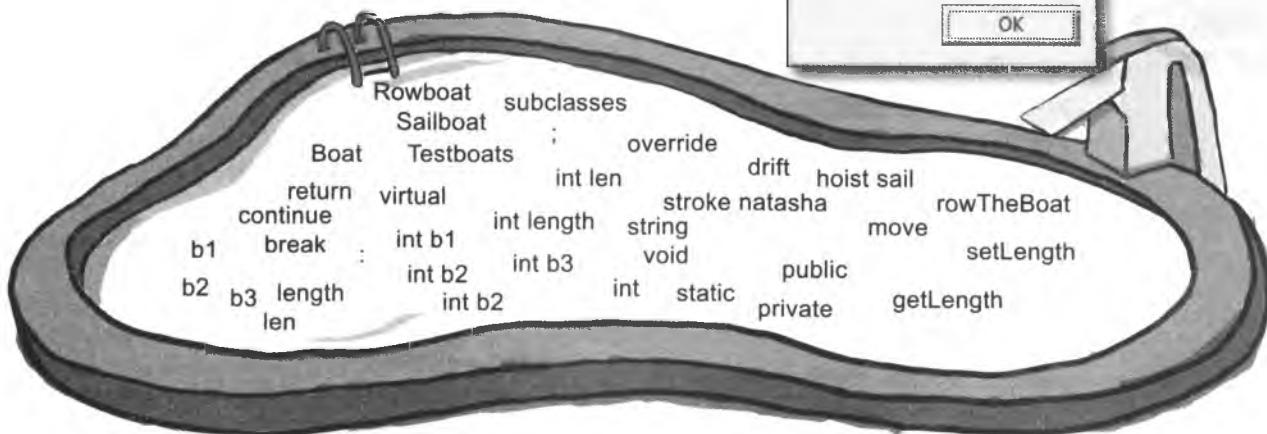
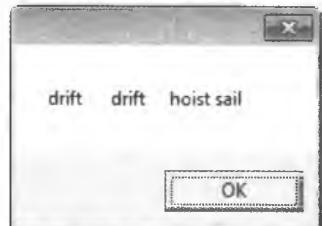
class ..... {
    private int ..... ;
    ..... void ..... (.....) {
        length = len;
    }
    public int getLength() {
        ..... ;
    }
    public ..... move() {
        return " ..... ";
    }
}
```

```
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:





## сМеCь сОoежeHий

```
class A {
    public virtual string m1() { ... }
    public virtual string m3() { ... }
}

class B : A {
    public override string m1() { ... }
}

class C : B {
    public override string m3() { ... }
}
```

a = 6; → 56  
b = 5; → 11  
a = 5; → 65

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();

Diagram illustrating method resolution order (MRO) for the code above:

- Top row:** q += b.m1();  
q += c.m2();  
q += a.m3(); } → A's m1, A's m2, C's m3, 6
- Second row:** q += c.m1();  
q += c.m2(); } → B's m1, A's m2, A's m3, A's m1, B's m2, C's m3, 6
- Third row:** q += a.m1();  
q += b.m2(); } → B's m1, A's m2, C's m3, 13  
q += c.m3(); } → B's m1, C's m2, A's m3, A's m1, B's m2, A's m3,
- Fourth row:** q += a2.m1(); } → B's m1, A's m2, C's m3, 6  
q += a2.m2(); } → B's m1, A's m2, C's m3, 13  
q += a2.m3(); } → 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, ИСР 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. 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.

**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.

**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.**



А я не понимаю, зачем нужны ключевые слова `virtual` и `override`. Если они отсутствуют, ИСР показывает предупреждение. Но программа все равно запускается! Так какая разница? Нет, я, конечно, могу писать эти слова, если так делать «правильно», но кажется, меня просто заставляют выполнять лишнюю работу.

### 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.

### Упражнение!

На этом 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` (Борз).

#### 2 Код для новых классов.

Добавьте следующий код:

```
class Jewels {  
    public string Sparkle() {  
        return "Sparkle, sparkle!";  
    }  
}  
Обратите внимание, что слово private скрывает переменные contents и combination.  
{  
    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);  
    }  
}
```

Если при создании нового проекта вместо варианта `Windows Forms application` выбрать вариант `Console Application`, ИСР 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.

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());
    }
}

```

## наследование

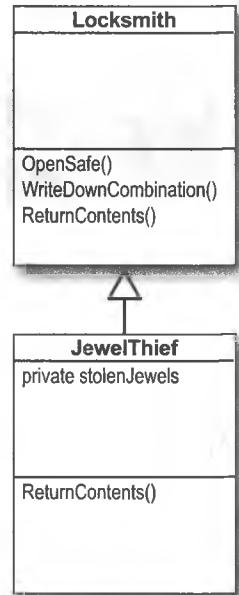
### ❸ Класс JewelThief наследует от класса Locksmith.

Воры 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); Метод OpenSafe() из класса
    } Locksmith подбирает ключ,
    private string writtenDownCombination = null; открывает сейф и возвращает
    public void WriteDownCombination(string combination) { его содержимое владельцу.
        writtenDownCombination = combination;
    }
}

```



```

public void ReturnContents(Jewels safeContents, Owner owner) {
    owner.ReceiveContents(safeContents);
}
}

```

```

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());
    }
}

```

### ❹ Метод Main() для класса Program.

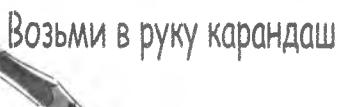
*Пока не запускайте программу!* Попробуйте угадать, что будет написано в консоли.

```

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.



Внимательно посмотрите код программы и запишите сообщение, которое появится в консоли. (Подсказка: Определите, какие свойства класс JewelThief наследует от класса Locksmith!)



*скрыть и обнаружить*

## Производный класс умеет скрывать методы

Запустите программу JewelThief. Так как это консольное приложение, вывод результатов осуществляется через командное окно. Вот как это выглядит:



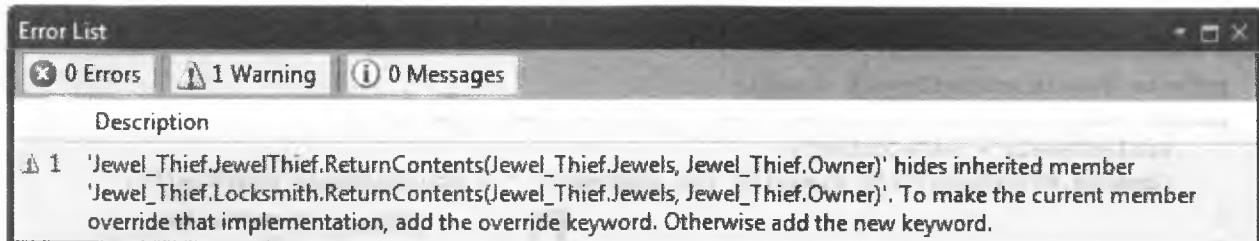
Вы ожидали, что программа напишет кое-что другое? Например:

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 посыпает приложение:



Так как класс JewelThief наследует от класса Locksmith и по идеи 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 совпадает 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 вызывает
// 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?**

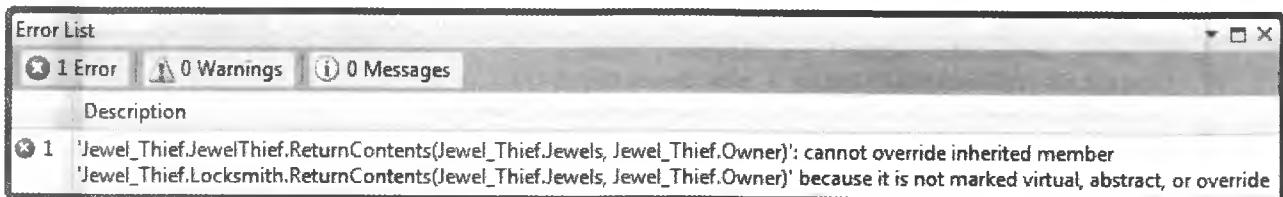
так вот зачем нужны эти ключевые слова

## 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:



Этого мы и добивались!



**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!

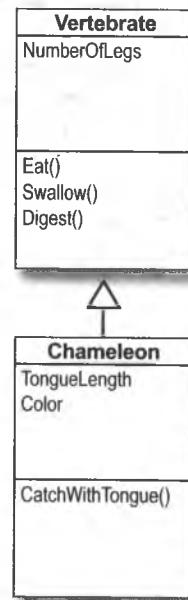
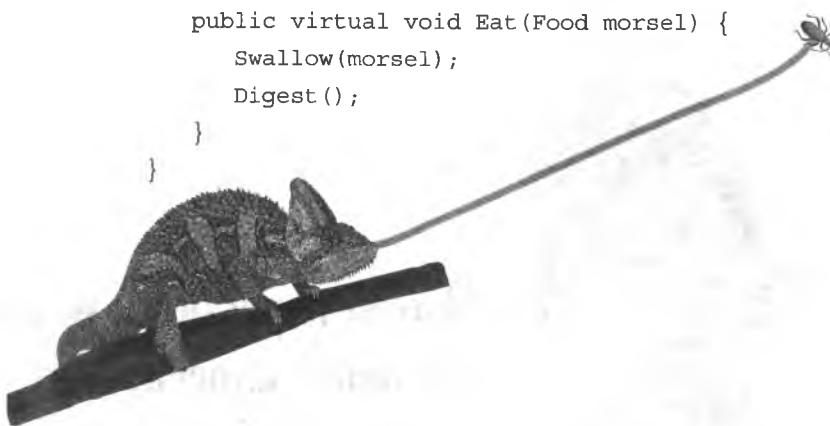
**Чтобы перекрыть метод  
базового класса, всег-  
да помечайте его клю-  
чевым словом `virtual`.  
И всегда используй-  
те ключевое слово  
`override`, когда хотите  
перекрыть метод в  
производном классе.  
Если это не сделать,  
некоторые методы вне-  
запно могут оказаться  
скрытыми.**

## 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, поэтому 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 оставаться 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 конструктор производного класса, и при 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.

**Конструктор базового класса вызывается первым**

Убедитесь в этом сами!

Упражнение!

### 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:

```
Выберите ко-  
манду Build >>  
Build Solution,  
и вы полу-  
ти сообщение  
об ошибке.  
class MySubclass : MyBaseClass{
    public MySubclass(string baseClassNeedsThis, int anotherValue) {
        MessageBox.Show("This is the subclass: " + baseClassNeedsThis
            + " and " + anotherValue);
    }
}
```

Эта ошибка означает, что производный класс не вызвал конструктор из базового класса.

1 No overload for method 'MyBaseClass' takes '0' arguments

### 3 Заставьте сначала вызывать конструктор из базового класса

Затем инициализируйте производный класс и посмотрите, в каком порядке появятся два окна диалога!

```
class MySubclass : MyBaseClass{
```

Так мы послали базовому классу параметр, который требуетсѧ его конструктору.

→ : base(baseClassNeedsThis)

Эта строка вызовет конструктор базово-го класса. После этого сообщение об ошибке исчезнет, и программа начнет работать.

## Теперь мы готовы завершить программу для Кэтлин!

После последнего обновления программы 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.

DinnerParty	BirthdayParty
NumberOfPeople	NumberOfPeople
CostOfDecorations	CostOfDecorations
CostOfBeveragesPerPerson	CakeSize
HealthyOption	CakeWriting
CalculateCostOfDecorations()	CalculateCostOfDecorations()
CalculateCost()	CalculateCost()
SetHealthyOption()	

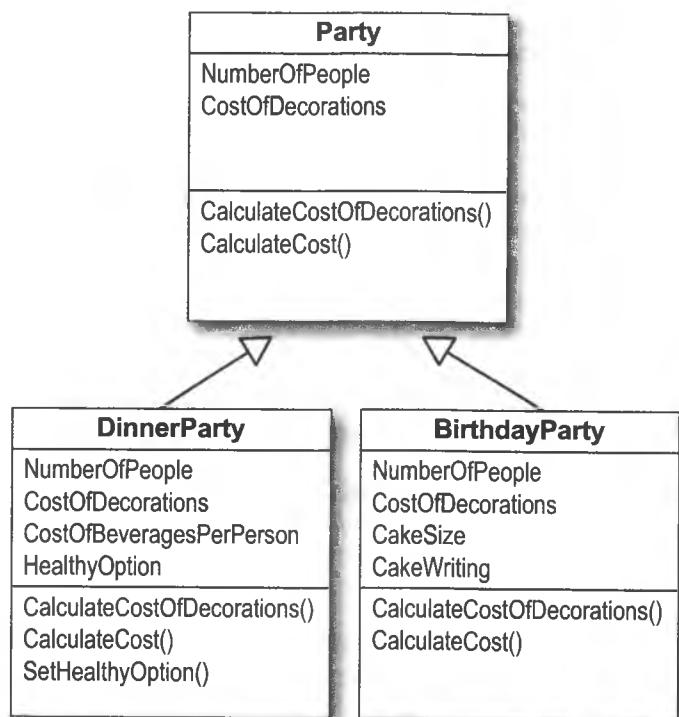


Если все сделать правильно, достаточно будет отредактировать два класса, не затрагивая саму форму!

1

### Новая модель классов

У нас останутся классы DinnerParty и BirthdayParty, но теперь они будут наследовать от единого класса Party. Так как эти методы должны иметь одинаковые свойства, методы и поля, редактировать форму не потребуется. Просто некоторые параметры переместятся в базовый класс Party, и мы будем перекрывать их по мере необходимости.



3

### Базовый класс Party

наследование

Создайте класс `Party` и укажите уровень доступа `public`. Рассмотрим диаграмму классов и решим, какие свойства и методы следует перенести из классов `DinnerParty` и `BirthdayParty` в класс `Party`.

Вскоре вы по-  
знакомитесь  
с ключевым  
словом `protected`.  
Оно открывает  
поле для произ-  
водного класса,  
оставляя его  
закрытым для  
всего осталь-  
ного.

- ★ Переместите свойства `NumberOfPeople` и `CostOfDecorations`, так как они определены как для `DinnerParty`, так и для `BirthdayParty`.
- ★ Проделайте то же самое с методами `CalculateCostOfDecorations()` и `CalculateCost()`. Не забудьте перенести и закрытые поля, необходимые для этих методов. (Помните, что производные классы видят только поля общего доступа. После переноса закрытого поля в класс `Party`, классы `DinnerParty` и `BirthdayParty` потеряют к нему доступ.)
- ★ Еще вам потребуется конструктор. Внимательно посмотрите на конструкторы `BirthdayParty` и `DinnerParty`: не имеют ли они что-нибудь общее.
- ★ Теперь добавим платеж **\$100** к мероприятиям с количеством гостей более 12. Это касается как званых обедов, так и дней рождения, поэтому поместим этот платеж в класс `Party`.

3

### Пусть класс `DinnerParty` наследует от класса `Party`

Теперь, когда класс `Party` взял на себя часть функций класса `DinnerParty`, в последнем можно оставить только функции, связанные с обедами.

- ★ Убедитесь, что конструктор работает. Он делает что-то, чего не делает конструктор класса `Party`? Оставьте только эти функции.
- ★ Все параметры и методы варианта `Healthy Option` должны остаться в классе `DinnerParty`.
- ★ Если мы хотим сохранить код формы, то не сможем перекрыть метод `CalculateCost()`, так как форма должна передать в него логическую переменную `healthyOption`. Поэтому мы просто добавим в класс еще один метод `CalculateCost()`, работающий с другими параметрами. Используйте для его объявления код из начала данной главы. Теперь запись `base.CalculateCost()` даст доступ к методу `CalculateCost()` в классе `Party`.

Эта процедура называется перегрузкой. Она будет подробно рассмотрена в главе 8.

4

### Пусть класс `BirthdayParty` также наследует от класса `Party`

Для класса `BirthdayParty` проведите аналогичную процедуру, перенеся все общие признаки в базовый класс.

- ★ Какие действия конструктора `BirthdayParty` не являются частью класса `Party`?
- ★ Вам нужно посчитать стоимость торта внутри класса `BirthdayParty`. Это затрагивает метод и свойства, поэтому их потребуется перекрыть.
- ★ Свойства тоже можно перекрывать! Присваивая значение переменной `base.NumberOfPeople`, вы обращаетесь к методу записи в базовом классе. Ключевое слово `base` потребуется вам как для чтения, так и для записи.



## 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
{
    const int CostOfFoodPerPerson = 25;
    private bool fancyDecorations;
    public decimal CostOfDecorations = 0;

    public Party(int numberOfPeople, bool fancyDecorations)
    {
        this.fancyDecorations = fancyDecorations;
        this.NumberOfPeople = numberOfPeople;
    }

    private int numberOfPeople;
    public virtual int NumberOfPeople {
        get { return numberOfPeople; }
        set {
            numberOfPeople = value;
            CalculateCostOfDecorations(fancyDecorations);
        }
    }

    public void CalculateCostOfDecorations(bool fancy) {
        fancyDecorations = fancy;
        if (fancy)
            CostOfDecorations = (NumberOfPeople * 15.00M) + 50M;
        else
            CostOfDecorations = (NumberOfPeople * 7.50M) + 30M;
    }

    public virtual decimal CalculateCost() {
        decimal TotalCost = CostOfDecorations + (CostOfFoodPerPerson * NumberOfPeople);
        if (NumberOfPeople > 12)
        {
            TotalCost += 100M;
        }
        return TotalCost;
    }
}
```

Этот код был перемещен из классов DinnerParty и BirthdayParty в класс Party.

Конструктор класса Party содержит функции, которые раньше были конструкторами классов DinnerParty и BirthdayParty.

Перед параметром NumberOfPeople требуется вставить ключевое слово `virtual`, так как класс BirthdayParty должен его перекрыть (в этом случае изменение количества гостей меняет диаметр торта).

Стоимость оформления для мероприятий обоих типов рассчитывается по одной и той же схеме, поэтому ее логично поместить в класс Party.

Метод вычисления стоимости мероприятия нужно пометить словом `virtual`, так как его необходимо перекрыть в классе BirthdayParty.

```

class BirthdayParty : Party {
    public int CakeSize;

    public BirthdayParty(int number_of_people, bool fancy_decorations, string cake_writing)
        : base(number_of_people, fancy_decorations) {
        CalculateCakeSize();
        this.CakeWriting = cake_writing;
        CalculateCostOfDecorations(fancy_decorations);
    }

    private void CalculateCakeSize() {
        if (Number_of_People <= 4)
            CakeSize = 8;
        else
            CakeSize = 16;
    }

    private string cakeWriting = "";
    public string CakeWriting {
        get { return this.cakeWriting; }
        set {
            int maxLength;
            if (CakeSize == 8)
                maxLength = 16;
            else
                maxLength = 40;
            if (value.Length > maxLength) {
                MessageBox.Show("Too many letters for a " + CakeSize + " inch cake");
                if (maxLength > this.cakeWriting.Length)
                    maxLength = this.cakeWriting.Length;
                this.cakeWriting = cakeWriting.Substring(0, maxLength);
            } else
                this.cakeWriting = value;
        }
    }

    public override decimal CalculateCost() {
        decimal CakeCost;
        if (CakeSize == 8)
            CakeCost = 40M + CakeWriting.Length * .25M;
        else
            CakeCost = 75M + CakeWriting.Length * .25M;
        return base.CalculateCost() + CakeCost;
    }

    public override int Number_of_People {
        get { return base.Number_of_People; }
        set {
            base.Number_of_People = value;
            CalculateCakeSize();
            this.CakeWriting = cakeWriting;
        }
    }
}

```

Конструктор оставляет основной объем работы базовому классу. И вызывает метод `CalculateCakeSize()`, как делал и старый конструктор из класса `BirthdayParty`.

Метод `CalculateCakeSize()` помещен в класс `BirthdayParty`, так диаметр торта учитывается только при расчете стоимости дней рождения.

По этой же причине свойство `CakeWriting` остается в классе `BirthdayParty`.

Метод `CalculateCost()` нуждается в перекрытии, так как сначала мы вычисляем стоимость торта, а потом хотим добавить ее к сумме, которую посчитал метод `CalculateCost()` из класса `Party`.

Здесь свойство `Number_of_People` должно перекрывать одноименное свойство из класса `Party`, так как методу записи нужно пересчитывать размер торта. При этом вызывается `base.Number_of_People`, что провоцирует выполнение метода записи из класса `Party`.

→ II Родолжение на с. 272.



пражнение  
Решение

```

class DinnerParty : Party
{
    public decimal CostOfBeveragesPerPerson; // Это поле общего доступа используется только при расчете стоимости званых обедов и поэтому остается в этом классе.

    public DinnerParty(int numberofPeople, bool healthyOption,
                       bool fancyDecorations) // Чтобы воспроизвести функциональность старого класса DinnerParty, новый конструктор вызывает сначала конструктор Party, а затем метод SetHealthyOption().
    {
        base(numberofPeople, fancyDecorations);
        SetHealthyOption(healthyOption);
        CalculateCostOfDecorations(fancyDecorations);
    }

    public void SetHealthyOption(bool healthyOption) { // Метод SetHealthyOption() остается без изменений.
        if (healthyOption)
            CostOfBeveragesPerPerson = 5.00M;
        else
            CostOfBeveragesPerPerson = 20.00M;
    }

    public decimal CalculateCost(bool healthyOption) {
        decimal totalCost = base.CalculateCost()
            + (CostOfBeveragesPerPerson * NumberofPeople);

        if (healthyOption)
            return totalCost * .95M;
        else
            return totalCost;
    }
}

```

Программа работает!

Теперь мой бизнес пошел в гору,  
спасибо огромное!

О том, как именно работает перегрузка, вы узнаете в главе 8.

### Стойте! В программе все еще осталась потенциальная ошибка!

В классе DinnerParty сейчас два метода CalculateCost(): один унаследованный от класса Party, а другой созданный нами. Этот класс не инкапсулирован, поэтому остается возможность вызывать неправильный метод CalculateCost():

```

DinnerParty dinner = new DinnerParty(5, true, true);
decimal cost1 = dinner.CalculateCost(true);
decimal cost2 = dinner.CalculateCost();

```

cost1 получит значение 261.25, в то время как cost2 – значение 250. Проблема... Иногда в базовом классе существует код, к которому вы не хотите обращаться напрямую. Или вы не собирались создавать экземпляры класса Party, но такая возможность сохранилась. А что будет, если человек, редактирующий наш код, создаст экземпляр класса Party? Явно не то, что мы планировали.

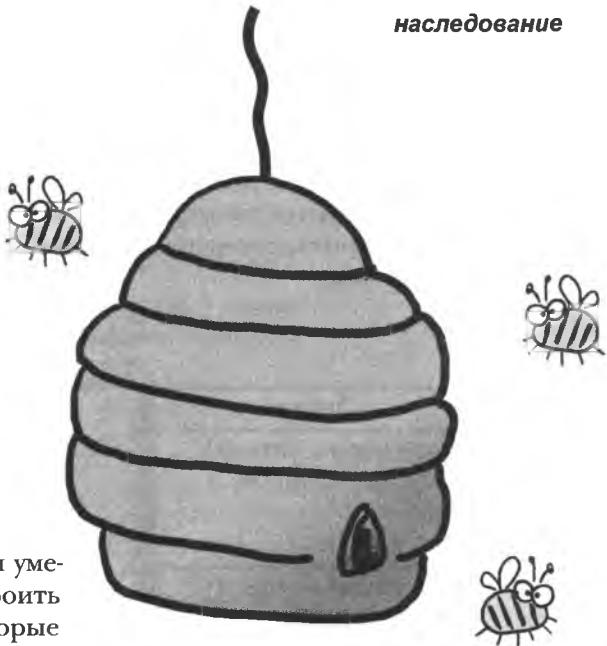
О том, как решить эту проблему, вы узнаете в следующей главе!



## Система управления ульем

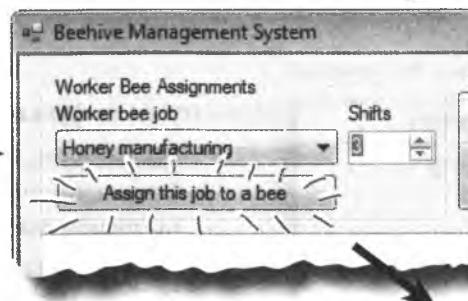
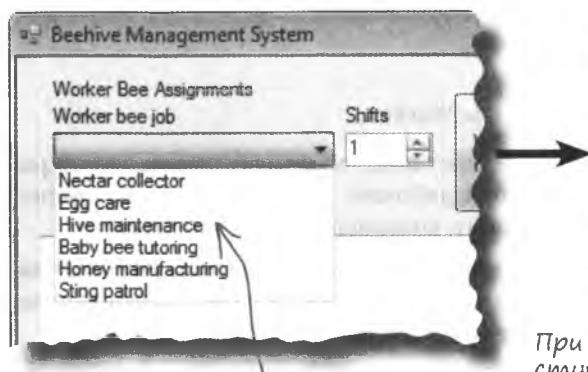
Теперь ваша помощь нужна пчелиной матке! Улей вышел из-под контроля и ей нужна программа, которая поможет им управлять. Улей полон рабочих пчел, имеется и список задачий. Нужно распределить задания между пчелами с учетом их специализации.

Постройте систему, управляющую поведением рабочих пчел. Вот как она должна функционировать:



### 1 Матка раздает задания рабочим

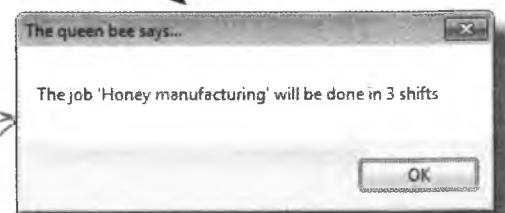
Существует шесть видов работ. Некоторые пчелы умеют собирать нектар и делать мед, другие могут строить улей и защищать его от врагов. Есть пчелы, которые вообще могут выполнять любую работу. Вам нужно написать программу, дающую пчеле задание, которое она в состоянии выполнить.



Пчелы труждаются посменно, а большинство работ выполняется в несколько смен. Матка вводит число смен в поле Shifts и щелкает на кнопке Assign this job, чтобы дать задание свободным пчелам.

При наличии доступных пчел программа дает им задание и отчитывается перед маткой, выводя окно:

«Продукция меда будет закончено за три смены».



Существует шесть видов работ. Матке все равно, чем занимается отдельная пчела. Она всего лишь указывает, что нужно сделать, а программа определяет наличие доступных рабочих и дает им задание.

### 2 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.



## Построение основ

Этот проект делится на две части. Начнем с обзора системы управления ульем. Вам потребуются два класса – Queen (Матка) и Worker (Рабочий), – которые следует инкапсулировать. Не обойдемся мы и без связанной с этими классами формой.

Иногда в диаграмме классов могут появиться закрытые поля и типы.



Объект Queen указывает, какую работу нужно сделать.

Queen
private workers: Worker[] private shiftNumber: int
AssignWork() WorkTheNextShift()

Свойства CurrentJob  
(Текущее Задание) и ShiftsLeft  
(Оставшиеся Смены)  
предназначены только для чтения.

Worker
CurrentJob: string ShiftsLeft: int
private jobsICanDo: string[] private shiftsToWork: int private shiftsWorked: int

Посмотрим на функции массива Worker.

- ★ Свойство CurrentJob дает понять объекту Queen, какую работу выполняет каждый рабочий. Если рабочий на момент проверки ничем 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.
- ★ Метод WorkOneShift() заставляет рабочего отработать одну смену, а также отслеживает оставшееся количество смен. Если работа закончена, возвращается пустая строка. Это означает, что он готов к выполнению следующего задания.

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.

Про то, как усовершенствовать систему управления ульем при помощи наследования, читайте на с. 282



## Упражнение

Итак, приступим к построению системы управления ульем, которая облегчит жизнь пчелиной матке.

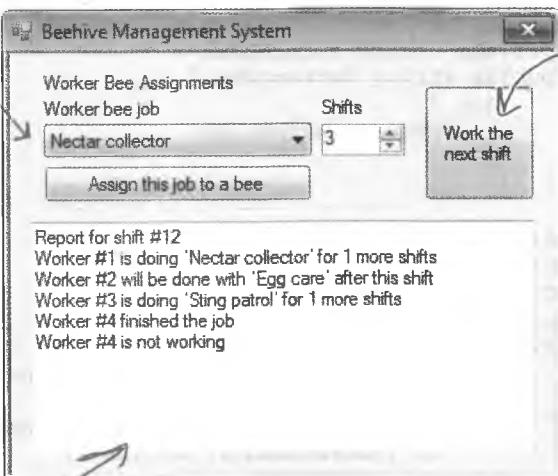
1

### Построение формы

Форма будет очень простой, всю работу предстоит выполнить классам Queen и Worker. Добавьте закрытое поле Queen и две кнопки, вызывающие методы AssignWork() и WorkTheNextShift(). Еще потребуется элемент ComboBox, управляющий работой пчел (пункты этого раскрывающегося списка были перечислены на предыдущей странице), элемент NumericUpDown, две кнопки и текстовое поле для отчета о сменах. Вид формы показан ниже.

Элемент ComboBox называется workerBeeJob. Используйте свойство Items для формирования списка. DropDownListStyle присвойте значение DropDownList, чтобы пользователь мог выбирать значения только из раскрывающегося списка. Поле Shifts является элементом NumericUpDown с именем shifts.

Дайте элементу TextBox имя report, а его свойству MultiLine значение true.



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 следуют списки дел работников. Используйте escape-последовательности "\r\n", чтобы добавить знак переноса в строку.

```
public Form1() {
    InitializeComponent();
    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)
}
```

Форме потребуется поле Queen с именем queen. Массив Worker нужно будет передать конструктору объекта Queen.

Конструктор каждого объекта Worker получает массив строк с информацией о том, какие работы может выполнять пчела.

2

### Создание классов Worker и Queen

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 да, закрытому полю shiftsToWork присваивается количество смен, параметру CurrentJob – название работы, а переменной shiftsWorked – значение «ноль». Отработанная смена увеличивает эту переменную на 1. При помощи предназначенного только для чтения свойства ShiftsLeft матка оценивает, сколько смен еще нужно 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 WorkOneShift() {
        if (String.IsNullOrEmpty(currentJob))
            return false;
        shiftsWorked++;
        if (shiftsWorked > shiftsToWork) {
            shiftsWorked = 0;
            shiftsToWork = 0;
            currentJob = "";
            return true;
        } else
            return false;
    }
}

```

Матка использует метод `DoThisJob()` объекта `worker` для назначения заданий рабочим, рабочий проверяет свойство `JobsICanDo`, чтобы понять, может ли он выполнить данную работу.

Матка использует метод `WorkOneShift()` 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 числа свободна.

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.

Оператор `!=` означающий НЕТ — проверяет, является ли строка пустой и имеет ли значения `null`. Именно таким образом осуществляется проверка несоблюдения условия.

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].WorkOneShift())
                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;
    }
}

```

*Метод WorkTheNextShift() объекта queen заставляет каждого рабочего трудиться следующую смену и добавляет строку в отчет.*

Вы знаете код для конструктора. Вот код для остальной части формы:

```

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();
}

```

*Кнопка nextShift заставляет матку раздать задания на следующую смену. При этом в текстовом поле формы появляется отчет.*

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.

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.



## Упражнение

Работа еще не закончена! Матке требуется система учета потребляемого в улье меда. Великолепный шанс применить на практике свои знания!

1

### Сколько же меда потребляет улей

Матке только что позвонила пчела-бухгалтер и сказала, что в улье недостаточно меда. Бухгалтеру нужна информация о количестве потребляемого меда, чтобы понять, не нужно ли часть рабочих, занимающихся заботой о потомстве, перевести на добывчу меда.

- ★ Все пчелы потребляют мед, поэтому его должно быть много.
- ★ Занятые рабочие потребляют больше меда. Больше всего его требуется в момент начала работы, потом потребление падает. В последнюю смену пчела съедает 10 единиц меда; в предпоследнюю – 11; перед этим – 12 и т. д. То есть если пчела работает (ее переменная `ShiftsLeft` больше нуля), количество потребляемого меда можно вычислить, прибавив 9 к `ShiftsLeft`.
- ★ Неработающая пчела (`ShiftsLeft` равно нулю) съедает 7.5 единиц меда за всю смену.
- ★ Это данные для обычных пчел. Если вес пчелы превышает 150 мг, она ест на 35% меда больше. (К матке эта информация не относится.)
- ★ Чем больше занятых рабочих, тем больше меда потребляет матка, так как больше сил тратится на управление ульем. Количество рабочих смен матки равно количеству смен пчелы, которой осталось работать дольше всего.
- ★ Матка потребляет в смену на 20 единиц меда больше, если работает не больше 2 пчел или меньше, или на 30 единиц больше, если работают 3 пчелы и более. Так как вес матки составляет 275 мг, к ней не относится правило 35%.
- ★ Данные о количестве потребляемого меда нужно добавить в конец отчета об отработанных сменах.

2

### Создайте класс Bee для учета потребления меда

Пчелы должны знать свой вес, чтобы определить, не съедают ли они на 35% меда больше.

- ★ Создайте метод `GetHoneyConsumption()`, вычисляющий потребление меда. Этот метод понадобится как рабочим пчелам, так и матке, у которой, впрочем, способ потребления немного отличается, поэтому класс `worker` может просто наследовать данный метод, а класс `queen` будет перекрывать его.
- ★ Методу `GetHoneyConsumption()` нужна информация о количестве смен, которые осталось отработать, поэтому добавим свойство `ShiftsLeft`, предназначенное только для чтения, и пометим его ключевым словом `virtual`, чтобы дать возможность перекрыть его в классе `worker`. Свойство это возвращает нулевое значение.
- ★ Так как потребление меда зависит от веса пчелы, конструктор `Bee` должен иметь поле для хранения данной информации. Другим классам эти сведения не потребуются, поэтому пометим их ключевым словом `private`.



Поля и свойства следует делать закрытыми по умолчанию и открывать, только когда к ним требуется доступ из других классов.

Подсказка: Воспользуйтесь появлением сообщением об ошибке «но overload!» Двойной щелчок на этом сообщении переместит вас в конструктору класса Worker. Очень удобно!



### 3 Класс Worker должен наследовать от класса Bee

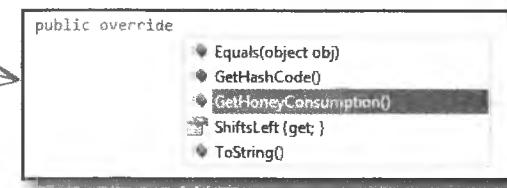
Конструктор класса Worker должен брать вес пчелы и передавать его конструктору базового класса. Вам остается только добавить ключевое слово `override` к методу `ShiftLeft`. После этого каждая рабочая пчела получит возможность вычислить свое потребление меда, и ваша работа по редактированию класса Worker будет завершена!

### 4 Класс Queen также должен наследовать от класса Bee

С классом Queen придется повозиться, так как именно здесь происходит подсчет количества меда и формируется отчет.

- ★ Перекройте метод `Bee.GetHoneyConsumption()` и добавьте к нему дополнительные вычисления. Нужно определить количество рабочих пчел, от этого зависит потребление меда маткой. Количество смен матки равно количеству смен самой занятой рабочей пчелы.
- ★ Обновите метод `WorkTheNextShift()`, добавив к отчету информацию о потреблении меда. Вам потребуется цикл, который будет добавлять эти сведения для каждого рабочего, а также вычислять пчелу с самым высоким потреблением, цикл должен располагаться **до** кода, отправляющего пчел на работу (тогда матка будет знать количество меда, съеденное во время текущей смены). Также нужно учесть потребление меда самой маткой и добавить в конец отчета строчку «`Total Honey Consumption: xxx units`» (Суммарное потребление меда).
- ★ Обновите конструктор для класса Queen, как вы это делали с конструктором для класса Worker.

В классе Queen наберите `public override` и нажмите пробел. Появится список доступных для перекрытия методов. После выбора нужного варианта название базового метода появится автоматически.



### 5 Корректный учет всех данных о пчелах

После редактирования конструкторов Queen и Worker требуется внести изменения и в способ их вызова. В конструкторах появился новый параметр `Weight`, который следует указать:

- ★ Worker Bee #1: 175mg; Worker Bee #2: 114mg; Worker Bee #3: 149mg; Worker Bee #4: 155mg; Queen Bee: 275mg

Это последнее изменение, которое требовалось внести в форму!



## упражнение Решение

Перед вами класс Bee, в котором происходит основной подсчет потребления меда. Затем эти данные используются классами Queen и Worker.

```
class Bee {
    public Bee(double weight) {
        this.weight = weight;
    }

    public virtual int ShiftsLeft {
        get { return 0; }
    }

    private double weight;

    public virtual double GetHoneyConsumption() {
        double consumption;
        if (ShiftsLeft == 0)
            consumption = 7.5;
        else
            consumption = 9 + ShiftsLeft;
        if (weight > 150)
            consumption *= 1.35;
        return consumption;
    }
}
```

Конструктор класса Bee задает поле Weight и метод HoneyConsumption(), рассчитывающий потребление меда рабочей пчелой.

Пчелы с весом более 150 мг, потребляют на 35% больше меда.

```
public Form1()
{
    InitializeComponent();
}
```

В форме меняется только конструктор.

```
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);
}
```

В конструкторы класса Worker добавляется информация о весе рабочей пчелы.

**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!**

```
class Worker : Bee
    public Worker(string[] jobsICanDo, int weight)
        : base(weight) {
            this.jobsICanDo = jobsICanDo;
    }
```

```
public override int ShiftsLeft {
    // ... the rest of the class is the same ...
```

Сначала мы делаем класс Queen производным от класса Bee.

```
class Queen : Bee {
    public Queen(Worker[] workers)
        : base(275) {
            this.workers = workers;
    }
```

```
public string WorkTheNextShift()
{
    double totalConsumption = 0;
    for (int i = 0; i < workers.Length; i++)
        totalConsumption += workers[i].GetHoneyConsumption();
    totalConsumption += GetHoneyConsumption();
```

// ... Здесь идет исходный код метода за исключением оператора return

```
    report += "Общее потребление меда: " + totalConsumption + " единиц";
    return report;
}
```

Остальная часть метода WorkTheNextShift() осталась без изменений. Вы только добавили в отчет данные о потреблении меда.

```
public override double GetHoneyConsumption() {
    double consumption = 0;
    double largestWorkerConsumption = 0;
    int workersDoingJobs = 0;
    for (int i = 0; i < workers.Length; i++) {
        if (workers[i].GetHoneyConsumption() > largestWorkerConsumption)
            largestWorkerConsumption = workers[i].GetHoneyConsumption();
        if (workers[i].ShiftsLeft > 0)
            workersDoingJobs++;
    }
    consumption += largestWorkerConsumption;
    if (workersDoingJobs >= 3)
        consumption += 30;
    else
        consumption += 20;
    return consumption;
}
```

Этот цикл определяет самое самое высокое потребление меда среди рабочих пчел.

Класс Worker мы сделали наследником класса Bee и отредактировали его конструктор. Теперь он передает параметр Weight в конструктор базового класса и перекрывает свойство Bee.ShiftsLeft, добавляя ключевое слово override в объявление свойства.

Матка весит 275 мг, именно этот параметр передает конструктору данного класса конструктору класса Bee.

В Верхней части метода WorkTheNextShift() имеется цикл, вызывающий метод GetHoneyConsumption() для каждого рабочего. А затем вызывается метод GetHoneyConsumption() данного класса, чтобы вычислить общее потребление.

В этом классе перекрывается метод GetHoneyConsumption() класса Bee. Чтобы учесть потребление меда маткой, вы находитите рабочего с самым высоким потреблением и добавляете 20 или 30 (в зависимости от количества занятых пчел).

Если трудятся 3 пчелы и более, матке требуется 30 дополнительных единиц меда; в противном случае ей нужны всего 20 дополнительных единиц.

## Совершенствуем систему управления ульем при помощи наследования

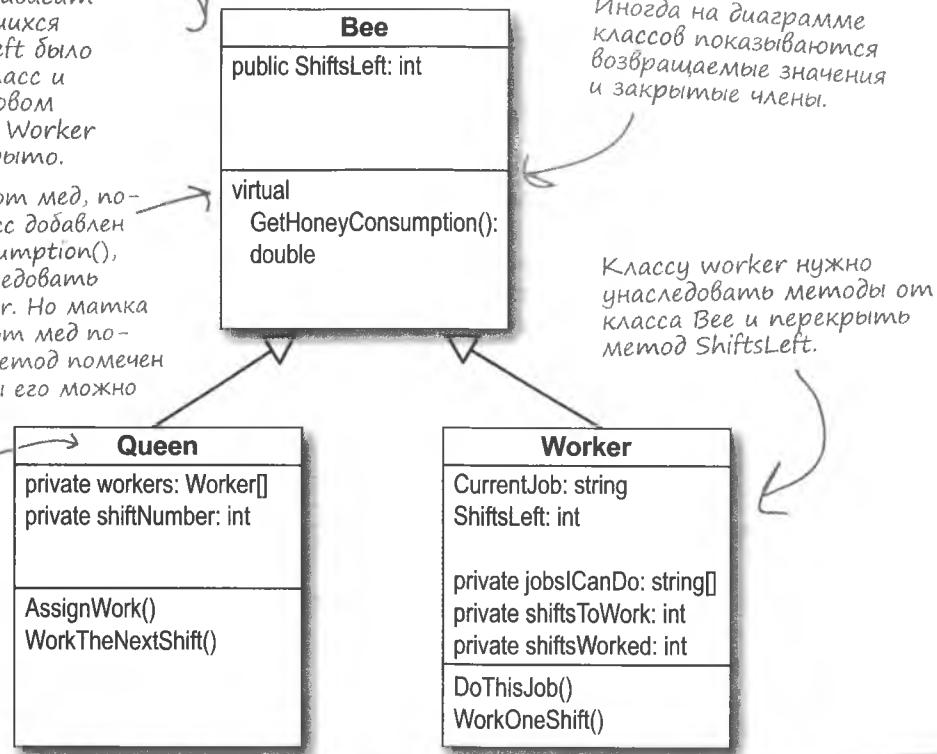
Основа нашей системы готова, теперь воспользуемся наследованием для учета количества меда, съедаемого пчелами. Каждая пчела потребляет мед, а больше всего его нужно матке. Поэтому создадим базовый класс Bee, от которого будут наследовать классы Queen и Worker.



Класс Bee содержит поведение, описывающее потребление меда. Так как оно зависит от количества оставшихся смен, свойство ShiftsLeft было перемещено в этот класс и помечено ключевым словом `virtual`, чтобы в классе Worker оно могло быть перекрыто.

Все пчелы потребляют мед, поэтому в базовый класс добавлен метод `GetHoneyConsumption()`, который могут наследовать классы queen и worker. Но матка и рабочие потребляют мед по-разному, поэтому метод помечен словом `virtual`, чтобы его можно было перекрывать.

В отчет для матери следует добавить данные о потреблении меда. Так как матери сама потребляет мед, она должна унаследовать от класса Bee метод `GetHoneyConsumption()` и перекрыть его.



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. ИСР скопирует эти файлы. Но не забудьте, что ИСР 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 поменять пространство имен вручную.



## 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 или функций.

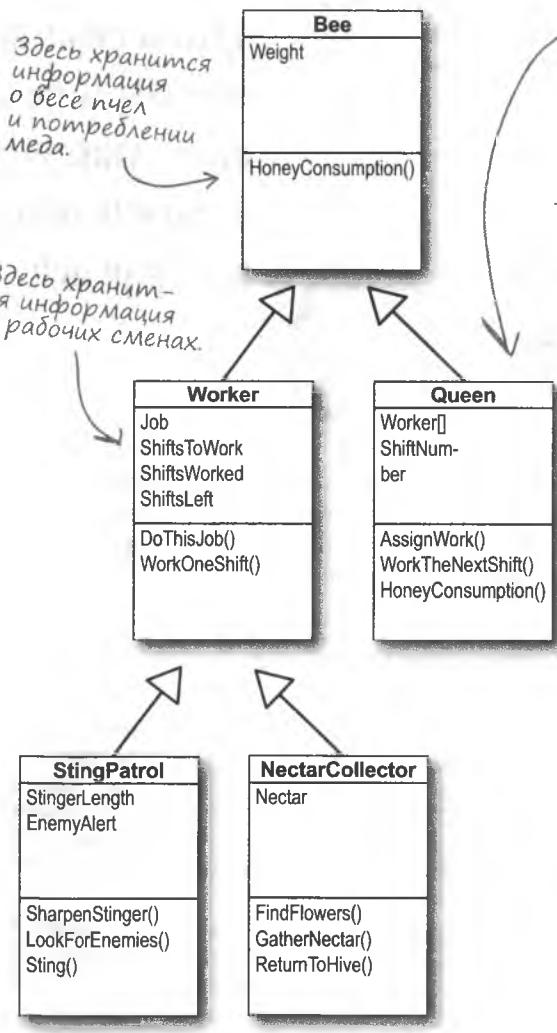
## Многое остается неизменным

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).



Помните, что Матке нужен дополнительный мед?  
Поэтому мы перекрыли метод HoneyConsumption().

Так будут выглядеть новые производные классы.

StingPatrol и NectarCollector наследуют от класса Worker.

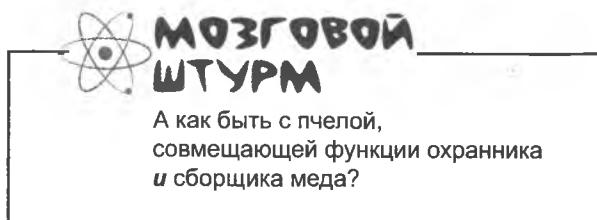
```

class StingPatrol : Worker
{
    int StingerLength;
    bool enemyAlert;
    public bool SharpenStinger (int Length)
    {...}
    public bool LookForEnemies(){...}
    public void Sting(string Enemy){...}
}
  
```

```

class NectarCollector : Worker
{
    int Nectar;
    public void FindFlowers (){...}
    public void GatherNectar(){...}
    public void ReturnToHive(){...}
}
  
```

Эти классы хранят информацию, относящуюся к отдельным заданиям.

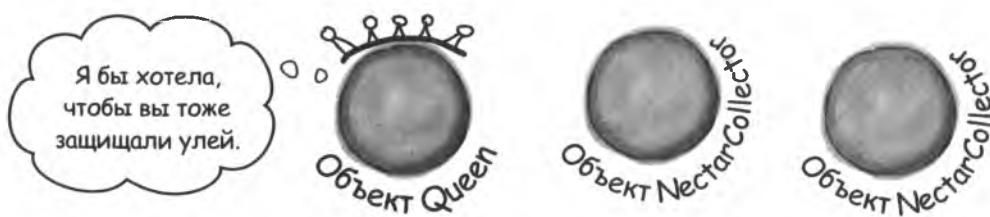


## Интерфейсы

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);
```

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.

Класс, реализующий интерфейс, должен включать в себя все методы и свойства, указанные в определении интерфейса.

Даже если матка добавит методы защищать 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 называнием `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.

## 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.

```

interface IStingPatrol
{
    int AlertLevel { get; }
    int StingerLength { get; set; }
    bool LookForEnemies();
    int SharpenStinger(int length);
}

Любому классу, реализующему
этот интерфейс, потребуется
написать метод
SharpenStinger()
с параметром типа
int.
}

interface INectarCollector
{
    void FindFlowers();
    void GatherNectar();
    void ReturnToHive();
}

```

**Интерфейс объявляется:**

**Интерфейсы не хранят данные, они не имеют полей... но они могут иметь свойства.**

**Любому классу, реализующему** **этот интерфейс, потребуется** **написать метод** **SharpenStinger()** **с параметром типа** **int.**

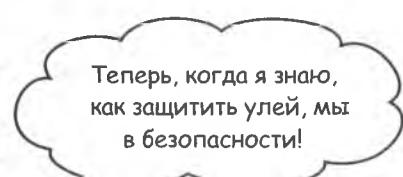
**В интерфейсе достаточно упомянуть имена методов, их код писать не требуется.** Код написан в классе, реализующем данный метод.

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 **данний** 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.



Интерфейсам стоит давать имена, начинающиеся с прописной буквы I. Немного правил, обязывающих вас так поступать, но это делает код более простым. Чтобы убедиться, насколько это облегчает жизнь, установите курсор на пустую строку внутри любого метода и наберите I — IntelliSense сразу же покажет вам список доступных интерфейсов .NET.

Любой класс, реализующий этот метод, должен иметь эти методы и свойства, иначе программа не будет компилироваться.

В интерфейсе достаточно упомянуть имена методов, их код писать не требуется. Код написан в классе, реализующем данный метод.

**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 { return alertLevel; }
        set {
            stingerLength = value;
        }
    }
    public int StingerLength {
        get { return stingerLength; }
        set {
            stingerLength = value;
        }
    }
    public bool LookForEnemies() {...}
    public int SharpenStinger(int length)
    {...}
    public void FindFlowers() {...}
    public void GatherNectar() {...}
    public void ReturnToHive() {...}
}
```

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.

Интерфейсы перечисляются через запятую.

Если врагов нет, исчезает прямое жало, вспомогательное поле StingerLength меняет свое значение.

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();
```

### 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`

В файл введите код интерфейса `IStingPatrol`, приведенный пару страниц назад. Программа при этом будет компилироваться.

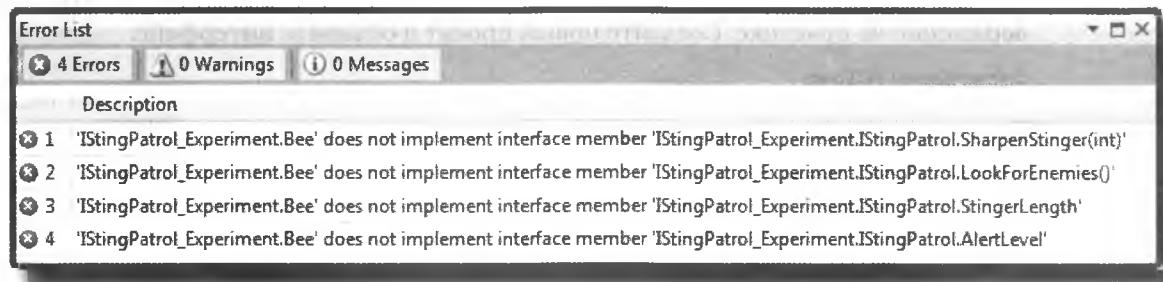
### 2 Добавьте к проекту класс `Bee`

Но пока не добавляйте ни свойств, ни методов. Заставьте этот класс реализовывать интерфейс `IStingPatrol`:

```
class Bee : IStingPatrol
{
}
```

### 3 Попытайтесь скомпилировать программу

Выберите команду Rebuild в меню Build. Компилятор не запустится:



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`. Пока они не должны выполнять никаких функций, они должны просто компилироваться. Добавьте метод чтения для переменной `AlertLevel` типа `int` и методы чтения и записи для переменной `StingerLength`. После этого программа снова начнет компилироваться!

## Учимся работать с интерфейсами

Использовать интерфейсы легко. Вы поймете это, немного по-практиковавшись. Начните с создания проекта Windows Forms Application. Перетащите на форму кнопку и приступим!



- 1 Это класс TallGuy (Высокий парень) и код для кнопки, нажатие которой создает объект и вызывает его метод TalkAboutYourself() (Рассказ о себе). Пока ничего нового:

```
class TallGuy {  
    public string Name;  
    public int Height;  
  
    public void TalkAboutYourself() {  
        MessageBox.Show("Меня зовут " + Name + " я ростом "  
            + Height + " см.");  
    }  
}  
  
private void button1_Click(object sender, EventArgs e) {  
    TallGuy tallGuy = new TallGuy() { Height = 190, Name = "Джимми" };  
    tallGuy.TalkAboutYourself();  
}
```

- 3 Создадим интерфейс IClown.

Вы уже знаете, что все элементы интерфейса должны быть открытыми. Проверим это утверждение на практике. Создайте новый проект и объявите интерфейс:

```
interface IClown
```

Теперь попробуйте объявить внутри него закрытый метод:

```
private void Honk();
```

Выбрав команду Build>>Build Solution, вы увидите сообщение:

Модификатор public  
внутри интерфей-  
са писать не нужно,  
доступ ко всем его  
методам и свойствам  
имеется по умолчанию.

1 The modifier 'private' is not valid for this item

Удалите модификатор private, сообщение об ошибке исчезнет.

- 3 Попробуйте создать интерфейс IClown самостоятельно и заставьте класс TallGuy его реализовывать. Начните с создания класса IClown.cs.

Интерфейс IClown должен иметь не возвращающий значений и не имеющий параметров метод Honk (Гудок) и предназначено только для чтения свойство FunnyThingIHave (Смотри, что у меня есть) типа string с методом чтения, но без 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 такой 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:

'TallGuy' does not implement interface member 'IClown.Honk()'

Объявив, что класс TallGuy реализует интерфейс IClown, вы подбирали 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 предназначеннное только для чтения свойство FunnyThingIHave с методом чтения, который всегда возвращает строку большие ботинки. И добавьте метод Honk(), вызывающий окно с надписью Би-би!

Вот, как это выглядит:

```
public string FunnyThingIHave {
    get { return "большие ботинки"; }
}

public void Honk() {
    MessageBox.Show("Би-би!");
}
```

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

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.

Это не работает...

```
IStingPatrol dennis = new IStingPatrol();
```

➊ 1 Cannot create an instance of the abstract class or interface

Попытка создать экземпляр интерфейса приведет к ошибкам компиляции.

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`.

Объект может выполнять много функций, но используя интерфейсную ссылку, вы получаете доступ только к методам, упомянутым в интерфейсе.



## 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!

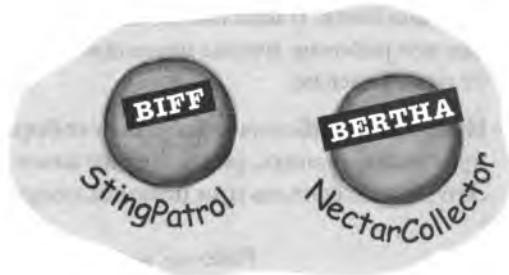
### 1 Создадим пару пчел

Вы уже не раз это делали.

```
StingPatrol biff = new StingPatrol();
NectarCollector bertha = new NectarCollector();
```



Пусть класс `StingPatrol` реализует интерфейс `IStingPatrol`, а класс `NectarCollector` — интерфейс `INectarCollector`.



### 2 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.



### 3 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`.



### 4 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`.

```

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)
    {
        Эта запись означает, что если
        данная пчела реализует интерфейс
        INectarCollector... нужно выполнить
        приведенный ниже код.
        bees[i].DoThisJob("Nectar Collector", 3)
    }
}

```

Рабочие представлены в виде массива Workers. Оператор `is` позволяет определить тип пчелы.

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.

Теперь, когда мы знаем, что эта пчела — сборщик нектара, ее можно отправить за нектаром.



### МОЗГОВОЙ ШТУРМ

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 обоих видов?

### 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 ли метод записи открытым или закрытым.)

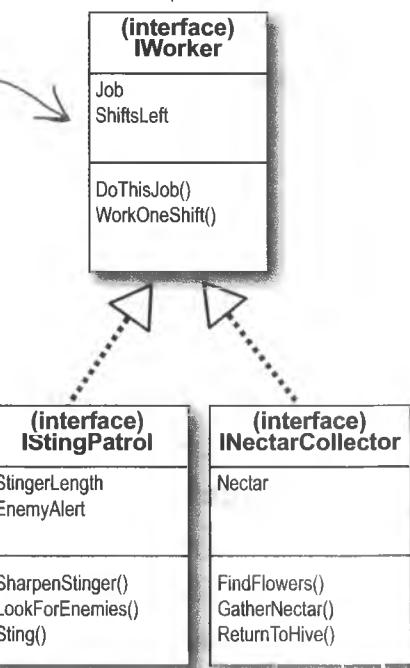
## Интерфейсы и наследование

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.



## Класс реализует все методы и свойства

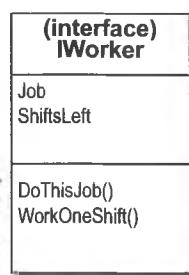
Класс, реализующий интерфейс, должен включить в себя все свойства и методы этого интерфейса. В ситуации, когда один интерфейс наследует от другого, все *их* свойства и методы также должны быть реализованы.

```
interface IStingPatrol : IWorker
{
    int AlertLevel { get; }
    int StingerLength { get; set; }
    bool LookForEnemies();
    int SharpenStinger(int length);
}
```

Класс, реализующий интерфейс IStingPatrol, должен реализовать не только эти методы...

...но и методы интерфейса IWorker, от которого проходит наследование.

Уже знакомый вам интерфейс IStingPatrol теперь наследует от интерфейса IWorker. Изменение кажется совсем небольшим, но для всех классов, реализующих IStingPatrol, ситуация коренным образом поменялась.



## RoboBee 4000 функционирует без мёда

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() { ... }
}
```

Базовый класс Robot, дающий новой пчеле возможность «питьаться» бензином.

```
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() {...}
}
```

Если класс RoboBee не будет реализовывать все упомянутое в интерфейсе IWorker, компиляция кода станет невозможной.

Остальные классы нашего приложения не «увидят» функциональной разницы между пчелой-роботом и обычной пчелой. Оба эти класса реализуют интерфейс IWorker, 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.

<b>RoboBee</b>
ShiftsToWork
ShiftsWorked
ShiftsLeft
Job
DoThisJob()



Класс RoboBee наследует от класса Robot и реализует интерфейс IWorker. В итоге мы получили робота, который может выполнять работу обычной пчелы.

Класс RoboBee реализует все методы интерфейса IWorker.

→

**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();

Мы перебираем пчел...
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 не можем вызывать для пчел методы интерфейса `INectarCollector`. Ведь пчелы принадлежат типу `IWorker` и ничего не знают о методах `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.

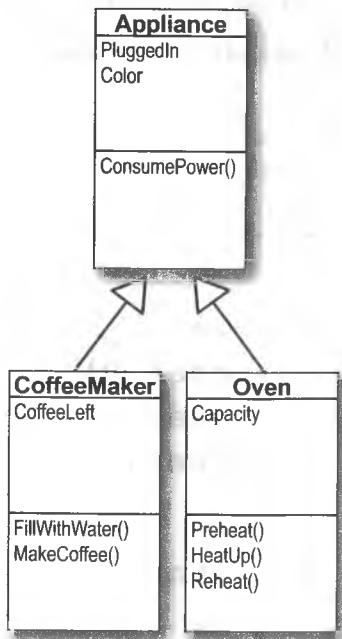
```
public void MonitorPower(Appliance appliance) {
    // код добавления данных в домашнюю
    // базу потребления энергии
}
```

Этот код отслеживает, сколько электроэнергии требуется для работы кофеварки.

Здесь должен быть метод, отслеживающий потребление электроэнергии в доме.

```
CoffeeMaker misterCoffee = new CoffeeMaker();
MonitorPower(misterCoffee);
```

Метод MonitorPower() требует ссылки на объект Appliance, но ему можно передать ссылку misterCoffee, так как класс CoffeeMaker является производным от класса Appliance.



Такое поведение вы уже наблюдали в предыдущей главе, когда передавали методу, ожидающему ссылку 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();
```

1. (Bees[i] is INectarCollector)

Метод  
NectarStinger()  
реализует  
интерфейс  
IStingPatrol.

0 и 6

2. (Bees[i] is IStingPatrol)

0, 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 уживаются 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();
```

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:

 'Appliance' does not contain a definition for 'MakeCoffee'

powerConsumer – это ссылка класса `Appliance` на объект `CoffeeMaker`.



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!
```

### 3 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.

Ссылка `javaJoe` ука-  
зывает на тот же  
объект, что и ссыл-  
ка `powerConsumer`.  
Но она относится  
к классу `CoffeeMaker`  
и дает возможность  
вызывать метод  
`MakeCoffee()`.

```
if (powerConsumer is CoffeeMaker) {  
    CoffeeMaker javaJoe = powerConsumer as CoffeeMaker;  
    javaJoe.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`. По-  
этому при попытке осуще-  
ствовать нисходящее приведе-  
ние ссылка `foodWarmer` даст  
`null`. Вот что произойдет при  
попытке использовать пу-  
стую ссылку...



## 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();
```

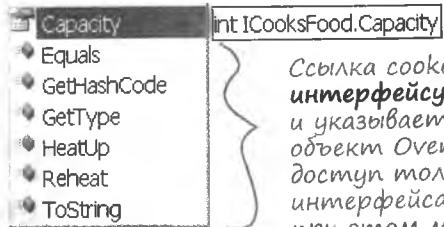
```
misterToasty.
```

Сразу после ввода точки функции `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`.

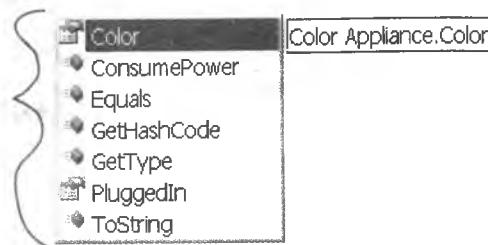
```
ICooksFood cooker;
if (misterToasty is ICooksFood)
    cooker = misterToasty as ICooksFood;
cooker.
```



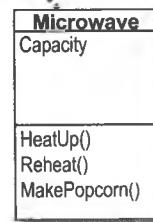
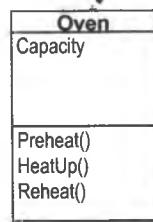
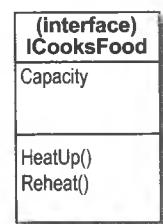
Ссылка `cooker` принадлежит **интерфейсу `ICooksFood`** и указывает на тот же объект `Oven`. Она дает доступ только к членам интерфейса `ICooksFood`, но при этом может указывать на объект `Microwave`.

```
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`.



Любой класс, реализующий интерфейс `ICooksFood`, относится к приборам, которые могут разогревать `HeatUp()` еду.



**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 операторов. Оператор в виде двоеточия, реализующий интерфейс, не имеет отношения к оператору, используемому 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 обиться 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 аттракциона катания вы хотите создать массив животных, которые могут тащить телегу. 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 телеги. 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 ИСР 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.**

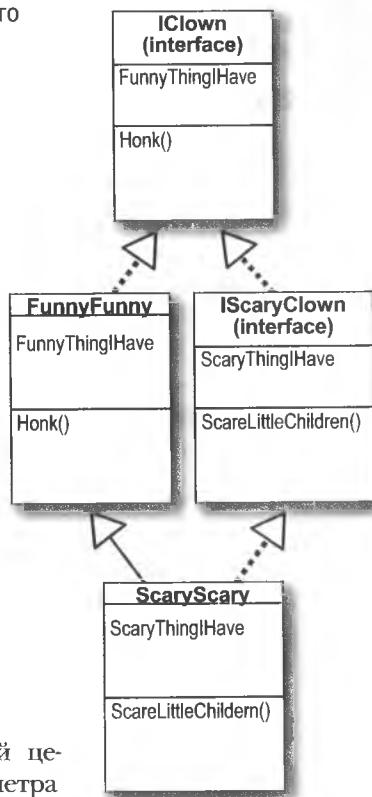


## Упражнение

Расширьте интерфейс `IClown` при помощи реализующих его классов.

Начните с интерфейса `IClown` из Упражнения! на с. 290:

```
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 «Би-би! 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 для кнопки. Вам нужно найти ошибки и заставить его работать.

```
private void button1_Click(object sender, EventArgs e) {
    ScaryScary fingersTheClown = new ScaryScary("большие ботинки", 14);
    FunnyFunny someFunnyClown = fingersTheClown;
    IScaryClown someOtherScaryclown = someFunnyClown;
    someOtherScaryclown.Honk();
}
```

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 "Би-би! У меня есть " + funnyThingIHave; }
    }
    public void Honk() {
        MessageBox.Show(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() {
        MessageBox.Show("Ага! Попался!");
    }
}

private void button1_Click(object sender, EventArgs e) {
    ScaryScary fingersTheClown = new ScaryScary("большие ботинки", 14);
    FunnyFunny someFunnyClown = fingersTheClown;
    IScaryClown someOtherScaryClown = someFunnyClown as ScaryScary;
    someOtherScaryClown.Honk();
}

```

**Метод Honk()**  
использует  
этот метод  
записи для  
отображения  
сообщения, что  
избавляет вас  
от дублирую-  
щегося кода.

Можно еще раз  
реализовать  
этот метод и  
свойство интер-  
фейса `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 принад-  
лежит к производному от `FunnyFunny`  
классу. Но ссылка `IScaryClown` не может  
указывать на произвольного клоуна, так  
как клоун может оказаться нестабильным.  
Поэтому вы используете оператор `as`.

Ссылку `someOtherScaryClown` можно использовать для  
вызыва 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 нужно.

Отсутствие модификатора доступа при объявлении члена класса означает, что будет использован вариантом `Private`.

### ★ `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.



### ★ `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.

Отсутствие модификатора доступа при объявлении класса или интерфейса означает, что будет использован вариантом `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 к некоторым классов на `internal` и посмотрите, что получится.

### ★ `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.

Ключевое слово `Sealed` не относится к модификаторам доступа.

### ★ `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.

## 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 "Би-би! У меня есть " + funnyThingIHave;  
    }  
    }  
    public void Honk() {  
        MessageBox.Show(this.FunnyThingIHave);  
    }  
}
```

Посмотрите, как модификатор `protected` влиял 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 ScareLittleChildren() осуществляет доступ к вспомогательному полю funnyThingIHave. Такое поведение связано с модификатором protected. При модификаторе private компиляция кода стала бы невозможной.

```
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() {
        MessageBox.Show("Ты не можешь забрать "
                        + base.funnyThingIHave);
    }
}
```

Ключевое слово base заставляет использовать значение из базового класса. Но в данном случае можно воспользоваться также ключевым словом this. Вы понимаете, почему?

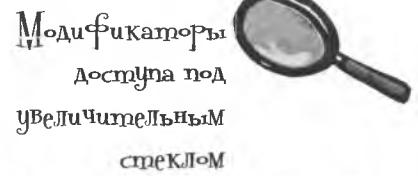
4 Эта кнопка создает экземпляры FunnyFunny и ScaryScary. Обратите внимание, как с помощью оператора as осуществляется нисходящее приведение someFunnyClown к ссылке IScaryClown.

```
private void button1_Click(object sender, EventArgs e) {
    ScaryScary fingersTheClown = new ScaryScary("большие ботинки", 14);
    FunnyFunny someFunnyClown = fingersTheClown;
    IScaryClown someOtherScaryClown = someFunnyClown as ScaryScary;
    someOtherScaryClown.Honk();
}
```

Так как запускающийся щелчком на кнопке обработчик событий не является частью классов FunnyFunny и ScaryScary, у него нет доступа к помеченному ключевым словом protected полю funnyThingIHave.

Программа искусственно удлинена, чтобы показать возможность восходящего приведения от ScaryScary к FunnyFunny, а затем — нисходящего приведения к IScaryClown. Вместо трех строчек можно написать всего одну. Попробуйте сделать это самостоятельно.

Кнопка не принадлежит ни одному из классов, поэтому операторы имеют доступ только к открытым членам объектов FunnyFunny или ScaryScary.



Модификаторы  
доступа под  
увеличительным  
стеклом

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.

Наличие при переменной funnyThingIHave модификатора private привело бы к сообщению об ошибке. Ключевое слово protected делает ее видимой из классов, производных от класса FunnyFunny.

**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.

## часто Задаваемые Вопросы

**В:** Зачем нужны свойства, если есть поля?

**О:** Интерфейс определяет способ, которым класс выполняет определенную 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 свой-  
ствах 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.

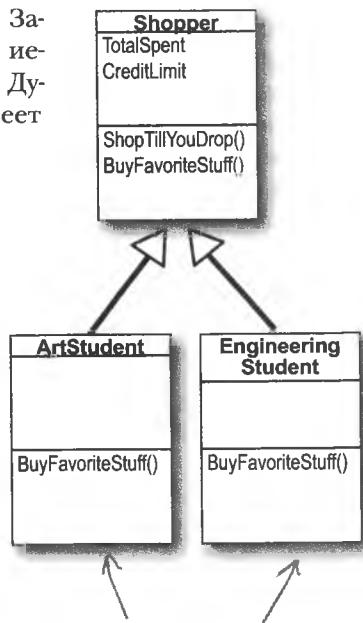
```
class Shopper {
    public void ShopTillYouDrop()
        while (TotalSpent < CreditLimit)
            BuyFavoriteStuff();
    }
    public virtual void BuyFavoriteStuff () {
        // Реализация здесь неуместна - мы не знаем,
        // что любит покупать наш студент!
    }
}
```

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();
    }
}
```



Классы ArtStudent и EngineeringStudent перекрывают метод BuyFavoriteStuff(). Они покупают разные книги.

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.



### 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!



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 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.



### 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.



Cannot create an instance of the abstract class or interface 'MyClass'

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.



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.

Класс PlanetMission клуб астрофизиков использует для отправки ракет к различным планетам.

Вот еще один пример...

Один полет совершается на Венеру, другой — на Марс.

```
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.";
    }
}

private void button1_Click(object s, EventArgs e) {
    Mars mars = new Mars();
    MessageBox.Show(mars.FuelNeeded());
}

private void button2_Click(object s, EventArgs e) {
    Venus venus = new Venus();
    MessageBox.Show(venus.FuelNeeded());
}

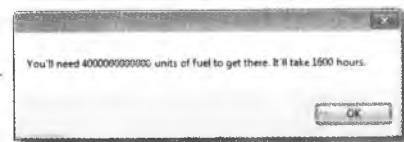
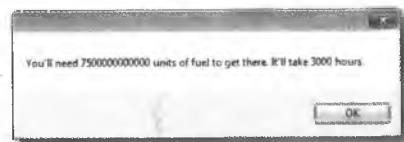
private void button3_Click(object s, EventArgs e) {
    PlanetMission planet = new PlanetMission();
    MessageBox.Show(planet.FuelNeeded());
}
```

Присваивать этим полям значения в базовом классе бесполезно, потому что мы не знаем, какая ракета куда полетит.

```
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, указанных в базовом классе Planet. Но поля не могут получить значения от экземпляра Planet. Что произойдет, если их попытается использовать метод FuelNeeded()?



Перед тем как перевернуть страницу, подумайте, что произойдет после щелчка на третьей кнопке...

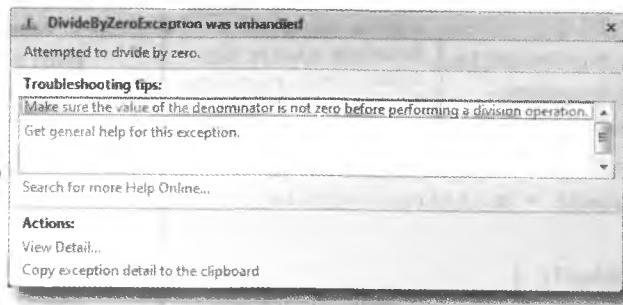
*абстрактные классы избегают порядка*

## Как уже было сказано, недопустимо создавать экземпляры некоторых классов

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());
}
```

В методе FuelNeeded() происходит деление на параметр RocketSpeedMPH, который равен нулю.



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...

### 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.

Теперь компиляция программы невозможна, пока мы не уберем строку, создающую экземпляр PlanetMission.

```
abstract class PlanetMission {
    public long RocketFuelPerMile;
    public long RocketSpeedMPH;
    public int MilesToPlanet;

    public long UnitsOfFuelNeeded() {
        return MilesToPlanet * RocketFuelPerMile;
    }

    // здесь определяется остальная часть класса
}
```

**МОЗГОВОЙ ШТУРМ**

Вернитесь к программе расчета стоимости мероприятий на с. 270-272 и подумайте над проблемой инкапсуляции. Придумайте ее решение при помощи абстрактных классов.

## 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. К счастью, достаточно напечатать «public override», и как только вы нажмете пробел, появится список всех доступных для перекрытия методов. Выберите вариант SetMissionInfo() и напишите:

```
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.

**Интерфейс содержит только абстрактные методы, поэтому ключевое слово `abstract` применяется только когда речь идет об абстрактных классах. Только такие классы могут иметь в своем составе абстрактные методы.**

Жизнь абстрактного метода ужасна. Ведь это жизнь без тела.



Попытавшись построить программу, вы получите сообщение об отсутствии реализации унаследованного абстрактного члена:



'VenusMission' does not implement inherited abstract member 'PlanetMission.SetMissionInfo(long, int, int)'

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.

## Возьми в руку карандаш

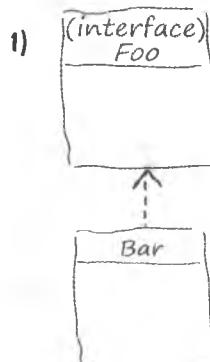


Продемонстрируйте свое искусство. Слева вы видите набор классов и объявлений интерфейсов. Нарисуйте справа соответствующие диаграммы классов, как показано в примере номер один. Не забудьте, что пунктирная линия показывает наследование от интерфейса, в то время как сплошная — наследование от класса.

Дано:

1) interface Foo { }  
class Bar : Foo { }

Диаграмма



2) interface Vinn { }  
abstract class Vout : Vinn { }

2)

3) abstract class Muffie : Whuffie { }  
class Fluffie : Muffie { }  
interface Whuffie { }

3)

4)  
class Zoop { }  
class Boop : Zoop { }  
class Goop : Boop { }

4)

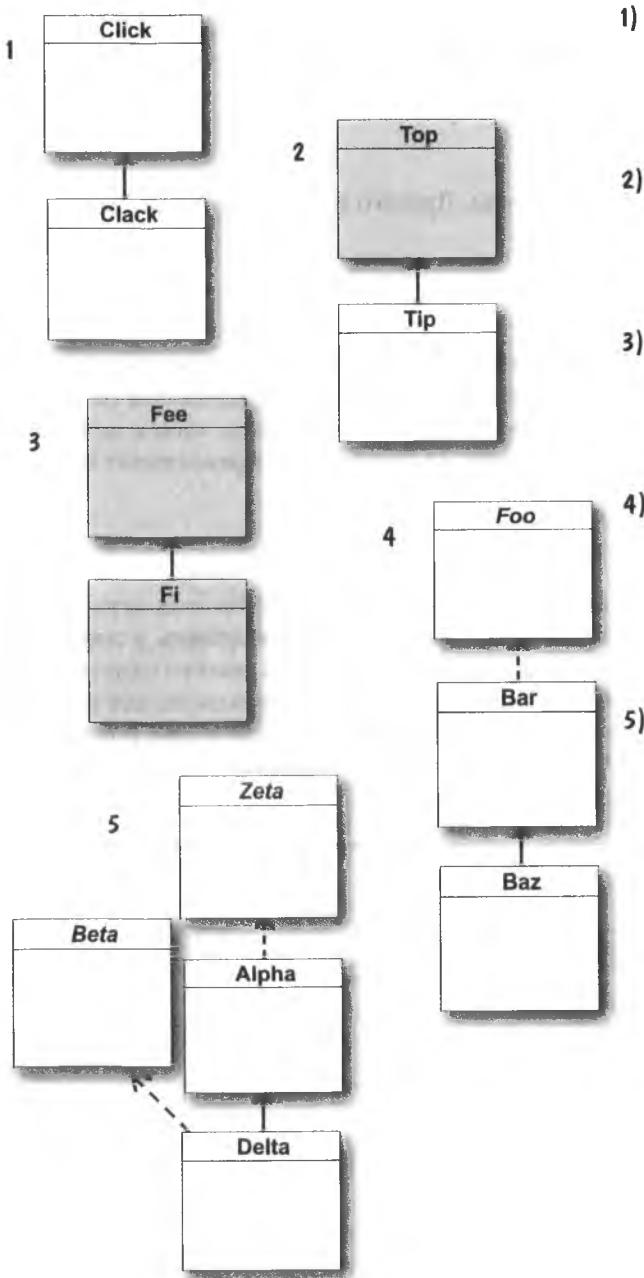
5)  
class Gamma : Delta, Epsilon { }  
interface Epsilon { }  
interface Beta { }  
class Alpha : Gamma, Beta { }  
class Delta { }

5)

А в этом задании слева приведены диаграммы классов, вам же нужно превратить их в объявления, как показано в первом примере.

**Дано:**

### Объявление



1) public class Click {}  
public class Clack : Click {}

2)

3)

4)

5)

### Обозначения

наследует



реализует



класс



интерфейс



абстрактный



## Беседа у камина



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.

Как ты вообще мог подумать, что можешь быть важнее меня? Ты даже не в состоянии наследовать, как полагается, тебя можно только реализовать.

Превосходит? Да ты сошел с ума. Я намного более гибок. Я могу иметь как обычные, так и абстрактные методы. И даже виртуальные методы, если захочу. Да, я не создаю экземпляров, но этого не можешь и ты. Зато моя функциональность ничем не хуже, чем у обычного класса.

### Интерфейс

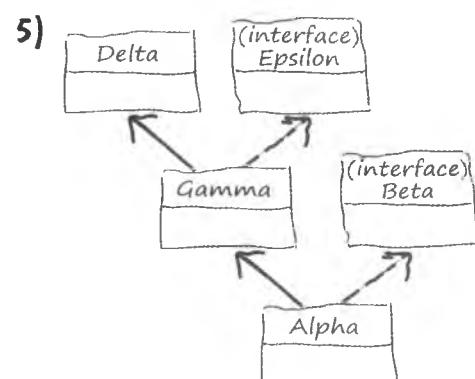
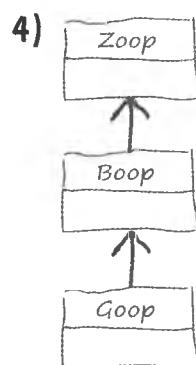
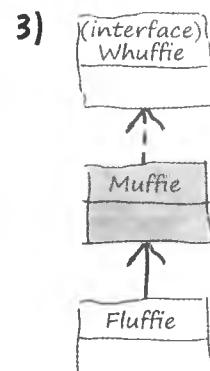
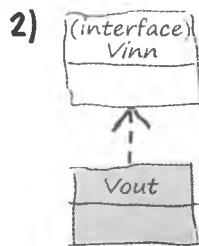
Прекрасно. Другого я от тебя и не ожидал!

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!

### Возьми в руку карандаш



### Решение



### Готовые диаграммы

## Абстрактный класс

Кажется, ты несколько переоцениваешь свою роль.

Такую чепуху можно услышать только от интерфейса. Нет ничего важнее кода! Именно он заставляет программу работать.

Да ну? По моим наблюдениям программистов очень даже волнует содержимое свойств и методов.

Да, конечно... расскажи кодеру, что он не может писать код.

## 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.

Девять из десяти, что программисту нужны определенные свойства и методы, но его при этом не волнует, как именно они реализуются.

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!

Не имеет значения!

2) abstract class Top {}  
class Tip : Top {}

4) interface Foo {}  
class Bar : Foo {}  
class Baz : Bar {}

3) abstract class Fee {}  
abstract class Fi : Fee {}

5) interface Zeta {}  
class Alpha : Zeta {}  
interface Beta {}  
class Delta : Alpha, Beta {}

Delta на-  
следует от  
Alpha и реа-  
лизует Beta.

## Корректные объявления



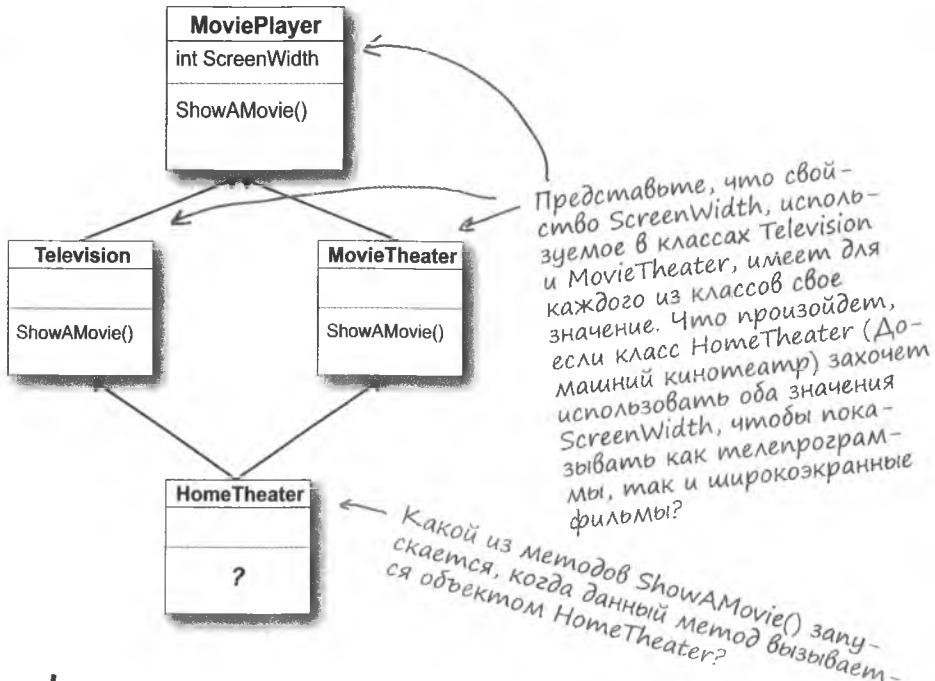
Досадно, что я не могу наследовать больше, чем от одного класса, и поэтому должна пользоваться интерфейсами. Это большой недостаток C#, не так ли?

### 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...

## Смертельный ромб!

И *Television* (Телевидение) и *MovieTheater* (Кинотеатр) наследуют от класса *MoviePlayer* (Показ кино), и оба эти класса перекрывают метод *ShowAMovie()* (Показ фильма). Кроме того, они наследуют свойство *ScreenWidth* (Ширина экрана).



### Избегайте неопределенностей!

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.



## 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. 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") { }
}

```

```

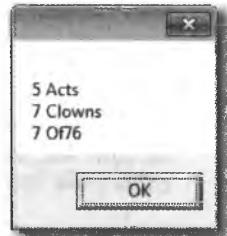
class ..... : .....
{
    public Acts() : base("Acts") { }
    public override ..... {
        return 5;
    }
}

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";
    }
    MessageBox.Show(result);
}

```

Точка входа находится здесь.



Результат

Каждый фрагмент кода можно использовать несколько раз.

Acts( );	;	i( )	class	5	class	Acts
Nose( );	;	i(x)	abstract	7	class	Nose
Of76( );	;	i[x]	interface	7	public class	Of76
Clowns( );	;		int Ear()	public		Clowns
Picasso( );	;		this			Picasso
Of76 [ ] i = new Nose[3];			this.	get	i.Ear(x)	
Of76 [ 3 ] i;			face	set	i[x].Ear()	
Nose [ ] i = new Nose( );			this.face	return	i[x].Ear(	
Nose [ ] i = new Nose[3];					i[x].Face	



Думаю, что теперь  
я хорошо умею  
управлять объектами!

Идея объединить данные  
и код в классы на мо-  
мент своего появления  
была революционной,  
но теперь это вполне  
обычная практика про-  
граммирования.

### Вы объектно-ориентированный программист.

То, чем вы занимаетесь, называется **объектно-ориентированное программирование** (OOP). 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 OOP, 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.

Классы и интерфейсы  
могут наследовать  
свойства друг друга.

Так называется создание  
объектов, которые от-  
слеживают свое состоя-  
ние при помощи закрытых  
полей, а для других классов  
предоставляют открытые  
свойства и методы. Та-  
ким образом другие классы  
видят только ту часть  
данных, которую им нужно  
видеть.

#### Наследование

#### Инкапсуляция

#### Абстракция

#### Полиморфизм

Создание модели, начинающейся  
с более общих — абстракт-  
ных — классов, от которых  
наследуют более подробные  
классы.

Это слово буквально  
означает «множе-  
ство форм». Мож-  
ете ли вы пред-  
ставить ситуацию,  
когда объект прини-  
мает разные фор-  
мы?

## Различные формы объекта

Помните, как вы использовали *Пересмешника* вместо *Животного* и *выдержанного Вермонта* вместо *Сыра*? 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 реализаций.

### 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, ожидающих 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`.

\*\*\*

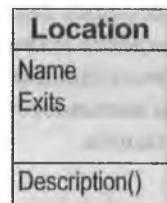


**Давайте построим дом!** В модели дома классы будут представлять комнаты и прочие помещения, а интерфейс пусть соответствует двери.

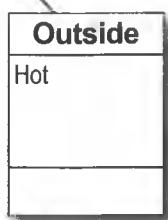
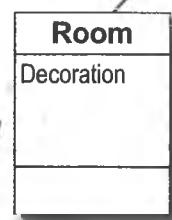
1

### 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 }.



Класс Location абстрактный. Именно поэтому на диаграмме он серый.



2

### Нарисуем план дома

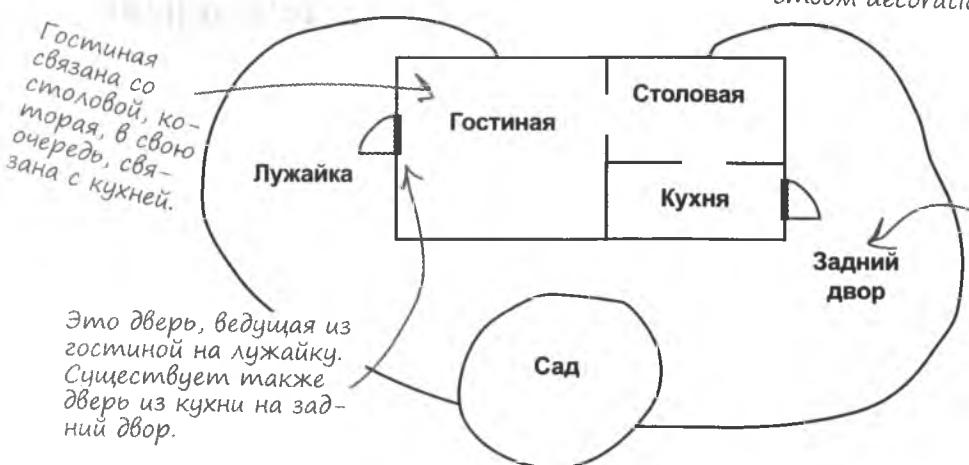
В доме три комнаты, лужайка, задний двор и сад. Внешних дверей две: одна ведет из гостиной на лужайку, а вторая – из кухни на задний двор.

Интерьер помещений определяется предназначенным только для чтения свойством decoration.

Определить, жарко ли снаружи, поможет предназначенное только для чтения булево свойство Hot.

Переместиться с лужайки на задний двор можно через сад.

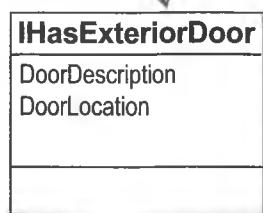
Все комнаты имеют двери, но только некоторые двери ведут наружу.



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:

```
abstract class Location {
    public Location(string name) {
        this.name = name;
    }
    private string name;
    public string Name {
        get { return name; }
    }
    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;
        }
    }
}
```

**Метод Description**  
виртуальный, его  
нужно пере-  
крыть.

**Свойство Description**  
возвращает строку  
с описанием комнаты  
и всех прилегающих  
к ней помещений (их  
список содержится  
в поле Exits[]). Так как  
в производных клас-  
сах описание будет  
меняться, свойство  
требуется перекрыть.

Конструктор задает значение  
 поля name, которое является  
предназначенным только для  
чтения полем свойства Name.

Открытое поле Exits являет-  
ся массивом ссылок Location,  
который отслеживает, какие  
помещения связаны с тем, в  
котором находитесь вы.

Класс Room  
перекрыва-  
ет и расши-  
ряет метод  
Description,  
добавляя к  
нему инте-  
рьер. К методу  
Outside он до-  
бавит темпе-  
ратуру.

Помните, что от  
класса Location можно  
наследовать, объявлять  
ссылочные перемен-  
ные типа Location, но  
нельзя создавать экзем-  
пляры.

Дополнительную информацию вы  
получите на следующей странице.

## 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:

```
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 упражне-  
ние... но мы обеща-  
ем, что вы получите  
удовольствие! И обя-  
зательно запомните  
новый материал.

Переверните страницу и продолжим!



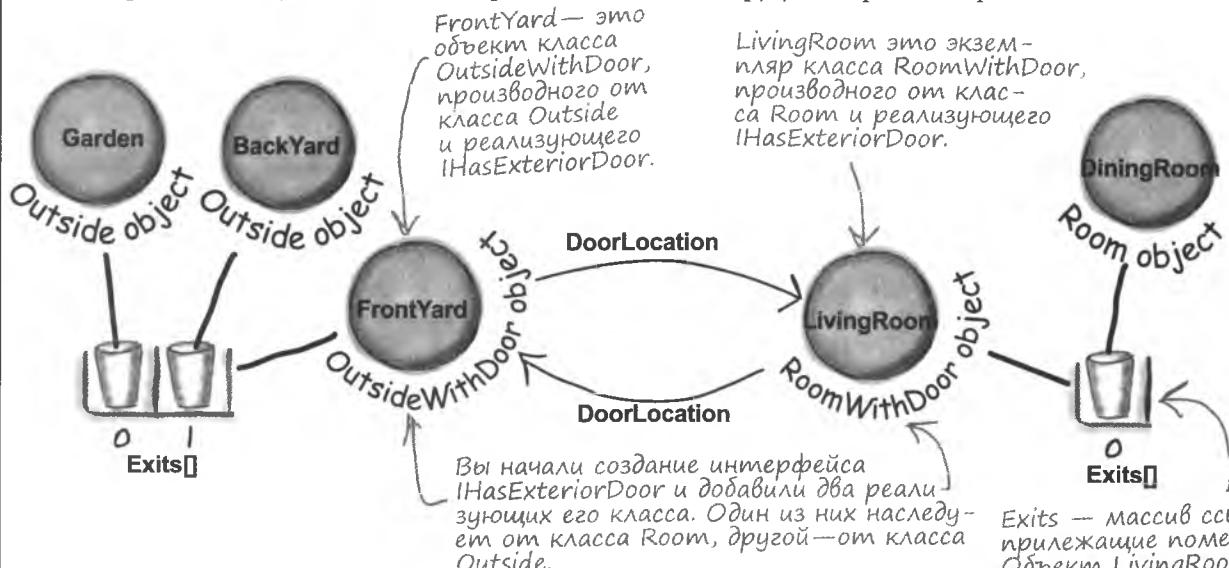
## Длинные упражнения

Пришло время создать объекты, представляющие различные части дома, и добавить форму для работы с ними.

6

### Как работают объекты

Рассмотрим архитектуру объектов frontYard и diningRoom. 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 предназначено 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.

Здесь нужны фигурные скобки.

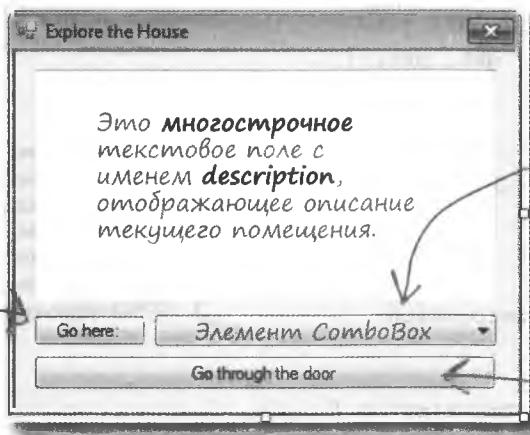
Exits (массив ссылок Location), создает массив из двух строк.

8

## Построение формы

Создадим простую форму для экскурсии по дому. Потребуется большое текстовое поле `description`, в котором будут появляться описания помещений. 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.

Кнопка `goHere`  
позволяет  
перейти в другое  
помещение.



Содержимое списка `ComboBox` задается здесь.

Элемент `ComboBox` содержит список выходов, поэтому присвоим ему имя `exits`. Выберите для свойства `DropDownStyle` вариант `DropDownList`.

Кнопка `goThroughTheDoor` появляется, если вы находитесь в комнате с дверью на улицу. Для изменения 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.

## Решение

## длинных

## упражнений

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)
    {
        this.doorDescription = doorDescription;
    }

    private string doorDescription;
    public string DoorDescription {
        get { return doorDescription; }
    }

    private Location doorLocation;
    public Location DoorLocation {
        get { return doorLocation; }
        set { doorLocation = value; }
    }
}
```

```

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;
    }

    private string doorDescription;
    public string DoorDescription {
        get { return doorDescription; }
    }

    private Location doorLocation;
    public Location DoorLocation {
        get { return doorLocation; }
        set { doorLocation = value; }
    }

    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; ←
        Форма отслеживает, в какой
        комнате вы находитесь
        в данный момент.

    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;
    }
}

При помощи этих ссылочных
переменных форма следит за
каждым помещением в доме.

Конструктор фор-
мы создает объект
и использует метод
MoveToANewLocation
для перехода в другое
помещение.

Создание объектов на-
чинается с создания
экземпляров и передачи
конструкторам этих
экземпляров нужной ин-
формации.

Здесь мы переда-
ем описание дверей
конструкторам
OutsideWithDoor.

Здесь заполняется мас-
сив Exits[] для каждого из
экземпляров. Данная про-
цедура возможна только
после создания всех экзем-
пляров!

Для объектов
IHasExteriorDoor нужно
указать положение
дверей.

```

```

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;
    } Делаем кнопку Go through the door невидимой, если текущее помещение не реализует IHasExteriorDoor.

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 null отображает первый пункт списка. Не забудьте присвоить свойству DropDownList значение DropDownStyle, чтобы пользователи не смогли добавлять в список свои значения.

Щелчок на кнопке Go here: перемещает в выбранное помещение.

Оператор as осуществляет неявное 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 прятаться 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!

→ Пак, начнем!



## 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.

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*.

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. Оказавшись 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:

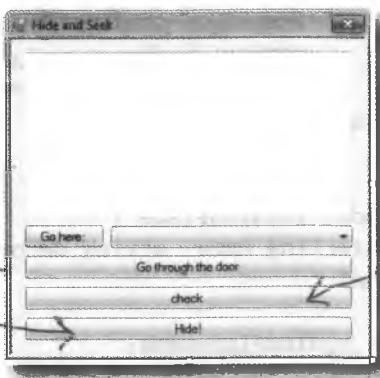
- ★ Добавьте лестницу с деревянными перилами, соединяющую гостиную с коридором второго этажа, где висит картина с собакой и стоит шкаф.
- ★ Верхний коридор ведет в три комнаты: главную спальню с большой кроватью, вторую спальню с маленькой кроватью и ванную с раковиной и туалетом. Прятаться можно как под кроватями, так и в душе.
- ★ Передний и задний дворы соединены проездом с гаражом, пригодным для укрытия. Прятаться можно и в сарае.

## 5 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 (Прячусь!). 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).

## 6 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 искаль!», 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 напел 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.

```

interface IHidingPlace {
    string HidingPlaceName { get; }
}

class RoomWithHidingPlace : Room, IHidingPlace {
    public RoomWithHidingPlace(string name, string decoration, string hidingPlaceName)
        : base(name, decoration)
    {
        this.hidingPlaceName = hidingPlaceName;
    }

    private string hidingPlaceName;
    public string HidingPlaceName {
        get { return hidingPlaceName; }
    }

    public override string Description {
        get {
            return base.Description + " Спрятаться можно " + hidingPlaceName + ".";
        }
    }
}

class RoomWithDoor : RoomWithHidingPlace, IHasExteriorDoor {
    public RoomWithDoor(string name, string decoration,
                        string hidingPlaceName, string doorDescription)
        : base(name, decoration, hidingPlaceName)
    {
        this.doorDescription = doorDescription;
    }

    private string doorDescription;
    public string DoorDescription {
        get { return doorDescription; }
    }

    private Location doorLocation;
    public Location DoorLocation {
        get { return doorLocation; }
        set { doorLocation = value; }
    }
}

```

*Новый интерфейс IHidingPlace содержит всего одно поле типа string с методом чтения, возвращающим название места, где можно спрятаться.*

*Класс RoomWithHidingPlace наследует от класса Room и реализует интерфейс IHidingPlace, добавляя свойство HidingPlaceName. Данное возможительное поле задается конструктором.*

*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.*

```

class OutsideWithHidingPlace : Outside, IHidingPlace {
    public OutsideWithHidingPlace(string name, bool hot, string hidingPlaceName)
        : base(name, hot)
    { this.hidingPlaceName = hidingPlaceName; }

    private string hidingPlaceName;
    public string HidingPlaceName {
        get { return hidingPlaceName; }
    }

    public override string Description {
        get {
            return base.Description + " Можно спрятаться " + hidingPlaceName + ".";
        }
    }
}

```

```

class Opponent {
    private Random random;
    private Location myLocation;
    public Opponent(Location startingLocation) {
        myLocation = startingLocation;
        random = new Random();
    }

    public void Move() {
        if (myLocation is IHasExteriorDoor) {
            IHasExteriorDoor LocationWithDoor =
                myLocation as IHasExteriorDoor;
            if (random.Next(2) == 1)
                myLocation = LocationWithDoor.DoorLocation;
        }
        bool hidden = false;
        while (!hidden) {
            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;
    }
}

```

Класс `OutsideWithHidingPlace` наследует от класса `Outside` и реализует метод `IHidingPlace` аналогично классу `RoomWithHidingPlace`.

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 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.

Конструктор Form1 создает объекты, определяет положение соперника и загружает 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;
}

```

```

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;

```

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!

## решение упражнения



### упражнение Решение

```
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;  
}
```

Это основной 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 check_Click(object sender, EventArgs e) {  
    Moves++;  
    if (opponent.Check(currentLocation))  
        ResetGame(true);  
    else  
        RedrawForm();  
}
```

Метод `ResetGame()` перезагружает игру. Он отображает финальное сообщение, а затем делает все кнопки, кроме кнопки «`Hide!`», невидимыми.

```
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);  
    }  
}
```

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 исходящего 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.

```
description.Text = "Я иду искать!";  
Application.DoEvents();  
System.Threading.Thread.Sleep(500);  
  
goHere.Visible = true;  
exits.Visible = true;  
MoveToANewLocation(livingRoom);  
}
```

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. 319

Вам требовалось взять фрагменты кода из бассейна и поместить их на пустые строки таким образом, чтобы получить показанный ниже результат.

Класс `Acts` вызывает конструктор базового для него класса `Picasso`. Он передает конструктору переменную `Acts`, которая сохраняется в свойстве `face`.

```
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 про-извольном месте класса! Код про-ще читать, если они сосредо-чены сверху, но в данном случае мы поместили свой-ство `face` в ниж-нюю часть класса `Picasso`.

```
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";
            }
            MessageBox.Show(result);
        }
    }
}
```

`Face` — это метод записи, возвращающий значения свойства `face`. И метод, и свойство определены в классе `Picasso` и наследуются производными классами.

## 8 перечисления и коллекции



# Большие объемы данных

Наконец-то я могу  
систематизировать  
объекты МойПарен!



### Пришла беда — отворяй ворота.

В реальном мире данные, как правило, не хранятся маленькими кусочками. Данные приходят 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.

```
Worker buzz = new Worker("Прокурор");  
Worker clover = new Worker("Кинолог");  
Worker gladys = new Worker("Диктор");
```

Код позволяет передавать 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.

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
}
```

Это имя перечисления.

После последнего пункта списка запятую ставим необязательно, но ее наличие облегчает процедуру изменения порядка элементов.

Каждая из этих работ может быть выполнена пчелой. И вы можете использовать ее в качестве значения Jobs.

Значения разделяются запятыми, а в конце ставится фигурная скобка.

Ссылка на элемент перечисления выглядит следующий образом:

```
Worker nanny = new Worker(Job.EggCare);
```

Название перечисления.

Нужное вам значение из перечисления.

Мы отредактировали конструктор Worker, и теперь он принимает параметры типа Worker.Jobs.

Вы не можете 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'

Сообщение об ошибке, которое выдаст компилятор.

## 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:

```
public 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 объявлять 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:

```
int value = (int)TrickScore.Fetch * 3;  
MessageBox.Show(value.ToString());  
TrickScore score = (TrickScore)value; ←  
MessageBox.Show(score.ToString());
```

Этот оператор присваивает переменной value значение 30.

Численное значение может быть преобразовано обратно в элемент перечисления. Так как value = 30, score присваивается значение TrickScore.Fetch. Соответственно метод score.ToString() возвращает значение Fetch.

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 :

```
public 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 явно не указана, при компиляции появится сообщение:

Cannot implicitly convert type 'long' to 'int'.



Воспользуемся полученными сведениями для построения класса, описывающего игральные карты

Card	
Suit	
Value	
Name	

### 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 бубей)).

### 2 Задайте масти и карты с помощью двух перечислений

Воспользуйтесь уже знакомой вам командой Add >> Class, заменив 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;
```

Переменная cardName должна иметь значение "Ace of Spades".

Чтобы это заработало,  
в класс Card нужно по-  
местить конструктор  
с двумя параметрами.

### 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 назы-  
вается **перегру-  
зкой (overloading)**.  
О ней мы погово-  
рим 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.

## 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, ИСР 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

часто  
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

```
enum Suits {
    Spades,
    Clubs,
    Diamonds,
    Hearts
}
```

Колода карт позволяет проиллюстрировать важность ограничения значений. Было бы странно обнаружить джокера крестей или 13 червей. Вот как выглядит код класса Card.

```
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
}
```

По умолчанию первый элемент списка приравнивается к нулю, второй – к единице, третий к двойке и т. д.

В данном случае значения по умолчанию переопределяются и Values.Ace равно 1.

Счет начинается с тузов.

```
class Card {
    public Suits Suit { get; set; }
    public Values Value { get; set; }
```

Класс Card имеет свойство Suit типа Suits и свойство Value типа Values.

```
public Card(Suits suit, Values value) {
    this.Suit = suit;
    this.Value = value;
}
```

Метод чтения свойства Name использует строку, полученную из имени при помощи метода ToString().

```
public string Name {
    get { return Value.ToString() + " of " + Suit.ToString(); }
}
```

Код кнопки, щелчок на которой вызывает окно с названием случайной карты.

```
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);
}
```

Перегруженный метод Random.Next() генерирует случайное число, которое мы присваиваем перечислению.

## 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).

Объявление массива сокращено для экономии места на странице. Но здесь перечислены все 52 карты в колоде.

```
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());
    }
}
```

## ... 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.



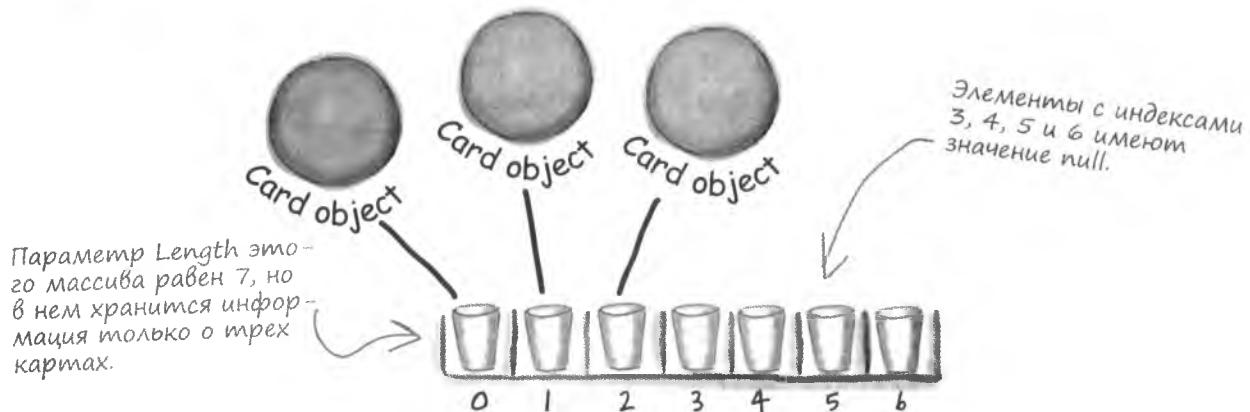
### МОЗГОВОЙ ШТУРМ

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 усложняется.

- Каждый массив имеет длину, которую вы должны знать. Для получения пустых элементов можно воспользоваться ссылкой на неопределенное значение `null`:



- 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.



- 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, больший 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.

## 1 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`.



Символ `<T>` указывает на **обобщенную коллекцию**.

Иногда для экономии места и упрощения код `<T>` будет опускаться.

Символ `T` заменяется типом, `List<int>` означает `List` значений типа `int`.

## 2 Добавление в коллекцию `List<T>`

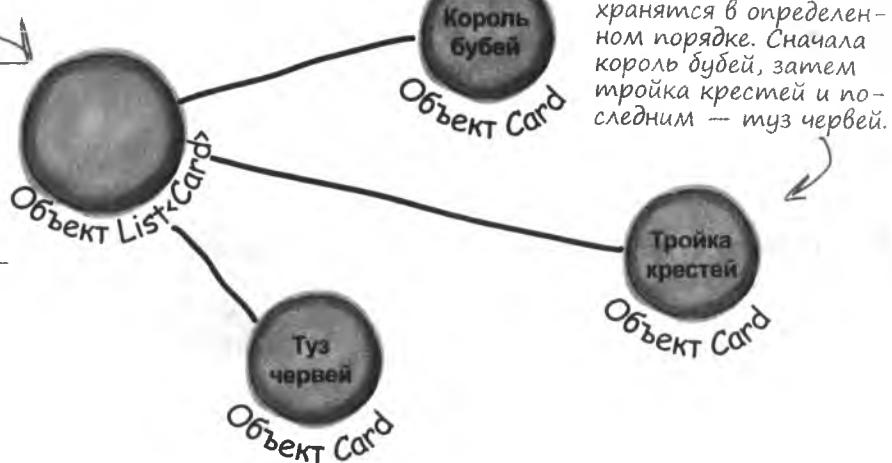
В коллекцию `List<T>` можно добавить произвольное количество элементов (главное, чтобы они были **полиморфны** с типом указанным при создании `List<T>`).

Они могут быть назначены: интерфейсам, абстрактным классам, базовым классам и т. п.

```
cards.Add(new Card(Suits.Diamonds, Values.King));
cards.Add(new Card(Suits.Clubs, Values.Three));
```

```
cards.Add(new Card(Suits.Hearts, Values.Ace));
```

В коллекцию `List` можно добавить произвольное число карт, достаточно воспользоваться методом `Add()`. Если количество объектов превышает количество свободных мест, размер коллекции увеличивается.



В коллекции, как и в массиве, элементы хранятся в определенном порядке. Сначала король бубей, затем тройка крестей и последним — туз червей.

какой прогресс!

## 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:

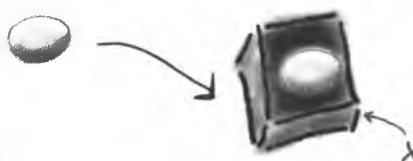
### 1 Можно создать коллекцию

```
List<Egg> myCarton = new List<Egg>();
```

Под созданный объектом List в куче выделяется память. Но в объекте пока ничего нет.

### 2 Добавить в нее объект

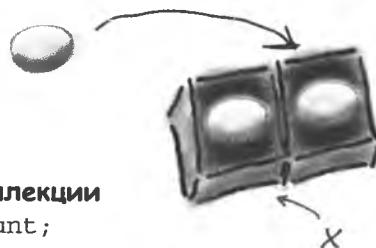
```
Egg x = new Egg();  
myCarton.Add(x);
```



Теперь в коллекции имеется объект Egg...

### 3 Добавить еще один объект

```
Egg y = new Egg();  
myCarton.Add(y);
```



...она увеличивается, чтобы принять второй объект Egg.

### 4 Узнать количество объектов в коллекции

```
int theSize = myCarton.Count;
```

Теперь коллекцию можно проверять на наличие объектов Egg. В данном случае оператор вернет значение true.

### 5 Узнать, содержатся ли в коллекции объекты определенного типа

```
bool Isin = myCarton.Contains(x);
```

Элемент x имеет индекс 0, в то время как элемент y — индекс 1.

### 6 Определить их положение

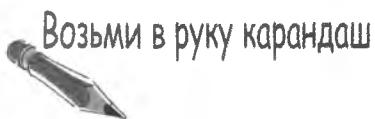
```
int idx = myCarton.IndexOf(y);
```



После удаления элемента у в коллекции остается только элемент x (размер коллекции уменьшается).

### 7 Удалить оттуда элемент

```
myCarton.Remove(y);
```



Возьми в руку карандаш  
Предположим, что эти  
операторы выполняются  
по порядку.

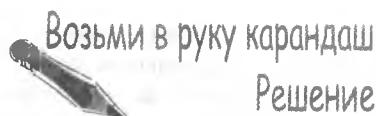
Заполните таблицу, сверяясь с расположенным слева кодом. Вам нужно показать, как выглядел бы код, если бы вместо коллекции использовался массив. Мы не ожидаем от вас 100% правильных результатов, просто попытайтесь догадаться.

### Коллекция

Мы начали выполнять  
упражнение...

### Обычный массив

<code>List&lt;String&gt; myList = new List &lt;String&gt;();</code>	<code>String [] myList = new String[2];</code>
<code>String a = "Yay!";</code>	<code>String a = "Yay!";</code>
<code>myList.Add(a);</code>	
<code>String b = "Bummer";</code>	<code>String b = "Bummer";</code>
<code>myList.Add(b);</code>	
<code>int theSize = myList.Count;</code>	
<code>Guy o = myList[1];</code>	
<code>bool isIn = myList.Contains(b);</code>	
<p>Подсказка: Помре- буется более одной строки кода.</p>	



Возьми в руку карандаш

Решение

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!" myList.Add(a);	String a = "Yay!"; myList[0] = a;
String b = "Bummer"; myList.Add(b);	String b = "Bummer"; myList[1] = b;
int theSize = myList.Count;	int theSize = myList.Length;
Guy o = myList[1];	Guy o = myList[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

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.

```
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";
}
```

Метод Remove() удаляет объект по ссылке, метод RemoveAt() — по индексу.

```
shoeCloset.RemoveAt(4);
Shoe thirdShoe = shoeCloset[3];
Shoe secondShoe = shoeCloset[2];
shoeCloset.Clear();
shoeCloset.Add(thirdShoe);
if (shoeCloset.Contains(secondShoe))
    Console.WriteLine("Удивительно!");
```

Эта строчка не будет запускаться, потому что метод Contains() возвращает значение false. В пустое перечисление мы добавили только элементом thirdShoe, но не элемент fifthShoe.

### Упражнение!

Мы объявляем коллекцию объектов Shoe (Туфля) с именем ShoeCloset (ОбувнойШкаф).

Оператор new можно использовать внутри метода List.Add().

**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 (плос 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 ИСР, 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()`.

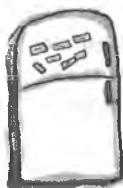
```
class List<T> : IList<T>
    ICollection<T> IEnumerable<T> IList,
    ICollection, IEnumerable
```

Отсюда берутся методы `Add()`, `Clear()`, `CopyTo()` и `Remove()`. (Это верно для всех обобщенных коллекций).

### 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.



## 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);
```

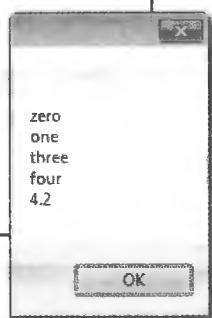
```
}
```

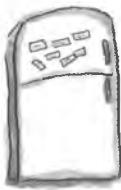
if (a.IndexOf("four") != 4) {  
 a.Add(fourth);  
}

```
printL(a);
```

```
string zilch = "zero";  
string first = "one";  
string second = "two";  
string third = "three";  
string fourth = "4.2";  
string twopointtwo = "2.2";
```

```
}
```

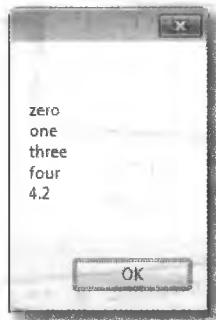




## Решение задачи с Магнитами

Помните, в главе 3 говорилось об интуитивно понятных именах? Сейчас мы их значительно 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(first);
    a.Add(second);
    if (a.Contains("three")){
        a.Add("four");
    }
    a.RemoveAt(2);
    if (a.IndexOf("four") != 4) {
        a.Add(fourth);
    }
    a.Add(twopointtwo);
}
printL(a);
    
```

Метод `RemoveAt()` удаляет элемент, следующий за элементом #2. То есть третий элемент коллекции.

Вы понимаете, почему «2.2» никогда не попадет в коллекцию, несмотря на то что этот элемент был объявлен?

Метод `printL()` использует цикл `foreach` для просмотра коллекции строк, добавления каждого элемента в общую строку и отображения результата в окне диалога.

```
public void printL (List<string>
a) {
```

```

    foreach (string element in a)
    {
        result += "\n" + element;
    }
    MessageBox.Show(result);
}
```

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. Перечисления 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 интерфейсов, 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.

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.

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.

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.

**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 хра-  
ниться. Сохранять  
можно значимые  
типы (`int`, `bool`  
или `decimal`) или  
класс.**

## Инициализаторы коллекций

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" });
```

Обратите внимание:  
каждому объекту `Shoe`  
присваивается началь-  
ное значение при помощи  
инициализатора.

Инициализатор коллекции  
может быть создан добав-  
лением каждого полученного  
при помощи метода `Add()`  
элемента к оператору,  
формирующему коллекцию.

Код, полученный при помощи инициализатора

```
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" },  
};
```

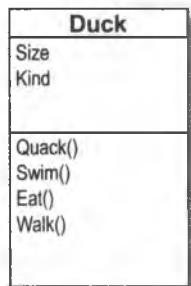
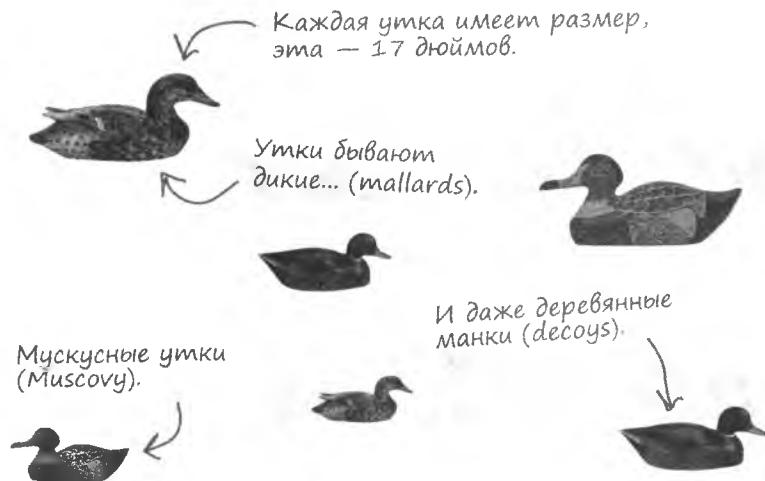
За оператором  
создания кол-  
лекции следуют  
 фигурные скобки,  
 в которых на-  
 ходятся разде-  
 ленные запятыми  
 операторы `new`.

Инициализатор  
может содер-  
жать не только  
набор опера-  
торов `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).



```
class Duck {
    public int Size;
    public KindOfDuck Kind;
}

enum KindOfDuck {
    Mallard,
    Muscovy,
    Decoy,
}
```

Перечисление KindOfDuck хранит информацию о видах уток в коллекции.

Добавьте к проекту класс Duck и перечисление KindOfDuck.

Вы добавляете к методу Main() код, выводящий результаты. Эта строка остается результатом видимым, пока вы не нажмете клавишу.

## 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();
```



## 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.

Уток можно расположить по размеру...



от самой маленькой к самой большой...



...или по виду...

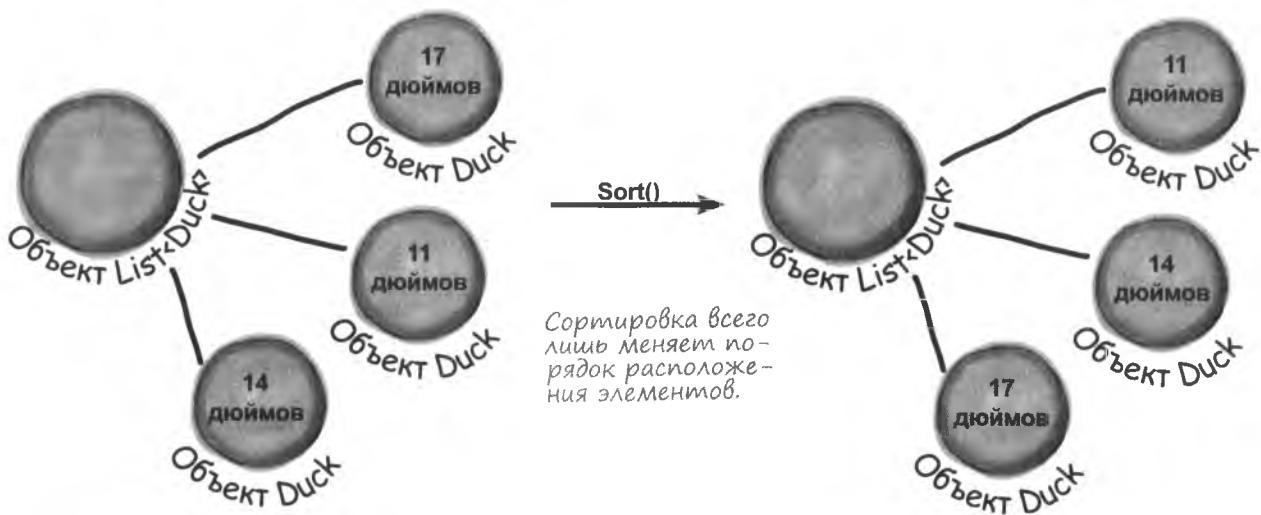


сортировка по виду утки..

### 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.

Сортировка объектов  
осуществляется с помощью  
интерфейса `IComparer<T>`,  
о котором мы поговорим  
чуть позже...



## 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.

### `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;
```

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.

```
    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.

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();
```



далее ▶

359

**Любой класс может работать со встроенным методом `Sort()` объекта `List` с помощью `IComparable<T>` и `CompareTo()`.**



## 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.

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 позволяет осуществлять любые типы сравнений.

0 означает совпадение размеров объектов.

Метод `Compare()` возвращает целое число и имеет два параметра, принадлежащих типу сортируемых объектов.

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`.

Это метод вывода уток в виде коллекции `List<Duck>`.

Добавьте метод `PrintDucks` в класс `Program`, чтобы обеспечить вывод данных.

```
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!");
}
```

Обновите метод `Main()` таким образом, чтобы он вызывался до и после сортировки.

## 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);
```

Добавьте этот код в метод `Main()`, чтобы видеть, как были отсортированы утки.



Методу `Sort()` передается ссылка на новый объект `DuckComparerBySize` как на параметр метода.

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:

```
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;
    }
}
```

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 перед манкой.

Пример со-вместного использования перечислений и коллекций.

Мы сравниваем свойство `Kind`, поэтому сортировка происходит по индексам перечисления `KindOfDuck`.

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);
```

Дополнительный код для метода `Main()`.



Сортировка по виду утки...

## Сложные схемы сравнения

Отдельный класс для сортировки уток позволяет применить более сложную логику сравнения, добавив члены, определяющие метод сортировки коллекции.

```

enum SortCriteria { Перечисление указывает объекту,
    SizeThenKind, каким именно способом будут
    KindThenSize, сортироваться утки. Первый
}                                Вариант «По размеру, затем
                                    по виду», второй — наоборот.

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);

```

Если метод Sort() для объекта IComparer<T> не указан, будет использован заданный по умолчанию метод сравнения значимых типов или ссылок.

Класс, выполняющий сравнение уток, более сложен. Метод Compare() использует те же самые параметры, но сначала проверяет значение открытоего 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 О возвращается не только когда утки имеют одинаковый размер, но и 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 вызывать 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.

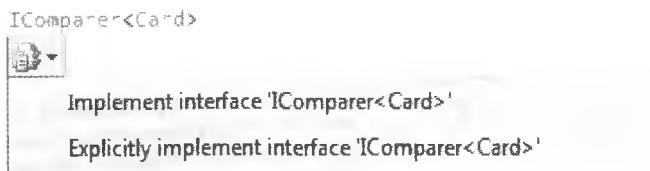
**2 Класс, который реализует сортирующий карты интерфейс IComparer<Card>**

Воспользуйтесь средствами ИСР, 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**.

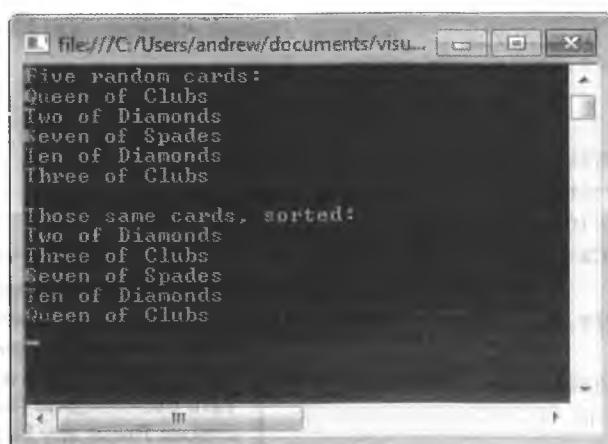


Выберите вариант Implement interface IComparer<Card>, и ИСР введет за вас все методы и свойства, необходимые для реализации данного интерфейса. Будет создан пустой метод Compare(), сравнивающий карты x и y. Пусть метод возвращает 1 (если x больше y), -1 (если меньше) и 0 (если карты одинаковы). Убедитесь, что король следует после валета, который следует за четверкой, которая следует 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 кла-  
виши, чтобы завер-  
шить программу.



Объект IComparer  
должен осуществлять  
сортировку таким  
образом, чтобы первыми  
в списке оказывались  
карты с наименьшим  
значением.





## Упражнение

## Решение

```

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;
    }
}

```

Если x имеет большее значение, метод возвращает 1. Если значение меньше, возвращается -1. Но оба оператора, возвращающие значение, завершают работу метода.

Встроенный метод List. Sort() берет объект IComparer, имеющий один метод: Compare(). Эта реализация проводит сравнение двух карт сначала по старшинству, потом по масти.

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, карты одинаковы. 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);
    }
}

```

Это обобщенная коллекция объектов Card. Ее легко сортировать с помощью интерфейса IComparer.

```

Console.WriteLine();
Console.WriteLine("Те же карты, отсортированные:");
cards.Sort(new CardComparer_byValue());
foreach (Card card in cards)
{
    Console.WriteLine(card.Name);
}
Console.ReadKey();
}

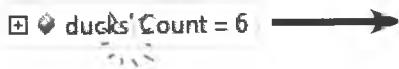
```

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:

 ducks' Count = 6 →

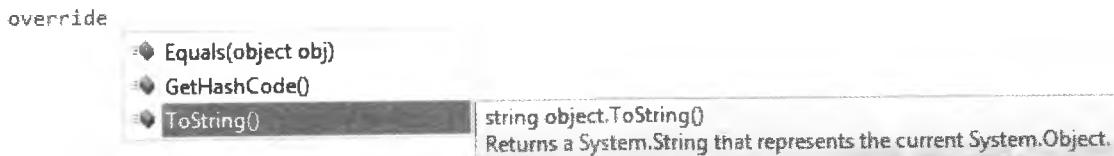
Метод `ToString()` вызывается при отображении объекта в окне `Watch`. Но метод `ToString()`, унаследованный классом `Duck` от класса `Object`, возвращает только имя класса. Мы можем сделать этот метод более информативным.

ducks Count = 6	
[+]	[0] {MyProject.Duck}
[+]	[1] {MyProject.Duck}
[+]	[2] {MyProject.Duck}
[+]	[3] {MyProject.Duck}
[+]	[4] {MyProject.Duck}
[+]	[5] {MyProject.Duck}
[+]	Raw View

Вместо передачи значения методам `Console.WriteLine()`, `String.Format()` и т. п., можно передать им объект. Его метод `ToString()` будет вызван автоматически. (Это работает и для значимых типов, таких как `int` и `enums`!)

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`!

ducks Count = 6	
[+]	[0] {A 17 inch Mallard}
[+]	[1] {A 18 inch Muscovy}
[+]	[2] {A 14 inch Decoy}
[+]	[3] {A 11 inch Muscovy}
[+]	[4] {A 14 inch Mallard}
[+]	[5] {A 13 inch Decoy}
[+]	Raw View

Чтобы показать объект, отладчик вызывает метод `ToString()`.

## 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 вызывает метод `ToString()` этого объекта.

Ведите этот код в программу `Ducks` и запустите ее. Список вывода не изменится. Но если вы захотите добавить свойство `Gender` для учета пола утки, достаточно 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 ИСР. В следующих главах мы рассмотрим преимущества, которые дает преобразование объектов в строки.

## Интерфейс `IEnumerable<T>`

Цикл `foreach` под  
увелличительным стеклом

Найдите переменную `List<Duck>` и воспользуйтесь функцией IntelliSense для рассмотрения метода `GetEnumerator()`. Напечатайте «`.GetEnumerator()`» и посмотрите на открывшееся меню:

`ducks.GetEnumerator()`

`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();
```

(Не беспокойтесь, что вы не понимаете последних двух строк. С интерфейсом `IDisposable` вы познакомитесь в главе 9.)

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!

Попробуйте, 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?

Инициализаторы  
коллекций работают  
с **ЛЮБЫМ** объектом  
`IEnumerable<T>`!

Реализуя интерфейс  
`IEnumerable<T>`,  
коллекция предо-  
ставляет цикл, пе-  
ребирающий ее эле-  
менты по очереди.

Здесь написан  
не весь код, но  
в данном случае  
главное, чтобы  
вы поняли  
основную идею...



никого кроме уток!

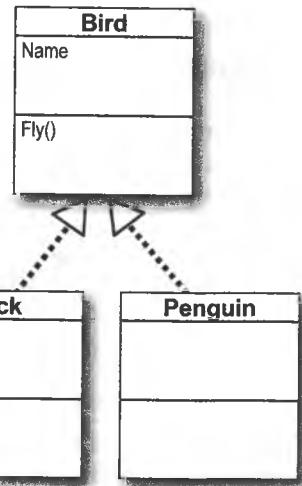
## Восходящее приведение с помощью `IEnumerable`

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.

```
class Bird {  
    public string Name { get; set; }  
    public void Fly() {  
        Console.WriteLine("Полетели!");  
    }  
    public override string ToString() {  
        return "Имя птицы " + Name;  
    }  
}
```

```
class Penguin : Bird {  
    public 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.

```
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 добавить 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) { Благодаря восходящему приведению уток к IEnumerable<Bird>  
    Console.WriteLine(bird); // их можно добавить в коллекцию объектов Bird.  
}
```



## Создание перегруженных методов

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 методов с различными параметрами.

Вместо редактирования пространства имен можно использовать оператор.



### 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. ИСР 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;
    }
}

public static bool DoesCardMatch(Card cardToCheck, Values value) {
    if (cardToCheck.Value == value)
        return true;
    } else {
        return false;
    }
}
```

Перегруженные методы не обязаны быть статическими, но мы решили, что вам будет полезно попрактиковаться в написании статических методов.

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);
MessageBox.Show(doesItMatch.ToString());
```

Метод ToString() нужен, так как параметр метода MessageBox.Show() должен принадлежать к типу string.

Как только будет напечатано DoesCardMatch(), ИСР 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 форма, которая будет его использовать.

### 1 Форма, позволяющая перемещать карты между колодами

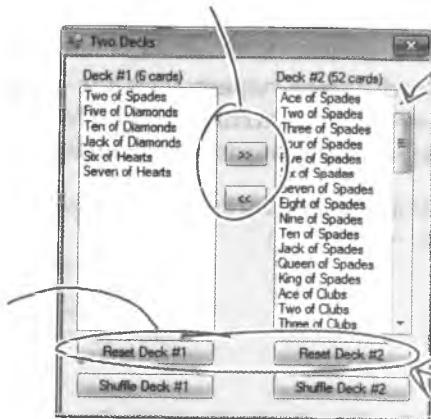
Создадим класс Deck (Колода) для хранения произвольного количества карт. В реальной колоде 52 карты, но в классе Deck может быть сколько угодно карт (или не быть ни одной).

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.

Кнопки moveToDeck2 (верхняя) и moveToDeck1 (нижняя) 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.

Кнопки reset1 и reset2 сначала вызывают метод ResetDeck(), а потом метод RedrawDeck().



Для показа обеих колод используются два элемента ListBox. При щелчке на кнопке moveToDeck1 выбранная карта перемещается из колоды #2 в колоду #1.

Имена этих кнопок shuffle1 и shuffle2. Они вызывают подходящий метод Deck.Shuffle(), чтобы перемешивать колоду, а затем перерисовывают ее.

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)";
    }
}
```

Обратите внимание, как карты добавляются в коллекцию при помощи цикла foreach.

Метод RedrawDeck() масает колоду, вытаскивает из нее случайные карты, и обновляет содержимое текстовых полей в соответствии с проделанным обменом.

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 содержащимо массива объектов Card.

```
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 void Sort() {
        cards.Sort(new CardComparer_bySuit());
    }
}
```

**Принадлежность параметра к типу `IEnumerable<Card>` позволяет передавать любую коллекцию, а не только `List<T>` или массив.**

**Хотя метод `GetCardNames()` возвращает массив, мы даем доступ к `IEnumerable<string>`.**

**Класс Deck хранит объекты в коллекции List, при этом они являются закрытыми.**

**Полная колода из 52 карт будет создана без передачи параметров конструктору.**

**Перегруженный конструктор в качестве параметра берет массив карт, загружая его в исходную колоду.**

Deck
Count
Add()
Deal()
GetCardNames()
Shuffle()
Sort()

**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.

**Еще одна подсказка:** Протестировать метод `Shuffle()` можно с помощью формы. Щелкните на кнопке “Reset Deck #1”, пока не получите колоду из трех карт. В этом случае легко убедиться, что метод перемешки 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.



## 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 решаема 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;
    }
}

```

```

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();
}

```

Метод RedrawDeck()  
вам уже встречался.



Сортировка по масти напоминает сортировку по старшинству. Единственное отличие в том, что масти сравниваются первыми, а сравнение значений карт происходит только при совпадении масти.



Вместо конструкции if/else if мы использовали набор операторов if. Это работает, так как каждый следующий оператор if выполняется только в случае невыполнения предыдущего.



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. Воссоздать колоду #2 еще проще. Достаточно создать экземпляр Deck().

→ Переверните страницу и продолжим!



## 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);
}

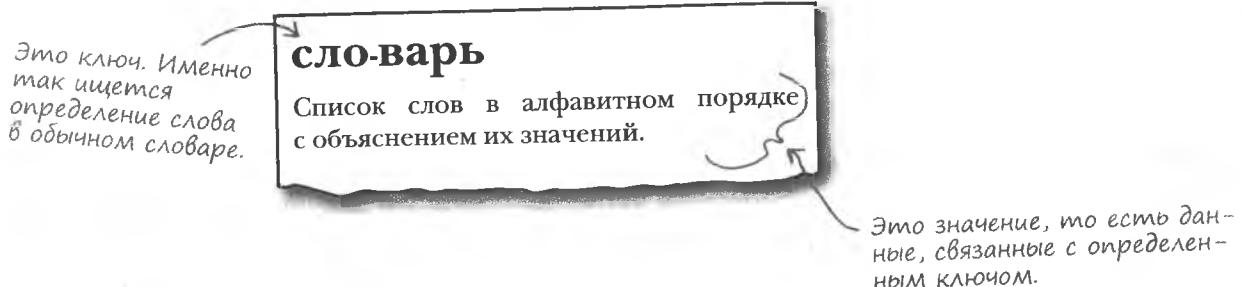
```

Функция этих кнопок проста — сначала восстановить или перемешать колоду, а затем перерисовать ее.

Свойство `SelectedIndex` элемента управления `ListBox` позволяет нажать, 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.



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 или число.");

    if (wordDefinition.ContainsKey ("Ключ")){
        MessageBox.Show(wordDefinition["Ключ"]);
    }
}
```

Ключи и значения до добавления в словарь при помощи метода Add().

Ключи и значения в этом словаре принадлежат к типу `string`, ведь мы моделируем обычный словарь, в котором за термином идет определение.

Метод Add() сначала берет ключ, а потом значение.

Метод ContainsKey() позволяет определить наличие в словаре указанного слова в качестве параметра.

## 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 так же 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.

Словари, сопоставляющие целые числа и объекты, используемые в случаях присвоения объектам уникальных идентификаторов.

```
Dictionary<int, Duck> duckDictionary = new Dictionary<int, Duck>();  
duckDictionary.Add(376, new Duck()  
{ Kind = KindOfDuck.Mallard, Size = 15 });
```

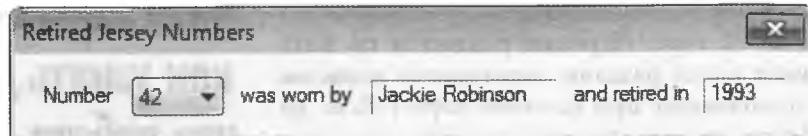
## Практическое применение словаря

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;
    }
}
```

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();
    }
}
```

Ключи из словаря добавляются в коллекцию элементов ComboBox.

Событие SelectedIndexChanged элемента ComboBox обновляет две метки на форме, выводя значение объекта JerseyNumber, взятое из словаря.

Объекты JerseyNumber передаются в словарь при помощи инициализатора коллекции.

Свойство SelectedItem элемента ComboBox является объектом. Так как ключ принадлежит типу int, вам потребуется операция приведения к этому типу.



## 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 которой отдельные фрагменты уже собраны. Код формы приведен в шаге #3. Вам остается только ввести его в ИСР. Но при этом все классы, которые вы пишите, должны *работать с этим кодом*. А вот этого добиться не так-то просто!

1

### 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 взяточ. Взяточкой считается набор из четырех одинаковых карт. Для выигрыша нужно собрать максимальное количество взяточ. Собранный набор из четырех карт выкладывается на стол.
- ★ Если выложенная взяточка 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 взяточ.
- ★ 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 спрашиваются про свои карты. Результаты каждой партии отображаются в текстовом поле.
- ★ Процедуры сдачи карт и взяточ 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.

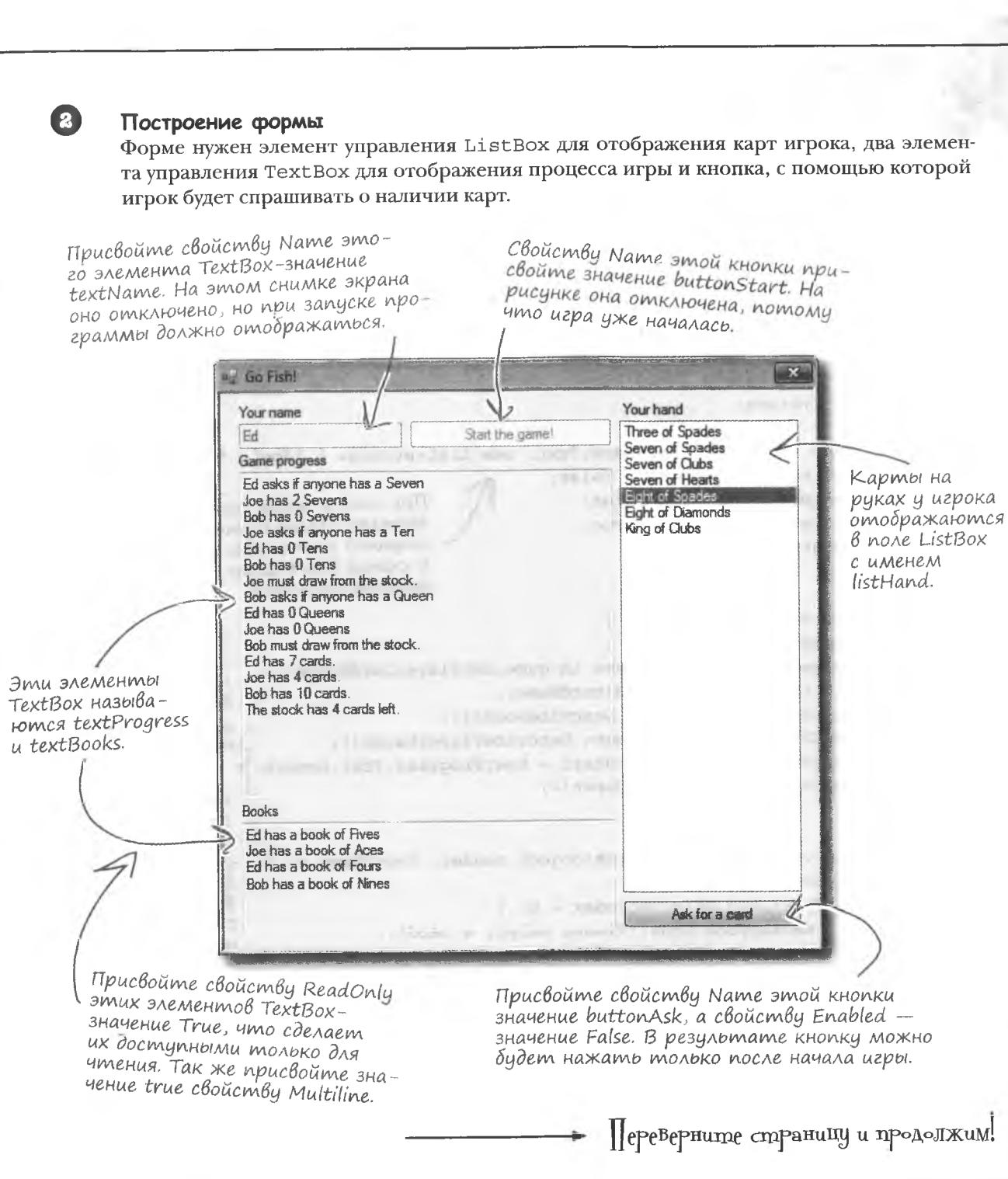
2

## 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 textBoxName. На этом снимке экрана это отключено, но при запуске программы должно отображаться.

Свойству Name этой кнопки присвойте значение buttonStart. На рисунке она отключена, потому что игра уже началась.





3

## Это код формы

Ведите его в ИСР.

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

    private Game game;
```

Это единственный класс, с которым взаимодействует форма. Именно он управляет всей игрой.

**Свойство Enabled** включает и отключает элементы управления формы.

```
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();
}
```

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.

```
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();
```

**Свойство SelectionStart** и метод **ScrollToCaret()** позволяют прокручивать текст, если он не умещается в текстовое поле.

```
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();
}
```

Игрок выбирает карту и нажимает на кнопку Ask, чтобы узнать, есть ли данная карта у кого-нибудь из соперников. Класс Game играет с помощью метода PlayOneRound().

**Свойство SelectionStart** определяет начало выделенного фрагмента, после этого метод **ScrollToCaret()** прокручивает содержимое текстового поля до места нахождения курсора.

4

**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 одинаковых 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!



## Длинные упражнения

5

### Это СЛОЖНАЯ часть: создание класса Player

Экземпляры класса Player существуют для всех игроков. Они создаются обработчиком событий кнопки buttonStart.

```
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()
    }

    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(); }
}
```

Внимательно читайте комментарии, там сказано, какие именно действия должны выполнять методы, для которых вы пишете код.

Метод `Peek()`, добавленный нами в класс `Deck`, очень полезен. Он позволяет программе посмотреть на карту, индекс которой вы указали, но, в отличие от метода `Deal()`, не удаляет карту из колоды.

```
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);
            for (int card = cards.Count - 1; card >= 0; card--)
                cards.Deal(card);
        }
    }
    return books;
}
```

Вам потребуются ДВЕ перегруженные версии метода `AskForACard()`. Первая будет использоваться вашими соперниками, она должна просматривать их карты и выбирать ту, о которой нужно спросить. Вторая версия используется, когда о карте спрашиваете вы. При этом обе версии опрашивают ВСЕХ игроков (как компьютер, так и человека).

## 6

**Еще один метод для класса `Card`**

Этот статический метод возвращает название карты (во множественном числе). Так как он статический, для вызова нужно указать имя класса – `Card`. `Plural()` – и от него невозможно 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 этого статического метода мы использовали разделяемый класс. В этом случае вам проще понять, что именно происходит. Но необходимости использовать разделяемый класс не было, так метод можно добавить в класс `Card`.

→ Скоро закончим, переверните страницу!



## Длинные упражнения

7

### Создание класса Game

Форма сохраняет экземпляр Game, управляющий игрой. Посмотрите, как он используется.

```
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();           // Это полезно и для инкапсуляции.
        Ifs использовать IEnumerable<T>           // вместо List<T>, вы не сможете
        players[0].SortHand();           // случайно отредактировать код.
    }

    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."
    }
}
```

В классах Player и Game имеются ссылки на элементы формы TextBox, в которых появляются сообщения о ходе игры. Убедитесь, что в верхней части файлов есть строка «using System.Windows.Forms;».

**Интерфейс IEnumerable<T>** делает классы более гибкими. Об этом следует помнить при дальнейшем редактировании кода. В данный момент для получения экземпляра класса Game можно написать `string[] List<string>`.

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 предоставляемый информацией о количестве сделанных им за игру взяточ. 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 взяточ присутствует у более чем одного игрока! Так что вам потребуется еще один цикл `foreach`, ищущий всех игроков в словаре `winners`, которые имеют максимальное количество взяточ. 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 взяточ
    // каждого игрока. Сначала цикл foreach (Values value in books.Keys)
    // заполняет словарь winners информацией о взятках. Затем
    // словарь просматривается на предмет поиска максимального
    // количества взяточ. 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 переносов 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, отформатированный 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);
}

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;
}

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 Deal() вызывается в начале игры, тасует колоду и раздает каждому игроку по пять карт. Затем он собирает взяточки, если таковые появляются.*

*Если после того как игрок спросил карту, образовалась взяточка, она у него забирается. Если взяточка 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.*

*Метод PullOutBooks() проверяет карты игрока на наличие взяточек. Обнаруженная взяточка добавляется в словарь. Если карт не осталось, возвращается значение true.*

```

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;
}

```

Так как в форме должен отображаться список взяточок, воспользуемся методом `DescribeTheBooks()`, чтобы превратить записи словаря в строки.

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 взяточек.

Затем определяется максимальное количество взяточек. Оно помещается в переменную `mostBooks`.

Теперь, когда мы знаем игрока с максимальным количеством взяточек, можно вывести строку с именем победителя (или победителей).

→ Переверните страницу и продолжим!

# Решение длинных упражнений

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;
}

public Values GetRandomValue() {
    Card randomCard = cards.Peek(random.Next(cards.Count));
    return randomCard.Value;
}

public Deck DoYouHaveAny(Values value) {
    Deck cardsIHave = cards.PullOutValues(value);
    textBoxOnForm.Text += Name + " has " + cardsIHave.Count + " "
        + Card.Plural(value) + Environment.NewLine; Метод DoYouHaveAny() использует
    return cardsIHave;                                метод PullOutValues() для извлече-
                                                                ния карт, которые соответствую-
                                                                ют заданным параметрам.

}

public void AskForACard(List<Player> players, int myIndex, Deck stock) {
    Values randomValue = GetRandomValue();
    AskForACard(players, myIndex, stock, randomValue);
}
```

**Дополнительное мини-упражнение:** Улучшите инкапсуляцию  
класса Player, заменив в этих двух методах List<Player> на  
IEnumerable<Player>, не повлияв на работу программы.

```
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) {
        textBoxOnForm.Text += Name +
            " must draw from the stock." + Environment.NewLine;
        cards.Add(stock.Deal());
    }
}
```

Это конструктор класса Player. Он задает значения приватных полей и добавляет в текстовое поле строку с информацией о присоединившемся игроке.

Метод GetRandomValue() с помощью метода Peek() выбирает случайную карту среди имеющихся у игрока.

Метод DoYouHaveAny() использует метод PullOutValues() для извлечения карт, которые соответствуют заданным параметрам.

В программе два перегруженных метода AskForACard(). Этот используется соперниками — выбирает случайную карту из имеющихся и вызывает второй метод AskForACard().

Метод AskForACard() проверяет всех игроков (за исключением сражающегося), вызывает их метод DoYouHaveAny() и добавляет найденные подходящие карты.

При отсутствии у соперников подходящих карт игрок берет карту из запаса при помощи метода Deal().

## 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, 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 обрабатывается тот, кто первым пришел.

↑  
Очередь работает по принципу «раньше вошел, раньше вышел». То есть объект, первым помещенный в очередь, первым и обрабатывается.

**Обобщенные коллекции являются важной частью .NET Framework**

Они настолько полезны, что ИСР 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`!

**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!

Стек работает по диаметрально противоположному принципу. То есть чем позже объект попадает в стек, тем раньше он обрабатывается.

**Очередь подобна списку. Новые объекты помещаются в его конец, а читается он сначала. Стек же дает доступ только к последнему помещенному в него объекту.**

любите ли вы стоять в очереди?

## Звенья следуют в порядке их поступления

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.

```
Создаем очередь строк.  
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");  
  
Метод Peek()  
позволяет видеть первый элемент  
в очереди, не удаляя его.  
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  
  
Метод Clear()  
удаляет из очереди все объекты.
```

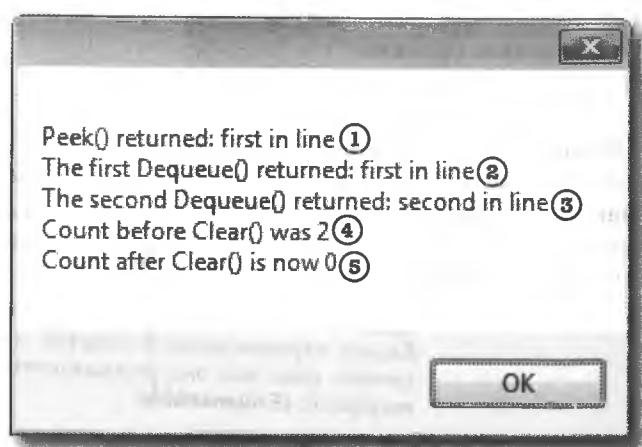
В очередь добавляются четыре элемента.

Первый метод `Dequeue()` удаляет из очереди первый элемент. После чего второй элемент сдвигается на первое место и следующий вызов метода `Dequeue()` удаляет уже его.

Свойство `Count` возвращает число элементов в очереди.



Объекты ждут своей очереди.



## Звенья следуют в порядке, обратном порядку их поступления

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 предыдущих.

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");

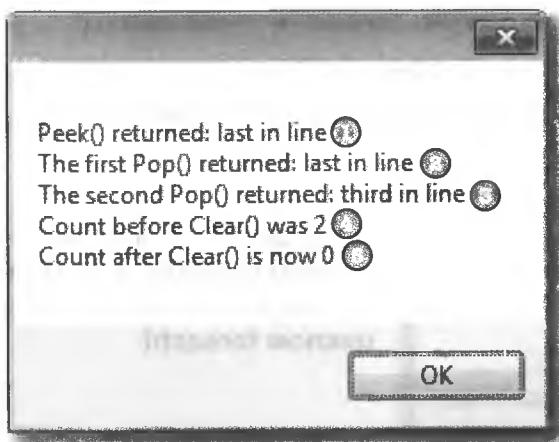
① string takeALook = myStack.Peek();
② string getFirst = myStack.Pop(); ←
③ string getNext = myStack.Pop();
④ int howMany = myStack.Count;
myStack.Clear();

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);
```

Метод Pop() удаляет из стека последний добавленный туда элемент.

Вместо символов \n можно воспользоваться константой Environment.NewLine, но мы предпочли сократить код.

5



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, реализующие 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, вернуть в состояние стека.

Все четыре элемента были скопированы в новые коллекции.



...для доступа ко всем членам очереди или стека достаточно воспользоваться циклом foreach!



## пражнение

Напишите программу, которая помогает хозяину кафе кормить лесорубов лепешками. Начните с класса `Lumberjack` (Лесоруб). Сконструируйте форму и добавьте к кнопкам обработчики событий.

- Добавьте в класс `Lumberjack` метод чтения для свойства `FlapjackCount` и методы `TakeFlapjacks` и `EatFlapjacks`.

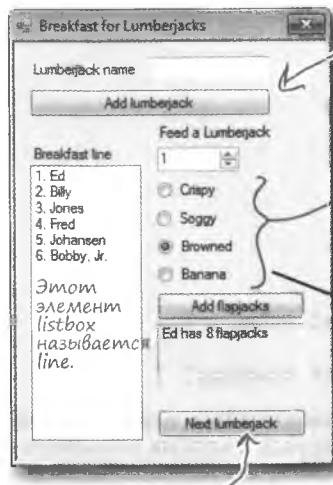
```
class Lumberjack {
    private string name;
    public string Name { get { return name; } }
    private Stack<Flapjack> meal;
    public Lumberjack(string name) {
        this.name = name;
        meal = new Stack<Flapjack>();
    }
    public int FlapjackCount { get { // возвращает число } }
    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 browned flapjack
Ed ate a soggy flapjack
Ed ate a soggy flapjack
Ed ate a soggy flapjack
Ed ate a crispy flapjack
Ed ate a soggy flapjack
Ed ate a banana flapjack
Ed ate a browned flapjack
```

- 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 выбирать только один переключатель за один раз. Узнать имена переключателей можно в методе `addFlapjacks_Click`.

```
private void addFlapjacks_Click(...) {
    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;
```

Заметили, что перечисление `Flapjack` использует строчные буквы (`Soggy`), но на выходе получаете прописные (`soggy`)? Подсказываем, как исправить ситуацию. Метод `ToString()` возвращает строку, одним из открытых членов которой является метод `ToLower()`, возвращающий только прописные буквы.

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 три кнопки. Подсказка: он использует цикл `foreach`.

```
Lumberjack currentLumberjack = breakfastLine.Peek();
currentLumberjack.TakeFlapjacks(food,
    (int)howMany.Value);
RedrawList();
```

Элемент управления `NumericUpDown` называется `howMany`, а `label` называется `nextInLine`.

Обратите внимание на особый синтаксис: `else if`.

Метод `Peek()` возвращает ссылку на первого лесоруба в очереди.

## решение упражнения



### упражнение Решение

```
private Queue<Lumberjack> breakfastLine = new Queue<Lumberjack>();
private void addLumberjack_Click(object sender, EventArgs e) {
    breakfastLine.Enqueue(new Lumberjack(name.Text));
    name.Text = "";
    RedrawList();
}
```

Метод RedrawList() при помощи цикла foreach убирает лесорубов из очереди и помещает сведения о них в список.

```
private void RedrawList() {
    int number = 1;
    line.Items.Clear();
    foreach (Lumberjack lumberjack in breakfastLine) {
        line.Items.Add(number + ". " + lumberjack.Name);
        number++;
    }
}
```

Этот элементом listBox называется двумя кнопками носит имя nextInLine (следующий в очереди).

```
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";
}
}
```

Этот оператор if обновляет метку в соответствии с информацией о следующем лесорубе в очереди.

```
private void nextLumberjack_Click(object sender, EventArgs e) {
    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>();
    }
}
```

Метод TakeFlapjacks обновляет содержимое стека Meal (Прием пищи).

```
public int FlapjackCount { get { return meal.Count; } }
```

```
public void TakeFlapjacks(Flapjack food, int howMany) {
    for (int i = 0; i < howMany; i++) {
        meal.Push(food);
    }
}
```

Именно здесь перечисление Flapjack впервые пишется прописными буквами. Постарайтесь понять, как это происходит.

```
public void EatFlapjacks() {
    Console.WriteLine(name + "'s eating flapjacks");
    while (meal.Count > 0) {
        Console.WriteLine(name + " ate a "
            + meal.Pop().ToString().ToLower() + " flapjack");
    }
}
```

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.



- 1** Добавьте в класс **Lumberjack** метод чтения для свойства **FlapjackCount** и методы **TakeFlapjacks** и **EatFlapjacks**.

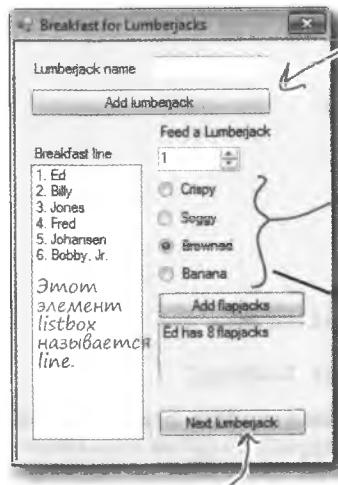
```
class Lumberjack {
    private string name;
    public string Name { get { return name; } }
    private Stack<Flapjack> meal;
    public Lumberjack(string name) {
        this.name = name;
        meal = new Stack<Flapjack>();
    }
    public int FlapjackCount { get { // возвращает число } }
    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 browned flapjack
Ed ate a soggy flapjack
Ed ate a soggy flapjack
Ed ate a soggy flapjack
Ed ate a crispy flapjack
Ed ate a soggy flapjack
Ed ate a banana flapjack
Ed ate a browned flapjack
```

- 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** внутри 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(...) {
    Flapjack food;
    if (crispy.Checked == true)
        food = Flapjack.Crispy;
    else if (soggy.Checked == true)
        food = Flapjack.Soggy;
    else if (browned.Checked == true)
        food = Flapjack.Brown;
    else
        food = Flapjack.Banana;
```

Обратите внимание на особый синтаксис: **else if**. Метод **Peek()** возвращает ссылку на первого лесоруба в очереди.

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 три кнопки. **Подсказка:** он использует цикл **foreach**.

```
Lumberjack currentLumberjack = breakfastLine.Peek();
currentLumberjack.TakeFlapjacks(food,
    (int)howMany.Value);
```

Элемент управления **NumericUpDown** называется **howMany**, а **label** называется **nextInLine**.

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.

## 9 Чтение и запись файлов



### Сохрани массив байтов и спаси мир



#### 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

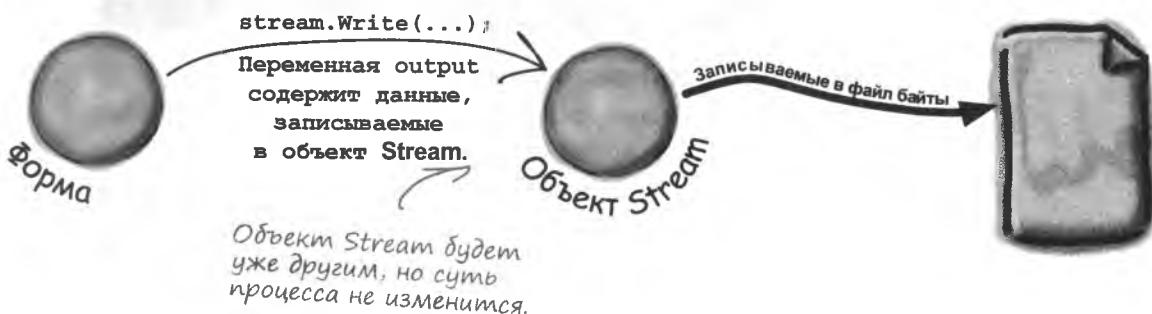
Поток (stream) — это способ, которым .NET Framework обменивается данными с программой. Каждый раз, когда программа читает или записывает файл, соединяется с другим компьютером сети или выполняет другие действия, связанные с передачей байтов из одного места в другое, речь идет о потоке данных.

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.

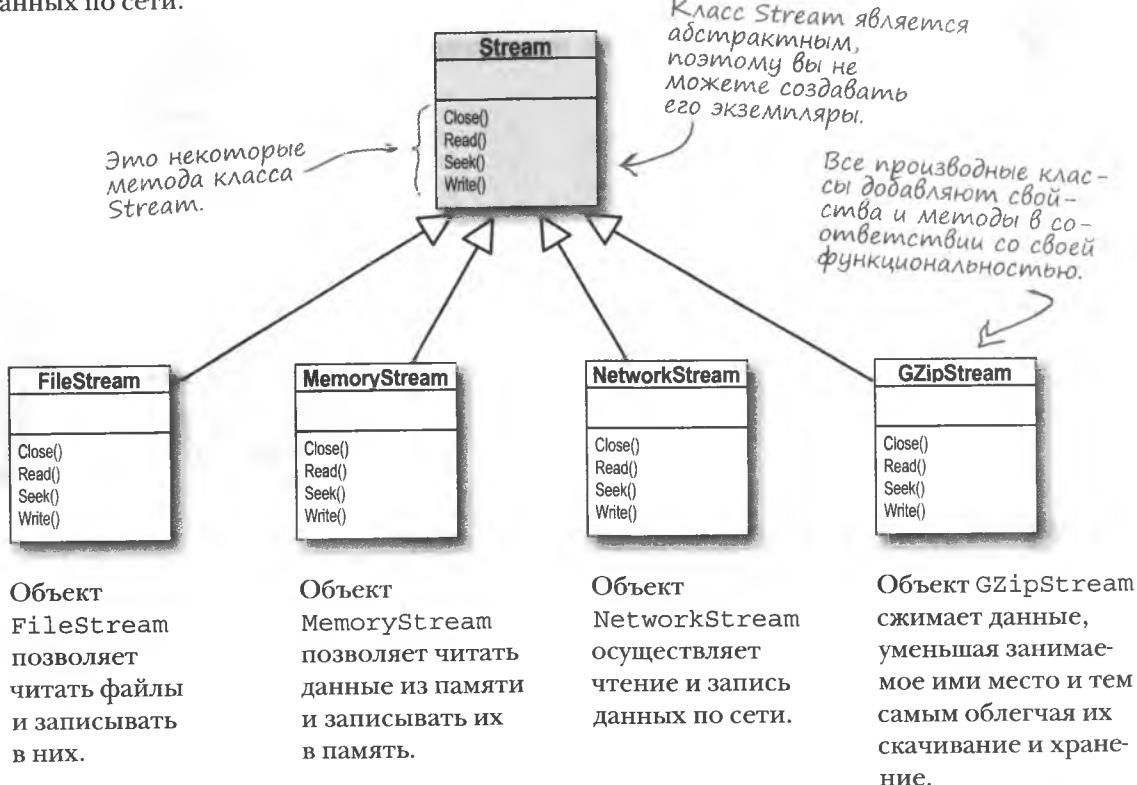


Для записи данных в файл используется другой объект Stream.



## 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.



### 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.
- 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

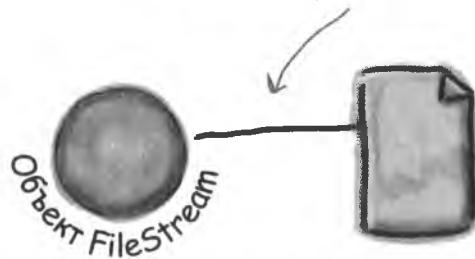
Вот что происходит при записи в файл нескольких строк текста:

В верхней части любой программы, работающей с потоками, должна присутствовать строчка  
using System.IO;

- 1 Создается объект FileStream, получающий команду записывать в файл.

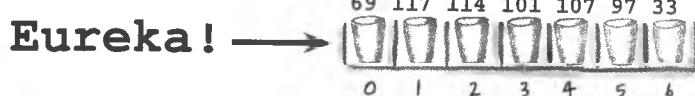


- 2 Объект FileStream присоединяется к файлу.



- 3 Строки, которые требуется записать, следует преобразовать в массив типа byte.

Эта процедура называется  
перекодированием, мы  
поговорим о ней чуть позже...



- 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()**.

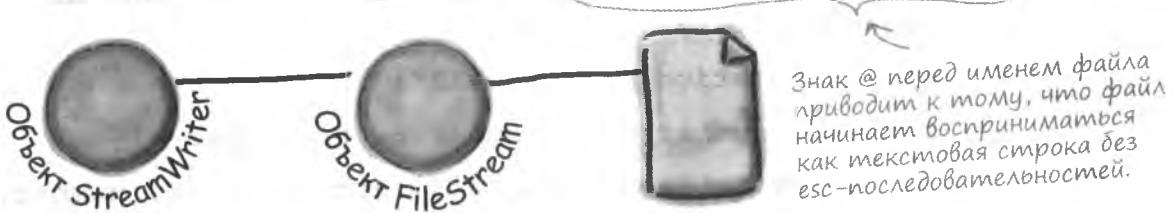
**Класс StreamWriter** автоматически создает объект **FileStream** и управляет им.

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);
```

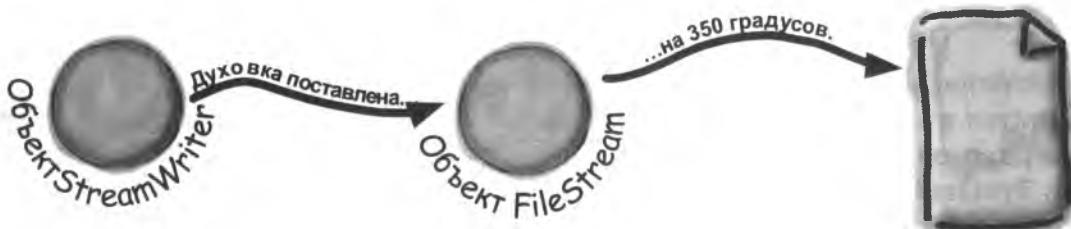


2

### 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 открытym 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()`:

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 `Close()` разрывает связь с файлом и прочими ресурсами, с которыми работаем объектом `StreamWriter`. Если не закрыть поток, запись текста осуществлена не будет.

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.

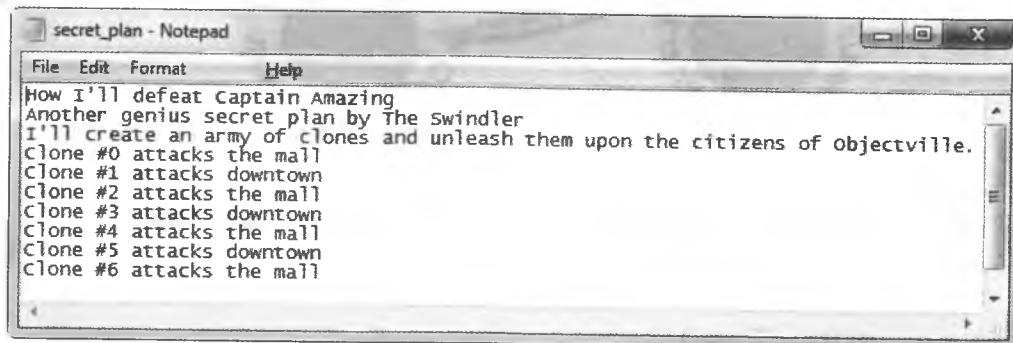
Вы понимаете, что происходит с переменной `location`?

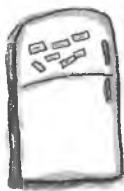
Скобки {} внутри текста передают переменные в записываемую строку. {0} заменяется первым параметром после строки, {1} вторым — и т. д.

Метод `WriteLine()` переносит строки после записи, в то время как метод `Write()` отправляет обычный текст.

Объект `StreamWriter` находится в пространстве имён `System.IO`, поэтому в верхней части программы должна быть строка `using System.IO;`

Вот такой результат получается на выходе.





## 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!

```
private void button1_Click(object sender, EventArgs e) {
    Flobbo f = new Flobbo("blue yellow");
    StreamWriter sw = f.Snobbo();
    f.Blobbo(f.Blobbo(sw), sw);
}
```

```
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;
}
```

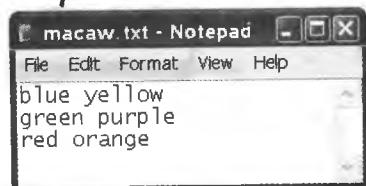
```
class Flobbo {
```

```
if (Already) {
```

```
} else {
```

```
public StreamWriter Snobbo() {
```

**Результат:**





## Решение задачи с Магнитрами

Вам требовалось сконструировать класс Flobbo, работающий определенным образом.

```
private void button1_Click(object sender, EventArgs e) {
    Flobbo f = new Flobbo("blue yellow");
    StreamWriter sw = f.Snobbo();
    f.Blobbo(f.Blobbo(sw), sw);
}
```

```
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 {

```

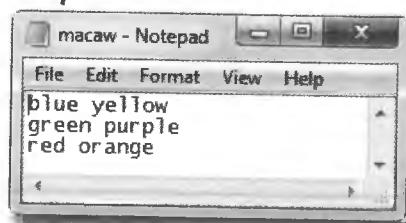
```
        sw.WriteLine(Zap);
        Zap = "red orange";
        return true;
    }
}
}
```

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.

Метод Blobbo() пере-  
гружен, он имеет два  
объявления с двумя  
различными параметрами.

После завершения работы не  
забудьте закрыть файлы.

### Результат:



## 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:

```
StreamReader reader =
    new StreamReader(@"c:\secret_plan.txt");
StreamWriter writer =
    new StreamWriter(@"c:\emailToCaptainAmazing.txt");
```

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@objectiville.net");
writer.WriteLine("Subject: Can you save the day... again?");
writer.WriteLine(); // Пустой метод WriteLine() записывает пустую строку.
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();
```

Вы должны закрыть все открытые папки, даже если всего лишь читаете файл.

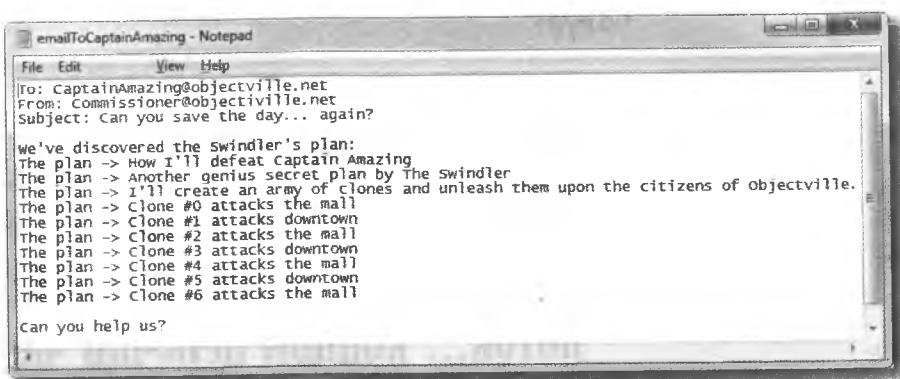
Объекты StreamReader и StreamWriter после создания их экземпляров открывают собственные папки. Чтобы закрыть их, мы два раза вызываем метод Close().

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?

Передайте файл, который требуется прочитать конструктору класса StreamReader.

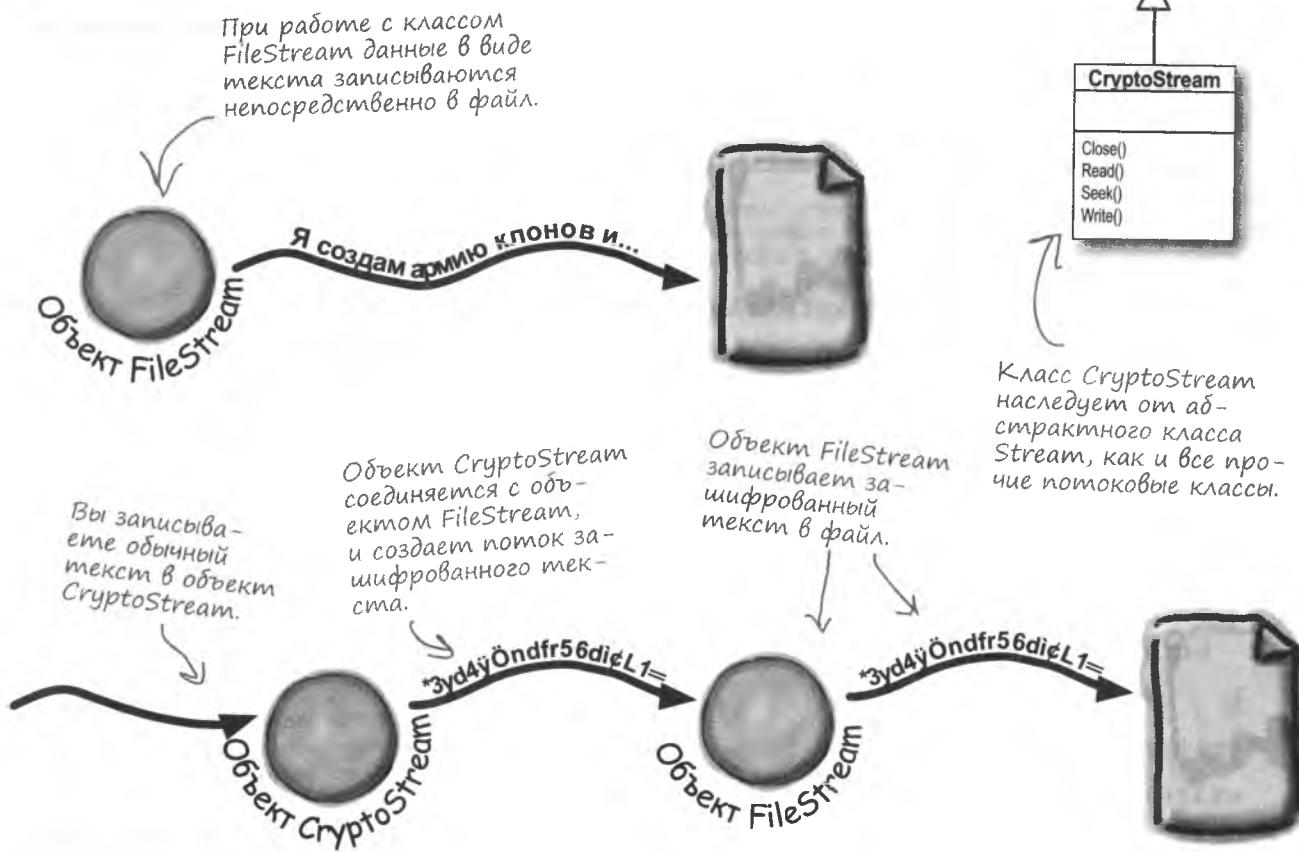
Свойство EndOfStream, позволяющее определить, остались ли в файле непрочитанные данные.

Цикл читает строку при помощи считывающего устройства и записывает ее при помощи устройства записи.



## Данные могут проходить через несколько потоков

Большим преимуществом работы с потоками в .NET является возможность пропустить 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.

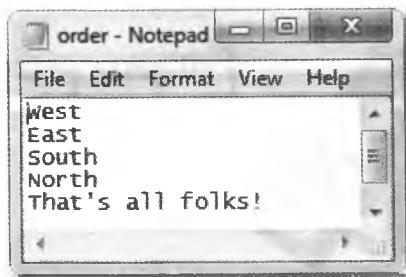
```

class Pineapple {
    const _____ d = "delivery.txt";
    public _____
        { North, South, East, West, Flamingo }
    public static void Main() {
        _____ 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._____ ();
    }
}

```

Каждый фрагмент  
можно использо-  
вать нескольким  
разам

int	long	string	enum	class	ReadLine	WriteLine	Stream reader	writer	StreamReader	StreamWriter	public	private	this	for	while	foreach	=	>=	Fargo
HowMany	HowMuch	HowBig	HowSmall				reader	writer	Open	Close	class	static					<=	Utah	
							StreamReader	StreamWriter								!=	Idaho		
							Open	Close								==	Dakota		
																++	Pineapple		
																--			



```

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._____ ();
    }
}

```



## Решение ребуса в бассейне

Это перечисление (особенно метод `ToString()`) используется для вывода конечного результата.

```
class Pineapple {
    const string d = "delivery.txt";
    public enum Fargo { North, South, East, West, Flamingo }
    public static void Main() {
        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();
        }
    }
}
```

Это точка входа в программу. Здесь создается объект `StreamWriter`, который передается в класс `Pizza`. Затем члены перечисления `Fargo` в цикле передаются методу `Pizza.Idaho()` для вывода в форму.

```
class Pizza {
    private StreamWriter writer;
    public Pizza(StreamWriter writer) {
        this.writer = writer;
    }
    public void Idaho(Pineapple.Fargo f) {
        writer.WriteLine(f);
        writer.Close();
    }
}
```

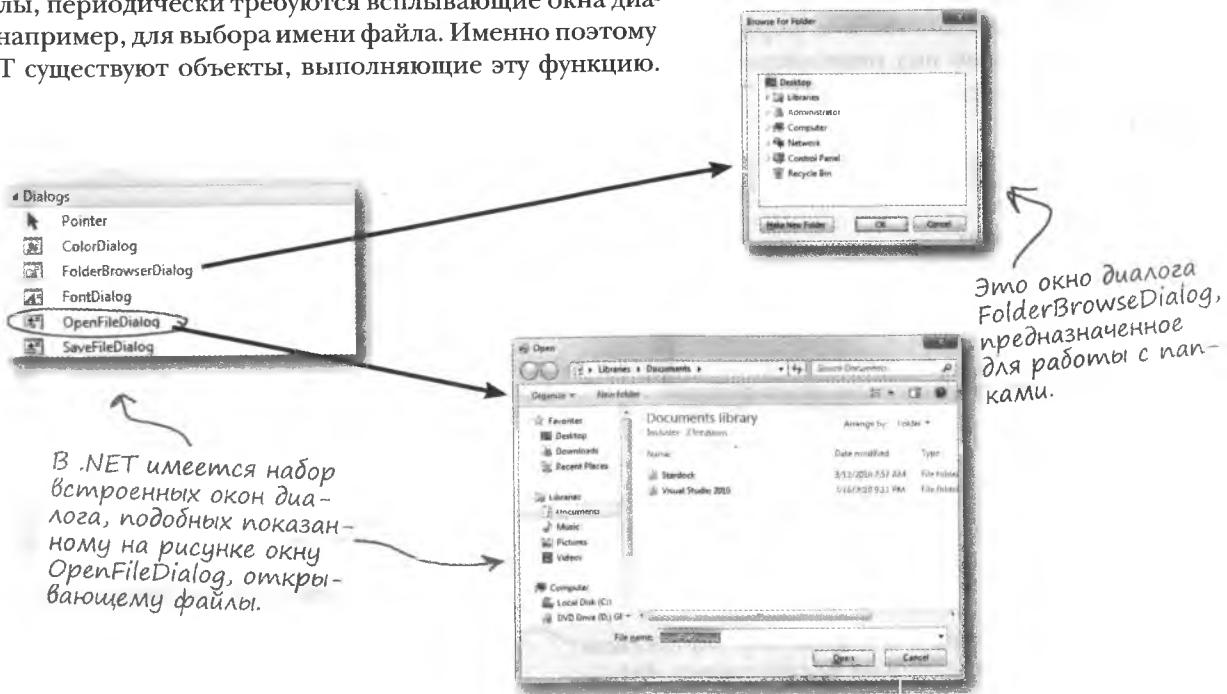
Класс `Pizza` использует объект `StreamWriter` как закрытое поле, а его метод `Idaho()` записывает в файл элементы перечисления `Fargo`, используя их методы `ToString()`, автоматически вызываемые методом `WriteLine()`.

В классе `Party` имеется поле `StreamReader`, и метод `HowMuch()` читает строки, записывая их в поле `StreamWriter`.

```
class Party {
    private StreamReader reader;
    public Party(StreamReader reader) {
        this.reader = reader;
    }
    public void HowMuch(StreamWriter q) {
        q.WriteLine(reader.ReadLine());
        reader.Close();
    }
}
```

## 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:

- 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 элементы .NET

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 и любые другие элементы.

«Невизуальный» означает всего лишь то, что он не появляется на форме после перемещивания из окна Toolbox.



Перемещенные из окна Toolbox на форму компоненты отображаются под редактором формы.

Свойство InitialDirectory определяет папку, которая предлагается для открытия по умолчанию.

```
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);  
}
```

Свойство Filter позволяет менять фильтры, показываемые в нижней части окна диалога, например, определяющие, какие типы файлов будут показываться.

Эти свойства ответственны за появление сообщения об ошибке в случаях, когда пользователь пытается открыть несуществующий файл.

Окно диалога отображается при помощи метода ShowDialog(), возвращающего перечисление DialogResult. Оно проверяет, нажал ли пользователь кнопку OK. В этом случае появляется значение DialogResult.OK. Значение DialogResult.Cancel соответствует кнопке Cancel.

## Окна диалога – это объекты

Объект `OpenFileDialog` показывает стандартное окно Windows Open, а объект `SaveFileDialog` – стандартное окно Save. Их можно отобразить и 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 перемаскивания элемента `SaveFileDialog` из окна Toolbox на форму к методу формы `InitializeComponent()` добавляется эта строка.

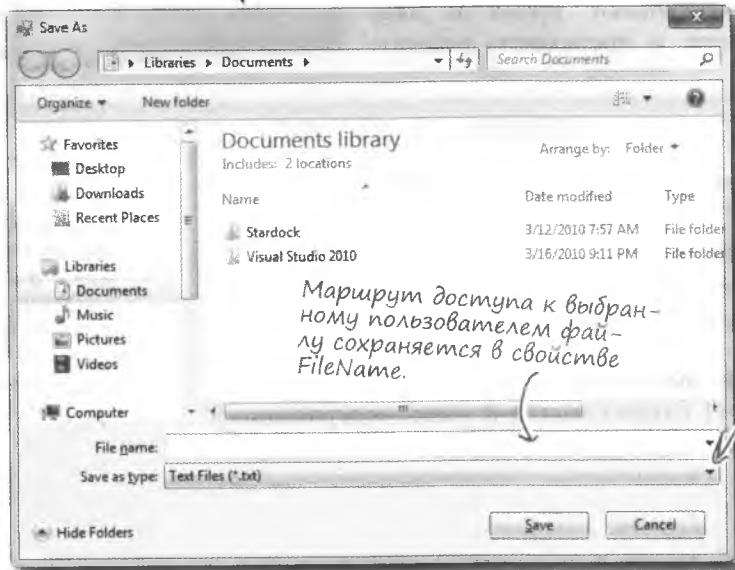
Понять смысл свойства `Filter` несложно. Справление написанное между символами | с тем, что показывается в раскрывающемся списке внизу окна.

Метод `ShowDialog()` и свойство `FileName` имеют те же функции, что и объект `OpenFileDialog`.

Объект `SaveFileDialog` соответствует стандартному окну диалога `Save as...`

Свойство `Title` меняет текст в строке заголовка.

Метод `ShowDialog()` вызывает окно диалога, открытое на панке, заданной свойством `InitialDirectory`.



Редактирование раскрывающегося списка `Save as type` осуществляется при помощи свойства `Filter`.

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 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()`.

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`.

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, `Е` соответствует 69, а `а` — 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 вызвать ее метод `ShowDialog()`, и новое окно диалога готово. Подробно этот процесс будет рассмотрен в главе 13.

**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()`.

**напишите это самостоятельно**

## Возьми в руку карандаш

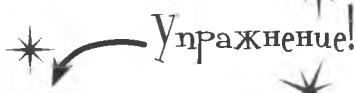


Для работы с файлами и папками в .NET существуют два встроенных класса с многочисленными статическими методами. Класс `File` содержит методы работы с файлами, а класс `Directory` дает возможность работать с папками. Напишите, как, с вашей точки зрения, работают представленные слева строки кода.

Код	Его функция
<pre>if (!Directory.Exists(@"c:\SYP")) {     Directory.CreateDirectory(@"c:\SYP"); }</pre>	
<pre>if (Directory.Exists(@"c:\SYP\Bonk")) {     Directory.Delete(@"c:\SYP\Bonk"); }</pre>	
<code>Directory.CreateDirectory(@"c:\SYP\Bonk");</code>	
<code>Directory.SetCreationTime(@"c:\SYP\Bonk",     new DateTime(1976, 09, 25));</code>	
<pre>string[] files = Directory.GetFiles(@"c:\windows\",     "*.log", SearchOption.AllDirectories);</pre>	
<pre>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?</pre>	
<code>File.Copy(@"c:\SYP\Bonk\weirdo.txt",         @"c:\SYP\copy.txt");</code>	
<pre>DateTime myTime =     Directory.GetCreationTime(@"c:\SYP\Bonk");</pre>	
<code>File.SetLastWriteTime(@"c:\SYP\copy.txt", myTime);</code>	
<code>File.Delete(@"c:\SYP\Bonk\weirdo.txt");</code>	

# 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.



## 1 Создание простой формы.

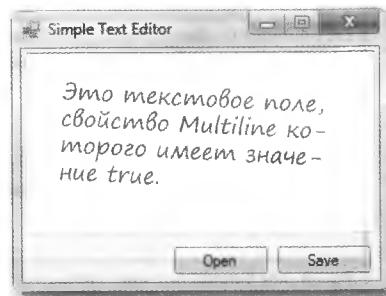
Вам потребуется текстовое поле и две кнопки. Перетащите на форму также элементы OpenFileDialog и SaveFileDialog. Двойным щелчком на кнопках создайте обработчики событий и добавьте закрытое строковое поле name. Добавьте оператор using для System.IO.

## 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 должен показывать только текстовые файлы, с помощью свойства Filter.

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 алтернативный 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 ИСР Go To Definition для просмотра определения интерфейса `IDisposable`. Введите «`IDisposable`» в произвольном месте внутри класса, щелкните на этой строке правой кнопкой мыши и выберите в меню команду Go To Definition. Откроется вкладка с кодом. Вот что вы увидите:

```
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(); ←
    }
}
```

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 ИСР 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.

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.



↓

**При объявлении  
объектов в разделе  
`using` вызов метода  
`Dispose()` таких объ-  
ектов будет осущест-  
вляться автоматически.**

рас-пре-де-лять, гл.  
раздавать ресурсы или  
обязанности для опре-  
деленных целей. Упра-  
ляемый распределил все  
конференц-залы под беспо-  
лезный семинар по менедж-  
менту.

## 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:

За оператором `using` всегда следует объявление объекта...

```
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");  
}  
  
После завершения работы оператора using, вызывается метод Dispose() используемого объекта.
```

...и блок кода в фигурных скобках.

В данном случае на используемый объект указывает ссылка `sw`, объявленная внутри оператора `using`. Поэтому будет запущен метод `Dispose()` класса `Stream`... который и закроет поток.

Эти операторы используют объект, созданный оператором `using`, как и любой другой.

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 StreamWriter("email.txt"))  
{  
    // операторы, использующие устройства чтения и записи  
}
```

Вам больше не требуется метод `Close()`, чтобы закрыть поток, так как оператор `using` закроет его автоматически.

В данном случае речь идет **вовсе не о тех операторах `using`, которые располагаются в верхней части кода**.

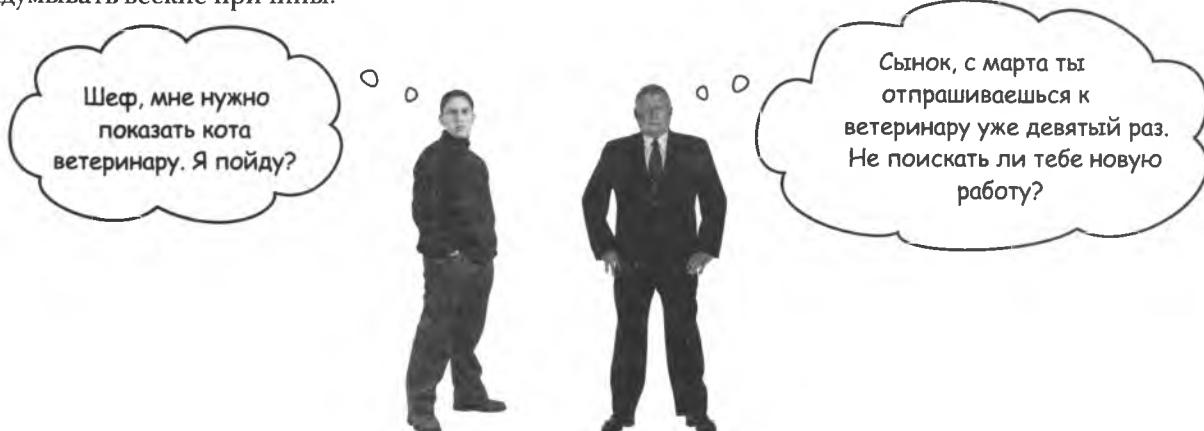
...и блок кода в фигурных скобках.

Эти операторы используют объект, созданный оператором `using`, как и любой другой.

**Потоки ВСЕГДА следует объявлять внутри оператора `using`. Это гарантирует, что после завершения работы они будут закрыты!**

## Проблемы на работе

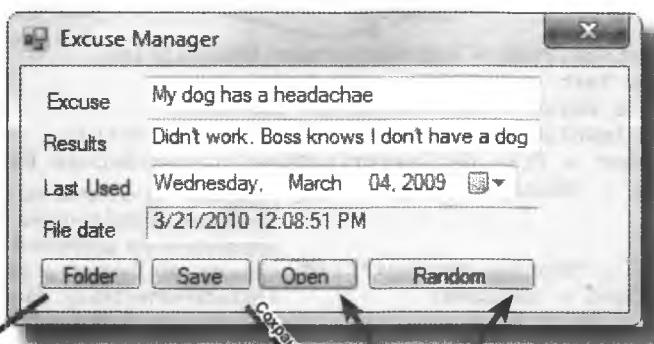
Перед вами Брайан. Он разработчик программ на C# и любит свою работу, но *обожает* устраивать не запланированные выходные. Его начальник **ненавидит** такие ситуации, поэтому Брайану приходится придумывать веские причины.



### 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.

Брайан хочет хранить перечень причин ухода с работы в одном месте, поэтому выделим для него папку.

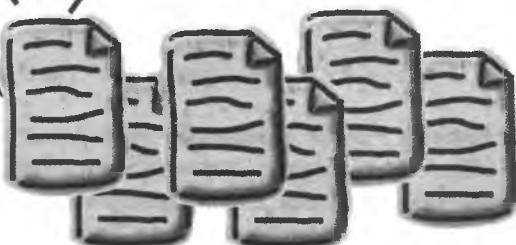


Объяснение: У моей собаки болела голова.  
Результат: Не сработало.  
Начальник знает, что у меня нет собаки.

Иногда Брайану лень выдумывать. Поэтому добавим кнопку Random, которая будет случайным образом выбирать причину прогула.



Папка содержит текстовые файлы, в каждом из которых записано одно оправдание. Щелчок на кнопке Save сохраняет текущее объяснение Брайана в такой файл. Чтобы его открыть, используйте кнопку Open.





## пражнение

Постройте для Брайана программу, выбирающую оправдание.

1

## 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.

2

## 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):

```
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;
}
```

*Помните, что символ ! означает НЕТ – то есть здесь проверяется, не является ли маркером доступа к файлу с оправданием пустым или со значением null.*

*Этот параметр определяет, вносились ли изменения в форму. Вам потребуется поле для его хранения.*

*Дважды щелкните на элементах ввода для создания обработчика событий Changed. Три обработчика событий сначала будут меняться экземпляром Excuse, а затем вызовут метод UpdateForm(true), дальше способ изменения любой формы выбираете только вы.*

Присвойте начальное значение параметру LastUsed в конструкторе формы:

```
public Form1() {
    InitializeComponent();
    currentExcuse.LastUsed = lastUsed.Value;
}
```

3

## 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.

Excuse
Description: string
Results: string
LastUsed: DateTime
ExcusePath: string
OpenFile(string)
Save(string)

*Дважды щелкнув на переташенном на форму текстовом поле, вы создадите для него обработчик события Changed.*

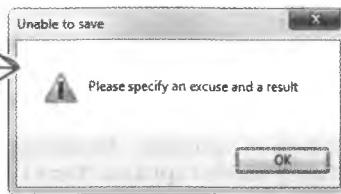
4

#### 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 воскли-  
цательным знаком воспользуйтесь  
перегруженным методом `MessageBox.  
Show()`, который позволяет задавать  
параметр `MessageBoxIcon`.

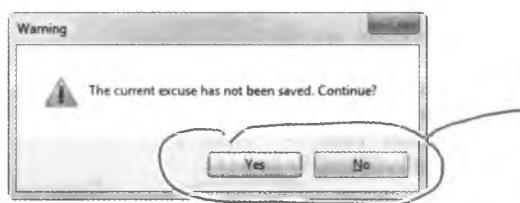


6

#### 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 за-  
давать параметр `MessageBoxButtons.  
YesNo`. При щелчке пользователя на  
кнопке No метод `Show()` возвращает  
`DialogResult.No`.

6

#### 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.



## упражнение

### Решение

```

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 папки форма вателем папки форма сохраняет ее имя и включает три оставшиеся кнопки.

Это символ оператора ИЛИ, выражение имеет значение true (если не указано оправдание) ИЛИ результатом.

Здесь задаются файлы 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(), добавив звездочку в строку заголовка и присвоив 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 единным 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

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 операторов **if/else**:

```
enum BodyPart
{
    Head,
    Shoulders,
    Knees,
    Toes,
}

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;
    }
}
```

Ключевое слово **break**; означает, где заканчивается один оператор **case** и начинается следующий.

Начните с ключевого слова **switch**, за которым следует сравниваемая переменная.

Оператор **switch** не имеет никаких особыхностей, предназначенных для работы с файлами. Это всего лишь полезный инструмент, который мы можем использовать в текущей ситуации.

**Оператор switch сравнивает одну переменную с множеством значений.**

Завершить оператор **case** можно знаком переноса строки. Программа все равно будет компилироваться, так как один оператор **case** заканчивается там, где начинается следующий.

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.



## 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.

```
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 совпадало ни с одним из предложенных вариантов, будут выполнены операторы, стоящие после ключевого слова default.*

**Оператор switch позволяет сравнивать одно значение с набором переменных.**

## 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.

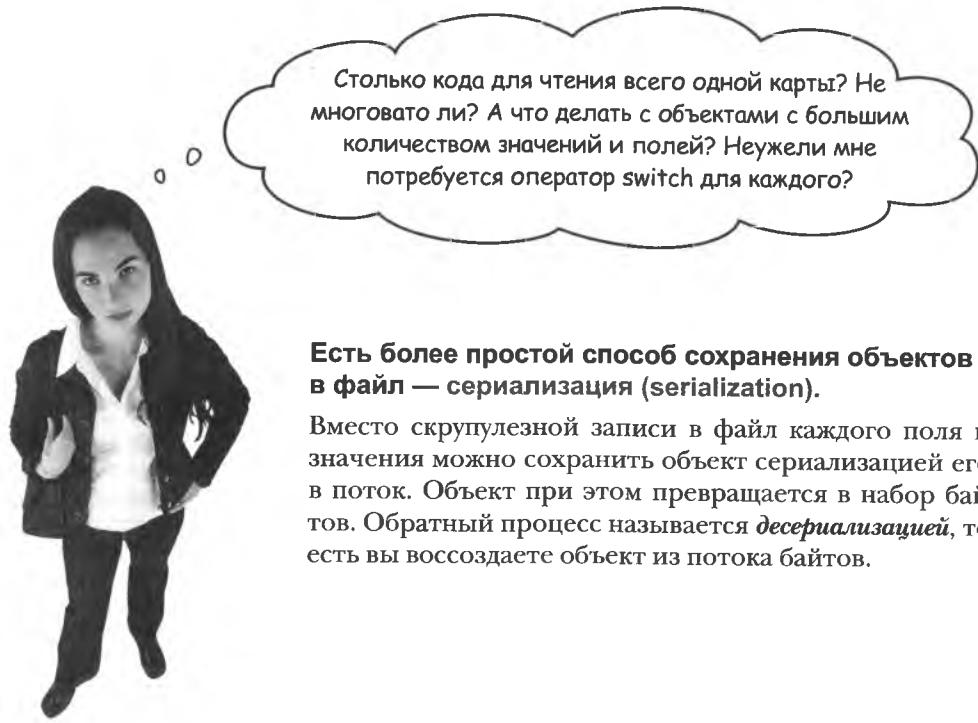
Вот крайне полезный метод Split(). Он разбивает каждую строку на массив более мелких строк, передавая ей массив char[] разделительных знаков.

```
public Deck(string filename) {
    cards = new List<Card>();
    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 совпадшее значение присваивается переменной suit.



**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 в помещение в файл кажется таинственным, на самом деле там все очень просто.

### 1 Объект в куче

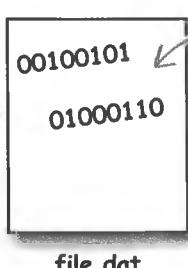


Экземпляр объекта имеет некоторое **состояние**. Объект при этом «знает» только, что делает экземпляр одного класса отличным от другого экземпляра этого же класса.

### 2 Сериализованный объект



При сериализации **полностью сохраняется состояние объекта**, поэтому каждый экземпляр может быть возвращен в кучу.



Экземпляры переменных для ширины и высоты были сохранены в файл `file.dat`, вместе с дополнительной информацией, необходимой для дальнейшего восстановления объекта (например, информацией о типе самого объекта и каждого из его полей).

### Объект снова в куче



### 3 И затем...

Затем, может быть, через много дней и в другой программе, вы можете провести **десериализацию** объекта. Исходный класс будет восстановлен из файла **ровно в то 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 сложней, объект, тем сложнее его сериализация. Как 37, так и 70, всего лишь байты, значимые типы, которые 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 объекты?

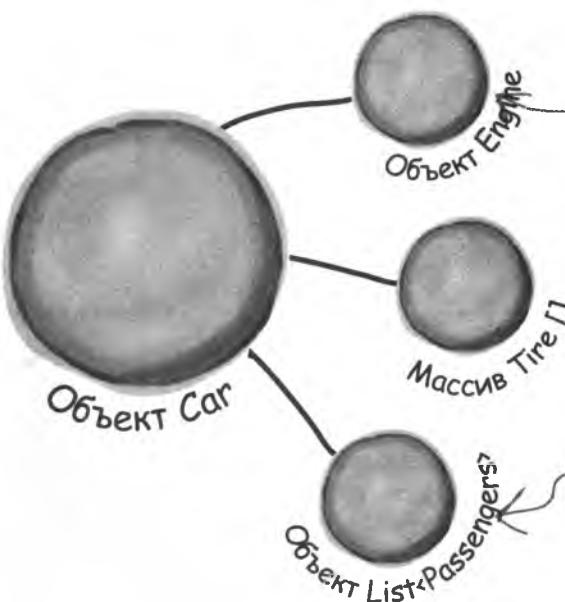
Подумайте об этом. Какая часть объекта является потенциально уникальной? Представьте, что именно нужно восстановить, чтобы получить объект, который был сохранен. Так или иначе, но все содержимое кучи нужно записывать в файл.



### НАПРЯГИ МОЗГИ

Каким образом следует сохранить объект `Car`, чтобы потом его можно было восстановить в исходное состояние? Предположим, что в автомобиле едут три пассажира, он имеет трехлитровый двигатель и всесезонные шины... разве вся эта информация — не часть состояния объекта `Car`? И что с ней делать?

Объект `Car` имеет ссылку на объект `Engine` (Двигатель), массив объектов `Tire` (Шина) и перечисление `List<T>` из объектов `Passenger` (Пассажир). Это составные части его состояния. Что произойдет с ними при сериализации?



Объект `Engine` является закрытым. Нужно ли его сохранять?

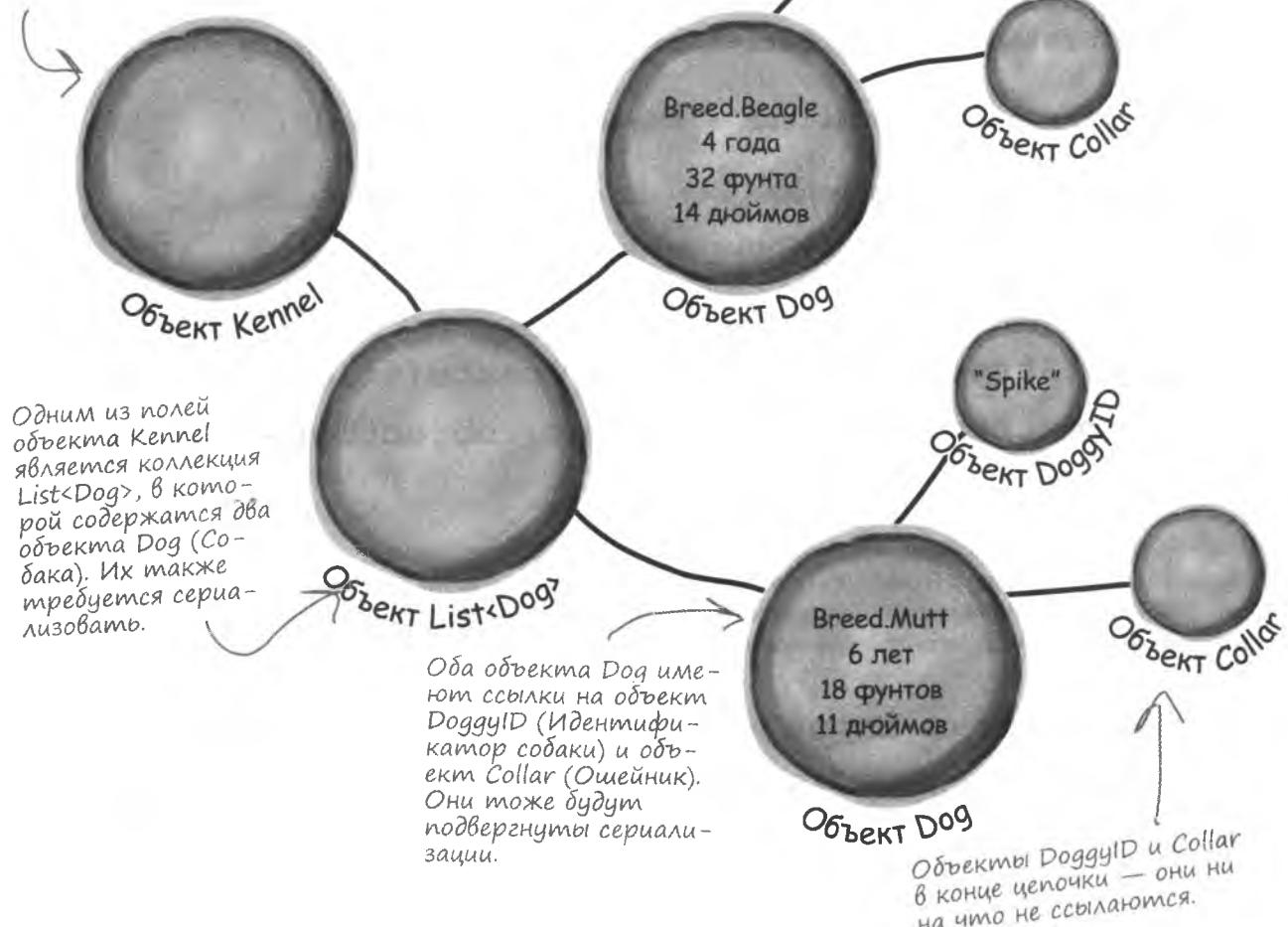
Каждый из объектов перечисления `Passenger` имеет ссылки на другие объекты. Следует ли их сохранять?

## 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»

При сериализации объекта Kennel (Конура) отслеживаются все поля, имеющие ссылки на другие объекты.



## 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.

### 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.

```
using (Stream output = File.Create(filenameString)) {  
    formatter.Serialize(output, objectToSerialize);  
}
```

Метод `File.Create()` создает новые файлы. Для открытия существующего файла используйте метод `File.OpenWrite()`.

Чтобы прочитать сериализованный объект, используйте метод `Deserialize()` объекта `BinaryFormatter`. Он возвращает ссылку, тип которой может не совпадать с переменной, предназначеннной для копирования. В этом случае используйте приведение.

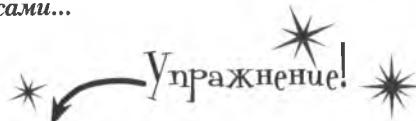
```
using (Stream input = File.OpenRead(filenameString)) {  
    SomeObj obj = (SomeObj)formatter.Deserialize(input);  
}
```

Метод `Serialize()` берет объект и записывает его в поток. Вы избавлены от необходимости делать это вручную!

Используя метод `Deserialize()` для чтения объекта из потока, не забывайте осуществить приведение возвращаемого значения к типу читаемого объекта.

## Атрибут [Serializable]

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, приводят к сообщению об ошибке. Убедитесь в этом сами...



1

### Создание и сериализация класса

Помните класс `Guy` из главы 3? СерIALIZУЕМ Джо, чтобы информация о количестве его наличности осталась в файле.

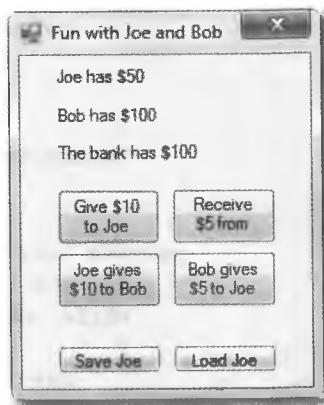
```
[Serializable]
class Guy
```

Этот атрибут должен присутствовать в верхней части любого файла класса, который Вы собираетесь сериализовать.

Этот код сериализует класс в файл `Guy_file.dat`, а также добавляет кнопки `Save Joe` (Сохранить Джо) и `Load Joe` (Загрузить Джо):

```
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` обязательно должны быть. Первая указывает пространство имен, в котором происходят операции ввода и вывода, вторая делает допустимой процедуру сериализации.



2

### Запуск и тестирование программы

Если Джо в результате обменных операций 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.

## 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, скопированным 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.

```
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("-----");
}
```

Создается пустая колода, в которую случайным образом добавляются карты из класса Card, созданного в предыдущей главе.

Метод DealCards() раздает карты из колоды и выводит результат на консоль.

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.

```
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");
}

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");
    }
}
```

Объект `BinaryFormatter` берет объекты, но меченные атрибутом `Serializable` — в нашем случае это объект `Deck` — и записывает их в поток с помощью метода `Serialize()`.

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.

В один поток можно сериализовать несколько объектов.

```
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");
        }
    }
}

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");
        }
    }
}
```

Обратите внимание, как в строке, читающей информацию из файла, осуществляется приведение результатов работы метода `Deserialize()` к объекту `Deck`.

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. Это была бы пустая трата места!

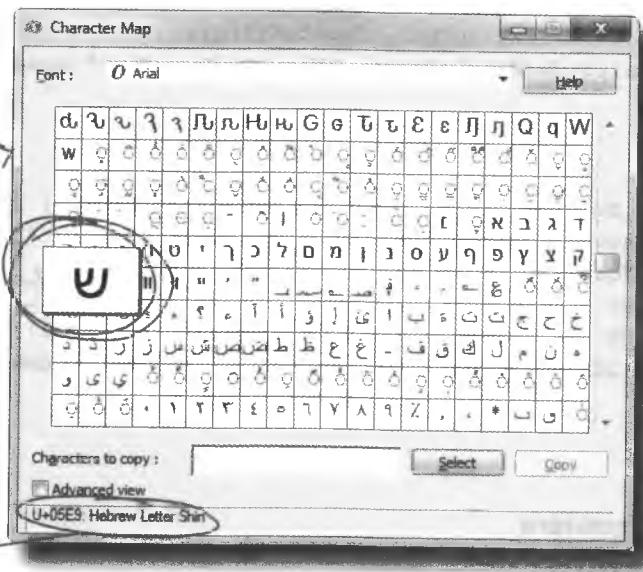
Для представления символов или строк в виде байтов в .NET используется Юникод. К счастью, в Windows имеется инструмент, позволяющий понять принцип работы Юникода. Откройте приложение Character Map (выберите в меню Start команду Run, введите «charmap.exe» и щелкните на кнопке OK).

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 Н) и преобразуются в байты (число 72). Ведь буквы, цифры, перечисления и другие данные хранятся в памяти именно в виде байтов. Узнать же соответствие между числами и символами можно в Таблице символов (Character Map).

Выберите в списке шрифтов Arial и прокрутите вниз 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.



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`, прочитайте в него 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

3

### 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 вашем приложении **Simple Text Editor**, 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.

Этот код создает массив байтов с названием «приветствие», открывает входящий поток и читает данные в байты массива от нулевого до шестого.



```
byte[] greeting;  
greeting = File.ReadAllBytes(filename);
```



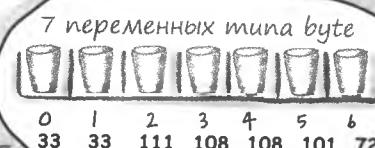
Вот так выглядит в Юникоде слово Hello!!

Это статический метод для объекта Arrays, меняющий порядок байтов на обратный. Здесь он используется, чтобы показать, как внесенные в массив байтов изменения влияют на вид читаемой из файла информации.

```
Array.Reverse(greeting);  
File.WriteAllBytes(filename, greeting)
```



После записи массива байтов в файл текст также оказывается перевернутым.



Порядок байтов поменялся на обратный.

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 фиш?

## Класс BinaryWriter

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. Добавим в программу еще одну кнопку, чтобы посмотреть на способ работы с классом **BinaryWriter()**.

Объект **StreamWriter** так же зашифровывает данные, но он работает только с текстом.

## Упражнение!

- 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.

- При помощи метода **File.Create()** откроем новый поток:

```
using (FileStream output = File.Create("binarydata.dat"))
using (BinaryWriter writer = new BinaryWriter(output)) {
```

- 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

- Используем написанный ранее код для чтения только что записанного.

```
byte[] dataWritten = File.ReadAllBytes("binarydata.dat");
foreach (byte b in dataWritten)
    Console.WriteLine("{0:x2} ", b);
Console.WriteLine(" - {0} bytes", dataWritten.Length);
Console.ReadKey();
```

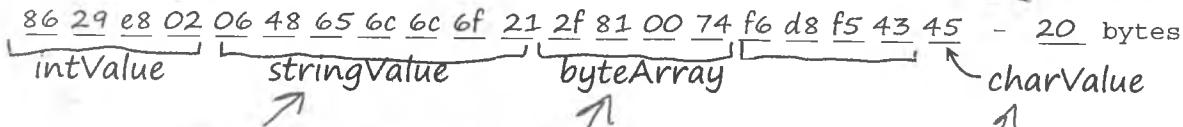
Напишите результат в свободные поля внизу. Вы понимаете, **какие байты соответствуют** каждому из пяти операторов **Write()**? Пометьте каждую группу байтов именем переменной.

**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.

bytes

## смесь байтов

Значения типов float и int занимают по 4 байта, в то время как типы long и double требуют по 8 байт.



## 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 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 еще одну кнопку и займемся чтением только что записанных данных:

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();
```

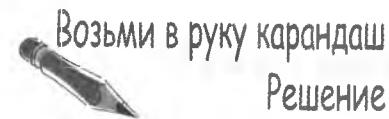
3

Выведем результат чтения данных на консоль:

```
Console.WriteLine("int: {0} string: {1} bytes: ", intRead, stringRead);
foreach (byte b in byteArrayRead)
    Console.Write("{0} ", b);
Console.WriteLine(" float: {0} char: {1} ", floatRead, charRead);
}
Console.ReadKey();
```

Вот как он будет выглядеть:

```
int: 48769414 string: Hello! bytes: 47 129 0 116 float: 491.695 char: E
```



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.

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.

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.

## Чтение и запись сериализованных файлов вручную

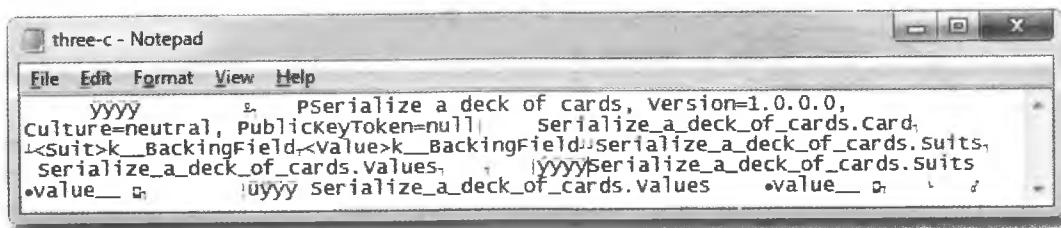
Открытые в приложении Блокнот сериализованные файлы выглядят не очень красиво. Все записанные вами файлы находятся в папке bin\Debug. Давайте посмотрим на дополнительные приемы работы с ними.



### 1 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, имеют одинаковый размер и откроите их в Блокноте:

В файле встречаются знакомые слова, но по большей части он не читаем.



### 2 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 сами 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!

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!

далее →

439

## 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.**

3

### Рассмотрим выведенные на консоль результаты

Они показывают, чем различаются файлы:

```
Byte #322: 1 versus 3  
Byte #382: 3 versus 6
```

Вернитесь к перечислению Suits в предыдущей главе, и вы увидите, что масти Club соответствует значение 1, а масти Heart – значение 3. Это первое различие. Второе различие, как легко догадаться, – старшинство карт. Различному количеству байт 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 #322 в сериализованном файле соответствует масти, мы сможем поменять масть карты, прочитав файл, изменив один байт и снова записав информацию в файл. (Имеите в виду, что в вашем сериализованном файле информация о масти может быть сохранена в другом месте.)

4

### Вручную создадим новый файл с королем пик.

Отредактируем один из прочитанных массивов и снова запишем его в файл.

```
firstFile[322] = (byte) Suits.Spades;  
firstFile[382] = (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.

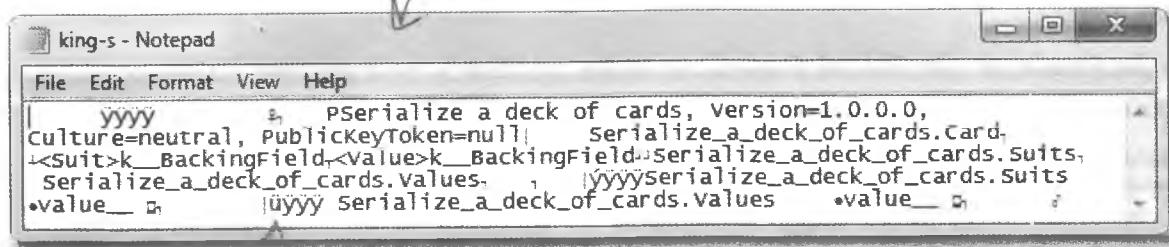
Теперь десериализуйте файл **king-s.dat** и убедитесь, что в нем содержится король пик!

Зная, какие именно байты содержат информацию о масти и значениях 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.

Перед вами сериализованная карта, открытая в Блокноте. Вряд ли кто-то сможет воспользоваться этой информацией.



Понять можно немногое: имена перечислений (Suit и Value), пространство имен, в котором мы работаем (Serialize\_a\_deck\_of\_cards). Но этой информации явно недостаточно.

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:

Можно сразу увидеть численное значение каждого байта.

Число в начале каждой строки указывает на сдвиг первого байта вперед.

A screenshot of a command-line window titled "C:\Windows\system32\cmd.exe". The window displays a hex dump of the serialized data. The left column shows the byte address (e.g., 0000, 0010, ..., 0140) and the right column shows the corresponding hex values. A red arrow points from the text "Число в начале каждой строки указывает на сдвиг первого байта вперед." to the first byte of the first line. Another red arrow points from the text "Вы можете читать исходный текст, а мусор заменен точками." to the right side of the window where the original text is partially visible as dots.

```

0000: 00 01 00 00 00 fd fd fd -- fd 01 00 00 00 00 00 00 .....?Chapter9.
0010: 00 0c 02 00 00 00 3f 43 -- 68 61 70 74 65 72 39 2c Version=1.0.0.0
0020: 20 56 65 72 73 69 6f 6e -- 3d 31 2e 30 2e 30 2e 30 . Culture=neutra
0030: 2c 20 43 75 6c 74 75 72 -- 65 3d 6e 65 75 74 72 61 l. PublicKeyToke
0040: 6c 2c 20 50 75 62 6c 69 -- 63 4b 65 79 54 6f 6b 65 n=null .....Chap
0050: 68 3d 3e 75 6c 6c 05 01 -- 00 00 00 0d 43 68 61 70 ter9.Card....Su
0060: 74 65 72 39 2e 43 61 72 -- 64 02 00 00 00 04 53 75 it.Value....Chapt
0070: 69 74 05 56 61 6c 75 65 -- 04 04 13 43 68 61 70 74 er9.Card+Suits...
0080: 65 72 39 2e 43 61 72 64 -- b2 53 75 69 74 73 02 00 ...Chapter9.Card
0090: 00 00 14 43 68 61 70 74 -- 65 72 39 2e 43 61 72 64 +Values...
00a0: 2b 56 61 6c 75 65 73 02 -- 00 00 00 02 00 00 00 05 .....Chapter9.Ca
00b0: fd fd fd 13 43 68 61 -- 70 74 65 72 39 2e 43 61 rd+Suits...val
00c0: 72 64 2b 53 75 69 74 73 -- 01 00 00 00 07 76 61 6c ue...
00d0: 75 65 5f 5f 00 08 02 00 -- 00 00 01 00 00 00 00 fd .....Chapter9.Car
00e0: fd fd fd 14 43 68 61 70 -- 74 65 72 39 2e 43 61 72 d+Values...val
00f0: 64 2b 56 61 6c 75 65 73 -- 01 00 00 00 07 76 61 6c ue...
0100: 75 65 5f 5f 00 08 02 00 -- 00 00 03 00 00 00 00 02 .....Chapter9.Car
0110: 60 00 00 03 00 00 00 0b -- 73 65 72 69 61 6c 69 7a d+Values...val
0120: 65 72 2e 43 61 72 64 2b -- 56 61 6c 75 65 73 01 00 ue...
0130: 00 00 07 76 61 6c 75 65 -- 5f 5f 00 08 02 00 00 00 .....serializ
0140: 03 00 00 00 0b -- er.Card+Values...

```

## 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 них 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
0010:	20	74	68	65	20	55	6e	69	--	74	65	64	20	53	74	61	74
0020:	65	73	2c	20	69	6e	20	4f	--	72	64	65	72	20	74	6f	20
0030:	66	6f	72	6d	20	61	20	6d	--	6f	72	65	20	70	65	72	66
0040:	65	63	74	20	55	6e	69	6f	--	6e	2e						

We the People of  
the United Stat  
es, in Order to  
form a more perf  
ect Union...

Число в начале 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 шестнадцать 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 про-  
блем 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 блоки читать именно такого размера.

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.

Метод называется **ReadBlock()**, потому что при вызове он «блокируется» (то есть не возвращается в программу, а работает), пока не прочитает все заказанные вами символы или файл до конца.

```

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.WriteLine("{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.

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 име-
ни файла приводит к сообще-
нию об ошибке.

При попытке передать приложению
имя несуществующего файла
появляется сообщение об ошибке.

Дами данных
существующего
файла выводит-
ся на консоль.

Однако вывод на
консоль осущес-
твляется методом
Console.WriteLine().
Но мы восполь-
зовались мето-
дом Console.Error.
WriteLine() для
вывода сообщений
об ошибке.

C:\Windows\system32\cmd.exe
C:\temp>hexdumper
usage: hexdumper file-to-dump

C:\temp>hexdumper does-not-exist.dat
File does not exist: does-not-exist.dat
usage: hexdumper file-to-dump

C:\temp>hexdumper king-s.dat
0000: 00 01 00 00 ff ff ff -- ff 01 00 00 00 00 00 00
0010: 00 0c 02 00 00 00 50 53 -- 65 72 69 61 6c 69 7a 65
0020: 20 61 20 64 65 63 6b 20 -- 6f 66 20 63 61 72 64 73
0030: 2c 20 56 65 72 73 69 6f -- 6e 3d 31 2e 30 2e 30 2e
0040: 30 2c 20 43 75 6a 24 75 -- 72 65 3d 6e 65 75 74 72
0050: 61 6c 2c 20 50 75 62 6c -- 69 63 4b 65 29 54 6f 6b
0060: 65 6e 3d 6e 75 6b 6c 05 -- 01 00 00 00 1e 53 65 72
0070: 69 61 6c 69 7a 65 5f 61 -- 5f 64 65 63 6b 5f 6f 66
0080: 5f 63 61 22 64 73 2e 43 -- 61 72 64 02 00 00 00 15
0090: 3c 53 75 69 24 3e 6b 5f -- 5f 42 61 63 6b 69 6e 67
00a0: 46 69 65 6c 64 16 3c 56 -- 61 6c 75 65 3e 6b 5f 5f
00b0: 42 61 63 6b 69 6e 67 46 -- 69 65 6e 64 04 04 1f 53
00c0: 65 72 69 61 6c 69 2a 65 -- 5f 61 5f 64 65 63 6b 5f
00d0: 6f 66 5f 63 61 72 64 73 -- 2e 53 75 69 24 73 02 00
00e0: 00 00 20 53 65 72 69 61 -- 6c 69 7a 65 5f 61 5f 64
00f0: 65 63 6b 5f 6f 66 5f 63 -- 61 72 64 73 2e 56 61 6c
0100: 75 65 73 02 00 00 00 02 -- 00 00 00 05 fd ff ff ff
0110: 1f 53 65 72 69 61 6c 69 -- 2a 65 5f 61 5f 64 65 63
0120: 6b 5f 66 5f 63 61 72 -- 64 73 2e 53 75 69 24 73
0130: 01 00 00 00 00 07 26 61 6c -- 75 65 5f 5f 00 08 02 00
0140: 00 00 00 00 00 00 00 05 6c -- ff ff ff 20 53 65 72 69
0150: 61 6c 69 7a 65 5f 61 6f -- 64 65 63 6b 5f 6f 66 5f
0160: 63 61 72 64 73 2e 56 61 -- 6c 75 65 73 01 00 00 00
0170: 07 76 61 6c 75 65 5f 6f -- 00 08 02 00 00 00 0d 00
0180: 00 00 0b

C:\temp>

```

Создание консольного приложения сопровождается построением класса `Program` с точкой входа, которая объявляется следующим образом: `static void Main(string[] args)`. В данном случае `args` 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.Substring(0, charactersRead));
            }
        }
    }
}

```

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).

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.

**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 нам 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`?

## часто Задаваемые Вопросы

**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 начале 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 выделяются два байта.

Это используемая в .NET по умолчанию кодировка называется UTF-8. Для смены кодировки нужно передать методу `File.WriteAllText()` другое значение.



## Упражнение

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;
```

Подсказка: Используйте внутри класса ключевое слово, которое возвращает ссылку на этот же класс.

### 2 Отредактируйте метод Excuse.Save()

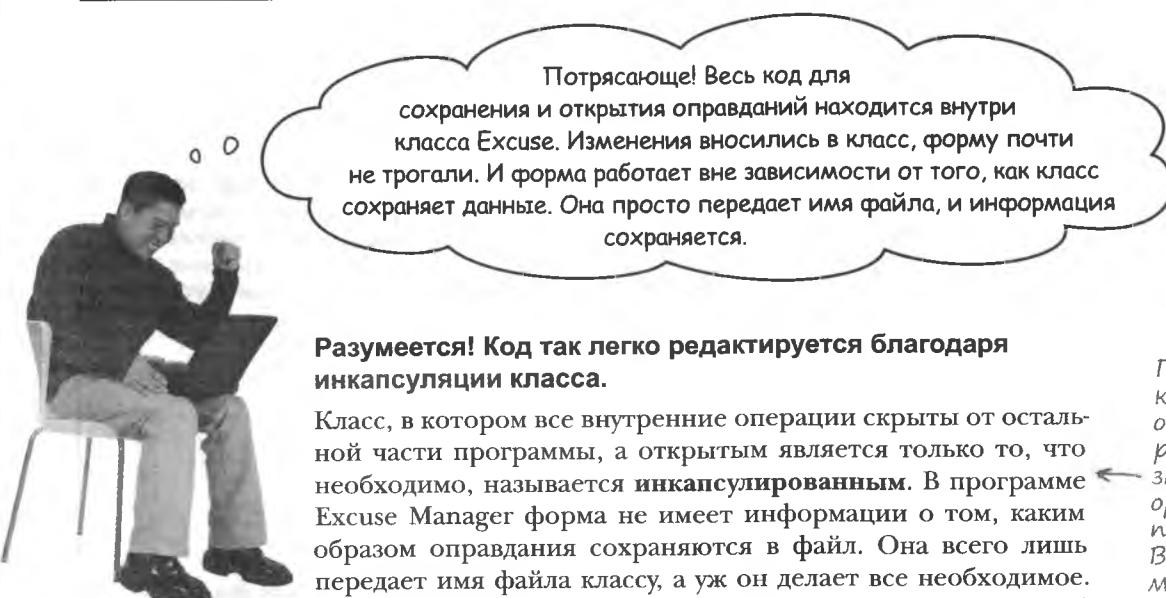
Теперь метод Save() вместо использования объекта StreamWriter должен открывать файл и сериализовывать его. Вам нужно понять, каким образом происходит десериализация текущего класса.

### 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.



### Разумеется! Код так легко редактируется благодаря инкапсуляции класса.

Класс, в котором все внутренние операции скрыты от остальной части программы, а открытый 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.

```
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
}
```

[Serializable] ← Это код всего класса Excuse.

```
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);
        }
    }
}
```

Здесь используются стандартные окна диалога Save и Open.

В форме мы поменяли только расширение файлов, которая она передает в класс Excuse.

Конструктор, загружающий случайные оправдания, теперь ищет расширение .excuse вместо расширения \*.txt.

Ключевое слово this тут фигурирует, так как требуется сериализовать именно этот класс.

## 10 обработка исключений



# Борьба с огнем надоедает

Я написал код обработки для  
Похмелье Исключение.



### Программисты не должны уподобляться пожарным.

Вы усердно работали, штудировали технические 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 выбирает из колеи, как необходимость устранять странные ошибки... но благодаря **обработке исключений** вы сможете написать код, который **сам будет разбираться с возможными проблемами**.

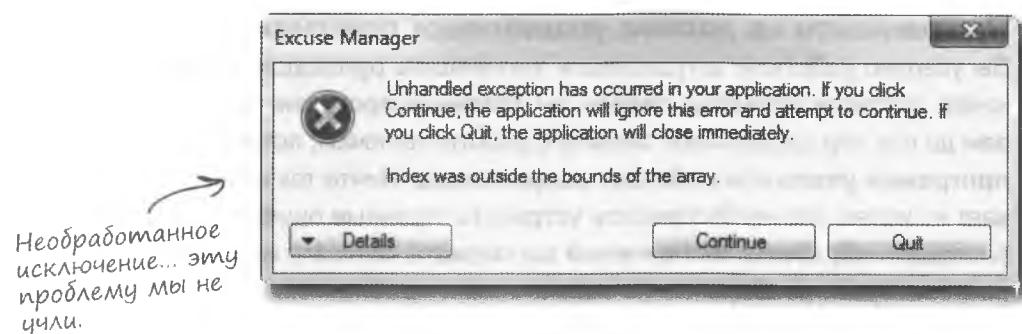
## Брайану нужны мобильные оправдания

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

Бот 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();
    }
}

```

Присвоение  
ссылке  
значения null  
означает, что  
она никуда не  
ведет.

Метод double.Parse("32")  
преобразует строку в  
переменную типа double  
со значением 32.

**OverflowException was unhandled** ①  
Value was either too large or too small for a Single.

**NullReferenceException was unhandled** ②  
Object reference not set to an instance of an object.

**InvalidOperationException was unhandled** ③  
Specified cast is not valid.

**DivideByZeroException was unhandled** ④  
Attempted to divide by zero.  
**Troubleshooting tips:**  
Make sure the value of the denominator is not zero before performing a division operation.  
Get general help for this exception.  
Search for more Help Online...

**FormatException was unhandled** ⑤

Input string was not in a correct format.

**Troubleshooting tips:**

Make sure your method arguments are in the right format.

When converting a string to DateTime, parse the string to take the date before putting each variable into the DateTime object.  
Get general help for this exception.

Search for more Help Online...

## Возьми в руку карандаш

### Решение

Вот какие строки кода стали причиной появления сообщений об ошибках.

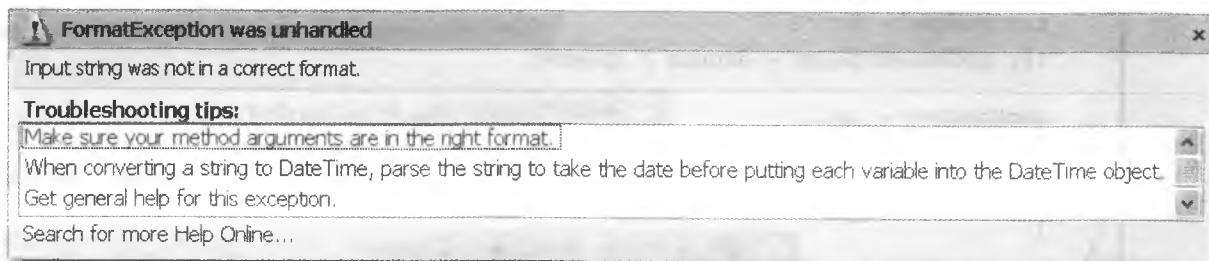
```
object myBee = new HoneyBee(36.5, "Zippo");  
float howMuchHoney = (float)myBee;
```

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.



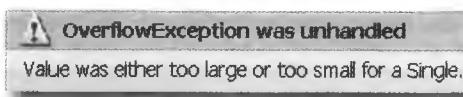
```
HoneyBee anotherBee = new HoneyBee(12.5, "Buzzy");  
double beeName = double.Parse(anotherBee.MyName);
```

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.



```
double totalHoney = 36.5 + 12.5;  
string beesWeCanFeed = "";  
for (int i = 1; i < (int) totalHoney; i++) {  
    beesWeCanFeed += i.ToString();  
}  
float f = float.Parse(beesWeCanFeed);
```

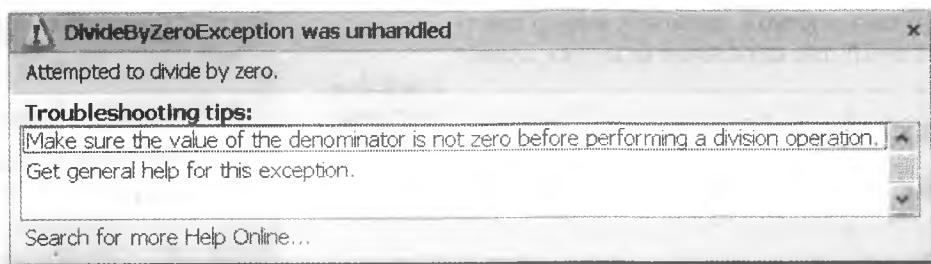
Цикл for создает строку beesWeCanFeed, содержащую число, которое состоит более чем из 60 цифр. Переменной типа float невозможно присвоить столь большое число, именно поэтому мы видим исключение OverflowException.



Разумеется, все эти исключения появляются по очереди, программа выводит первое из них и останавливается. Второе исключение вы можете увидеть, только после того как исправите ошибку, приведшую к первому исключению.

```
int drones = 4;
int queens = 0;
int dronesPerQueen = drones / queens;
```

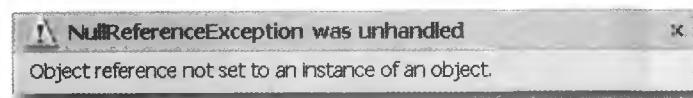
Увидеть такое сообщение об ошибке очень легко.  
Достаточно поделить любую переменную на ноль.



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.



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 шестнадцать (с индексом 15, так как отсчет ведется с нуля):

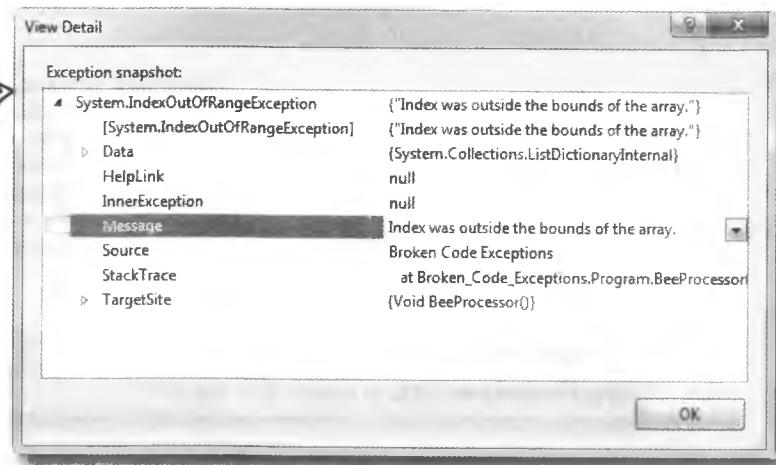
```
int[] anArray = {3, 4, 1, 11};  
int aValue = anArray[15];
```

Очевидно,  
что это  
ошибочный  
код.

При возникновении исключения создается объект, содержащий сведения об ошибке.

В строке `Message` объясняется смысл ошибки, а также содержится список всех обращений к памяти, приведших к событию, ставшему причиной ошибки.

Для просмотра подробной информации щелкните на строчке `View Detail` в окне с сообщением о необработанном исключении.



.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.

**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.

часто  
Задаваемые  
Вопросы

**В:** Почему исключений так много?

**О:** Существует много способов сделать ошибку. В случае общей формулировки («Проблема в строчке 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 или 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 ИСР существует **отладчик**. Он выполняет программу строка за строкой, оператор за оператором, показывает мгновенное значение каждой переменной и каждого поля. Это позволяет обнаружить, где именно код работает не так, как вы предполагали.

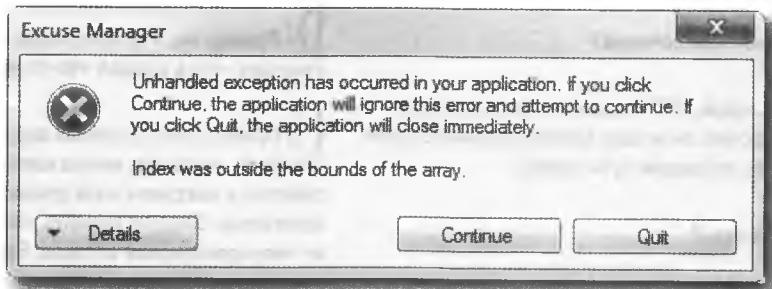
**Исключения помогают обнаружить и исправить код, который работает не так, как вы думали.**

*никто не ожидал, что...*

## **Ког 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 об необработанном исключении:



- 2 Окно сообщает, что индекс вышел за границы массива. Посмотрим на код для обработчика событий кнопки Random Excuse:

```
private void randomExcuse_Click(object sender, EventArgs e) {  
    if (CheckChanged()) {  
        currentExcuse = new Excuse(random, selectedFolder);  
        UpdateForm(false);  
    }  
}
```

- 3 Массивов тут нет. Зато при помощи одного из перегруженных конструкторов создается объект Excuse. Может быть, массив содержится в коде этого конструктора?

```
public Excuse(Random random, string Folder) {  
    string[] fileNames = Directory.GetFiles(Folder, "*.excuse");  
    OpenFile(fileNames[random.Next(fileNames.Length)]);  
}
```

Вот оно! Мы нашли массив. Должно быть, мы попытались указать индекс, выходящий за его границы.

**4**

Оказывается, метод `Directory.GetFiles()`, после того как вы указали на пустую папку, стал возвращать пустой массив. Исправить ситуацию можно, добавив **проверку содержимого папки** перед открытием файла. И вместо окна с информацией о необработанном исключении будет появляться окно с сообщением.

```
private void randomExcuse_Click(object sender, EventArgs e) {
    string[] fileNames = Directory.GetFiles(selectedFolder, "*.excuse");
    if (fileNames.Length == 0) {
        MessageBox.Show("Please specify a folder with excuse files in it",
                        "No excuse files found");
    } else {
        if (CheckChanged() == true) {
            CurrentExcuse = new Excuse(random, Folder);
            UpdateForm(false);
        }
    }
}
```



Проверив наличие файлов с оправданиями в папке до создания объекта `Excuse`, мы предотвращаем появление сообщения об исключении — и вызываем окно с вспомогательной информацией.

Я понял, что исключения — это не всегда плохо. Порой они указывают на ошибки, хотя в большинстве случаев мне просто сообщается, что все идет не так, как я думал.



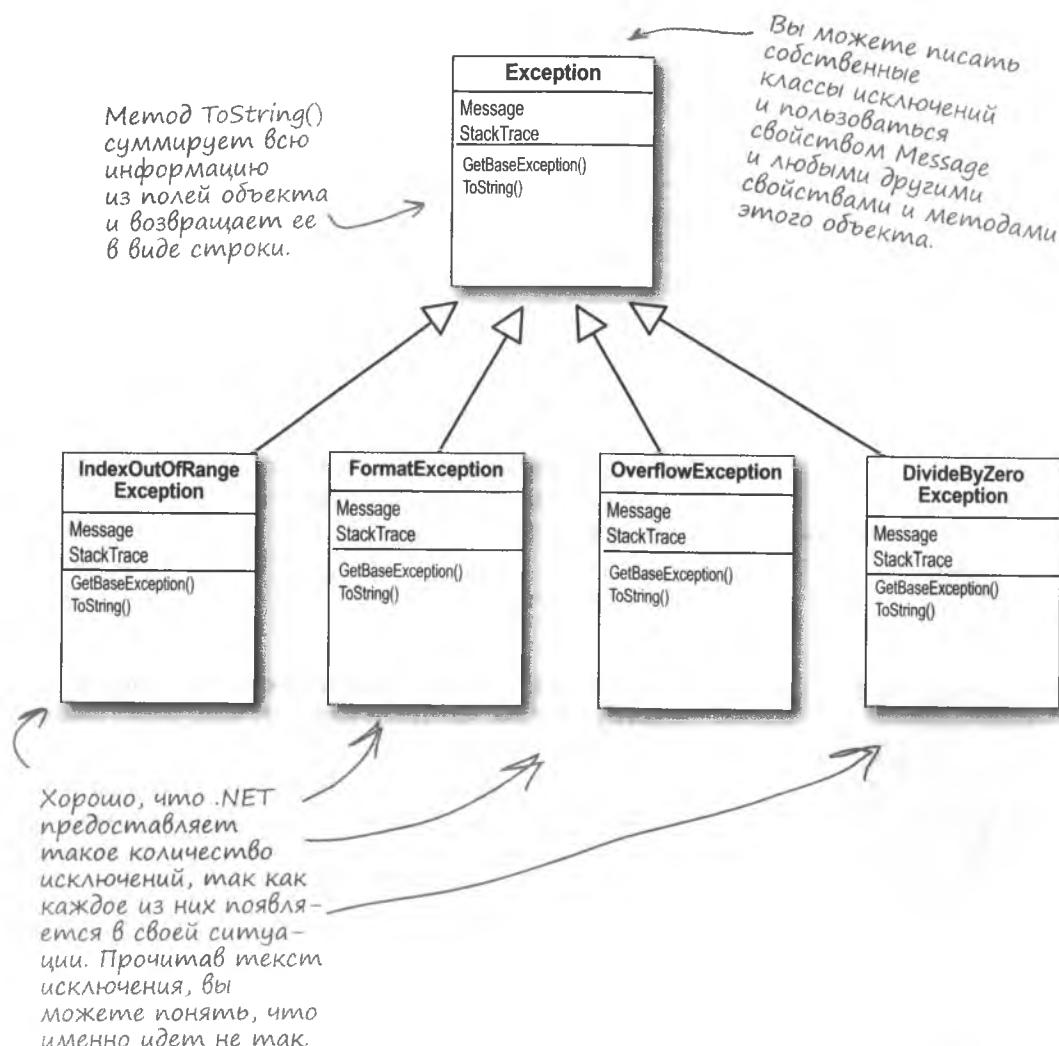
**Именно так. Исключения являются полезным инструментом, который находит код, работающий неожиданным для нас способом.**

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

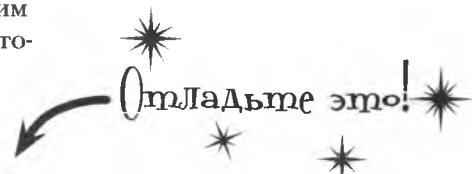
В .NET существует множество исключений. Так как многие из них схожи, имеет смысл говорить о наследовании. .NET определяет базовый класс `Exception`, от которого и наследуют все типы исключений.

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.



## 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

### Точка останова у обработчика событий кнопки Random

Вы знаете, с чего начать: исключение появилось после щелчка на кнопке Random Excuse. Поэтому откройте код этой кнопки, поместите курсор в первую строчку метода и выберите в меню Debug команду Toggle Breakpoint. Запустите программу. Выделите пустую папку и щелкните на кнопке Random для перехода к точке останова:

```
private void randomExcuse_Click(object sender, EventArgs e)
{
    string[] fileNames = Directory.GetFiles(selectedFolder, "*.excuse");
    if (fileNames.Length == 0)    ← fileNames.Length == 0 true
    {
        MessageBox.Show("Please specify a folder with excuse files in it",
                        "No excuse files found");
    }
    else
    {
        if (CheckChanged())
        {
            currentExcuse = new Excuse(random, selectedFolder);
            UpdateForm(false);
        }
    }
}
```

Наведите указатель мыши на свойство fileNames.Length, а когда появится окно, щелкните на кнопке со значком скрепки, чтобы его зафиксировать.

2

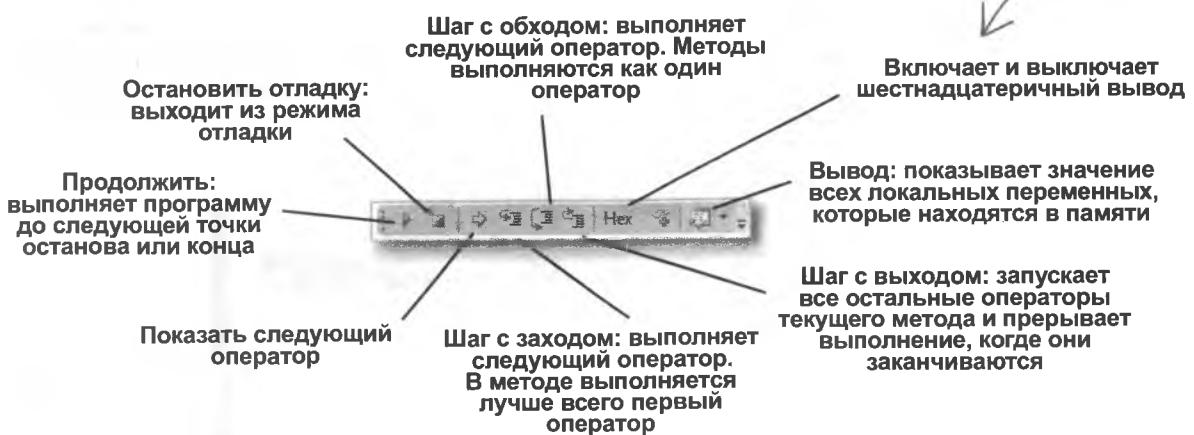
### Обработчик событий и конструктор Excuse

Воспользуйтесь командой Step Into для просмотра программы строка за строкой. Так как вы выделили пустую папку, вы увидите, что после выполнения директивы MessageBox.Show() произойдет выход из обработчика событий.

Теперь выделите папку, содержащую файлы оправданий, и снова щелкните на кнопке Random. Пошагово просмотрите код. (Вы должны пользоваться функцией Step Into, а не Step Over, хотя и имеет смысл обойти метод CheckChanged().) Перед созданием объекта Excuse управление будет передано конструктору. Обойдите первую строчку, так как в ней задается значение переменной fileNames variable. Затем наведите указатель мыши на переменную, чтобы увидеть ее значение.

## Работа с отладчиком

Перед тем как добавлять обработку исключения, нужно понять, какие именно операторы стали причиной его появления. В этом вам поможет встроенный в ИСР отладчик. Вам уже приходилось им пользоваться, но давайте теперь рассмотрим его подробно. После его запуска появляется панель инструментов с кнопками. По очереди наведите курсор на каждую из них и посмотрите на результат:



Панель инструментов Debug появляется только в режиме отладки программы.

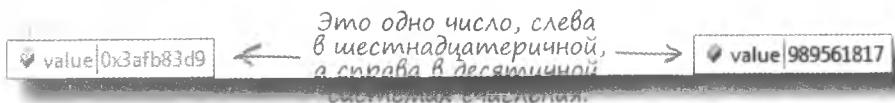
### Bug панели Debug в режиме Expert

По умолчанию Visual Studio 2010 Express находится в режиме Basic Settings, который замечательно подходит для начинающих. Но теперь давайте перейдем к расширенным параметрам. Для этого выберите в меню Tools команду **Settings > Expert Settings** (переход в другой режим может занять некоторое время). После этого на панели инструментов Debug вы обнаружите две новые кнопки:



### Шестнадцатеричный вывод

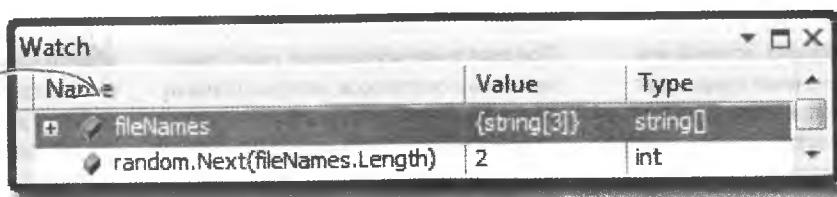
Нажмите Нех для включения режима шестнадцатеричного вывода и наведите курсор на любое поле или переменную. Снова щелкните на кнопке для возвращения в обычный режим. Преобразование значения в шестнадцатеричную систему счисления будет осуществлено автоматически. В прошлой главе вы узнали, зачем это нужно.



**3****Работа с окном Watch**

Шелкните правой кнопкой мыши на переменной `fileNames` и выберите команду **Expression: 'fileNames' >> Add Watch**. Затем щелкните на пустой строчке под переменной `fileNames` и введите `random.Next(fileNames.Length)`, чтобы отладчик добавил контрольное значение. Вот как будет выглядеть окно Watch для папки с тремя файлами оправданий (когда переменная `fileNames` имеет длину 3).

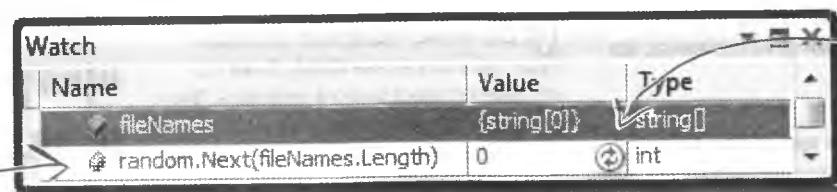
Окно Watch используется для воспроизведения ситуации, приведшей к появлению исключения. Мы начали с добавления массива `fileNames`.

**4****Присвоим переменной `fileNames` пустой строковый массив**

Окно Watch позволяет менять значения отображаемых переменных и полей. Вы даже можете выполнять методы и создавать новые объекты, в эти моменты появляется значок (⌚), щелчок на котором приводит к повторному выполнению строки. Ведь метод Random каждый раз дает другой результат.

Дважды щелкните на строчке `fileNames`, чтобы выделить текст `{string[3]}`. Замените его на `new string[0]`. Тут же исчезнет квадратик со знаком «плюс» рядом с переменной `fileNames`, так как теперь этот массив пуст. А рядом со строчкой `random.Next()` появится значок (⌚). Щелкните на нем, чтобы еще раз выполнить метод. Он должен вернуть 0.

Так как источником проблем являемся пустой массив `fileNames`, мы смоделируем эту ситуацию в окне Watch.

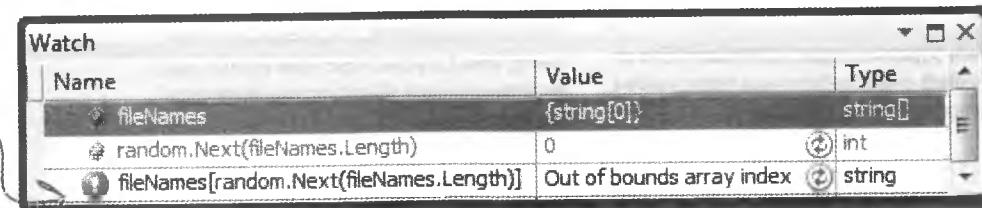


Этот значок инициирует повторное выполнение метода `Next()`.

**5****Воссоздание проблемной ситуации**

Добавьте в отладчик строку с оператором, который стал причиной исключения: `fileNames[random.Next(fileNames.Length)]`. Это выражение будет обсчитываться в окне Watch... и вы увидите сообщение об исключении. Слева появится значок с восклицательным знаком, а в столбце Value – текст исключения.

Этот значок в окне Watch сообщает об обнаружении исключения.



**Вы всегда можете воспроизвести с помощью отладчика ситуацию, ставшую причиной появления исключения.**

часто  
Задаваемые  
Вопросы

**В:** Почему сообщение о необработанном исключении, которое получил Брайан, отличается от моего?

**О:** Запуская программу в ИСР, вы работаете из отладчика, который при обнаружении исключения **прерывает выполнение** (как при нажатии кнопки Break All или при вставке точки останова). При этом появляется окно с сообщением. Это позволяет исследовать объект `Exception`, а также поля и переменные вашей программы, и понять, в чем именно состоит проблема.

Программа, с которой работал Брайан, была установлена на его компьютер. Помните, вы делали это в главе 1 для приложения со списком контактов? Для запуска программы вне ИСР достаточно построить решение, то есть получить исполняемый файл, который находится в папке `bin/`, вложенной в папку проекта. После этого вы будете видеть такие же окна с сообщениями об исключениях, как и Брайан.

**В:** То есть если программа работает вне ИСР, с появлением сообщения об исключении ее выполнение просто останавливается, и пользователь ничего не может сделать?

**О:** Да, программа останавливается, столкнувшись с **необработанным** исключением. Но ведь нигде не сказано, что все исключения относятся к необработанным! О способах обработки исключений и о том, каким образом создать программу без сообщений о необработанных исключениях, мы поговорим чуть позднее.

**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 последствием. 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, любые корректные операторы будут работать в окне Watch. Даже те, которые не имеет смысла запускать. Запустите программу, прервите ее выполнение и добавьте в окно Watch строку: `System.Threading.Thread.Sleep(2000)`. (Как вы помните, этот метод вызывает двухсекундную задержку работы программы.) Использовать его нет смысла, но интересно посмотреть, что произойдет: на две секунды, которые занимает выполнение этого метода, появится изображение песочных часов. Так как метод `Sleep()` не возвращает значения, в окне Watch появится сообщение, `Expression has been evaluated and has no value`, информирующее об отсутствии возвращаемых значений.

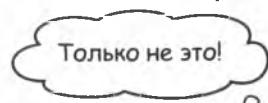
**В:** То есть я могу запустить в окне Watch что-то, меняющее способ работы программы?

**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 поменять 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** (этот шутка понята только физикам и котам в ящике).

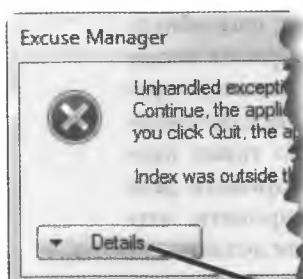
**Запущенная в ИСР  
программа при  
появлении необ-  
работанных иску-  
чений прерывается,  
как при достиже-  
нии точки останова.**

## А как все равно не работает...

Брайан успешно пользовался Excuse Manager при наличии папки, наполненной файлами с оправданиями, которую он создал во время написания программы, но он забыл про эту папку  *перед*  сериализацией программы. И вот что случилось....



- ➊ Приложение Блокнот позволяет воссоздать файлы с оправданиями. Первая строка — оправдание, вторая — результат его применения, третья — дата его последнего использования («10/4/2007 12:08:13 PM»).
- ➋ Вызовите Excuse Manager и откройте оправдание. При появлении исключения щелкните на кнопке Details. Обратите внимание на **стек вызова** — именно так называется метод, вызывающий другой метод, который, в свою очередь, тоже вызывает метод и т. д.



Кажется, у нас проблема с классом **BinaryFormatter**. Это предположение имеет смысл, так как приложение пытается десerialize текстовый файл.

Многое можно узнать из **стека вызова**, ведь там показывается список запущенных методов. Вы видите, что метод **OpenFile()** класса **Excuse** вызывается его конструктором (**.ctor**), который, в свою очередь, вызывается обработчиком событий кнопки **Random Excuse**.

Программа сообщает, что исключение появляется при сериализации. Можно ли из подробного описания понять, какая строка кода является причиной его появления?

```
***** Exception Text *****

System.Runtime.Serialization.SerializationException: End of Stream encountered before parsing was completed.

at System.Runtime.Serialization.Formatters.Binary._BinaryParser.Run()
at System.Runtime.Serialization.Formatters.Binary.ObjectReader.Deserialize(HeaderHandler handler, _BinaryParser serParser, Boolean fCheck, Boolean isCrossAppDomain, IMethodCallMessage methodCallMessage)
at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream serializationStream, HeaderHandler handler, Boolean fCheck, Boolean isCrossAppDomain, IMethodCallMessage methodCallMessage)
at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream serializationStream)
at Chapter10.Excuse.OpenFile(String ExcusePath) in C:\Documents and Settings\Administrator\My Documents\Visual Studio 2005\Projects\Chapter10\Chapter10\Excuse.cs:line 40
at Chapter10.Excuse..ctor(Random random, String Folder) in C:\Documents and Settings\Administrator\My Documents\Visual Studio 2005\Projects\Chapter10\Chapter10\Excuse.cs:line 30
at Chapter10.Form1.randomExcuse_Click(Object sender, EventArgs e) in C:\Documents and Settings\Administrator\My Documents\Visual Studio 2005\Projects\Chapter10\Chapter10\Form1.cs:line 146
```

- ➌ Щелчок на кнопке Details позволяет многое узнать о причинах возникшей проблемы. Есть идеи, что со всем этим делать?

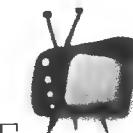


Разумеется, программа должна была прекратить работу, ведь я дала ей не тот файл. Пользователи все время делают что-то не то, и с этим невозможно бороться.

### 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 не делали.

**ро-баст-ный, прил.**  
прочный; способный  
противостоять неблаго-  
приятным условиям.



Будьте | **Класс BinaryFormatter сообщает  
осторожны!** об исключении в случае  
проблем с сериализован-  
ным файлом.

Получить такое исключение для программы *Excuse Manager* проще простого — дайте ей файл, который не является сериализованным объектом *Excuse*. Класс *BinaryFormatter* ожидает, что файл будет содержать сериализованный объект нужного типа. Если же файл содержит что-то другое, метод *Deserialize()* отобразит исключение *SerializationException*.

## Ключевые слова try и catch

Происходящее в C# можно описать фразой «Протестируйте (try) этот код и при появлении исключения прервите (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.

Обработка исключения начинается с ключевого слова **try**.  
В данном случае мы поместили за ним существующий код.

```
private void randomExcuse_Click(object sender, EventArgs e)
{
    // ... здесь код, добавленный несколько страниц назад...

    try {
        if (CheckChanged() == true) {
            currentExcuse = new Excuse(random, selectedFolder);
            UpdateForm(false);
        }
    }

    catch (SerializationException) {
        MessageBox.Show(
            "Your excuse file was invalid.",
            "Unable to open a random excuse");
    }
}
```

Ключевое слово **catch** означает, что следующий за ним блок операторов содержит **обработчик исключения**.

Код, который может стать причиной исключения, поместите внутрь блока **try**. Если исключение не появляется, выполнение пойдет обычным образом, а операторы блока **catch** будут проигнорированы. Но если какой-то из операторов станет причиной исключения, остальные операторы в блоке **try** выполняться не будут.

Исключение вызывает немедленный переход к операторам блока **catch**.

Это простейший способ обработки исключения: остановить программу, написать сообщение и затем продолжить выполнение программы.



Если исключение немедленно передает управление операторам блока **catch**, что происходит с объектами и данными, с которыми вы работали до этого?

## Вызов сомнительного метода

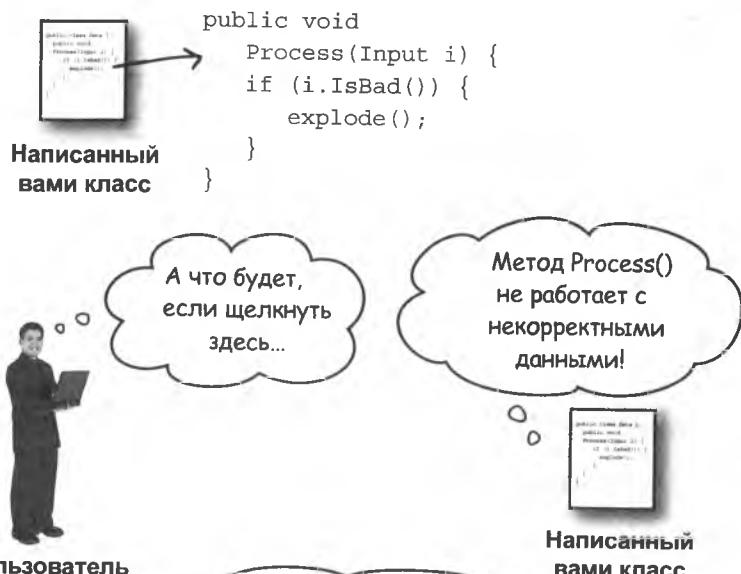
Пользователи непредсказуемы. Они вводят в программу странные данные, щелкают на кнопках. И это прекрасно, ведь вы можете справиться с последствиями ввода таких данных, обрабатывая исключения.

- ① Предположим, пользователь вводит не те данные.**



- ② Метод делает что-то странное, что может и не сработать во время прогона.**

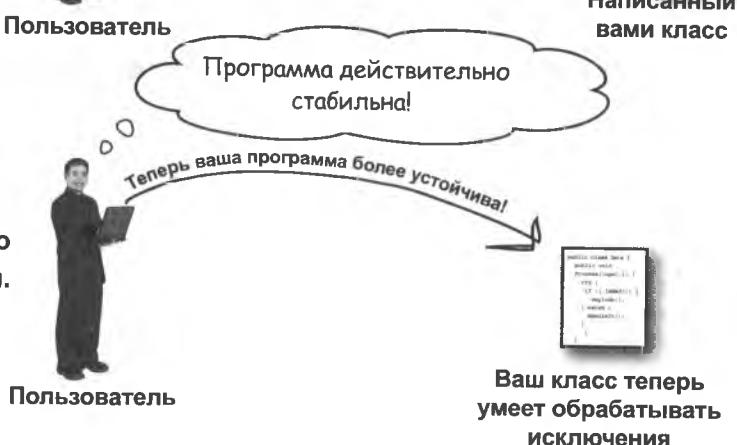
Временем прогона (Runtime) называется время работы вашей программы. Исключения иногда еще называют «ошибками при исполнении» (runtime errors).



- 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...*



- Весь предоставленный в этой главе код нужно встроить в обработчик событий кнопки Random Excuse. Точку останова следует поместить в первую строчку. И запустить программу. Щелкните на кнопке Folder и выберите папку с единственным файлом в ней. Убедитесь, что это некорректный файл с оправданием (несмотря на расширение .excuse). Щелкните на кнопке Random Excuse, а затем шесть раз щелкните на кнопке Step Over (или нажмите F10) для перехода к оператору, вызывающему конструктор Excuse. Вот как должен выглядеть ваш экран на этом этапе:

Это точка останова, которую мы поместили в первую строчку обработчика событий.

Нажмайте F10, пока не дойдете до строки, предшествующей созданию объекта Excuse.

```
private void randomExcuse_Click(object sender, EventArgs e)
{
    string[] fileNames = Directory.GetFiles(selectedFolder, "*.excuse");
    if (fileNames.Length == 0)
    {
        MessageBox.Show("Please specify a folder with excuse files in it",
                        "No excuse files found");
    }
    try
    {
        if (CheckChanged() == true)
        {
            currentExcuse = new Excuse(random, selectedFolder);
            UpdateForm(false);
        }
    }
    catch (SerializationException)
    {
        MessageBox.Show(
            "Your excuse file was invalid.",
            "Unable to open a random excuse");
    }
}
```

Используйте команду Step Over (F10) для обхода метода CheckChanged().

- Нажмите F11 для просмотра оператора `new`. Отладчик поместит желтую полоску над строкой объявления в коде конструктора `Excuse`. Нажмите F11, чтобы попасть в метод `OpenFile()` и посмотреть, что случится на строчке `Deserialize()`.

После оператора `new` отладчик начинает просматривать код конструктора.

```
public Excuse(Random random, string folder)
{
    string[] fileNames = Directory.GetFiles(folder, "*.excuse");
    OpenFile(fileNames[random.Next(fileNames.Length)]);
}
```

3

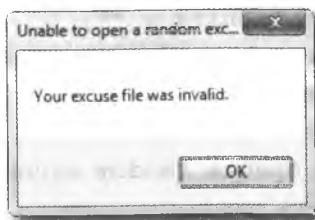
- После выполнения оператора Deserialize() появится исключение, и программа, проигнорировав вызов метода UpdateForm(), перейдет непосредственно к блоку catch.

Отладчик выделит желтым строчку с ключевым словом catch, в то время как остальной код блока будет помечен серым, то есть готовым к выполнению.

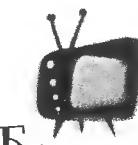
```
private void randomExcuse_Click(object sender, EventArgs e)
{
    string[] fileNames = Directory.GetFiles(selectedFolder, "*.excuse");
    if (fileNames.Length == 0)
    {
        MessageBox.Show("Please specify a folder with excuse files in it",
                        "No excuse files found");
    }
    try
    {
        if (CheckChanged() == true)
        {
            currentExcuse = new Excuse(random, selectedFolder);
            UpdateForm(false);
        }
    }
    catch (SerializationException)
    {
        MessageBox.Show(
            "Your excuse file was invalid.",
            "Unable to open a random excuse");
    }
}
```

4

- Нажмите клавишу F5 для запуска программы. Она начнет выполняться со строчки, помеченной желтым, в нашем случае с блока catch.



Совет на будущее: Иногда с поисками на должность программиста часто включают в себя вопрос о том, как следует поступать с исключениями в конструкторе.



Будьте

осторожны!

**Аккуратнее 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 этот 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 нужно закончить обработчик событий кнопки Random Excuse:

```
private void randomExcuse_Click(object sender, EventArgs e) {
    string[] fileNames = Directory.GetFiles(selectedFolder, "*.excuse");
    if (fileNames.Length == 0) {
        MessageBox.Show("Please specify a folder with excuse files in it",
                        "No excuse files found");
    } else {
        try {
            if (CheckChanged() == true) {
                currentExcuse = new Excuse(random, selectedFolder);
            }
        }
        catch (SerializationException) {
            currentExcuse = new Excuse();
            currentExcuse.Description = "";
            currentExcuse.Results = "";
            currentExcuse.LastUsed = DateTime.Now;
            MessageBox.Show(
                "Your excuse file was invalid.",
                "Unable to open a random excuse");
        }
        finally {
            UpdateForm(false);
        }
    }
}
```

Блок `finally` гарантирует, что метод `UpdateForm()` будет выполнен вне зависимости от исключения. Согласно, метод `UpdateForm()` будет вызван не только если конструктор `Excuse` успешно прочитал оправдание, но и в случае, когда появилось исключение и файл с оправданием был очищен.

При появлении исключения в конструкторе `Excuse` устанавливается значение переменной CurrentExcuse невозможно, так как экземпляр `Excuse` отсутствует. Поэтому в блоке `catch` создается этот объект и очищаются все его поля.

`SerializationException` находится в пространстве имен `System.Runtime.Serialization`, поэтому в верхнюю часть файла формы нужно добавить строчку `using System.Runtime.Serialization;`

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 обработчик событий кнопки Random Excuse, вставив туда код с предыдущей страницы. Поместите точку останова на первую строку метода и отладьте программу.
- 2** Запустите программу и убедитесь, что когда программа указывает на папку с файлами оправданий, кнопка Random Excuse работает корректно. Программа прервется на заданной вами точке останова:

Когда желтая строка достигает точки останова, над красной точкой на полях появляется желтая стрелка.

```

private void randomExcuse_Click(object sender, EventArgs e)
{
    string[] fileNames = Directory.GetFiles(selectedFolder, "*.excuse");
    if (fileNames.Length == 0)
    {
        MessageBox.Show("Please specify a folder with excuse files in it",
                        "No excuse files found");
    }
    else
    {
        try
        {
            if (CheckChanged() == true)
            {
                currentExcuse = new Excuse(random, selectedFolder);
            }
        }
        catch (SerializationException)
        {
            currentExcuse = new Excuse();
            currentExcuse.Description = "";
            currentExcuse.Results = "";
            currentExcuse.LastUsed = DateTime.Now;
            MessageBox.Show(
                "Your excuse file was invalid.",
                "Unable to open a random excuse");
        }
        finally
        {
            UpdateForm(false);
        }
    }
}

```

- 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 укажите папку, содержащую всего один дефектный файл, и снова щелкните на кнопке Random Excuse. Из блока `try` при обнаружении исключения управление перейдет к блоку `catch`. После того как будут просмотрены все его операторы, начнет выполняться блок `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:** Безопасней 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 там 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.

**В:** Что произойдет, если не указать тип исключения рядом с ключевым словом `catch`?

**О:** Блок `catch` будет работать с любым исключением, которое появится в блоке `try`.

**В:** Но если блок `catch` может обрабатывать исключения любых типов, зачем мне указывать конкретный тип?

**О:** К сожалению, 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. Можно воспользоваться набором блоков `catch`, если вы хотите учесть различные виды ошибок. Этого блока может и не быть вовсе. В этом случае исключения обрабатывать не будут, просто код блока `finally` начнет выполняться даже при остановке из-за появления исключения в блоке `try`.

**Необработанное исключение означает неизвестную работу программы. Именно поэтому программа перестает работать при их появлении.**

# Ребус в бассейне



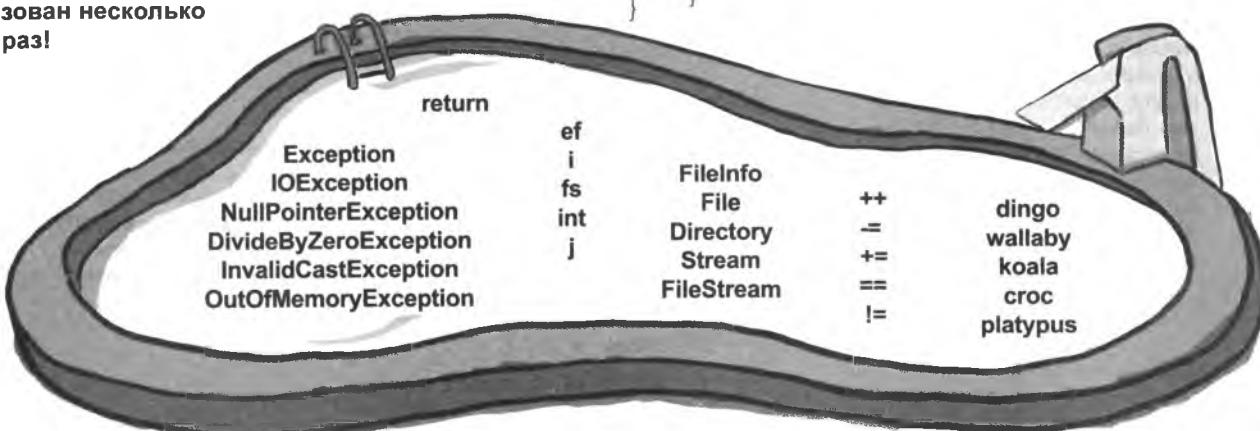
Поместите фрагменты кода из бассейна на пустые строчки программы. Каждый фрагмент может использоваться несколько раз. Имеются и лишние фрагменты. Нужно получить следующую строку.

**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!");
    }
}
```

Каждый фрагмент  
может быть исполь-  
зован несколько  
раз!

```
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;
            }
        }
        _____;
    }
}
```



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!

Метод Joey.Wombat() вызывался три раза и на третий раз вернул ноль. Поэтому метод WriteLine() вызвал исключение DivideByZeroException.

## Решение ребуса в бассейне



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.

```

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.

Вы уже знаете, что после завершения работы с файлом его требуется закрыть, иначе файл окажется заблокированным. Попытка снова его открыть станет причиной исключения IOException.

## Получения информации о проблеме

Мы уже несколько раз повторили, что при появлении исключения .NET создает объект `Exception`. Доступ к нему вы получаете через код блока `catch`. Вот как это работает:

- 1 Некий объект выполняет некие функции, и вдруг нештатная ситуация приводит к появлению исключения.

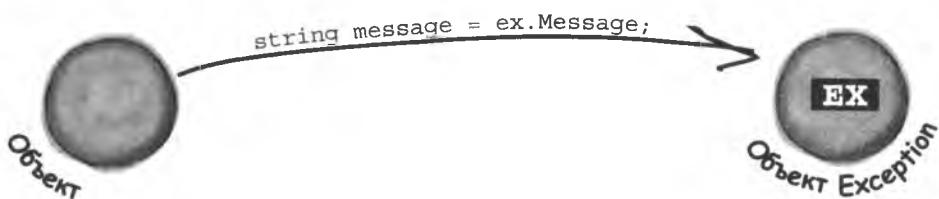


- 2 К счастью, срабатывает блок `try/catch`. Внутри блока `catch` исключению присваивается имя `ex`.

```
try {
    DoSomethingRisky();
}
catch (RiskyThingException(ex)) {
    string message = ex.Message;
    MessageBox.Show(message, "Я сильно рисковал!");
}
```

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()` исключения позволяет вывести в окно `MessageBox` относящиеся к делу данные.

```

public void ProcessNectar(NectarVat vat, Bee worker, HiveLog log) {
    Иногда try {
        требует-     NectarUnit[] units = worker.EmptyVat(vat);
        ся повтор-     for (int count = 0; count < worker.UnitsExpected, count++) {
        но вызвать     stream hiveLogFile = log.OpenLogFile();
        перехваченное     worker.AddLogEntry(hiveLogFile);
        исключение.     }
        Для этого     } Если вы не собираетесь исполь-
        использует-     зовать объект Exception, объяв-
        ся оператор     лять его не нужно.
        throw.     }
        catch (VatEmptyException) {
            vat.Emptied = true;
        }
        catch (HiveLogException ex) {
            throw;
        }
        catch (IOException ex) {
            Для объекта     worker.AlertQueen("An unspecified file error happened: "
                Exception в раз-     + "Message: " + ex.Message + "\r\n"
                личных блоках     + "Stack trace: " + ex.StackTrace + "\r\n"
                можно использо-     + "Data: " + ex.Data + "\r\n");
                вать одно и то же
                имя (ex).
            }
            finally {
                vat.Seal();
                worker.FinishedJob();
            }
        }
    }
}

```

В блоке `catch` объекту `exception` сопоставляют переменную `ex`, которая затем может использоваться для получения информации об объекте.

Этот оператор использует три свойства объекта `Exception`: `Message`, описывающее текущее исключение («Попытка десериализации на ноль»); `StackTrace`, возвращающий строковое представление стека вызова; и `Data`, иногда содержащее дополнительную информацию об исключении.

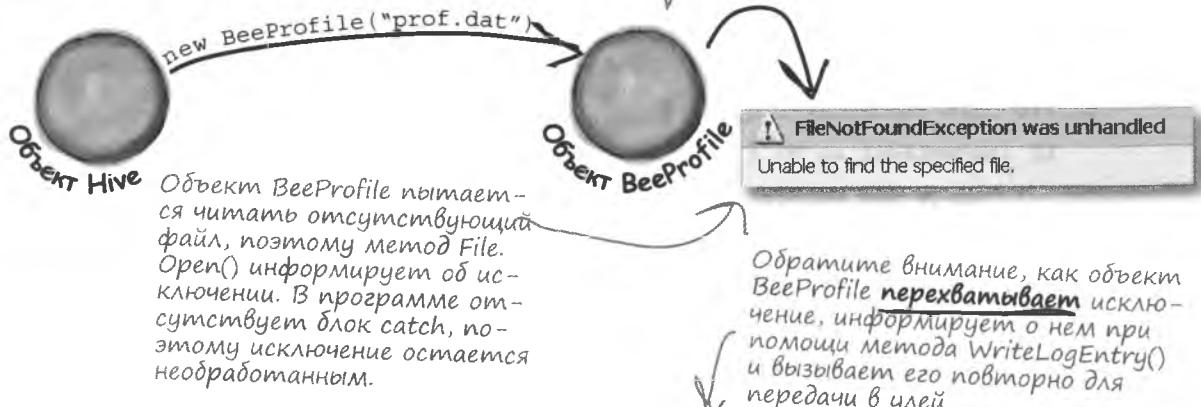
## Один класс создает исключение, другой его обрабатывает

В момент создания класса неизвестно, 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.

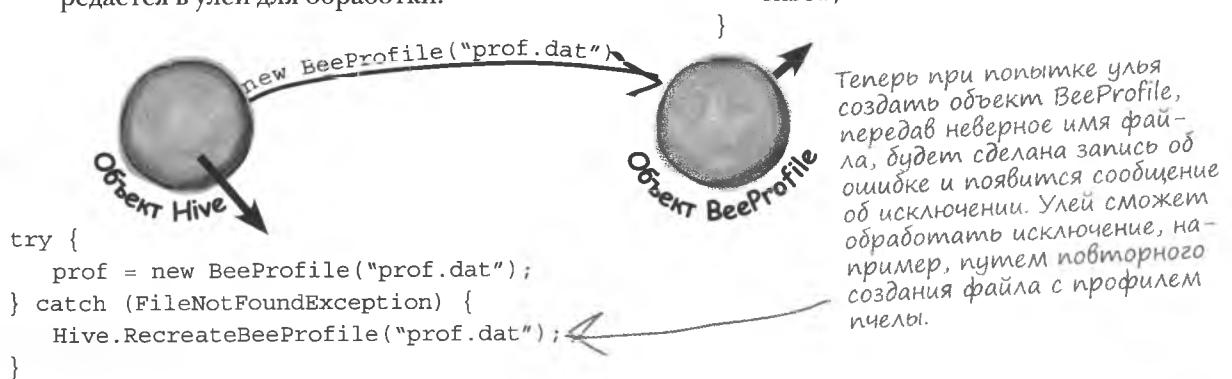
### Вместо того, чтобы...

Без обработки исключений программа перестает работать. Вот что происходит в программе управления профилями пчел.



### ...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.



Разумеется, бывает и такое, что один метод может формировать исключение, которое ликвидируется другим методом этого же класса.

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);
```

**FileNotFoundException was unhandled**  
Unable to find the specified file.

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.Open(profile);
} catch (FileNotFoundException ex) {
    WriteLogEntry("unable to find " +
        profile + ": " + ex.Message());
    throw;
}

```

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

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.

```
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();
        }
    }
}
```

Exception
Message
StackTrace
GetBaseException()
ToString()


your Exception
Message
StackTrace
GetBaseException()
ToString()

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.

478 глава 10

```
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");

```
public static void Main() {  
    Console.Write("when it ");  
    ExTestDrive.Zero("yes");  
    Console.Write(" it ");  
    ExTestDrive.Zero("no");  
    Console.WriteLine(".");  
}
```

```
class MyException : Exception { }
```

Эта строкка определяет пользовательское исключение MyException, которое обрабатывается в блоке catch.



## Решение задачи с Магнитами

Расположите магниты с кодом таким образом, чтобы на консоль был выведен следующий результат.

### Результат:

when it thaws it throws.

```
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 строкка выполняется только в случае, когда метод doRisky() не становится причиной появления исключения.

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 так же всегда попадает в список вывода.

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`. Страйтесь 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 вызывать 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 больше информации пользователю.

## 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 скращенную запись с оператором `using`.

небольшое предупреждение

## 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.WriteLine(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 внутри метода будет выполнено в конце оператора using.

Метод Dispose()  
уже написан,  
так что он  
может быть  
вызван произ-  
вольное коли-  
чество 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 реализации интерфейса IDisposable 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:

```
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);  
}
```

Вложенные операторы using используются при необходимости объявить две ссылки на интерфейс IDisposable в одном блоке кода.

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:** Я правильно понимаю, что объекты, реализующие интерфейс `IDisposable`, используются только внутри оператора `using`?

**О:** Да. Интерфейс `IDisposable` предназначен для работы с оператором `using`, и добавление этого оператора эквивалентно созданию экземпляра класса, только тут всегда вызывается метод `Dispose()`.

**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`. Поэтому 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 DivideByZeroException.

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.

## 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 исключений, но лучше, чем ничего.

Вот временное решение для калькулятора:

```
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());  
            };  
        }  
    }  
}
```

Я понял! Мы используем обработку исключений, чтобы пометить проблемную область.

← Проблема никуда не исчезла, но, по крайней мере, стало ясно, где она возникла. Лучше всего разобраться, почему ваш метод Divide вызывает-ся при нулевом знаменателе и устраниить эту возмож-ность.



**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!**

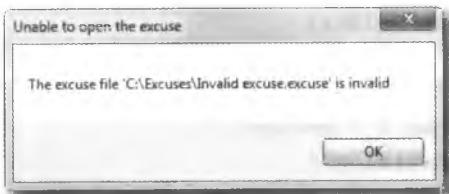
Или другими элементами, реализующими интерфейс `IDisposable`.



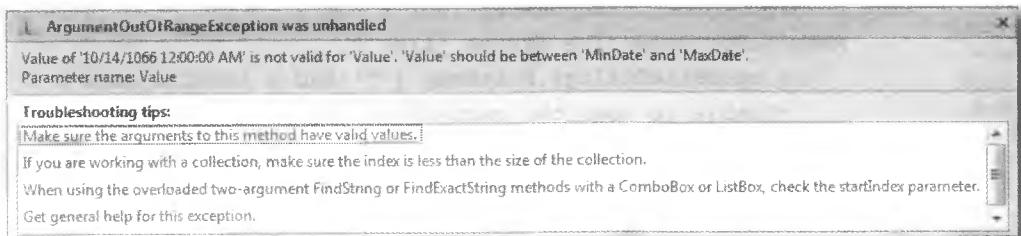
## Упражнение

Воспользуйтесь блоком `try/catch/finally` для обработки исключений в программе Excuse Manager Брайана.

- Добавьте к обработчику события Click кнопки Open обработку исключения. Достаточно блока `try/catch`, вызывающего окно с сообщением. Оно должно появляться при попытке открыть файл неверного формата:



- Но это еще не все. Создадим отдельный некорректный файл с объяснением. Поместите точку останова в первую строчку метода `Excuse.Save()`, запустите программу и сохраните оправдание. Когда программа прервет работу, добавьте контрольное значение к свойству `LastUsed`. В окне Watch присвойте ему значение `DateTime.Parse("October 14, 1066")`. Нажмите клавишу F5, чтобы продолжить отладку. Должно появиться исключение `ArgumentOutOfRangeException`:



Оно появилось потому, что свойство `Value` элемента `DateTimePicker` получило значение меньшее, чем `MinDate`. Но перед этим класс `Excuse` записал файл. Это **очень полезная техника**: генерация файлов с заранее известными неверными данными. Потом эти файлы можно использовать для тестирования программы.

- Загрузите только что созданный файл, и вы получите то же самое исключение. Для получения другого исключения нужно попытаться открыть файл, который не содержит оправдания. Добавьте блок обработки исключения, *вложенный внутрь добавленного на шаге 2*, чтобы загрузить файл, не содержащий оправдания:

- Объявите булеву переменную `clearForm` перед блоком `try/catch`. Она должна иметь значение `true` при наличии исключения и использоваться при проверке необходимости очистки формы.
- Добавьте еще один блок `try/catch` внутрь блока кнопки Open.
- Добавьте блок `finally` к внешней конструкции `try/catch`, чтобы вернуть форму в исходное, пустое состояние. Если переменная `clearForm` имеет значение `true`, присвойте `LastUsed.Value` свойство `DateTime.Now` (оно возвращает текущую дату).

упражнение  
Решение

Вот каким образом ваши знания о конструкции try/catch/finally помогли улучшить программу Брайана.

```

private void open_Click(object sender, EventArgs e) {
    if (CheckChanged()) {
        openFileDialog1.InitialDirectory = selectedFolder;
        openFileDialog1.Filter =
            "Excuse files (*.excuse)|*.excuse|All files (*.*)|*.*";
        openFileDialog1.FileName = description.Text + ".excuse";
        DialogResult result = openFileDialog1.ShowDialog();
        if (result == DialogResult.OK) {
            bool clearForm = false;
            try {
                currentExcuse = new Excuse(openFileDialog1.FileName);
                try {
                    UpdateForm(false);
                }
                catch (ArgumentOutOfRangeException) {
                    MessageBox.Show("The excuse file ''" +
                        openFileDialog1.FileName + "' had a invalid data",
                        "Unable to open the excuse");
                    clearForm = true;
                }
                catch (SerializationException ex) {
                    MessageBox.Show("An error occurred while opening the excuse '"
                        + openFileDialog1.FileName + "'\n" + ex.Message,
                        "Unable to open the excuse", MessageBoxButtons.OK,
                        MessageBoxIcon.Error);
                    clearForm = true;
                }
                finally {
                    if (clearForm) {
                        description.Text = "";
                        results.Text = "";
                        lastUsed.Value = DateTime.Now;
                    }
                }
            }
        }
    }
}

```

В данном случае не используется объект exception, поэтому в операторе catch указывается только тип исключения, а имя переменной опускается.

В этом блоке try/catch block создается окно с сообщением об ошибке, на случай возникновения проблемы при вызове формой конструктора Excuse для загрузки оправдания.

Это вложенный блок try/catch. Он обрабатывает исключение, возникающее, если данные в загружаемом файле выходят за пределы заданного диапазона.

Это окно с сообщением, созданное внешним блоком try/catch block. Оно содержит информацию об исключении.

Оба блока catch присваивают переменной clearForm значение true, поэтому в этом блоке finally форма перезагружается.

## Наконец Брайан получил свой отды...

Теперь, после того как Брайан позаботился об исключениях, он получил свой заслуженный (и одобренный шефом!) выходной.



## ...и положение дел улучшилось!

Ваше умение работать с исключениями не просто предотвратило проблему. Оно прежде всего гарантировало, что шеф Брайана не узнает о его прогулках!



Старина Брайан никогда  
не уходит с работы без  
уважительной причины.

**Правильная обра-  
ботка исключений  
незаметна польза-  
телям. Программа  
не останавливает  
работу, а проблемы  
обрабатываются акку-  
ратно, без сообщений  
об ошибках.**



# Что делает ваш код, когда вы на него не смотрите

Надо подписаться на событие  
ВдругПоявилосьДерево,  
иначе может быть вызван  
метод СломатьНогу().



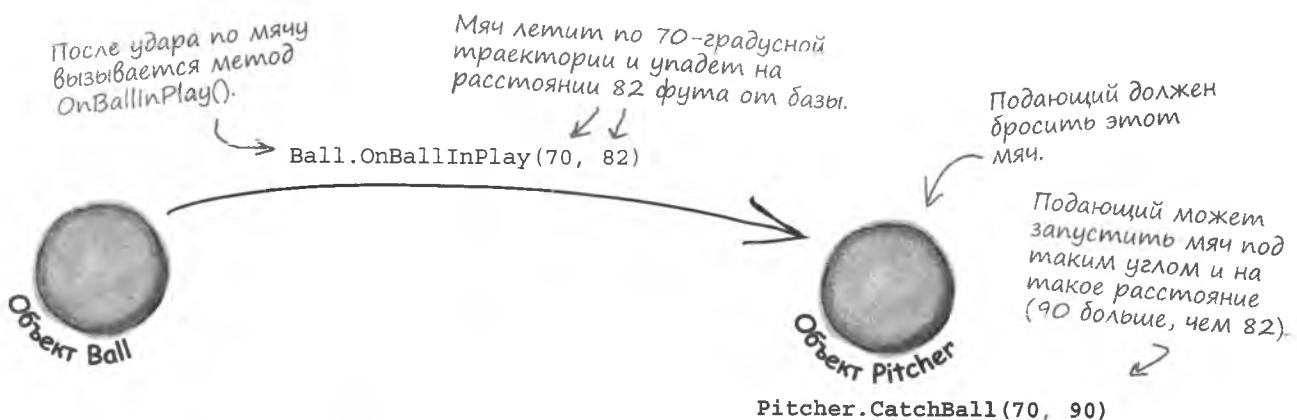
**Невозможно все время контролировать созданные объекты.**  
Иногда что-то... происходит. И хотелось бы, чтобы объекты умели реагировать на происходящее. Здесь вам на помощь приходят события. Один объект их публикует, другие объекты на них подписываются, и система работает. А для контроля подписчиков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

## Хотите, что объекты научились думать сами?

Предположим, вы пишите симулятор игры в бейсбол. Вы собираетесь 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.

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:

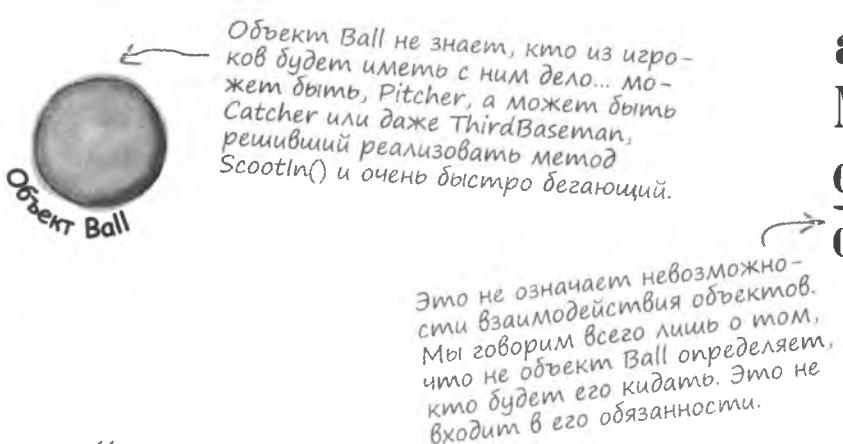
Это повсеместно распространенный способ именования методов. Мы обсудим его чуть позже.



## 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».

**Объекты должны заботиться о себе, а не о других.  
Мы за разделение сферы влияния объектов.**



## 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.

Удар по мячу становится причиной события BallInPlay.

На это событие может подписать́сь любой объект... при этом объект Ball список подписчиков неизвестен.

**Вызвано событие BallInPlay**

В ИСР события помечаются значком в виде молнии. Вы уже могли его видеть рядом с событиями в окнах IntelliSense и Properties.

Нападающий и другие игроки стараются получить мяч.

Судья проверяет, по правилам ли обрабатывается каждый мяч, и отмечает происходящее на поле.

со-бы-ти-е, сущ.

то, что случается.

Солнечное затмение – это **событие**, которое нельзя пропустить.

Подписавшись на событие BallInPlay, мы сразу узнаем, когда мяч окажется в игре.

Фанаты подписываются на случай, если мяч попадет на трибуны.

## 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

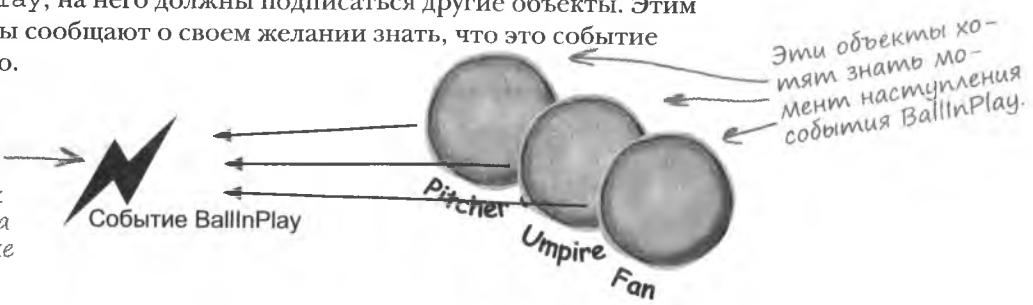
Посмотрим, как в C# функционируют события, их обработчики и подписки:

①

### Сначала объекты подписываются на событие

До момента, когда объект Ball сможет вызвать событие BallInPlay, на него должны подписать другие объекты. Этим они как бы сообщают о своем желании знать, что это событие наступило.

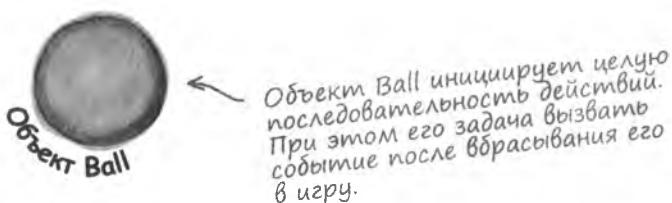
Каждый объект создает свой собственный обработчик события. Помните, как в качестве реакции на событие Click к кнопке добавлялся метод button1\_Click().



②

### Событие запускается

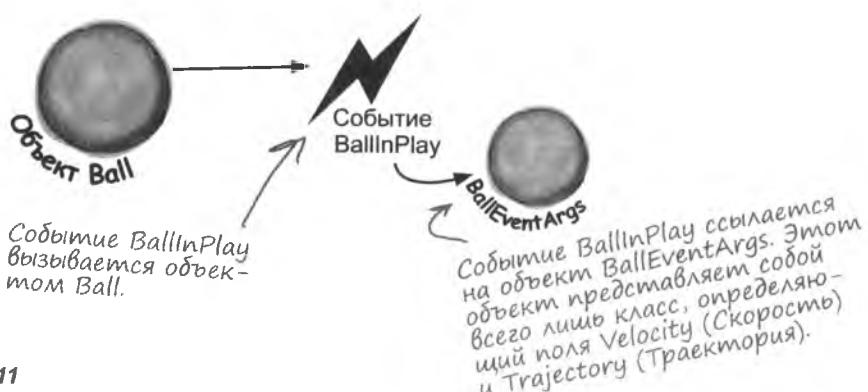
По мячу наносится удар. Именно в этот момент объект Ball вызывает новое событие.



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.

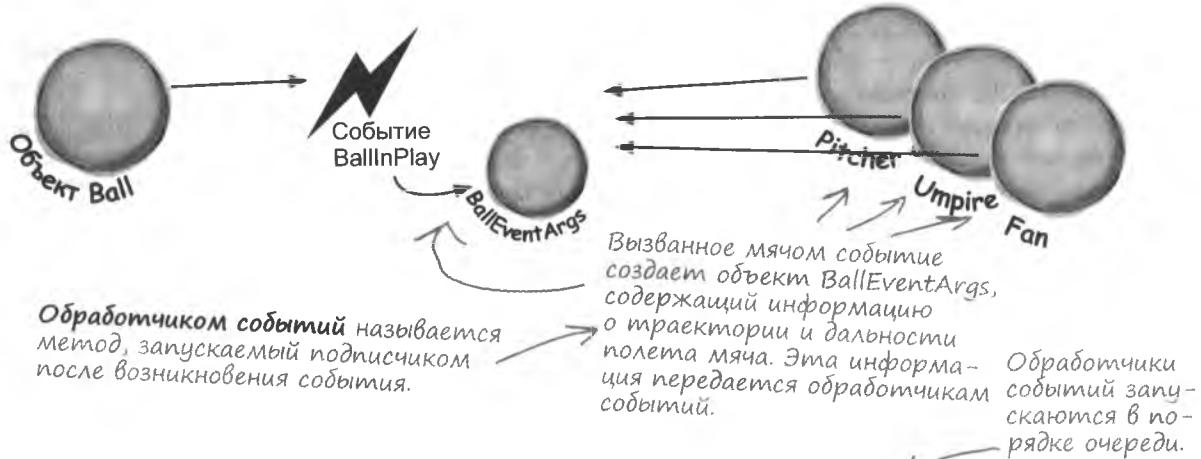


## Обработка события

После возникновения события все подписанные на него объекты получают уведомления и могут совершать различные действия.

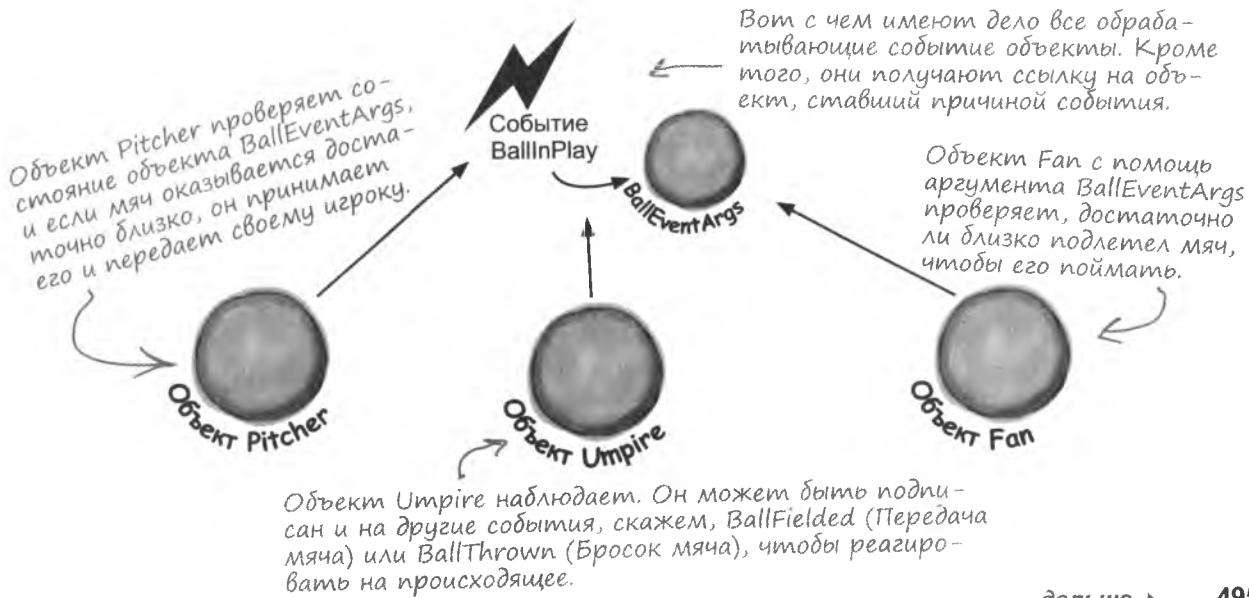
### 4 Подписчики получают уведомление

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 имеет общее представление о том, что происходит, рассмотрим более подробно процесс соединения разрозненных кусков.

Имеет смысл (хотя это и не обязательно) делать объекты аргументов событий производными от класса EventArgs.

### 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
```

Это означает возможность **восходящего приведения** объекта EventArgs в случае, когда его нужно переслать событию, не умеющему его обрабатывать.



Эти свойства позволяют мячу передавать обработчикам событий информацию о месте броска в игру.

### ② Нужно объявить событие внутри вызвавшего его класса

В классе Ball присутствует строчка 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, ИСР добавляла его в класс. Ровно то же 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 ➔

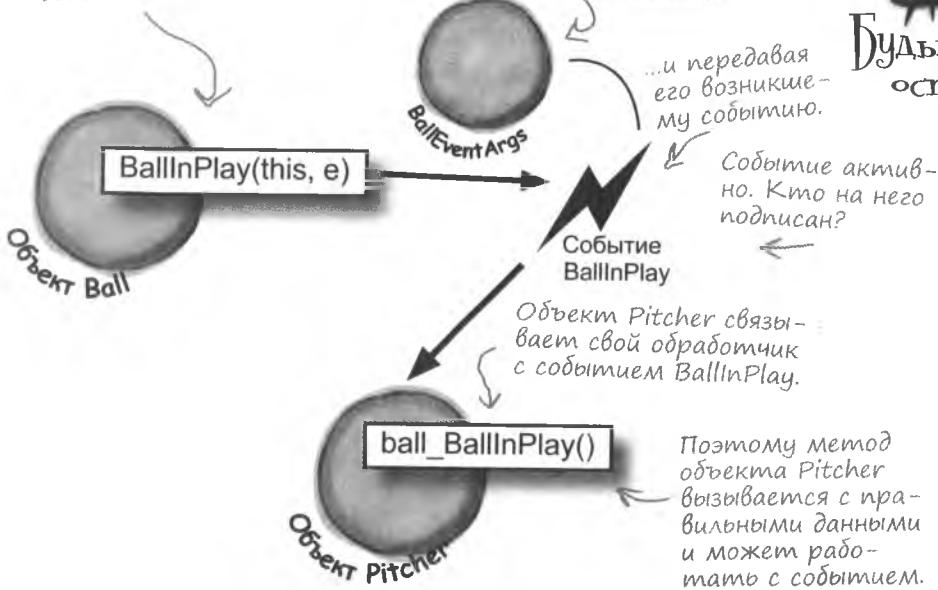
## 5 Объект Ball оповещает подписчиков, что он в игре, с помощью события

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 гарантировано не имеет значения null и используется для вызова события.

По мячу наносится удар, и объект Ball начинает действовать...



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 формы 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 Form вызывает множество событий, а каждое событие имеет набор вызывающих его методов. Метод формы **OnDoubleClick()** вызывает событие DoubleClick, и это выглядит логично. Поэтому объект Ball имеет метод **OnBallInPlay**, использующий в качестве параметра объект BallEventArgs. Симулятор вызывает этот метод каждый раз, 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 обработчиками 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 строкке 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 значений.

**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 последнюю строку, предлагаемую функцией IntelliSense. 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 позднее.

**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 обработчикам событий имена 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 чтение кода другими пользователями.

К счастью ИСР позволяет без проблем следовать этому неписанному правилу. Ведь она умеет **автоматически добавлять обработчики событий**. Вы уже сталкивались с этой функцией, но давайте рассмотрим ее еще раз.



1

### Создайте приложение Windows Form 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****ИСР поможет вам сэкономить время**

Как только вы введете оператор +=, появится окно:

```
public Pitcher(Ball ball) {
    ball.BallInPlay +=  
}
```

new EventHandler(ball\_BallInPlay); (Press TAB to insert)

Нажмите tab для завершения ввода оператора. Вот как он выглядит:

```
public Pitcher(Ball ball) {
    ball.BallInPlay += new EventHandler(ball_BallInPlay);
```

Двойной щелчок на кнопке в конструкторе форм приводит к автоматическому добавлению обработчика события, но в данном случае код добавляется к методу InitializeComponent() в файле Form1.Designer.cs, а не в конец файла класса.

**4****Обработчик событий тоже будет добавлен автоматически**

Нужно добавить еще один метод в цепочку события. К счастью, это тоже можно сделать средствами ИСР.

```
new EventHandler(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();
```

ИСР по умолчанию вставляет исключение NotImplementedException() в качестве заполнителя. Если вы не введете сюда нужный код и запустите программу, появится сообщение о том, что метод нереализован.

**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();
    }
}
```

Эти методы будут добавлены через минуту.

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.

**Упражнение**

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.**

Методы CatchBall() и CoverFirstBase() должны выводить информацию о том, что защитник или ловит мяч или бежит на первую базу.

```
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();
        }
    }
}
```

*Эти два метода должны выводить результат на консоль.*

**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 (хоум-ран), и при превышении значений пытаться поймать мяч. В противном случае болельщик кричит. Выведите результат на консоль.

Хоум-ран — удар, при котором мяч пролетает все поле и вылетает за его пределы.



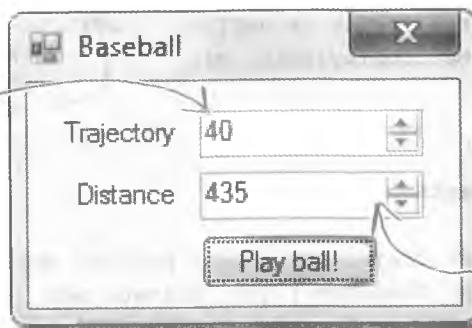
*Точный список вывода вы увидите на рисунке на следующей странице.*



**3****Построение простого симулятора.**

Создайте новую форму с двумя элементами NumericUpDown: один для расстояния, другой для траектории. Добавьте кнопку Play ball! Щелчок на ней соответствует удару по мячу, после чего мяч летит с указанными в счетчиках параметрами. Форма должна выглядеть примерно вот так:

Диапазон этого значения может меняться от 0 до 100, поэтому свойству Minimum присвойте значение 0, свойству Maximum — значение 100, а свойству Value — значение 20.

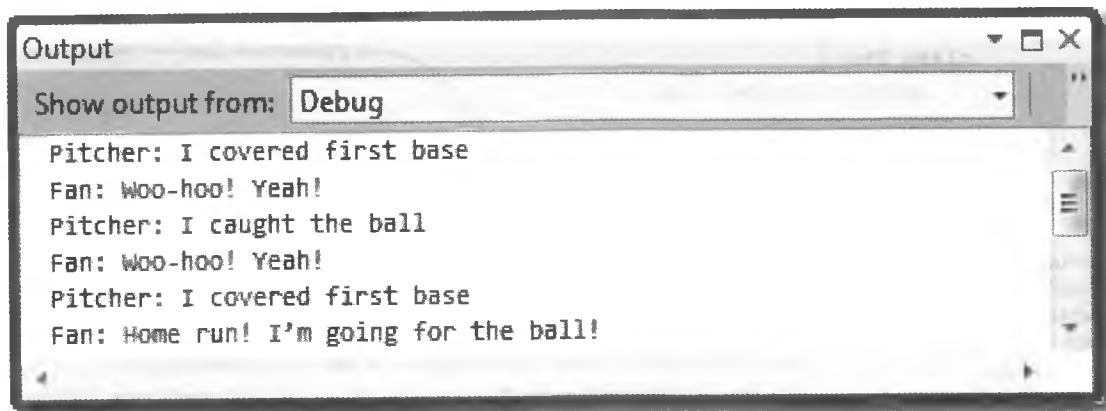


Не забудьте осущест-  
вить приведе-  
ние свойства Value  
к типу int.

Расстояние может  
меняться от 0 до  
500. В качестве  
значения по умолча-  
нию выберите 100.

**4****Список вывода.**

Вот какой результат должен выдать симулятор после трех мячей:

**Мяч 1:**

Траектория: .....

Расстояние: .....

**Мяч 2:**

Траектория: .....

Расстояние: .....

**Мяч 3:**

Траектория: .....

Расстояние: .....



## упражнение Решение

Вот как выглядит код с написанными классами Ball и Pitcher и добавленным классом 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

В качестве аргументов события прекрасно подходят автоматически добавляемые, предназначенные только для чтения свойства. Ведь обработчики событий только читают передаваемый им данные.

```
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;
    }
}
```

Конструктор объекта Fan привязывает свой обработчик события к событию BallInPlay.

```
class Fan {
    public Fan(Ball ball)
    {
        ball.BallInPlay += new EventHandler(ball_BallInPlay);
    }
}
```

Обработчик события BallInPlay класса Fan высматривает высоко летящие мячи, брошенные на слишком длинную дистанцию.

```
void ball_BallInPlay(object sender, EventArgs e)
{
    if (e is BallEventArgs) {
        BallEventArgs ballEventArgs = e as BallEventArgs;
        if (ballEventArgs.Distance > 400 && ballEventArgs.Trajectory > 30)
            Console.WriteLine("Fan: Home run! I'm going for the ball!");
        else
            Console.WriteLine("Fan: Woo-hoo! Yeah!");
    }
}
```

```

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();
        }
    }
    private void CatchBall() {
        Console.WriteLine("Pitcher: I caught the ball");
    }
    private void CoverFirstBase() {
        Console.WriteLine("Pitcher: I covered first base");
    }
}

public partial class Form1 : Form {
    Ball ball = new Ball();
    Pitcher pitcher;
    Fan fan;

    public Form1() {
        InitializeComponent();
        pitcher = new Pitcher(ball);
        fan = new Fan(ball);
    }

    private void playBallButton_Click(object sender, EventArgs e) {
        BallEventArgs ballEventArgs = new BallEventArgs(
            (int)trajectory.Value, (int)distance.Value);
        ball.OnBallInPlay(ballEventArgs);
    }
}

```

В классе Pitcher уже имеется обработчик события BallInPlay. Он проверяет низко лежащие мячи.

Форме требуется один мяч, один болельщик и один нападающий. В своем конструкторе она связывает болельщика и нападающего с мячом.

Щелчок на кнопке заставляет нападающего выполнить подачу, в результате чего мяч вызывает свое событие BallInPlay. Это, в свою очередь, вызывает обработчики событий объектов Pitcher и Fan.

Вот значения, которые мы использовали, чтобы получить показанный на рисунке результат. Вы можете попробовать и другие комбинации.

**Мяч 1:**Траектория:.....**75**.....  
Расстояние:.....**105**.....**Мяч 2:**Траектория:.....**48**.....  
Расстояние:.....**80**.....**Мяч 3:**Траектория:.....**40**.....  
Расстояние:.....**43.5**.....

## Обобщенный EventHandler

Посмотрим на объявление события в классе Ball:

```
public event EventHandler BallInPlay;
```

А теперь рассмотрим объявление события Click для кнопки формы:

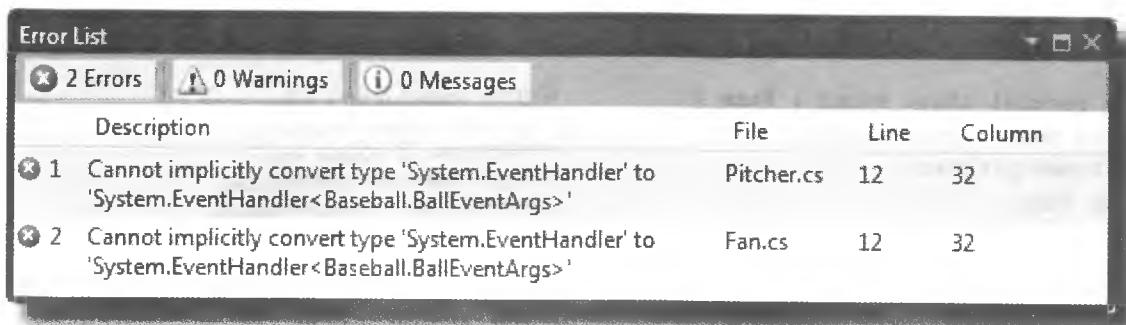
```
public event EventHandler Click;
```

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, .NET имеется инструмент, позволяющий легко сообщить эту информацию, — обобщенный EventHandler. Измените обработчик события BallInPlay вот таким образом:

```
public event EventHandler<BallEventArgs> BallInPlay;
```

Обобщенный аргумент EventHandler должен быть производным от EventArgs.

Вам понадобиться отредактировать и метод OnBallInPlay, заменив EventHandler на EventHandler<BallEventArgs>. Но при попытке построить код в окне Error List появится сообщение о двух ошибках:



Дело в том, что после изменений в объявлении события нужно обновить классы Pitcher и Fan, заставив их передавать обработчику обобщенный аргумент:

```
ball.BallInPlay += new EventHandler<BallEventArgs>(ball_BallInPlay);
```

## Неявное преобразование

Автоматически созданный обработчик событий будет обязательно содержать ключевое слово new, за которым следует его тип. Если же убрать это ключевое слово и тип обработчика, C# осуществит **неявное преобразование** и определит тип за вас:

```
ball.BallInPlay += ball_BallInPlay;
```

Замените код в конструкторах классов Pitcher и Fan указанным выше способом. Вы увидите, что на работе программы это не отразится.

## Все формы используют события

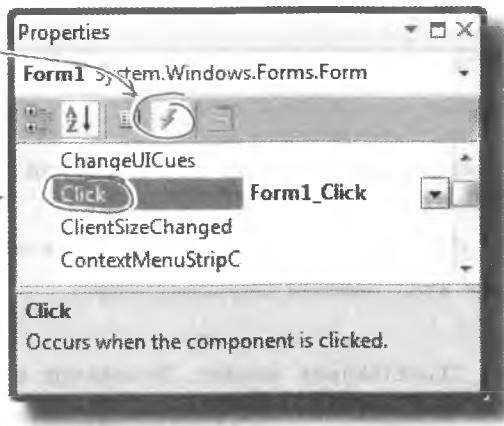
Создавая кнопку, дважды щелкнув на ней в конструкторе формы и прописывая код для метода `button1_Click()`, вы работаете с событиями.



- 1 Создайте проект Windows 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 строчку `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 строчке 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

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 ИСР, 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.



Обработчики событий должны быть «связаны».

Будьте  
осторожны!

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, ИСР возьмет заданный по умолчанию обработчик `button1_Click_1()` и добавит его к кнопке.

часто  
Задаваемые  
Вопросы

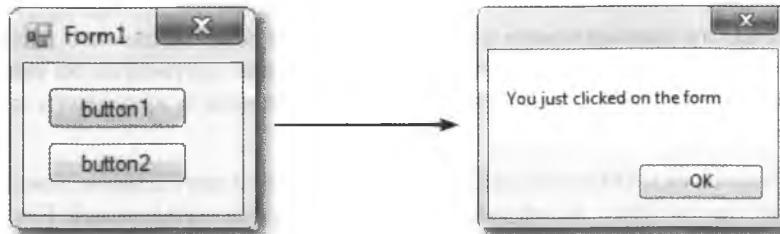
В: Почему после добавления нового обработчика событий к объекту Pitcher появилось сообщение об исключении?

О: ИСР 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.



Запустите программу и выполните следующие действия:

- ★ Щелкните на форме. Появится окно с текстом «You just clicked on the form».



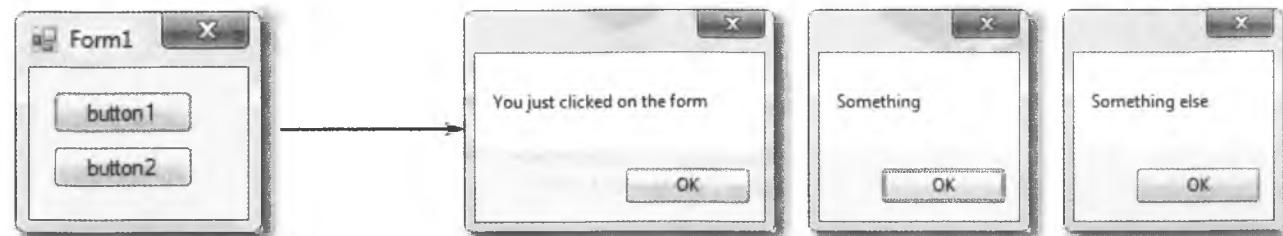
Обработчик события Click формы вызвал окно с сообщением «Вы только что щелкнули на форме».

- ★ Теперь щелкните на кнопке button1, а затем снова на форме. Появятся два окна с текстом: «You just clicked on the form» и «Something».



Каждый щелчок на кнопке приводит к появлению еще одного окна диалога.

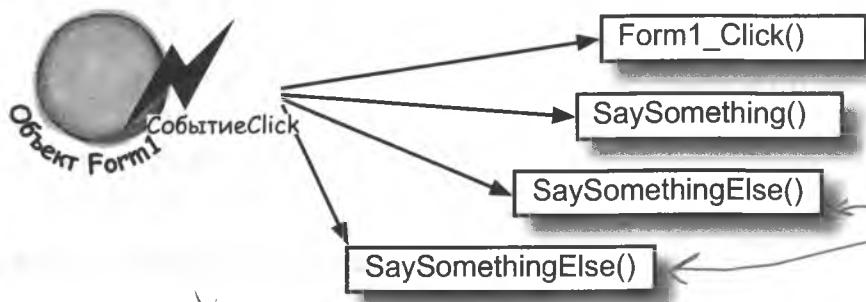
- ★ Дважды щелкните на кнопке button2, а затем снова на форме. Появятся четыре окна: «You just clicked on the form», «Something», «Something else» и «Something else».



### 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 щелчки 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.

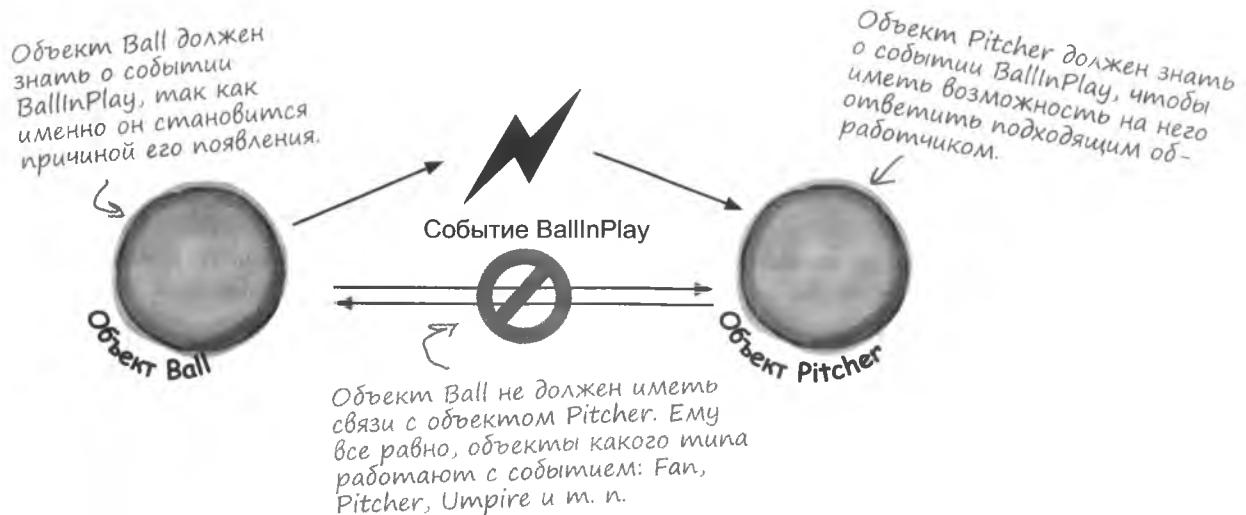
[далее ▶](#)

509

## Связь между издателями и подписчиками

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 он ему передает. И **подписчик (receiver)** 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 пред-  
ставлять других. Президент отправил делегата на саммит.

## Делегат замещает методы

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.

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.

Данный делегат может ссылаться на любой метод, использующий в качестве параметров object и EventArgs и не возвращающий значения.



## 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 ConvertsToString.cs. Введите одну строчку:

```
delegate string ConvertsToString(int i);
```

Еще одним добавленным в проект делегатом является ConvertsToString. 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 ReturnsAString.

Заполните метод Main():

```
static void Main(string[] args) { ConvertsToString someMethod = new ConvertsToString(HiThere); string message = someMethod(5); Console.WriteLine(message); Console.ReadKey(); }
```

someMethod — это переменная типа ConvertsToString. От обычной ссылочной переменной она отличается тем, что указывает не на объект в куче, а на метод.

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

Работать с делегатами легко, они не требуют большого объема кода. Попробуем с их помощью помочь владельцу ресторана рассортировать секретные ингредиенты с кухни шеф-повара.



1

### Добавьте делегат в новый проект

Делегаты обычно располагаются вне классов, поэтому добавьте к проекту новый файл классов GetSecretIngredient.cs и введите в него строку:

```
delegate string GetSecretIngredient(int amount);
```

(Полностью удалите объявление класса.) Этот делегат станет основой переменной, указывающей на метод, который, взяв параметр типа int, возвращает строку.

2

### Добавьте класс для первого шеф-повара, Сюзанны

Suzanne.cs содержит класс, следящий за секретными ингредиентами, которые использует Сюзанна. Он снабжен закрытым методом SuzanneSecretIngredient() с сигнатурой, совпадающей с GetSecretIngredient. Имеется и предназначено только для чтения свойство, возвращающее GetSecretIngredient. С его помощью остальные объекты могут получить ссылку на метод SuzanneIngredientList().

```
class Suzanne {  
    public GetSecretIngredient MySecretIngredientMethod {  
        get {  
            return new GetSecretIngredient(SuzannesSecretIngredient);  
        }  
    }  
    private string SuzanneSecretIngredient(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.

3

### 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, так же берет переменную amount типа int и возвращает строку. Но это уже другая строка.

4

**Добавьте к проекту делегаты**

Постройте форму.

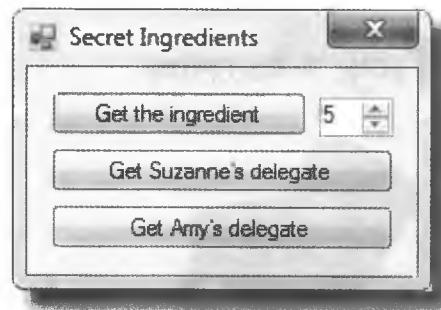
Это ее код:

```
GetSecretIngredient ingredientMethod = null;
Suzanne suzanne = new Suzanne();
Amy amy = new Amy();

private void useIngredient_Click(object sender, EventArgs e) {
    if (ingredientMethod != null)
        Console.WriteLine("I'll add " + ingredientMethod((int)amount.Value));
    else
        Console.WriteLine("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);
}
```



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 формы (являющемуся делегатом GetSecretIngredient) значение, возвращаемое свойством GetSecretIngredient. Это новый экземпляр типа GetSecretIngredient, указывающий на метод SuzanneSecretIngredient().
- ★ Снова щелкните на кнопке «Get the ingredient». Теперь, когда поле ingredientMethod указывает на SuzanneSecretIngredient(), 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 строчки 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



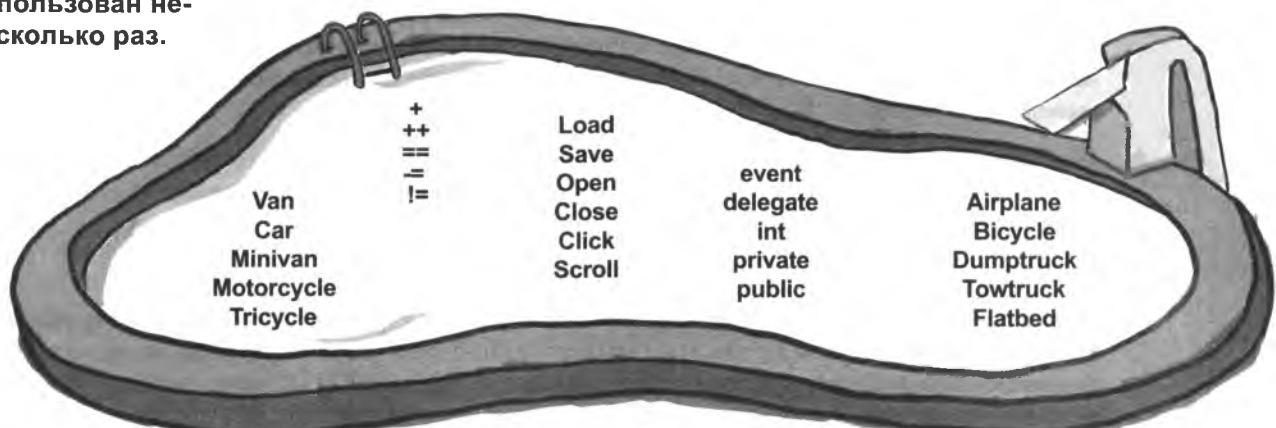
Возьмите фрагменты кода из бассейна и заполните пропуски. Каждый фрагмент может быть использован несколько раз. В бассейне есть лишние фрагменты. Вам нужно, чтобы при щелчке на кнопке button1 на консоль выводился следующий результат:

### Результат:

Fingers is coming to get you!

```
public Form1() {  
    InitializeComponent();  
  
    this._____ += new EventHandler(Minivan);  
    this._____ += new EventHandler(______);  
}  
  
void Towtruck(object sender, EventArgs e) {  
    Console.WriteLine("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.WriteLine("Fingers ");  
}
```

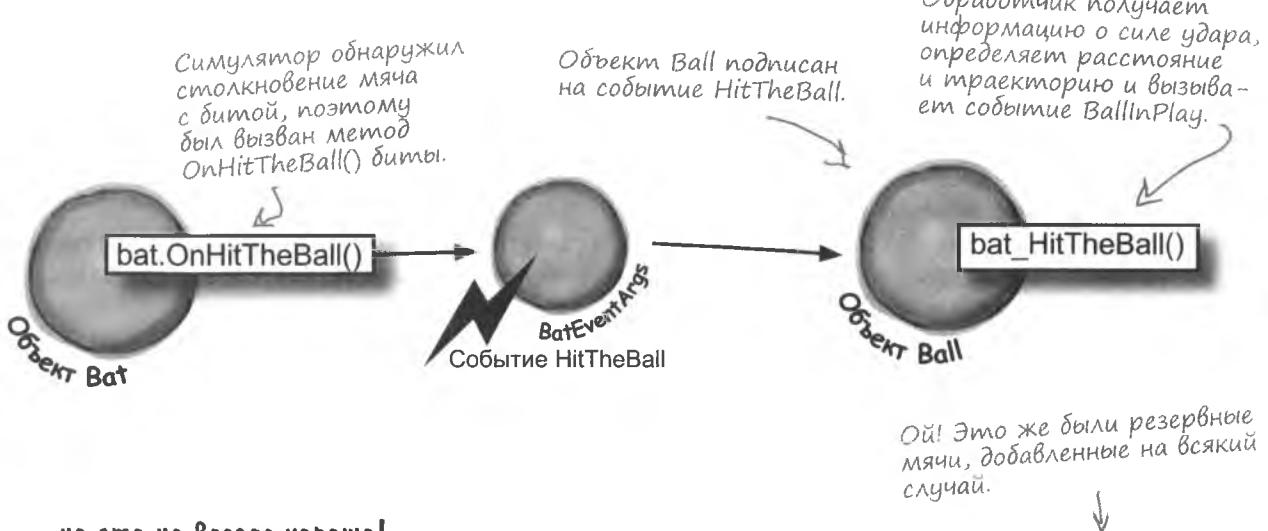
Каждый фрагмент  
может быть ис-  
пользован не-  
сколько раз.



## Объект может подписаться на событие...

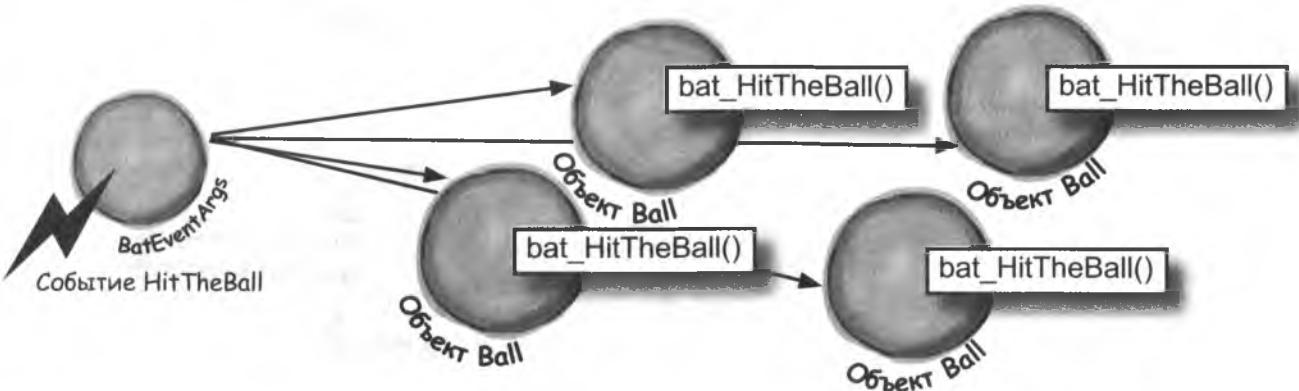
Предположим, для симулятора был создан класс Bat (Бита), который добавляет событие HitTheBall. Обнаружив, что игрок ударил по мячу, симулятор вызывает метод OnHitTheBall() объекта Bat, который вызывает событие HitTheBall.

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. 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.

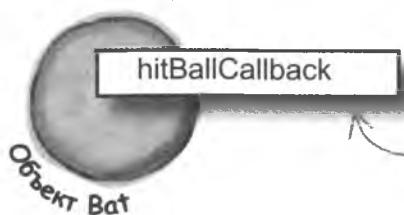
В этом нам поможет **обратный вызов (callback)**. Так называется техника работы с делегатами, при которой вместо события, доступного для подписки любым объектам, используется метод (часто конструктор), с хранящимся в закрытом поле делегатом в качестве аргумента. Обратный вызов позволит нам гарантировать, 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

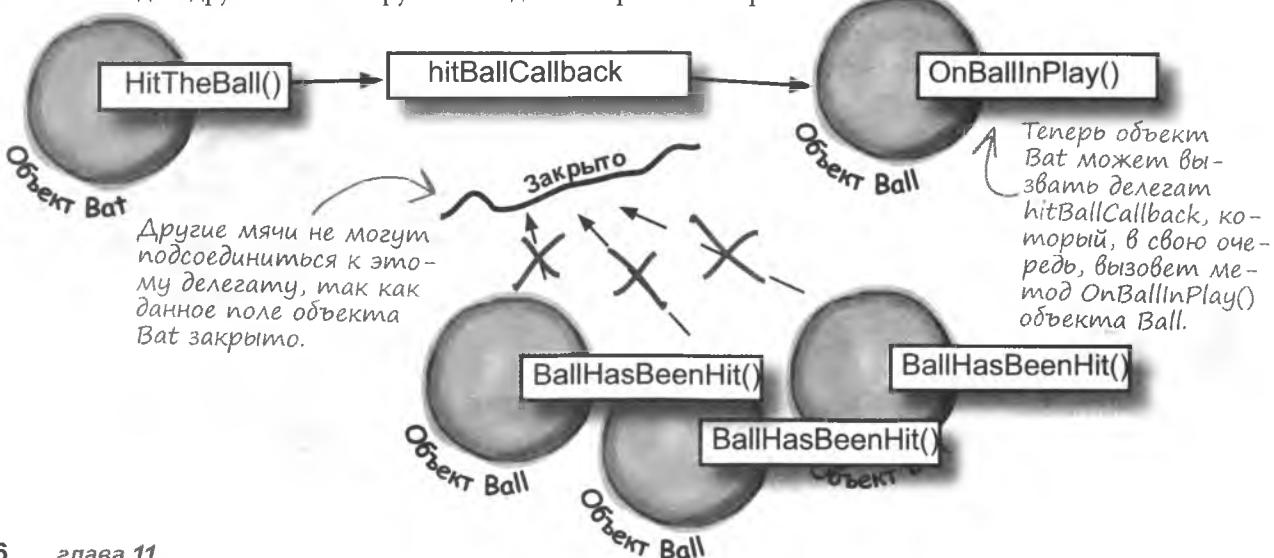
Когда мяч оказывается в игре, конструктор создает экземпляр биты и передает указатель на него методу OnBallInPlay(). Это **метод обратного вызова**, так как объект Bat использует его для вызовы объекта, который его создал.



Объект Ball передает ссылку на делегат своему собственному методу OnBallInPlay() в конструкторе объекта Bat. Последний сохраняет данный делегат в закрытом поле hitBallCallback.

### 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!



## ПятиМинутная трэйна



### Случай с золотым крабом

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 так же 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 посыпается 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. 521.

Конструктор последовательно добавляем два обработчика к событиям `Load`. В итоге они вызываются сразу после загрузки формы.

```
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 ");
}
```

Щелчок на кнопке вызывает цепочку из трех связанных с ней обработчиков событий.

### Решение ребуса в бассейне



Два обработчика события `Load` привязывают к обработчику события `Click` кнопки три других, отдельных обработчика.

## Обратный вызов как способ работы с делегатами

Обратный вызов является еще одним способом работы с делегатами. Он описывает **шаблон** — способ, при котором вы используете делегаторов в своих классах таким образом, что один объект может сообщить 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 необязательно. 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);  
    }  
}
```

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 — в отслеживании вызывающим объектом адресатов. В случае события, объекты требуют оповещения, добавляя обработчики, в то время как при обратном вызове объекты перебирают делегаторов и просят об оповещении.

### 3 Связем 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));  
}
```

Мы настроили обратный вызов в конструкторе класса Bat. Но бывают случаи, когда имеет смысл поместить его в открытый метод или метод записи.

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.

4

**Инкапсулируем класс Ball**

Методы, вызывающие события, названия которых начинаются с On... не бывают открытыми.

Попробуйте в форме вызвать событие OnClick() кнопки playBall, вы не сможете это сделать, так как оно защищено (в итоге производный класс может перекрыть его). Сделаем такой уровень доступа и для метода 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

**Осталось связать метод с формой**

Форма больше не может вызывать метод OnBallInPlay() объекта Ball, как мы и хотели. Для этого был вставлен метод Ball.GetNewBat(). Теперь форма должна попросить у объекта Ball новую биту для удара по мячу. При этом метод OnBallInPlay() нужно связать с обратным вызовом биты.

```
private void playBallButton_Click(object sender, EventArgs e)
{
    Bat bat = ball.GetNewBat();
    BallEventArgs ballEventArgs = new BallEventArgs(
        (int)trajectory.Value, (int)distance.Value);
    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 делегата, 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 делегатов для ссылки и вызова методов других объектов.
- Чтобы понять, как работают делегаты, используйте отладчик.

часто  
Задаваемые  
Вопросы

**В:** Чем обратный вызов отличается от события?

**О:** События и делегаты — это часть .NET. Это способ, которым одни объекты оповещают других 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 снаженное ключевым словом `event`.

**О:** Обратный вызов похож на событие, потому что они оба используют **делегатов**. 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.

## Случай с золотым крабом

### 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(NewLocationArgs e);
    public event Escape RunForCover;
    public void SomeonesNearby() {
        NewLocationArgs e = new NewLocationArgs("Под камнем");
        RunForCover(e);
    }
}
class NewLocationArgs {
    public NewLocationArgs(HidingPlace newLocation) {
        this.newLocation = newLocation;
    }
    private HidingPlace newLocation;
    public HidingPlace NewLocation { get { return newLocation; } }
}
```

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 += new GoldenCrab.Escape(treasure_RunForCover);
    }
    void treasure_RunForCover(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 добавят 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.

Пятыминутная тайна  
раскрыта



Как только к крабу кто-то приближается, метод *SomeonesNearby()* инициирует событие *RunForCover*, и краб прячется в другом месте.

## Возьми в руку карандаш

Заполните пробелы, чтобы игра «Убей моль» заработала. Вам нужен код, поддерживающий обратные вызовы. Когда программа будет готова, проверьте ее работу в ИСР!

```

public partial class Form1 : Form {
    Mole mole;
    Random random = new Random();
    public Form1() {
        InitializeComponent();

        mole = new Mole(random, new Mole._____());
        timer1.Interval = random.Next(500, 1000);
        timer1.Start();
    }
    private void timer1_Tick(object sender, EventArgs e) {
        timer1.Stop();
        ToggleMole();
    }
    private void ToggleMole() {
        if (mole.Hidden == true)
            mole.Show();
        else
            mole.HideAgain();
        timer1.Interval = random.Next(500, 1000);
        timer1.Start();
    }
    private void MoleCallBack(int moleNumber, bool show) {
        if (moleNumber < 0) {
            timer1.Stop();
            return;
        }
        Button button;
        switch (moleNumber) {
            case 0: button = button1; break;
            case 1: button = button2; break;
            case 2: button = button3; break;
            case 3: button = button4; break;
            default: button = button5; break;
        }
        if (show == true) {
            button.Text = "HIT ME!";
            button.BackColor = Color.Red;
        } else {
            button.Text = "";
            button.BackColor = SystemColors.Control;
        }
        timer1.Interval = random.Next(500, 1000);
        timer1.Start();
    }
    private void button1_Click(object sender, EventArgs e) {
        mole.Smacked(0);
    }
}

```

Этот метод управляет появлением и исчезновением моли по результатам работы таймера.

Блок switch гарантирует изменение текста и цвета у правильной кнопки.

Добавьте эти обработчики событий обычным способом, то есть двойным щелчком на кнопках в конструкторе формы.

Форма передает делегата указывающий на метод обратного вызова в конструкторе моли.

Чтобы создать обработчик событий для таймера, переместите этот элемент на форму и дважды щелкните на нем. Таймеры последовательно вызывают событие Tick. Более подробно мы поговорим о них в следующей главе.

Добавьте пять обработчиков событий для кнопок. button2\_click() вызывает mole.Smacked(1) и заставляет button3 вызывать mole.Smacked(2), что заставляет button4 вызывать mole.Smacked(3) и button5 вызывать mole.Smacked(4).

```

using System.Windows.Forms;
class Mole {
    public _____ void PopUp(int hole, bool show);
    private _____ popUpCallback;
    private bool hidden;
    public bool Hidden { get { return hidden; } }
    private int timesHit = 0;
    private int timesShown = 0;
    private int hole = 0;
    Random random;
    public Mole(Random random, PopUp popUpCallback) {
        if (popUpCallback == null)
            throw new ArgumentException("popUpCallback can't be null");
        this.random = random;
        this._____ = _____;
        hidden = true;
    }
    public void Show() {
        timesShown++;
        hidden = false;
        hole = random.Next(5);
        _____(hole, true);
    }
    public void HideAgain() {
        hidden = true;
        _____(hole, false);
        CheckForGameOver();
    }
    public void Smacked(int holeSmacked) {
        if (holeSmacked == hole) {
            timesHit++;
            hidden = true;
            CheckForGameOver();
        }
        _____(hole, false);
    }
    private void CheckForGameOver() {
        if (timesShown >= 10) {
            popUpCallback(-1, false);
            MessageBox.Show("You scored " + timesHit, "Game over");
            Application.Exit();
        }
    }
}

```

Внешнее делегат и поле, в котором он будет храниться, оба параметра должны находиться в верхней части класса Mole.

Здесь мы проверяем не-равенство обратного вызова null, в противном случае объект Mole вызовет исключение ArgumentException.

При создании нового объекта Mole форма передает ссылку на его метод обратного вызова. Посмотрите, как именно вызывается конструктор формы, и заполните это поле.

После появления моли должен быть вызван метод формы, который меняет цвет кнопки на красный и отображает текст «HIT ME!»

Методы HideAgain() и Smacked() также пользуются делегатом обратного вызова, чтобы вызывать метод формы.

В игре используется таймер со случайной задержкой от 0.5 до 1.5 секунд. После того как время вышло, он вызывает моль. Обратный вызов объекта Mole заставляет форму показать или скрыть моль в одном из пяти полей. Форма использует таймер, чтобы через промежуток времени (от 0.5 до 1.5 секунд) убрать моль.

Игра заканчивается после того, как моль появится 10 раз. Ваши очки показывают, сколько раз вам удалось ее приклоннуть.

## Возьми в руку карандаш



## Решение

Вот как должен выглядеть код программы «Убей моль».

```
public partial class Form1 : Form {
    private void Form1_Load(object sender, EventArgs e) {
        mole = new Mole(random, new Mole._____ (_____));
        timer1.Interval = random.Next(500, 1000);
        timer1.Start();
    }
}
```

```
class Mole {
    public _____ void PopUp(int hole, bool show);
```

```
    private _____ popUpCallback;
```

...

```
public Mole(Random random, Popup popUpCallback) {
    this.random = random;
```

```
    this._____ = _____;
    hidden = true;
}
```

```
public void Show() {
    timesShown++;
    hidden = false;
    hole = random.Next(5);
```

```
    _____ (hole, true);
}
```

```
public void HideAgain() {
    hidden = true;
```

```
    _____ (hole, false);
    CheckForGameOver();
}
```

```
public void Smacked(int holeSmacked) {
    if (holeSmacked == hole) {
        timesHit++;
        hidden = true;
        CheckForGameOver();
    }
}
```

```
    _____ (hole, false);
}
```

**PopUp** **MoleCallBack**

Форма передает ссылку на метод **MoleCallBack()** объекту **Mole**, что позволяет моли вызывать данный метод.

Моль определяет свой делегат и использует его для задания закрытого поля, в котором будет храниться ссылка на метод в форме, меняющий цвет кнопок.

Создав новый экземпляр объекта **Mole**, форма передает конструктору в качестве параметра ссылку на метод **MoleCallBack()**. Эта строка в конструкторе копирует ссылку в поле **popUpCallback**. Методы конструктора при помощи этого поля могут вызывать метод **MoleCallBack()** в форме.

Когда моль показывается, прячется или оказывается убитой, объект **Mole** использует делегат **popUpCallback** для вызова метода, меняющего цвет и текст кнопок формы.

## 12 обзор и предварительные результаты



# \* Знания, сила и построение классных приложений



Помнится, я что-то читал о том, как восходящее и нисходящее приведения облегчают обработку событий...

### **Знания нужно применять на практике.**

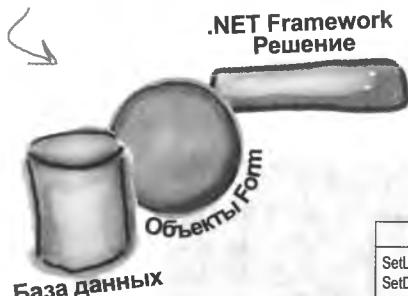
Пока вы не начнете писать работающий код, вряд ли можно быть уверенными, что вы на самом деле усвоили сложные понятия C#. Сейчас мы займемся применением на практике полученных ранее знаний. Кроме того, вы найдете предварительные сведения о понятиях, которые будут рассматриваться в следующих главах. Мы начнем построение **действительно сложного приложения**, чтобы закрепить ваши навыки.

так много информации

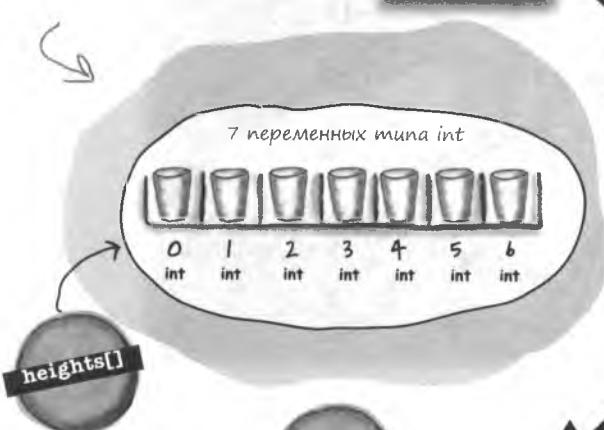
## Вы прошли долгий путь

Много времени прошло с момента, когда вы взялись за свой первый код для объектильской фирмы по производству бумаги. Ниже перечислено многое из того, чему вы научились за несколько сотен страниц.

Вы создавали формы, пользовались средствами .NET Framework и даже обменивались данными с базой.



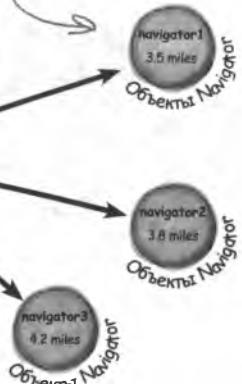
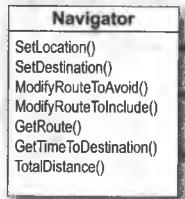
Вы научились работать даже с такими сложными типами данных как массивы.



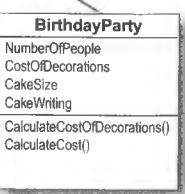
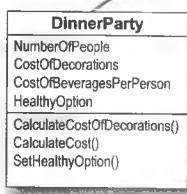
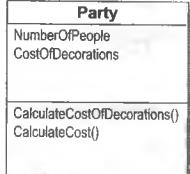
Вы боретесь с проблемами при помощи отладки и исключений.

```
private void randomExcuse_Click(object sender, EventArgs e)
{
    string[] fileNames = Directory.GetFiles(selectedPath, ".excuse");
    if (fileNames.Length == 0)
    {
        MessageBox.Show("Please specify a folder with excuse files in it",
                       "No excuse files found");
    }
}
```

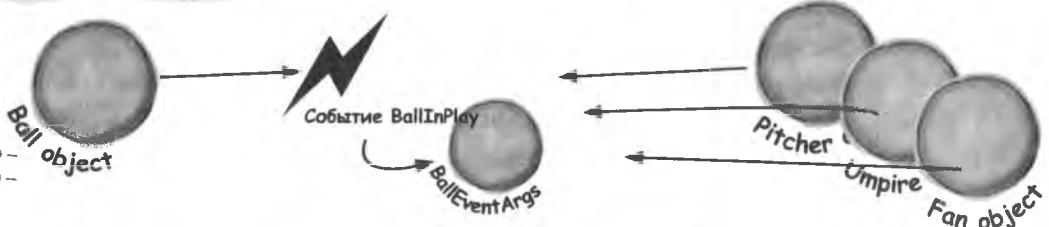
Объекты, классы, экземпляры... все эти странные термины прочно вошли в вашу повседневную практику программирования.



Вы использовали наследование и интерфейсы для построения генеалогического дерева объектов.

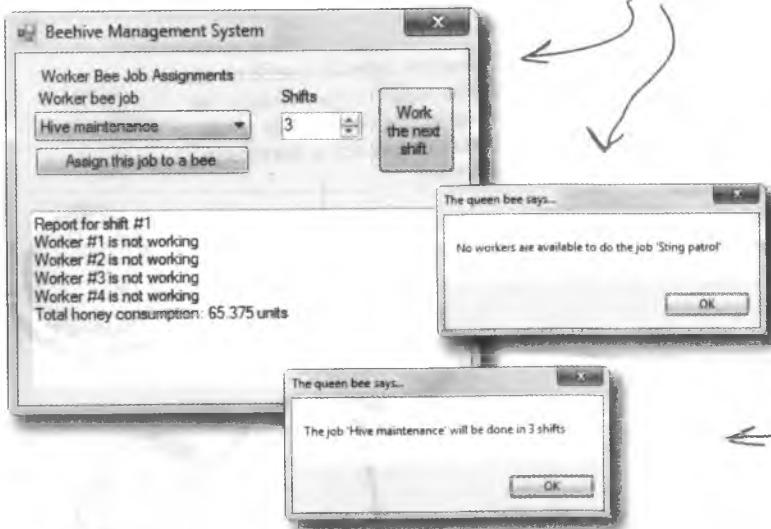


Вы использовали события для оповещения объектов о происходящих действиях.



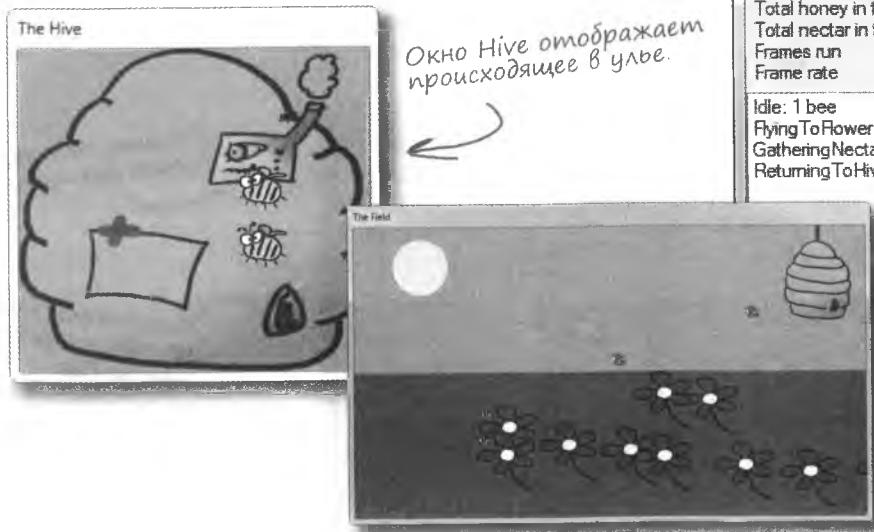
## Вы стали пчеловодом

Помните, как в главе 6 вы моделировали работу улья?



## Но можно сделать Все намного лучше...

Вы многому научились за это время. Поэтому начнем сначала и построим **анимированный симулятор улья**. Создадим пользовательский интерфейс, показывающий улей и поля, над которыми летают пчелы, а также статистику произведенных пчелами действий.



Окно статистики позволяет отслеживать работу симулятора.

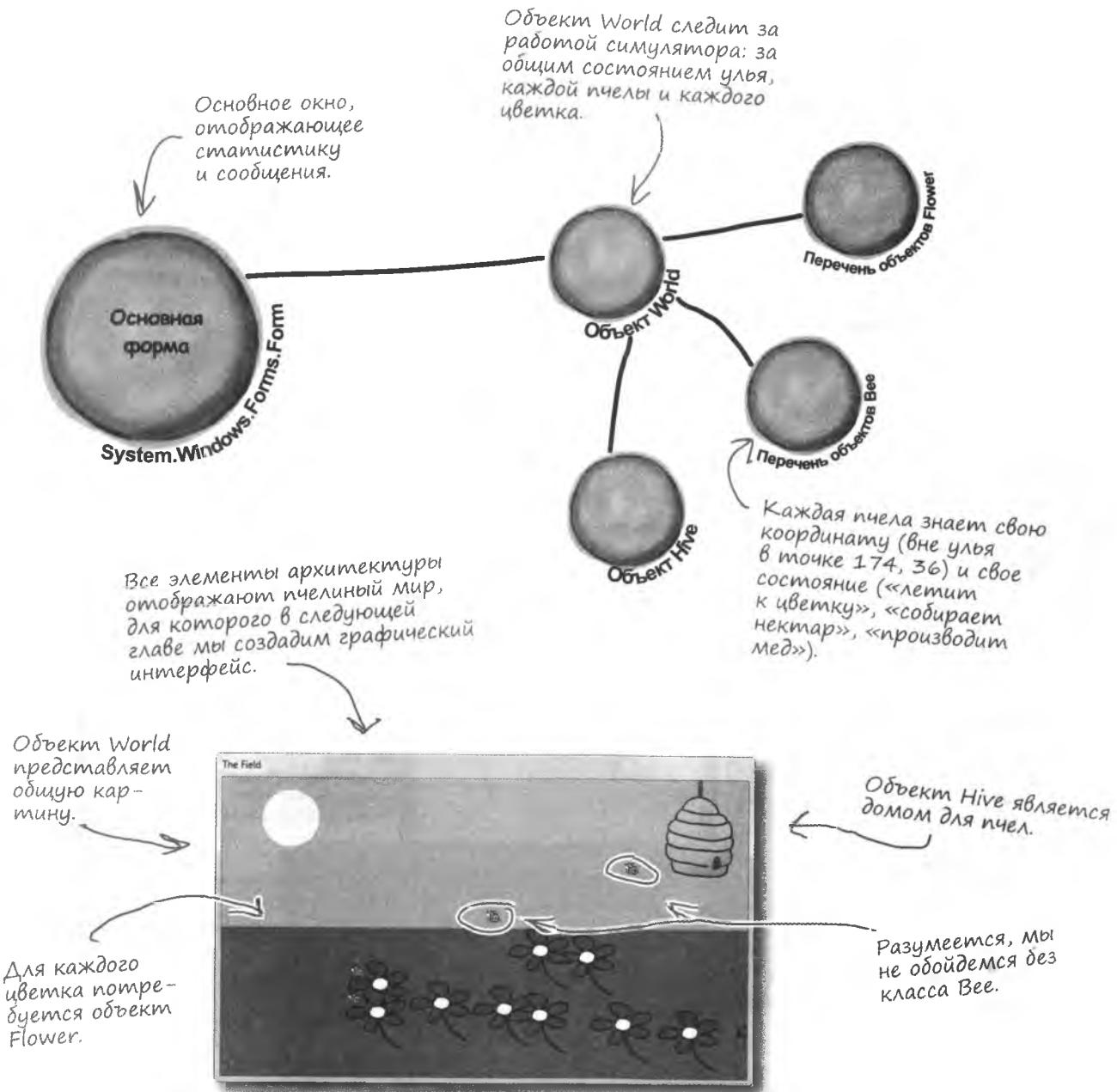
Beehive Simulator	
Pause simulation	Reset
# Bees	6
# Flowers	11
Total honey in the hive	1.200
Total nectar in the field	34.390
Frames run	983
Frame rate	16 (62.5ms)
Idle: 1 bee	
FlyingToFlower: 2 bees	
GatheringNectar: 1 bee	
ReturningToHive: 2 bees	

Можно даже наблюдать за работой пчел в поле.

не слишком сложно... да?

## Архитектура симулятора улья

Вот как выглядит улей изнутри. Несмотря на необходимость контролировать множество пчел, общая модель довольно проста.



## Построение симулятора улья

Мы пока не создавали проектов такого уровня сложности, поэтому на подготовительную работу придется затратить пару глав. Вам предстоит научиться работать с таймерами, освоить встроенный язык запросов LINQ и приемы работы с графикой.

Вот чем вам предстоит заняться в этой главе:

- 1** Создать класс `Flower`, в котором цветы распускаются, дают нектар, вянут и умирают.
- 2** Создать класс `Bee` с набором различных состояний (сбор нектара, возвращение в улей) и набором заданий в зависимости от состояния.
- 3** Создать класс `Hive`, содержащий вход, выход, инкубатор пчел и фабрику для переработки собранного нектара в мед.
- 4** Создать класс `World`, управляющий ульем, цветами и пчелами в каждый момент времени.
- 5** Построить форму для собора статистики о состоянии классов и управления ульем.



## Упражнение Решение

Перейдем к написанию кода. Начнем с класса Flower. Он содержит информацию о положении цветка, возрасте и продолжительности жизни. Нужно сделать так, чтобы цветы со временем старели и умирали. Этим вы сейчас и займитесь.

Скелетом класса называется объявление полей, свойств и методов, для которых пока не написана реализация.

1

### Основа класса Flower

На основе диаграммы класса Flower напишите его скелет. Location (Местоположение), Age (Возраст), Alive (Живой), Nectar (Нектар) и NectarHarvested (Собранный нектар) – **автоматические свойства**. Последнее свойство перезаписываемо, в то время как первые четыре предназначены только для чтения.

Все эти свойства, кроме NectarHarvested, предназначены только для чтения.

Это поле используется только внутри класса, поэтому его нужно сделать закрытым.

Flower
Location: Point
Age: int
Alive: bool
Nectar: double
NectarHarvested: double
lifespan: int
HarvestNectar(): double
Go()

После двоеточия указан тип переменной...

...или тип возвращаемого методом значения.

2

### Необходимые константы

Добавим в класс Flower шесть констант:

Константы обычно не показываются на диаграмме классов.

- LifeSpanMin – минимальная продолжительность жизни цветка.
- LifeSpanMax – максимальная продолжительность жизни цветка.
- InitialNectar – сколько нектара изначально содержит цветок.
- MaxNectar – максимальное количество нектара, которое можно собрать с цветка.
- NectarAddedPerTurn – количество нектара, добавляемое по мере роста цветка.
- NectarGatheredPerTurn – как много нектара удается собрать за один цикл.

Нужно, взяв за основу значение каждой константы, выбрать для нее тип. Цветы живут от 15 000 до 30 000 циклов, и в начальный момент имеют 1.5 единицы нектара. Количество нектара может доходить до 5 единиц. За один цикл прибавляется 0.01 единицы нектара, и 0.3 единицы может быть собрано.

Анимированные симуляторы создаются кадр за кадром. Термины «кадр», «цикл» и «очередь» в нашем случае являются взаимозаменяемыми.

В верхней части класса, использующего тип `Point`, должна быть строка `using System.Drawing;`.

3

### Построение конструктора

Конструктор должен передавать структуру `Point`, задающую положение цветка, и экземпляр класса `Random`. Нужно указать, что возраст цветка равен 0, что он живой, а также присвоить ему начальное количество нектара. Наконец, требуется рассчитать продолжительность жизни цветка. Вот строчка, которая вам поможет:

```
lifeSpan = random.Next(LifeSpanMin, LifeSpanMax + 1);
```



Это работает только при условии правильного именования переменных, констант и аргументов, передаваемых конструктору `Flower`.

4

### Метод `HarvestNectar()`

При каждом вызове этого метода проверяется, достигло ли количество собранного нектара максимума, который могут дать цветы. В этом случае метод возвращает 0. В противном случае возвращается количество собранного нектара, которое добавляется к переменной `NectarHarvested`, хранящей информацию об общем сборе с каждого цветка.



Подсказка: В этом методе используются только константы `NectarGatheredPerTurn`, `Nectar` и `NectarHarvested`.

5

### Метод `Go()`

Каждый вызов этого метода соответствует проживанию цветком одного цикла, значит, должен обновляться и его возраст. Нужно проверять, не достигла ли эта переменная заданного для цветка времени жизни. В случае положительного результата цветок умирает.

К живым цветкам нужно добавить количество произведенного за цикл нектара. Проверяйте, не превышает ли оно максимально возможное значение.

Результатом будет **анимация** с маленькими летающими фигурками пчел. Метод `Go()` вызывается для каждого кадра, при том что показывается несколько кадров в секунду.



Решение на следующей странице... но сначала попробуйте написать и скомпилировать код!



## Упражнение

### Решение

Вот как выглядит класс Flower для симулятора улья.

```

class Flower {
    private const int LifeSpanMin = 15000;
    private const int LifeSpanMax = 30000;
    private const double InitialNectar = 1.5;
    private const double MaxNectar = 5.0;
    private const double NectarAddedPerTurn = 0.01;
    private const double NectarGatheredPerTurn = 0.3;
    public Point Location { get; private set; }
    public int Age { get; private set; }
    public bool Alive { get; private set; }
    public double Nectar { get; private set; }
    public double NectarHarvested { get; set; }
    private int lifeSpan;

    public Flower(Point location, Random random) {
        Location = location;
        Age = 0;
        Alive = true;
        Nectar = InitialNectar;
        NectarHarvested = 0;
        lifeSpan = random.Next(LifeSpanMin, LifeSpanMax + 1);
    }

    public double HarvestNectar() {
        if (NectarGatheredPerTurn > Nectar)
            return 0;
        else {
            Nectar -= NectarGatheredPerTurn;
            NectarHarvested += NectarGatheredPerTurn;
            return NectarGatheredPerTurn;
        }
    }

    public void Go() {
        Age++;
        if (Age > lifeSpan)
            Alive = false;
        else {
            Nectar += NectarAddedPerTurn;
            if (Nectar > MaxNectar)
                Nectar = MaxNectar;
        }
    }
}

```

Свойства  
Location, Age,  
Alive и Nectar  
предназначены  
только для  
чтения.

Цветы име-  
ют различную  
продолжитель-  
ность жизни.

Метод Go() вызы-  
вается для каждого  
кадра. Возраст  
цветка за один цикл  
увеличивается не-  
значимельно, но по  
мере работы симу-  
лятора эти значе-  
ния суммируются.

Flower
Location: Point
Age: int
Alive: bool
Nectar: double
NectarHarvested: double
lifeSpan: int
HarvestNectar(): double
Go()

Доступ к полю  
NectarHarvested  
должны иметь  
и другие классы.

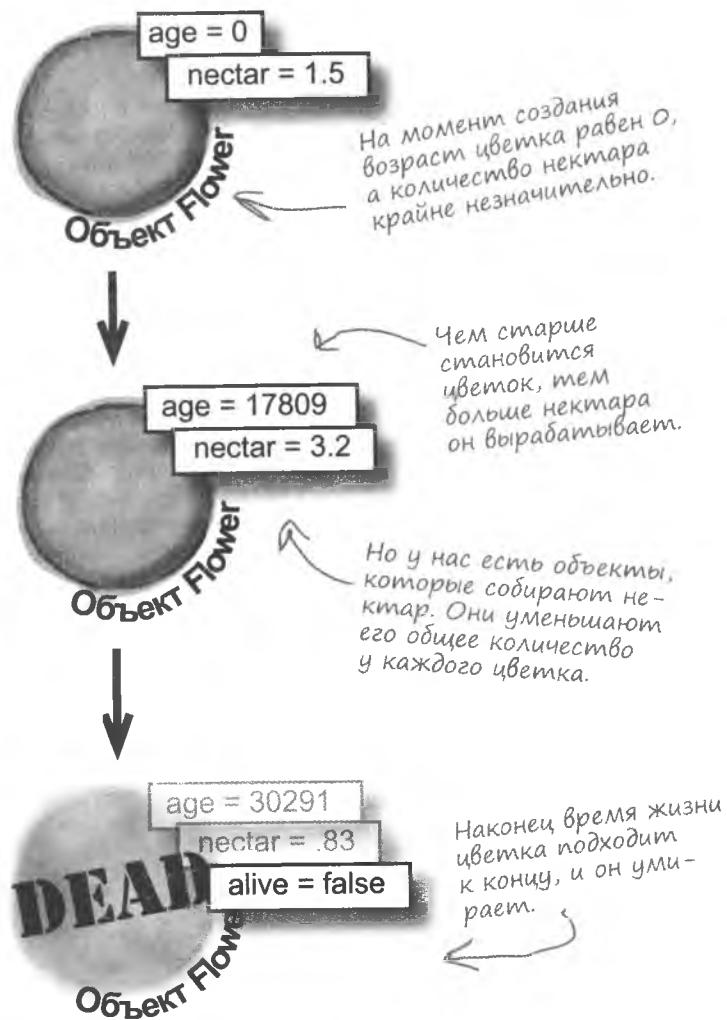
Пчелы вызывают метод  
HarvestNectar() для сбора нектара.  
За один раз пчела забирает совсем  
немного нектара, поэтому метод  
отслеживает состояние цветка  
на протяжении ряда кадров, пока  
нектар не закончится.

Нектар следует  
добавлять только  
живым цветкам.

Переменные типа Point принадлежат пространству имен  
System.Drawing, Поэтому не забудьте добавить строчку using  
System.Drawing; в верхнюю часть файла класса.

## Жизнь и смерть цветка

Каждый цветок проходит базовый цикл, растет, накапливает нектар, отдает этот нектар пчелам и наконец умирает:



### часто задаваемые вопросы

**В:** Переменная `NectarHarvested` только увеличивает свое значение, а больше нигде в классе не используется. Зачем она нужна?

**О:** Это заготовка на будущее. Симулятор должен отслеживать состояние цветов и количество собранного нектара. Эта переменная была оставлена в открытом доступе, именно потому что она потребуется другим классам.

**В:** Почему все автоматические свойства предназначены только для чтения?

**О:** Помните разговор в главе 5 об уровнях доступа? В данном случае другие объекты симулятора, то есть улей и пчелы, должны иметь возможность чтения этих полей, но не записи в них. При этом вы можете их редактировать внутри класса при помощи закрытого метода записи.

**В:** Мой код отличается от вашего. Я сделал какую-то ошибку?

**О:** Вы могли написать код по-другому, но если он **функционирует** так же, как и наш, все в порядке. Особенность инкапсуляции в том, что внутренняя структура классов не важна для других классов, нужно только, чтобы класс выполнял свою функцию.

## МОЗГОВОЙ ШТУРМ

Метод `Go()` увеличивает возраст цветка на 1 при продолжительности жизни от 15 000 до 30 000 циклов. То есть метод `Go()` для одного цветка будет вызван по меньшей мере 15 000 раз. Как обработать такое количество вызовов?  
А в случае 10 цветов? 100? 1000?

## Класс Bee

Для цветов, готовых дать нектар, понадобится класс Bee, в котором содержатся сведения о возрасте пчелы, ее местоположении и возможном количестве собираемого нектара. В класс был также добавлен метод, перемещающий пчелу в нужную точку пространства.

```
class Bee {
    private const double HoneyConsumed = 0.5;
    private const int MoveRate = 3;
    private const double MinimumFlowerNectar = 1.5;
    private const int CareerSpan = 1000;

    public int Age { get; private set; }
    public bool InsideHive { get; private set; }
    public double NectarCollected { get; private set; }

    private Point location;
    public Point Location { get { return location; } }

    private int ID;
    private Flower destinationFlower;

    public Bee(int id, Point location) {
        this.ID = id;
        Age = 0;
        this.location = location;
        InsideHive = true;
        destinationFlower = null;
        NectarCollected = 0;
    }

    public void Go(Random random) {
        Age++;
    }
}
```

Как и в случае с классом Flower, вам нужно объявить несколько констант, связанных только с этим классом.

При помощи константы MinimumFlowerNectar пчела определяет пригодные для сбора нектара цветы.

Для задания местоположения использовалось вспомогательное поле. Автоматические свойства в данном случае не помогут, так как метод MoveTowardsLocation() не в состоянии задать значения своих членов напрямую (Location.X -= MoveRate).

Пчела должна знать свой ID и свое начальное местоположение.

Пчелы начинают свои действия из улья. В начальный момент времени нет цветов, на которые можно лететь, как нет и собранного нектара.

Метод Go() еще не закончен. Но для начала этого достаточно.

Метод Math.Abs() вычисляет модуль разности между пунктом назначения и текущим положением.

```
private bool MoveTowardsLocation(Point destination) {
    if (Math.Abs(destination.X - location.X) <= MoveRate &&
        Math.Abs(destination.Y - location.Y) <= MoveRate)
        return true;
```

**Метод возвращает значение true при достижении пчелой пункта назначения.**

```
if (destination.X > location.X)
    location.X += MoveRate;
else if (destination.X < location.X)
    location.X -= MoveRate;
```

```
if (destination.Y > location.Y)
    location.Y += MoveRate;
else if (destination.Y < location.Y)
    location.Y -= MoveRate;
```

```
return false;
```

Метод возвращает значение false, так как пункты назначения не достигнут и нужно продолжать движение.

Для начала мы проверяем, не находится ли пчела на меньшем, чем MoveRate расстоянии от пункта назначения.

Если пчела находится недостаточно близко к пункту назначения, она смещается в его направлении на величину, определяемую коэффициентом смещения.

Метод MoveTowardsLocation() смещает положение пчелы, меняя ее координаты X и Y. Как только пчела достигает пункта назначения, он возвращает значение true.



## Упражнение

Ниже перечислен список действий, которые должны выполнять пчелы. Создайте для объекта Bee перечисление BeeState (Состояние пчелы). Для каждой из пчел потребуется предназначено только для чтения автоматическое свойство CurrentState (Текущее состояние). В начальный момент времени пчела находится в состоянии idle (Незанятый). Добавьте к методу Go() оператор switch с параметрами для каждого элемента перечисления.

Элемент	Функция
Idle	Пчела ничем не занята
FlyingToFlower	Пчела летит на цветок
GatheringNectar	Пчела собирает нектар
ReturningToHive	Пчела возвращается в улей
MakingHoney	Пчела производит мед
Retired	Пчела прекращает работу



## Упражнение Решение

На основе данного на предыдущей странице списка пчелиных занятий для класса Bee было создано перечисление BeeState. Закрытое поле currentState предназначено для отслеживания состояния каждой пчелы.

```
enum BeeState {  
    Idle,  
    FlyingToFlower,  
    GatheringNectar,  
    ReturningToHive,  
    MakingHoney,  
    Retired  
}  
  
class Bee {  
    // объявление констант  
    // объявление переменных  
  
    public BeeState CurrentState { get; private set; }  
  
    public Bee(int ID, Point initialLocation) {  
        this.ID = ID;  
        Age = 0;  
        location = initialLocation;  
        InsideHive = true;  
        CurrentState = BeeState.Idle; ←  
        destinationFlower = null; ← Сначала пчелы ничего  
        NectarCollected = 0; ← не делают.  
    }  
}
```

Это перечисление возможных состояний пчелы.

Переменная, отслеживающая состояние каждой пчелы.

Вы не забыли добавить в верхнюю часть файла класса строку `using System.Drawing;?`

```

public void Go(Random random) {
    Age++;
    switch (CurrentState) {
        case BeeState.Idle:
            if (Age > CareerSpan) {
                CurrentState = BeeState.Retired;
            } else {
                // Что мы делаем в состоянии idle?
            }
            break;
        case BeeState.FlyingToFlower:
            // Перемещение пчелы к выбранному цветку
            break;
        case BeeState.GatheringNectar:
            double nectar = destinationFlower.HarvestNectar();
            if (nectar > 0)
                NectarCollected += nectar;
            else
                CurrentState = BeeState.ReturningToHive;
            break;
        case BeeState.ReturningToHive:
            if (!InsideHive) {
                // Перемещение в улей
            } else {
                // Что пчела делает внутри улья?
            }
            break;
        case BeeState.MakingHoney:
            if (NectarCollected < 0.5) {
                NectarCollected = 0;
                CurrentState = BeeState.Idle;
            } else {
                // Перерабатываем нектар в мед
            }
            break;
        case BeeState.Retired:
            // Ничего не делаем. Работа закончена!
            break;
    }
}

```

**Оператор switch() обрабатывает состояние каждой пчелы.**

Мы ввели код обработки некоторых состояний. Ничего страшного, если вы не смогли написать его самостоятельно, просто воспользуйтесь нашим вариантом.

Когда переменная age сравнивается с переменной lifespan, пчела уходит на пенсию. Но перед этим она должна закончить все данные ей задания.

Этот код будет написан немного позже.

Мы собираем нектар с выбранного цветка...

...если нектара еще остался, добавьте его к уже собранному...

...а если нектара не осталось, возвращайтесь в улей.

Возвращение в улей зависит от того, где в данный момент находится пчела.

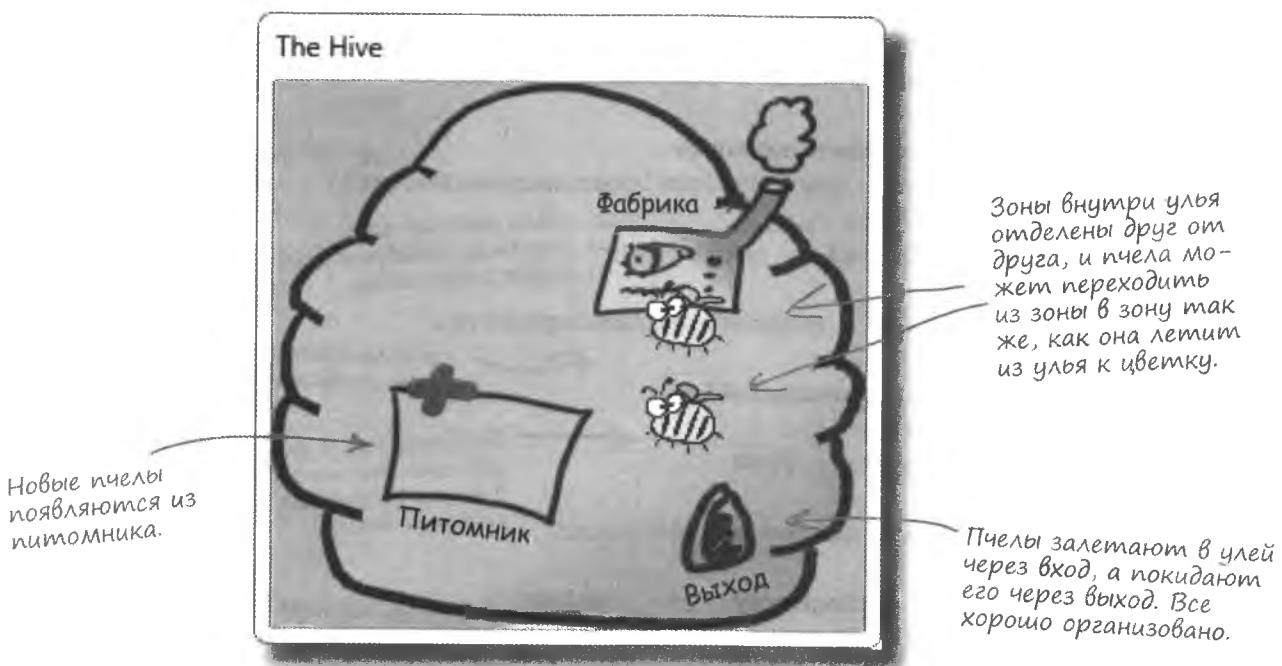
Пчела отдает на перерабатывающую фабрику половину единицы нектара за раз. С меньшими количествами нектара фабрика не работает, поэтому пчела от них просто избавляется.

Требуется обработать все возможные состояния.

## Программисты против бездомных пчел

У нас есть пчелы и полные нектара цветы. Нужно написать код для сбора нектара, но перед этим задумаемся о том, откуда появились пчелы? И куда они понесут весь этот нектар? Ответом на вопрос будет класс Hive.

Улей – не просто место, куда возвращаются пчелы. Внутри он разделен на зоны различного назначения. Существуют вход и выход, питомник для выращивания молодых пчел и фабрика по переработке нектара в мед.



## Для жизни улья нужен мед

Существование улья зависит от количества хранимого в нем меда. Мед нужен для функционирования старых и создания новых пчел. Соответственно, нам потребуется фабрика, перерабатывающая собранный пчелами нектар в мед. Из единицы нектара получается 0.25 единицы меда.

Обдумайте эту информацию... мед нужен для функционирования улья и создания новых пчел. Уже имеющиеся пчелы собирают нектар, который перерабатывается в мед и позволяет улью работать дальше.

Именно это (с небольшой помощью с нашей стороны) вам предстоит смоделировать.



## Упражнение

Ваша задача — создать класс Hive.

1

### Напишите скелет класса

Как и в случае класса Flower, нужно начать со скелета. Диаграмма класса показана справа. Сделайте свойство Honey пред назначенным только для чтения, поле locations должно быть закрытым, как и поле beeCount.

Hive
Honey: double
locations: Dictionary<string, Point>
beeCount: int
InitializeLocations()
AddHoney(Nectar: double): bool
ConsumeHoney(amount: double): bool
AddBee(random: Random)
Go(random: Random)
GetLocation(location: string): Point

2

### Определите константы

Укажите начальное количество пчел (6), меда (3.2), максимально количество меда (15), коэффициент переработки нектара в мед (.25), максимальное количество пчел (8) и минимальное количество меда, при котором появляются новые пчелы (4).

3

### Код, работающий с полем Location

Напишите метод GetLocation(). Он должен брать строку, сравнивать ее со словарными значениями и возвращать структуру point. При отсутствии строки должно появляться исключение ArgumentException.

Затем напишите метод InitializeLocations(), задающие координаты элементов улья:

- Вход (600, 100)
- Питомник (95, 174)
- Фабрика (157, 98)
- Выход (194, 213)

Все это привязано к точкам в двумерном пространстве улья.

Присвойте константам имена InitialBees, InitialHoney, MaximumHoney, NectarHoneyRatio, MaximumBees и MinimumHoneyForCreatingBees соответственно. При выборе типов констант помните не только о начальных значениях, но и о значениях выражений, в которых эти константы будут фигурировать.

В нашем симуляторе используется улей с фиксированными координатами. При написании симулятора с набором ульев можно при желании задавать относительные, а не абсолютные координаты.

4

### Конструктор Hive

Конструктор должен задавать начальное количество меда в улье и положение внутренних элементов, а также создавать экземпляр Random. Затем для каждой рождающейся пчелы нужно вызывать метод AddBee(), передавая ему только что созданный экземпляр Random.

Объект Random добавляет случайное значение к положению объекта Nursery. Это позволит избежать появления пчел в одной точке.



## Решение

Вам нужно было начать построение класса Hive.

Убедитесь в наличии строчки «`using System.Drawing;`» так как код использует значения типа `Point`.

Вы можете выбрать для констант и другие имена.

```

class Hive {
    private const int InitialBees = 6;
    private const double InitialHoney = 3.2;
    private const double MaximumHoney = 15.0;
    private const double NectarHoneyRatio = .25;
    private const double MinimumHoneyForCreatingBees = 4.0;
    private const int MaximumBees = 8;

    private Dictionary<string, Point> locations;
    private int beeCount = 0;

    public double Honey { get; private set; }

    private void InitializeLocations() {
        locations = new Dictionary<string, Point>();
        locations.Add("Entrance", new Point(600, 100));
        locations.Add("Nursery", new Point(95, 174));
        locations.Add("HoneyFactory", new Point(157, 98));
        locations.Add("Exit", new Point(194, 213));
    }

    public Point GetLocation(string location) {
        if (locations.ContainsKey(location))
            return locations[location];
        else
            throw new ArgumentException("Unknown location: " + location);
    }

    public Hive() {
        Honey = InitialHoney;
        InitializeLocations();
        Random random = new Random();
        for (int i = 0; i < InitialBees; i++)
            AddBee(random);
    }

    public bool AddHoney(double nectar) { return true; }
    public bool ConsumeHoney(double amount) { return true; }
    private void AddBee(Random random) { }
    public void Go(Random random) { }
}

Можно добавить исключение NotImplementedException ко всем методам, для которых пока не написана реализация. Это позволит обнаружить места, в которые следует дописать код.
  
```

Константа `MaximumHoney` меняется в диапазоне от `InitialHoney` (3.2) до этого значения, поэтому мы выбрали тип `double`. Ведь именно этому типу принадлежит константа `InitialHoney`.

В нашем словаре хранятся строки, описывающие положение в улье.

Не забудьте создать экземпляр `Dictionary`, или это не будет работать.

Остальная часть метода очевидна.

Этот метод не позволяет другим классам вносить изменения в словарь, то есть перед нами пример инкапсуляции.

Метод `AddBee()` должен быть вызван для каждой зарождающейся в улье пчелы.

Этот код еще не написан, используйте пустые методы в качестве местозаполнителя.



Странный способ  
написания кода. Пчелы пока не знают  
о цветах, а улей полон пустых объявлений  
методов. И ничего не работает, не так ли?

#### Код составляется из кусочков.

Было бы прекрасно, если бы код класса писался за один раз, компилировался, тестиировался и откладывался в сторону, чтобы перейти к работе над следующим классом. К сожалению, это невозможно.

В большинстве случаев код пишется фрагмент за фрагментом. Мы практически написали класс Flower, а вот для класса Bee это сделать не получилось. Вам нужно дописать, что следует делать в каждом из состояний.

И класс Hive остался с множеством пустых методов. Кроме того, пчелы пока никак не связаны с ульем. Плюс нерешенная проблема с многократным вызовом метода Go () ...

Но мы еще даже не приступили  
к соединению классов друг с другом!  
Мы разобрались с архитектурой  
и приступили к построению.

#### Сначала разрабатываем, затем строим

Мы начали проект с понимания того, что именно мы хотим получить в итоге: симулятор улья. Мы многое знаем о взаимодействии таких объектов как пчелы, цветы, улей и окружающий мир. Именно поэтому мы начали с **архитектуры**, дающей представление о том, как классы работают друг с другом. Только после этого можно переходить к разработке каждого из классов.

Работать над проектом намного проще, если вы **заранее** представляете конечный результат. Это вполне очевидно. И именно это позволяет внести в конечный продукт значительные изменения.



## Построение класса Hive

Вернемся к классу Hive и напишем код для отсутствующих пока методов:

```

class Hive {
    // объявление констант
    // объявление переменных

    // InitializeLocations()
    // GetLocation()
    // Hive constructor

    public bool AddHoney(double nectar) {
        double honeyToAdd = nectar * NectarHoneyRatio;
        if (honeyToAdd + Honey > MaximumHoney)
            return false;
        Honey += honeyToAdd; ← При наличии места
        добавим мед.
        return true;
    }

    public bool ConsumeHoney(double amount) {
        if (amount > Honey)
            return false; ← Если количество меда меньше, чем требуется
        else { ← и пчелам, метод возвращает значение false.
            Honey -= amount; ← Если меда достаточно, он изымается
            return true; ← из хранилища, и метод возвращает
            } ← значение true.
        }
    }

    private void AddBee(Random random) {
        Создаваем beeCount++;
        int r1 = random.Next(100) - 50; ←
        жем только int r2 = random.Next(100) - 50; ←
        ко экземпляру Hive. Point startPoint = new Point(locations["Nursery"].X + r1,
                                                       locations["Nursery"].Y + r2);
        Bee newBee = new Bee(beeCount, startPoint);
        // Новую пчелу следует добавить в существующую систему
    }

    public void Go(Random random) { }
}

```

Сначала определим, какое количество меда может быть получено из нектара...

...и проверим, есть ли в улье место для такого количества меда.

Этот метод берет переменную amount (количество меда) и отправляет мед из хранилища потребителям.

Здесь создается точка, отстоящая от питомника на 50 единиц по осям X и Y.

Новая пчела появляется в месте с вычисленной координатой.

Скоро мы закончим метод AddBee() и напишем метод Go()...

## Метод Go() для улья

Мы уже писали метод Go() для классов Flower и Bee (хотя и не закончили эту работу). Вот метод Go() для класса Hive:

```
public void Go(Random random) {
    if (Honey > MinimumHoneyForCreatingBees)
        AddBee(random);
}
```

Единственным ограничением (по крайней мере на данный момент) является необходимое для создания новых пчел количество меда.

← Тот же экземпляр Random, который передавался методу Go(), передается методу AddBee().

К сожалению, эта модель далека от реальности. Часто пчелиная матка улья оказывается настолько занята, что не имеет времени на создание новых пчел. У нас нет класса QueenBee, но предположим, что при наличии необходимого количества меда новые пчелы появляются в 10% случаев. Это можно написать так:

```
public void Go(Random random) {
    if (Honey > MinimumHoneyForCreatingBees
        && random.Next(10) == 1)
        AddBee(random);
}
```

Таким образом легко смоделировать создание пчелы в одном из 10 случаев. Метод Next возвращает случайное число от 0 до 9. Если это число равно 1, создается новая пчела.

### Задаваемые вопросы

**В:** Может ли улей создать бесконечное количество пчел?

**О:** В настоящий момент да, но такая ситуация далека от реальности. Позднее мы добавим ограничение на количество пчел, которые могут одновременно существовать в улье.

**В:** Можно ли назначить экземпляру Random свойству класса вместо передачи его методу AddBee()?

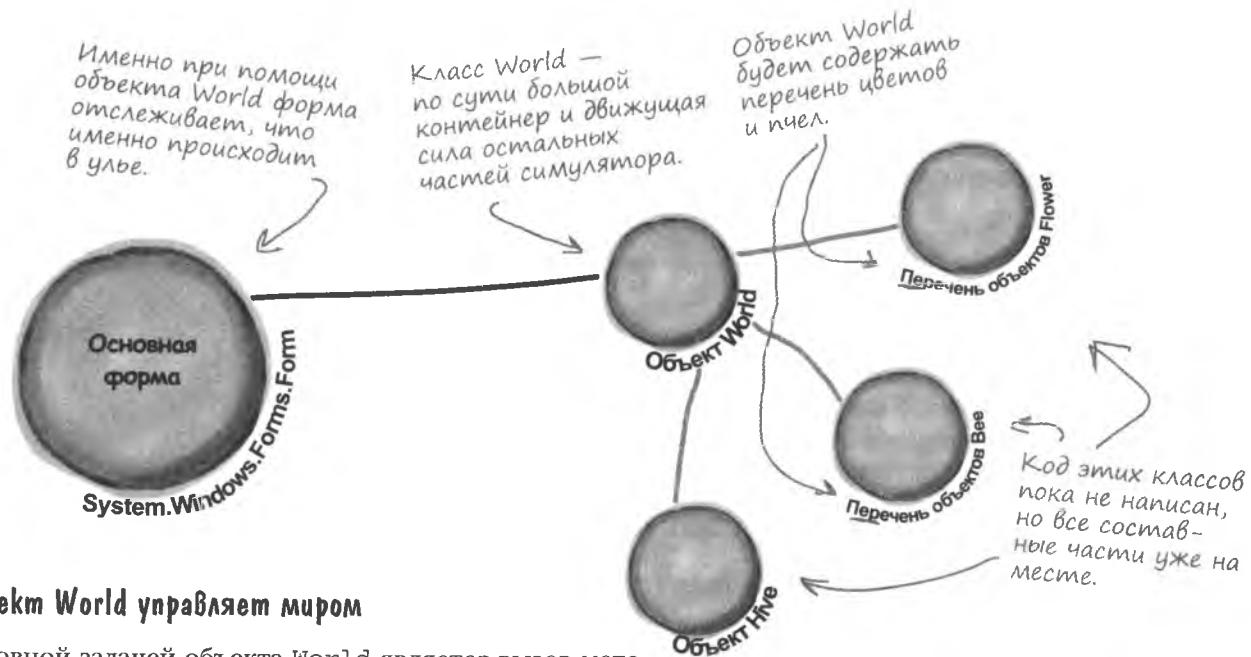
**О:** Разумеется. После этого метод AddBee будет использовать это свойство вместо передаваемого ему параметра. Но вряд ли имеет смысл так поступать.

**В:** Я все еще не понимаю, как будут вызываться все эти методы Go().

**О:** Мы как раз собираемся рассмотреть этот вопрос. Для начала нам потребуется класс World, отслеживающий происходящее в улье, состояние всех пчел и даже всех цветов.

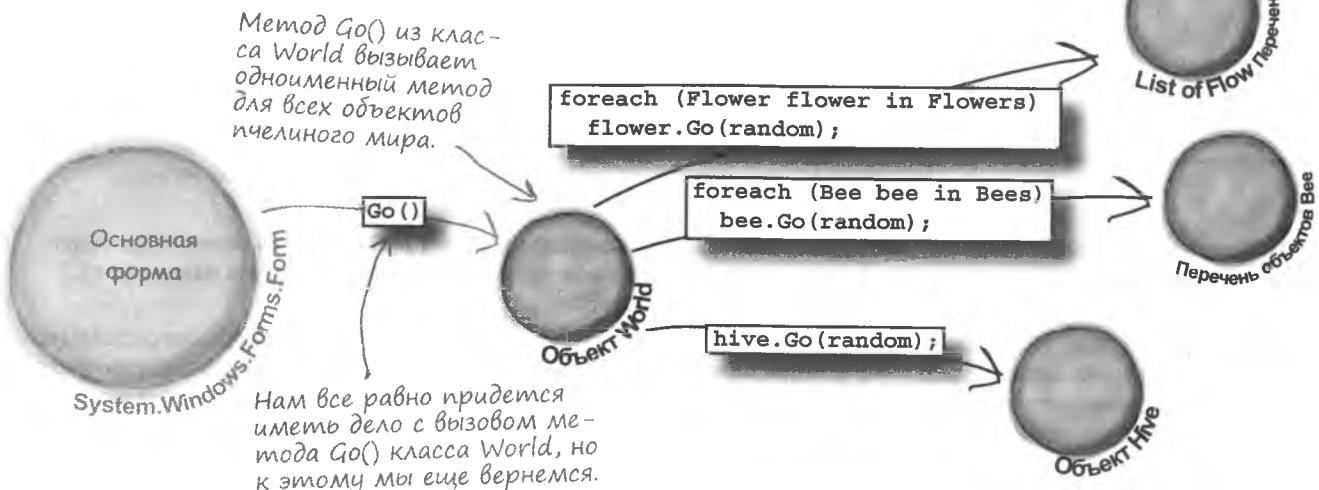
## Все готово для класса World

Имея практически готовые классы Hive, Bee и Flower, можно приступить к построению класса World, предназначенного для координации составных частей нашего симулятора. Именно он будет поддерживать жизнедеятельность пчел, определять наличие места для новых пчел, размещать цветы и т. п.



### Объект World управляет миром

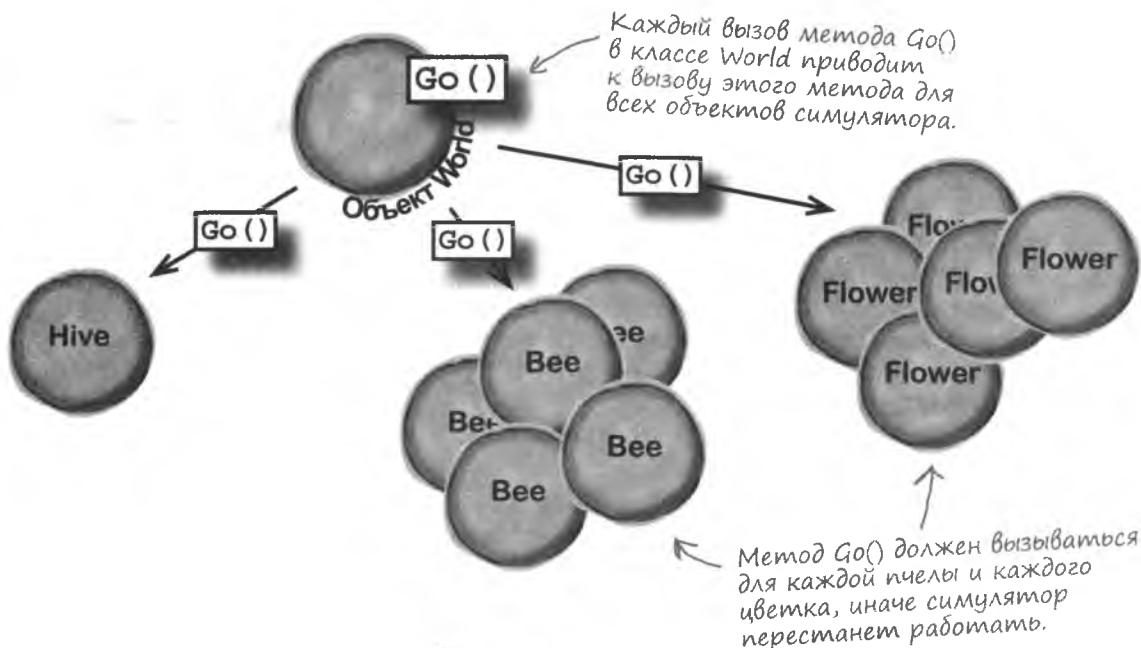
Основной задачей объекта World является вызов метода Go () в каждом кадре для каждого экземпляра Flower, Bee и Hive. Другими словами, объект World обеспечивает существование пчелиного мира.



## Система, основанная на кадрах

Метод `Go()` для каждого из объектов должен вызываться в **каждом кадре** нашего симулятора. Под кадром в данном случае подразумевается произвольное количество времени: например, картинка может обновляться каждые 10 секунд или каждые 60 секунд либо каждые 10 минут.

Смена кадров влияет на состояние каждого объекта. Улей стареет и проверяет, не требуются ли новые пчелы. Пчелы пролетают небольшой отрезок пути по направлению к пункту назначения, выполняют маленький кусок работы или стареют. Каждый цветок производит небольшое количество нектара и тоже стареет. Класс `World` отслеживает, чтобы при каждом вызове метода `Go()` с каждым объектом происходили необходимые изменения.



### Возьми в руку карандаш



При построении симулятора будет использован один из основных принципов объектно-ориентированного программирования — инкапсуляция. Приведите по два примера инкапсуляции для каждого из созданных нами классов.

#### Hive

1. ....

2. ....

#### Bee

1. ....

2. ....

#### Flower

1. ....

2. ....

## Код для класса World

Класс World – один из самых простых в нашем симуляторе. Вот отправная точка для написания кода. Если присмотреться, вы увидите отсутствие отдельных фрагментов, которые мы совсем скоро добавим.

```
using System.Drawing;
class World {
    private const double NectarHarvestedPerNewFlower = 50.0;
    private const int FieldMinX = 15;
    private const int FieldMinY = 177; } Координаты,
    private const int FieldMaxX = 690; } определяющие границы
    private const int FieldMaxY = 290; } цветочного поля.

    public Hive Hive; ← В мире существует
    public List<Bee> Bees; ← всего один улей,
    public List<Flower> Flowers; ← множество пчел
                                и цветков.

    public World() {
        Bees = new List<Bee>();
        Flowers = new List<Flower>();
        Random random = new Random();
        for (int i = 0; i < 10; i++)
            AddFlower(random);
    }

    public void Go(Random random) { Здесь мы просто вызываем для улья метод Go(),
        Hive.Go(random); ← передавая ему экземпляр Random.

        for (int i = Bees.Count - 1; i >= 0; i--) {
            Bee bee = Bees[i]; ← Метод Go() вызывается для всех
            bee.Go(random); ← имеющихся в наличии пчел.
            if (bee.CurrentState == BeeState.Retired)
                Bees.Remove(bee); ← Вышедшую на пенсию
                                пчелу нужно удалить
                                из нашего мира.

        double totalNectarHarvested = 0;
        for (int i = Flowers.Count - 1; i >= 0; i--) {
            Flower flower = Flowers[i]; ← Метод Go() вызывается для каждого
            flower.Go(); ← цветка.
            totalNectarHarvested += flower.NectarHarvested; ← Нужно отслеживать
            if (!flower.Alive) ← количество нектара,
                Flowers.Remove(flower); ← собранное на каждом
                                этапе. Поэтому мы
                                суммируем нектар,
                                собранный с каждого
                                цветка.

        }
    }
}
```

Обратите внимание на открытые поля Hive, Bees и Flowers. Другой класс может случайно присвоить им значение null, что станет причиной серьезных проблем! Подумайте, какими свойствами и методами можно воспользоваться для инкапсуляции.

# Возьми в руку карандаш

## Решение

Вот те, которые пришли нам в голову. У вас есть другие варианты?

### Hive

1. Закрыт словарь Locations.
2. Это дает пчелам метод добавления меда.

При написании симулятора использовался один из основных принципов объектно-ориентированного программирования, инкапсуляция. Вот примеры инкапсуляции для каждого из созданных нами классов.

### Bee

1. Положение пчелы в пространстве является свойством только для чтения.
2. Аналогично с ее возрастом. Эти свойства нельзя изменять из другого класса.

### Flower

1. Этот класс обеспечивает метод сбора некоторого количества нектара.
2. Логическое значение alive является закрытым.

```
if (totalNectarHarvested > NectarHarvestedPerNewFlower) {
```

```
    foreach (Flower flower in Flowers)
        flower.NectarHarvested = 0;
```

```
    AddFlower(random);
```

```
}
```

При достаточном количестве нектара на поле появляется новый цветок.

В процессе сбора некоторого количества нектара пчелы опыляют цветы. В нашем мире считается, что собрав достаточно количество нектара, пчелы инициируют появление нового цветка.

```
private void AddFlower(Random random)
```

```
{
```

```
    Point location = new Point(random.Next(FieldMinX, FieldMaxX),
                                random.Next(FieldMinY, FieldMaxY));
```

```
    Flower newFlower = new Flower(location, random);
```

```
    Flowers.Add(newFlower);
```

```
}
```

Обработчик выбирает положение на поле по случайному принципу...

...и помещает туда новый цветок.

## часто задаваемые вопросы

**В:** Почему бы не воспользоваться циклом `foreach` для удаления увядших цветов и отслуживших пчел?

**О:** Потому что удалить элемент коллекции изнутри просматривающего ее цикла `foreach` невозможно. Попытка сделать это приведет к появлению исключения.

**В:** А почему циклы `for` считать от конца списка к началу?

**О:** Цикл не должен нарушать нумерацию. Предположим, мы начали просматривать с начала список из пяти цветов, и оказалось, что один из них увял. Удалив цветок с индексом, например #3,

вы получите список из четырех пунктов, новый цветок, который будет записан под индексом #3, в следующий раз будет пропускаться, так как цикл перескочит на индекс #4.

Начав же просмотр с конца, вы никогда не пропустите цветок, помещенного на освободившееся место.



Все четыре класса написаны, теперь их нужно соединить. Следуйте данной ниже инструкции. Помните, что вам придется внести изменения практически в каждый класс.

**1 Обновите класс Bee.**

Пчелы должны узнать о существовании улья и внешнего мира. Поместите в конструктор класса Bee в качестве параметров ссылки на улей и мир и сохраните их для дальнейшего использования.

**2 Обновите класс Hive.**

Пчелы должны знать, что улей существует, улей же должен знать о существовании внешнего мира. Поместите в конструктор класса Hive ссылку на объект World и сохраните ее. Обновите код создания новых пчел, передав в него ссылки на улей и на мир.

**3 Обновите класс World.**

Обновите класс World таким образом, чтобы при создании нового улья ему передавалась ссылка на мир.



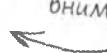
**СТОП!** На данный момент код уже должен компилироваться. Если этого не происходит, .....  
поиските ошибки.

**4 Внесите ограничение на создание пчел.**

В классе Hive есть константа MaximumBees, указывающая, сколько пчел в состоянии поддерживать улей (считается количество насекомых внутри и снаружи). Теперь, когда объект Hive имеет доступ к объекту World, ограничение можно усилить.



Подсказка: Обратите внимание на код создания или добавления пчел. В классе Hive он появляется дважды, поэтому будьте внимательны.



**5 Мир должен знать о создании новых пчел.**

Класс World использует перечисление List для слежения за имеющимися пчелами. Убедитесь, что новые пчелы попадают в список, состояние которого отслеживается объектом World.

## часто Задаваемые Вопросы

**В:** Зачем нужно исключение в методе `GetLocation()` класса `Hive`?

**О:** Для обработки некорректных данных, передаваемых в качестве параметра. Внутри улья есть несколько мест, но в метод `GetLocations()` можно передать произвольную строку. Что произойдет, если переданное значение отсутствует в словаре? Какой результат должен вернуть метод?

В случае некорректных параметров проще всего вызвать исключение `ArgumentException`. Вот как это делает метод `GetLocation()`:

```
throw new ArgumentException(  
    "Unknown location: " + location);
```

Этот оператор заставляет класс `Hive` показать исключение `ArgumentException` с сообщением «Неизвестное местоположение» и указанием места, которое программа не может найти.

Таким образом пользователь немедленно получает информацию о вводе некорректного параметра. Указав в сообщении этот параметр, вы предоставляете сведения, позволяющие немедленно устранить проблему.

**В:** Почему для описания местоположения была выбрана структура `Point`, если мы ничего не рисуем?

**О:** Вне зависимости от того, отмечаем мы местоположение пчелы на экране или нет, оно является ее характеристикой. Объект `Bee` отслеживает, где именно находится каждая из пчел. При каждом вызове метода `Go()` пчела перемещается на небольшое расстояние в направлении пункта назначения.

Нам нужно отслеживать положение пчел внутри улья и на поле, так как иначе невозможно узнать, достигли та или иная пчела пункта назначения.

**В:** Но почему для хранения информации о местоположении используется именно структура `Point`? Разве она предназначена не для рисунков?

**О:** Да, именно эту структуру используют визуальные элементы управления при описании своего свойства `Location`. Она применяется и при создании анимации. Но это вовсе не означает, что с ее помощью невозможно следить за положением других объектов. Можно, конечно, создать отдельный класс `BeeLocation` с целочисленными полями `X` и `Y`. Но зачем изобретать колесо, если можно воспользоваться уже имеющейся структурой?

**Всегда проще переделать существующий класс, функциональность которого, по большей части, совпадает с нужной вам, чем писать с нуля новый.**



## Упражнение Решение

Здесь вы увидите, каким же способом наши четыре класса соединяются друг с другом.

1

### Обновление класса Bee

Пчелы должны узнать о существовании улья и внешнего мира. Поместите в конструктор класса Bee в качестве параметров ссылки на улей и мир и сохраните их для дальнейшего использования.

```
class Bee {
    // объявление констант
    // объявление переменных
    private World world;
    private Hive hive;

    public Bee(int ID, Point InitialLocation, World world, Hive hive) {
        // код
        this.world = world;
        this.hive = hive; ←
    }
}
```

Это вполне очевидно...  
присвойте эти значения  
закрытым полям.

2

### Обновление класса Hive

Пчелы должны знать, что существует улей, улей же должен знать о существовании внешнего мира. Поместите в конструктор класса Hive ссылку на объект World и сохраните ее. Обновите код создания новых пчел, передав в него ссылку на улей и на мир.

```
class Hive {
    private World world; ←

    public Hive(World world) {
        this.world = world; ←
        // код
    }

    public void AddBee(Random random) {
        // код создания новых пчел
        Bee newBee = new Bee(beeCount, startPoint, world, this); ←
    }
}
```

Ссылка на мир дается первой, так как она используется в остальной части конструктора.

Новым пчелам  
нужна ссылка на  
окружающий мир,  
а также на улей.

**Если у вас не получается написать  
работающий код, скачайте готовую версию с сайта:  
<http://www.headfirstlabs.com/books/hfcsharp/>**

**3 Ограничение на создание пчел**

В классе Hive есть константа MaximumBees, указывающая, сколько пчел в состоянии поддерживать улей (учитывается количество насекомых внутри и снаружи). Теперь, когда объект Hive имеет доступ к объекту World, ограничение можно усилить.

```
public void Go(Random random) {  
    if (world.Bees.Count < MaximumBees  
        && Honey > MinimumHoneyForCreatingBees  
        && random.Next(10) == 1) {  
        AddBee(random);  
    }  
}
```

Объект World позволяет увидеть общее количество пчел и сравнить его с максимально возможным для данного улья.

Этот оператор сравнения мы ставим первым. Ведь если для новых пчел нет места, не имеет смысла проверять, достаточно ли в улье меда для их создания.

**4 Мир должен знать о создании новых пчел**

Класс World использует перечисление List для слежения за имеющимися пчелами. Убедитесь, что новые пчелы попадают в список, состояние которого отслеживается объектом World.

```
private void AddBee(Random random) {  
    beeCount++;  
    // Calculate the starting point  
    Point startPoint = // start the near the nursery  
    Bee newBee = new Bee(beeCount, startPoint, world, this);  
    world.Bees.Add(newBee);  
}
```

Это одна из причин необходимости ссылки на объект World в классе Hive.

Добавляется новая пчела.

**5 Обновление класса World.**

Обновите класс World таким образом, чтобы при создании нового улья ему передавалась ссылка на мир.

```
public World() {  
    Bees = new List<Bee>();  
    Flowers = new List<Flower>();  
    Hive = new Hive(this);  
    Random random = new Random();  
    for (int i = 0; i < 10; i++)  
        AddFlower(random);  
}
```

Здесь передается ссылка на объект Hive.

## Поведение пчел

У нас отсутствует метод Go() объекта Bee. Мы начали писать код для некоторых состояний, но многое осталось незаконченным (Idle, FlyingToFlower и частично MakingHoney).

Пришло время заполнить пробелы:

```

public void Go(Random random) {
    Age++;
    switch (CurrentState) {
        case BeeState.Idle:
            if (Age > CareerSpan) {
                CurrentState = BeeState.Retired;
            } else if (world.Flowers.Count > 0
                      && hive.ConsumeHoney(HoneyConsumed)) {
                Flower flower =
                    world.Flowers[random.Next(world.Flowers.Count)];
                if (flower.Nectar >= MinimumFlowerNectar && flower.Alive) {
                    destinationFlower = flower;
                    CurrentState = BeeState.FlyingToFlower;
                }
            }
            break;
        case BeeState.FlyingToFlower:
            if (!world.Flowers.Contains(destinationFlower))
                CurrentState = BeeState.ReturningToHive;
            else if (InsideHive) {
                if (MoveTowardsLocation(hive.GetLocation("Exit")))
                    InsideHive = false;
                location = hive.GetLocation("Entrance");
                Если пчела добралась до выхода, она может покинуть
                улей. Обновите ее положение. С поля же она будет
                лететь в сторону входа.
            }
            else
                if (MoveTowardsLocation(destinationFlower.Location))
                    CurrentState = BeeState.GatheringNectar;
            break;
        case BeeState.GatheringNectar:
            double nectar = destinationFlower.HarvestNectar();
            if (nectar > 0)
                NectarCollected += nectar;
            else
                CurrentState = BeeState.ReturningToHive;
            break;
    }
}

```

*Из состояния без-  
действия пчела  
должна переходить  
в состояние поиска  
цветка с нектаром.*

*Если остались цветы с несо-  
бранным нектаром и пчела  
собирает нужное для жиз-  
недеятельности количество  
меда. В противном случае  
пчела остается в состоянии  
бездействия.*

*Нужен другой  
живой цветок  
с нектаром.*

*Убедитесь, что за то  
время, пока пчела ле-  
тит к цветку, цветок  
не завянет.*

*Если пчела  
вылетела из  
улей, а цветок  
жив, она летит  
собирать с него  
нектар..*

*Если все условия  
соблюдены, пчела  
летит на новый  
цветок.*

*По этой причине  
мы передали  
ссылку на улей  
конструктору  
класса Bee.*



Это выход. Когда улей сохраняет положение Exit, это соответствует точке на форме объекта Hive, в которой располагается картинка с изображением выхода.



Это вход. Возвращаясь в улей, пчелы летят по направлению к входу на форме объекта field.

Поэтому в словаре location хранятся два отдельных местоположения Exit и Entrance.

```

case BeeState.ReturningToHive:
    if (!InsideHive) {
        if (MoveTowardsLocation(hive.GetLocation("Entrance")))
            InsideHive = true;
        location = hive.GetLocation("Exit");
    }
    else
        if (MoveTowardsLocation(hive.GetLocation("HoneyFactory")))
            CurrentState = BeeState.MakingHoney;
    break;
case BeeState.MakingHoney:
    if (NectarCollected < 0.5) {
        NectarCollected = 0;
        CurrentState = BeeState.Idle;
    }
    else
        if (hive.AddHoney(0.5))
            NectarCollected -= 0.5;
        else
            NectarCollected = 0;
    break;
case BeeState.Retired:
    // Do nothing! We're retired!
    break;
}

```

Отработавшая пчела должна ждать, пока объект Hive не удалит ее из перечисления. После этого она вольна делать, что хочет!

В случае проникновения в улей обновите положение и статус переменной insideHive.

Если вы находитесь в улье, отправляйтесь на фабрику переработки нектара.

Попытайтесь отдать собранный нектар.

Если улей может использовать этот нектар для производства меда...

...заберите его у пчелы.

Если улей полон, метод AddHoney() вернет значение false, и пчела выбросит собранный нектар, чтобы отправиться на следующее задание.

## МОЗГОВОЙ ШТУРМ

Каким образом следует отредактировать симулятор, чтобы пчела тратила два кадра на достижение цветка, а потом еще два кадра на возвращение в улей? Какие **методы** каких классов следует изменить, чтобы получить такое поведение?

## Основная форма

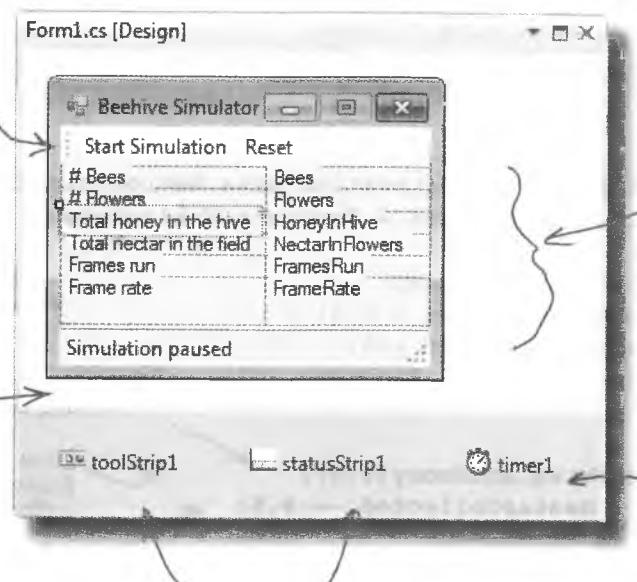
Как вы знаете при каждом вызове метода Go() все объекты нашего мира слегка меняются. Но как вызвать этот метод? Разумеется, при помощи формы! Пришло время заняться ее построением.

Добавьте форму к проекту и придайте ей показанный ниже вид. Она содержит новые элементы управления, назначение которых будет рассмотрено на следующих страницах.

Элементом ToolStrip помещаем в верхнюю часть формы панель инструментов. Воспользуйтесь раскрывающимся списком элемента в конструкторе и добавьте две кнопки. Присвойте параметру *DisplayStyle* кнопок значение *Text*.

Элементом StatusStrip добавляем в нижнюю часть формы строку состояния. Воспользуйтесь раскрывающимся списком этого элемента, чтобы добавить также *StatusLabel*.

Мы наконец-то пишем код для управления объектом *World*.



Метки в правом столбце будут отображать статистику. Присвойте им имена Bees, Flowers, HoneyInHive и т. п.

Каждая из этих меток занимает одну ячейку элемента управления TableLayoutPanel. Вы заполняете ее, как обычную таблицу в Microsoft Word. Щелкните на маленькой черной стрелке, чтобы добавить, удалить или поменять размер столбца или строки.

Добавьте элемент управления Timer. Он не является визуальным, поэтому появляется снизу от формы.

Элемент управления ToolStrip добавляет панель инструментов в верхнюю часть формы, а элемент StatusStrip — строку состояния в нижнюю. Но кроме этого, они появляются в виде значков под формой, чтобы дать вам доступ к редактированию своих свойств.

```
foreach (Flower flower in Flowers)  
    flower.Go(random);
```

```
foreach (Bee bee in Bees)  
    bee.Go(random);
```

```
hive.Go(random);
```



## Получение статистики

Отредактируем добавленные элементы управления. Мы не будем по очереди выделять каждый из них, а воспользуемся методом, обновляющим статистику симулятора:

```
private void UpdateStats(TimeSpan frameDuration) {
    Bees.Text = world.Bees.Count.ToString();
    Flowers.Text = world.Flowers.Count.ToString();
    HoneyInHive.Text = String.Format("{0:f3}", world.Hive.Honey);
    double nectar = 0;
    foreach (Flower flower in world.Flowers)
        nectar += flower.Nectar;
    NectarInFlowers.Text = String.Format("{0:f3}", nectar);
    FramesRun.Text = framesRun.ToString();
    double milliSeconds = frameDuration.TotalMilliseconds;
    if (milliSeconds != 0.0)
        FrameRate.Text = string.Format("{0:f0} ({1:f1}ms)",
            1000 / milliSeconds, milliSeconds);
    else
        FrameRate.Text = "N/A";
}
```

Имена методов должны быть теми же, что и в форме.

}

Добавьте этот метод в файл Form1.

А откуда взялся объект World, если мы его еще не создали? И к чему все эти разговоры о времени и кадрах?



### Сотворение Мира

Вы правы, нам нужно создать объект World. Добавьте в конструктор формы следующую строчку:

```
public Form1() {
    InitializeComponent();
    world = new World();
}
```

Добавьте к форме закрытое поле world.

Что же касается кода, связанного со временем... нам же требовался способ многократного вызова метода Go() в классе World... значит, нам требовался таймер.

Это продолжительность одного кадра... через несколько страниц мы получим этот параметр из другого места.

Большая часть этого кода получает данные от объекта world и обновляет метки.

Первый параметр выводится в виде числа без десятичной точки, затем после пробела выводится второй параметр с одним знаком после запятой, за которым следуют буквы ms (в скобках).

Частотой кадров называется количество кадров, показываемых за одну секунду. Информация о продолжительности одного кадра хранится в объекте TimeSpan. Чтобы узнать частоту, мы делим 1000 на количество миллисекунд, которое показывается один кадр.

Мы поговорим об этом подробнее в разделе, посвященном созданию объекта TimeSpan.

Здесь вы видите тот же самый метод String.Format(), который использовали в шестнадцатиричном примере. Но вместо вывода информации в x2, вы указываете f3 и отображаете число с тремя десятичными знаками.



# Таймеры

Такой полезный объект как **таймер** позволяет запускать одно и то же событие снова и снова, тысячу раз в секунду.

Создайте новый проект, чтобы посмотреть, как именно работают таймеры. Затем мы вернемся к симулятору и применим полученные знания на практике.



## 1 Создайте новый проект

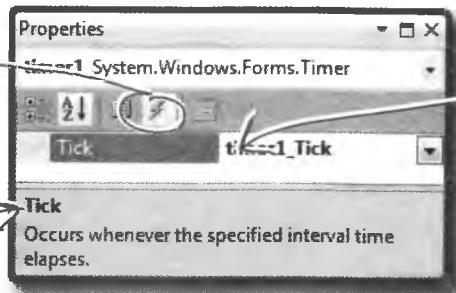
Откройте Visual Studio и создайте проект с формой. Перетащите на нее таймер и три кнопки. Щелкните на таймере и присвойте свойству Interval значение 1000. Измерения ведутся в миллисекундах, соответственно, событие будет запускаться один раз в секунду.

## 2 Щелкните на кнопке Events в окне Properties

(Напоминаем, что это кнопка со значком молнии.) Таймеру соответствует всего одно событие Tick. Щелчком выделите значок Timer в конструкторе, затем дважды щелкните на строке в окне Events. Будет автоматически создан обработчик событий, связанный со свойством.

Обработчик событий можно добавить двойным щелчком на значке Timer.

Кнопка Events в окне Properties дает доступ к событиям, связанным с выделенным элементом управления.



Описание события находится в нижней части окна.

С элементом управления Timer связано событие Tick. Двойной щелчок на этой строчке приводит к созданию обработчика события.

## 3 Код для события Tick и кнопок

Вот код, который позволит понять принцип работы таймера:

```
private void timer1_Tick(object sender, EventArgs e) {
    Console.WriteLine(DateTime.Now.ToString());
```

Эти кнопки позволяют вам под-  
играть со свойством Enabled и  
методами Start() и Stop(). Первая  
меняет значение свойства Enabled  
с true на false и

наоборот, две остальные вызывают методы Start() и Stop().

```
private void toggleEnabled_Click(object sender, EventArgs e) {
```

```
    if (timer1.Enabled)
        timer1.Enabled = false;
    else
        timer1.Enabled = true;
```

Свойство Enabled запускает и останавливает таймер.

```
private void startTimer_Click(object sender, EventArgs e) {
```

```
    timer1.Start();
    Console.WriteLine("Enabled = " + timer1.Enabled);
```

```
private void stopTimer_Click(object sender, EventArgs e) {
```

```
    timer1.Stop();
    Console.WriteLine("Enabled = " + timer1.Enabled);
```

Этот оператор выводит на консоль текущую дату и время. Вы убедитесь, что событие tick запускается 1 раз в секунду (каждые 1000 миллисекунд).

Метод Start() запускает таймер и присваивает свойству Enabled значение true. Метод Stop() останавливает таймер и присваивает свойству Enabled значение false.

## Таймеры используют обработчики событий

Каким образом таймер узнает, что делать в каждое мгновение? Почему метод `timer1_Tick()` запускается при каждом отсчете таймера? Тут мы возвращаемся к **событиям и делегатам**, с которыми вы познакомились в предыдущей главе. Воспользуйтесь функцией Go To Definition, чтобы вспомнить принцип работы делегата `EventHandler`:

За сценой



В предыдущем примере использовался стандартный обработчик событий.

### 4 Щелкните правой кнопкой мыши на переменной `timer1`...

... и выберите в меню команду Go To Definition для перехода к участку кода, в котором задается переменная `timer1`. Найдите строчку:

```
this.timer1.Tick += new System.EventHandler(this.timer1_Tick);
```

↑  
Это событие `Tick` элемента `timer`. Оно появляется каждые 1000 миллисекунд.

↑  
Один из делегатов класса `System`: базовый обработчик событий. Это делегат... а указатель на один или несколько методов.

↑  
Это только что написанный вами метод `timer1_Tick()`. Делегат указывает именно на него.

### 5 Теперь щелкните правой кнопкой мыши на делегате `EventHandler`...

... и снова выберите в меню команду Go To Definition. Обратите внимание на название новой вкладки: `EventHandler [from metadata]`. Оно означает, что код, определяющий `EventHandler`, написан не вами. Это встроенный код .NET Framework, а ИСР генерировала строчку, его представляющую:

```
public delegate void EventHandler(object sender, EventArgs e);
```

↑  
Каждое событие относится к типу `EventHandler`. Наше событие `Tick` сейчас указывает на метод `timer1_Tick()`.

Вот почему все события в C# имеют параметры `Object` и `EventArgs`, именно такую форму имеет делегат, определяющий обработку событий.

## МОЗГОВОЙ ШТУРМ

Какой код нужно написать, чтобы запустить метод `World's Go()` 10 раз в секунду?

## Таймер для симулятора

Добавим таймер в наш симулятор. У вас уже есть соответствующий элемент управления, вероятно, с именем timer1. Вручную свяжем его с методом обработчика событий RunFrame():

Объект TimeSpan обладает такими свойствами как Days (Дни), Hours (Часы), Seconds (Секунды) и Milliseconds (Миллисекунды), что позволяет измерять временные промежутки в различных единицах.

В .NET класс DateTime хранит информацию о времени, а свойство Now возвращает текущую дату и время. Вычислить разницу двух дат позволяет объект TimeSpan: он вычитает один объект DateTime из другого и возвращает объект TimeSpan, содержащий разницу.

```
public partial class Form1 : Form {
    World world; ← у вас уже должно быть свойство World.
    private Random random = new Random();
    private DateTime start = DateTime.Now; ← Здесь будет
    private DateTime end; ← вычисляться время работы
    private int framesRun = 0; ← Мы хотим знать,
                               сколько кадров
                               уже показано.

    public Form1() {
        InitializeComponent();
        world = new World();

        timer1.Interval = 50; ← Запуск каждые 50 миллисекунд.
        timer1.Tick += new EventHandler(RunFrame); ← Мы связали обработчик
        timer1.Enabled = false; ← с собственным
        UpdateStats(new TimeSpan()); ← Запуск таймера. методом RunFrame().
    }
}

private void UpdateStats(TimeSpan frameDuration) {
    // Предыдущий код для обновления статистики
}
```

Запуск каждые 50 миллисекунд.

Мы связали обработчик с собственным методом RunFrame().

В одной секунде 1000 миллисекунд, так что таймер будет срабатывать 20 раз в секунду.

```
public void RunFrame(object sender, EventArgs e) {
    framesRun++;
    world.Go(random); ← Увеличьте количество кадров
                       на 1 и запустите метод Go()
    end = DateTime.Now;
    TimeSpan frameDuration = end - start; ← Затем мы узнаем,
    start = end; ← сколько времени прошло
    UpdateStats(frameDuration); ← с момента проигрывания
                                последнего кадра.

}
```

Обновите статистику с новой продолжительностью времени.



## пражнение

Если вы еще не перетащили элементы ToolStrip и StatusStrip из окна toolbox на форму, сделайте это сейчас.

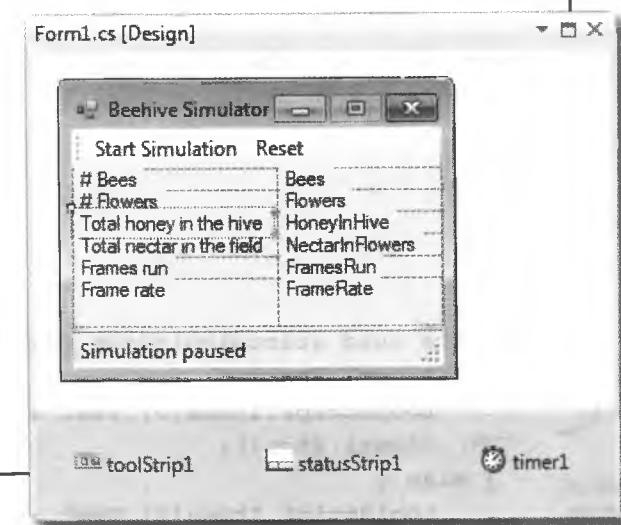
Напишите обработчики событий для кнопок **Start Simulation** и **Reset** элемента ToolStrip. Вот что делают кнопки:

- Изначально на первой кнопке должна быть надпись «Start Simulation». Щелчок на ней запускает симулятор, а надпись меняется на «Pause Simulation». При постановке симулятора на паузу надпись меняется на «Resume simulation».
- На второй кнопке должна быть надпись «Reset». Щелчок на ней приводит к перезагрузке мира. При поставленном на паузу таймере текст на первой кнопке должен меняться с «Resume simulation» на «Start Simulation».

На этом вопрос нет однозначного ответа. Мы просто хотим, чтобы вы подумали над дальнейшим построением симулятора.



Дважды щелкните на кнопке ToolStrip в конструкторе, чтобы добавить к ней обработчик событий, как к обычной кнопке.



## Возьми в руку карандаш

Как вы думаете, что осталось несделанным на этом этапе работы над симулятором? Запустите программу. Запишите, какие изменения нужно добавить, перед тем как переходить к работе с графикой.

.....

.....

.....

.....

.....

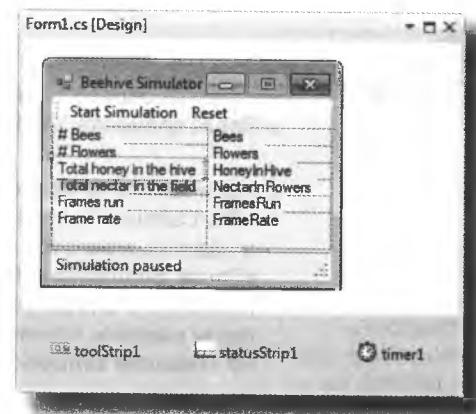
часть  
Задаваемые  
Вопросы

**В:** В чем отличие цикла от кадра?

**О:** Отличие чисто семантическое. Наша система работает циклически. Но как только мы вставим в проект графические фрагменты, разговор пойдет исключительно о кадрах.



Вот как выглядит код обработчиков событий для кнопок Start Simulation и Reset.



Код формы остается без изменений.

```

private void startSimulation_Click(object sender, EventArgs e) {
    if (timer1.Enabled) {
        toolStrip1.Items[0].Text = "Resume simulation";
        timer1.Stop();
    } else {
        toolStrip1.Items[0].Text = "Pause simulation";
        timer1.Start();
    }
}

private void reset_Click(object sender, EventArgs e) {
    framesRun = 0;
    world = new World();
    if (!timer1.Enabled)
        toolStrip1.Items[0].Text = "Start simulation";
}

```

Убедитесь, что имена элементов управления формы со-впадают с теми, ко-торые вы используете в своем коде.

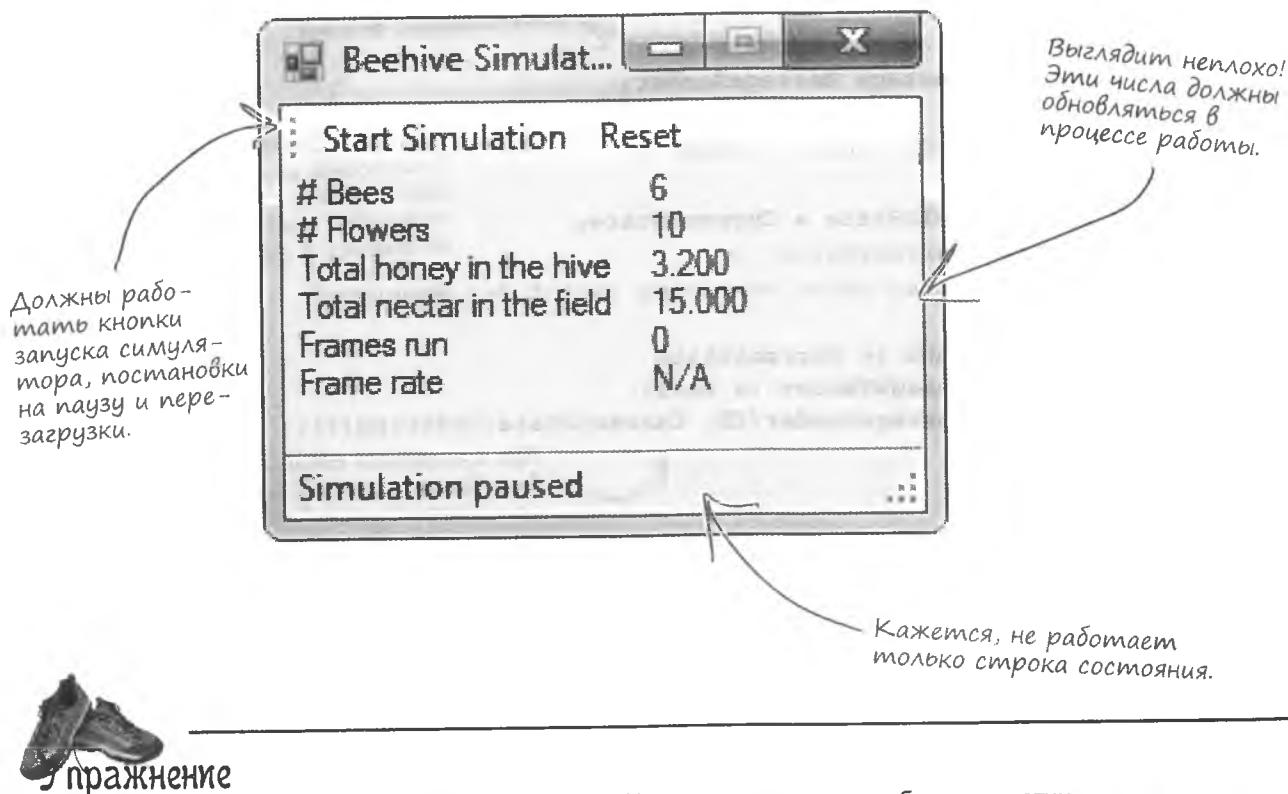
Переключи-  
те таймер  
и обновите  
сообщение.

Перезагруз-  
ка симулятора  
осуществляется  
повторным созда-  
нием экземпляра  
World и переза-  
грузкой метода  
framesRun.

Мемку на первой кнопке следует менять только в случае, когда на ней написано «Resume simulation». В случае надписи «Pause simulation» изменения вносить не нужно.

## Тестирование

Проделана большая работа. Скомпилируйте код, исправьте опечатки и запустите симулятор. Что у вас получилось?



## Упражнение

Вот шанс проявить свои знания. Нужно сделать так, чтобы пчелы отчитывались симулятору, и эта информация появлялась в строке состояния.

На этот раз вам предстоит не только написать большую часть кода, но и самостоятельно решить, что именно писать. Вам нужен метод, который будет вызываться при каждом изменении состояния пчелы.

Чтобы немного вам помочь, мы написали метод, который следует добавить к форме. Класс Bee должен вызывать его при изменении состояния пчел:

```
private void SendMessage(int ID, string Message) {
    statusStrip1.Items[0].Text = "Bee #" + ID + ":" + Message;
}
```

\*Еще одна подсказка:  
Редактировать придется все классы, кроме одного.

## упражнение Решение

Это было добавлено в класс Bee.

```

class Bee {
    // весь уже существующий код
    public BeeMessage MessageSender;

    public void Go(Random random) {
        Age++;
        BeeState oldState = CurrentState;
        switch (currentState) {
            // остальная часть оператора switch без изменений
        }
        if (oldState != CurrentState
            && MessageSender != null)
            MessageSender(ID, CurrentState.ToString());
    }
}

```

Мы воспользовались обратным вызовом для привязки пчел к методу SendMessage() формы.

Делегат BeeMessage берет в качестве параметров номер пчелы и сообщение. С его помощью пчела отправляет сообщение в форму.

При изменении статуса пчелы вызывается метод BeeMessage, на который указывает делегат.

Изменения, внесенные в класс Hive.

```

class Hive {
    // весь уже существующий код
    public BeeMessage MessageSender;

    public Hive(World world, BeeMessage MessageSender) {
        this.MessageSender = MessageSender;
        // существующий код конструктора
    }
}

```

Объекту Hive также требуется делегат, чтобы передать каждой пчеле метод, который будет вызываться после создания этой пчелы методом AddBee().

```

public void AddBee(Random random) {
    // код метода AddBee()
    Bee newBee = new Bee(beeCount, startPoint, world, this);
    newBee.MessageSender += this.MessageSender;
    world.Bees.Add(newBee);
}

```

AddBee() гарантирует, что каждая новая пчела получает метод, на который она будет указывать.

```
public delegate void BeeMessage(int ID, string Message);
```

В класс World также требовалось внести изменения.

```
class World {
    // весь существующий код

    public World(BeeMessage messageSender) {
        Bees = new List<Bee>();
        Flowers = new List<Flower>();
        Hive = new Hive(this, messageSender);
        Random random = new Random();
        for (int i = 0; i < 10; i++)
            AddFlower(random);
    }
}
```

BeeMessage — это наш делегат. Он совпадает с написанным нами для формы методом SendMessage().

Добавьте его в файл BeeMessage.cs, он должен находиться в пространстве имен, но вне любого класса.

Классу World делегаты не передаются. Он просто передает метод для вызова экземпляра Hive.

```
public partial class Form1 : Form {
    // объявление переменных

    public Form1() {
        InitializeComponent();
        world = new World(new BeeMessage(SendMessage));
        // остальная часть конструктора формы
    }
}
```

Новый делегат создается из класса Bee (убедитесь, что метод BeeMessage является открытым) и указывает на метод SendMessage().

```
private void reset_Click(object sender, EventArgs e) {
    framesRun = 0;
    world = new World(new BeeMessage(SendMessage));
    if (!timer1.Enabled)
        toolStrip1.Items[0].Text = "Start simulation";
}
```

Здесь то же самое... создается мир с методом для обратного вызова пчел.

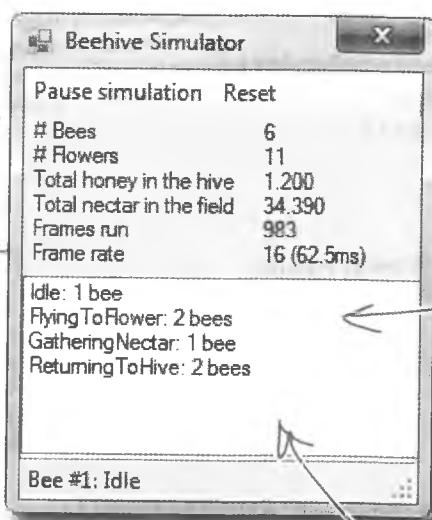
```
private void SendMessage(int ID, string Message) {
    statusStrip1.Items[0].Text = "Bee #" + ID + ":" + Message;
}
```

Это метод, который мы вам дали... не забудьте вписать его.

## Работа с группами пчел

Пчелы жужжат вокруг улья и на поле, симулятор работает! Здорово? Но пока визуальная часть симулятора не работает, этим мы займемся в следующей главе. Информацию мы можем получать только посредством сообщений, которые пчелы отправляют в форму при помощи обратного вызова. Добавим дополнительную информацию об их деятельности.

Форма обновляет статистические данные и отображает сообщения, которые пчелы посыпают как отчет о проделанной работе.



Добавим элемент ListBox для отображения дополнительной информации о пчелах.

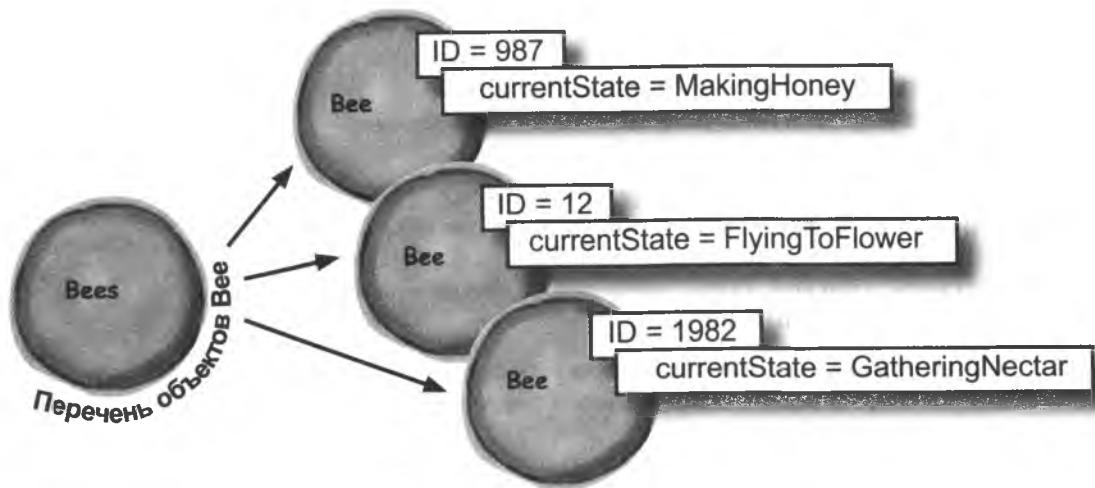
В каждый момент времени вокруг летает множество пчел. Новый ListBox будет показывать, сколько пчел выполняют определенное задание. В данном случае две пчелы летят к цветам, одна собирает нектар, две возвращаются в улей и одна ничего не делает.

### МОЗГОВОЙ ШТУРМ

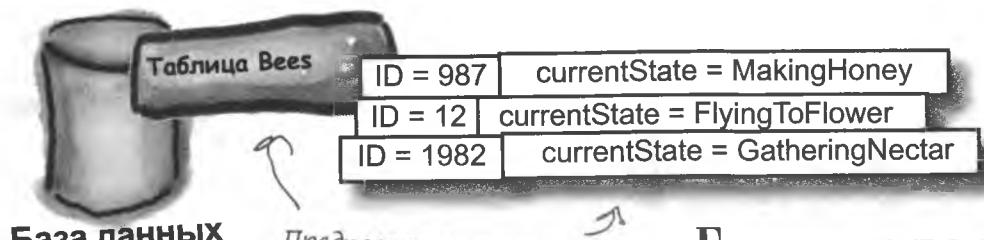
Вы знаете достаточно для вставки в форму элемента ListBox. Подумайте, как именно он функционирует. Это сложнее, чем кажется на первый взгляд. Что нужно сделать, чтобы определить количество пчел в каждом из состояний Bee.State?

## Коллекция коллекционирует... ДАННЫЕ

Пчелы сохранены в коллекции `List<Bee>`. Коллекции предназначены для хранения данных... практически, как базы данных. Каждую пчелу можно представить в виде строки, в которой указано состояние, идентификатор и т. п. Вот как выглядят пчелы в виде коллекции:



Поля объекта `Bee` содержат множество данных. Коллекцию объектов можно представить в виде строк базы данных. Каждый объект содержит данные в полях, так же как каждая строка базы содержит данные в столбцах.



**Большинство коллекций, особенно содержащих объекты, можно представить как своего рода базу данных.**



Какая разница, в каком виде мы можем представить коллекцию, если мы не можем использовать ее как базу данных?

Представьте запросы к коллекциям, базам данных и даже к XML-документам с одинаковым синтаксисом!

В C# имеется полезная функция LINQ (**L**anguage **I**ntegrated **Q**uery – встроенный язык запросов). Эта функция дает вам возможность работать с данными из массивов, списков, стека, очередей и других коллекций при помощи единых операций.

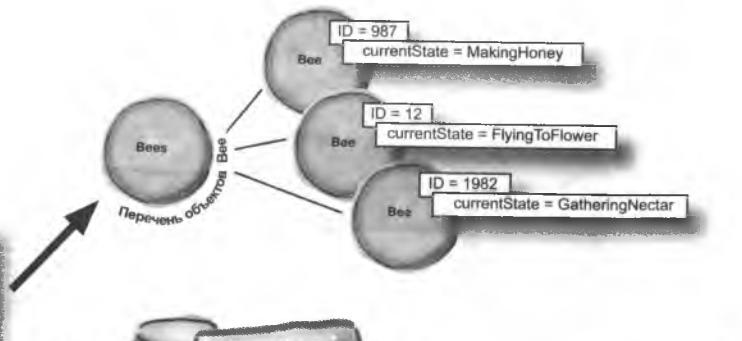
LINQ позволяет использовать при работе с коллекциями тот же синтаксис, что и при работе с базами данных.

Большая часть главы 15 посвящена именно LINQ.

Запросы LINQ одинаково работают с коллекциями и с базами данных.

```
var beeGroups =  
    from bee in world.Bees  
    group bee by bee.CurrentState  
    into beeGroup  
    orderby beeGroup.Key  
    select beeGroup;
```

**LINQ**



Если бы данные о пчелах находились в базе (или в файле XML)? LINQ работал бы с ними тем же способом.

```
<bee id="987" currentState="MakingHoney" />  
<bee id="12" currentState="FlyingToFlower" />  
<bee id="1982" currentState="GatheringNectar" />
```

## LINQ упрощает работу с коллекциями и базами данных

LINQ будет посвящена целая глава, а пока вы можете воспользоваться его возможностями и добавить в симулятор дополнительные функции. Введите этот код. Ничего страшного, если вы не понимаете его. О том, как он работает, мы поговорим в главе 15.



Готово  
к употреблению

```

private void SendMessage(int ID, string Message) {
    statusStrip1.Items[0].Text = "Bee #" + ID + ":" + Message;
    var beeGroups =
        from bee in world.Bees
        group bee by bee.CurrentState into beeGroup
        orderby beeGroup.Key
        select beeGroup;
    listBox1.Items.Clear();
}

Убедитесь, что это совпадает с именем элемента listbox вашей формы
foreach (var group in beeGroups) {
    string s;
    if (group.Count() == 1)
        s = "";
    else
        s = "s";
    listBox1.Items.Add(group.Key.ToString() + ":" +
        + group.Count() + " bee" + s);
    if (group.Key == BeeState.Idle
        && group.Count() == world.Bees.Count()
        && framesRun > 0)
        listBox1.Items.Add("Simulation ended: all bees are idle");
    toolStrip1.Items[0].Text = "Simulation ended";
    statusStrip1.Items[0].Text = "Simulation ended";
    timer1.Enabled = false;
}
}

```

Это запрос LINQ, группирующий всех пчел коллекции по свойству CurrentState.

Ключом является свойство пчелы CurrentState, так что состояния будут отображаться в форме именно в таком порядке.

Переменная beeGroups появилась из запроса LINQ. Мы можем посчитать членов этой группы и по очереди просмотреть их.

Этот код обеспечивает правильное употребление английского слова «пчела» во множественном числе.

Добавим в текстовое поле статус группы (ее ключ) и счетчик.

Так как количество незанятых пчел известно...

...можно проверить, не являются ли все пчелы незанятыми. Из положительного результата проверки следует отсутствие в улье Меда, а значит, завершение работы симулятора.



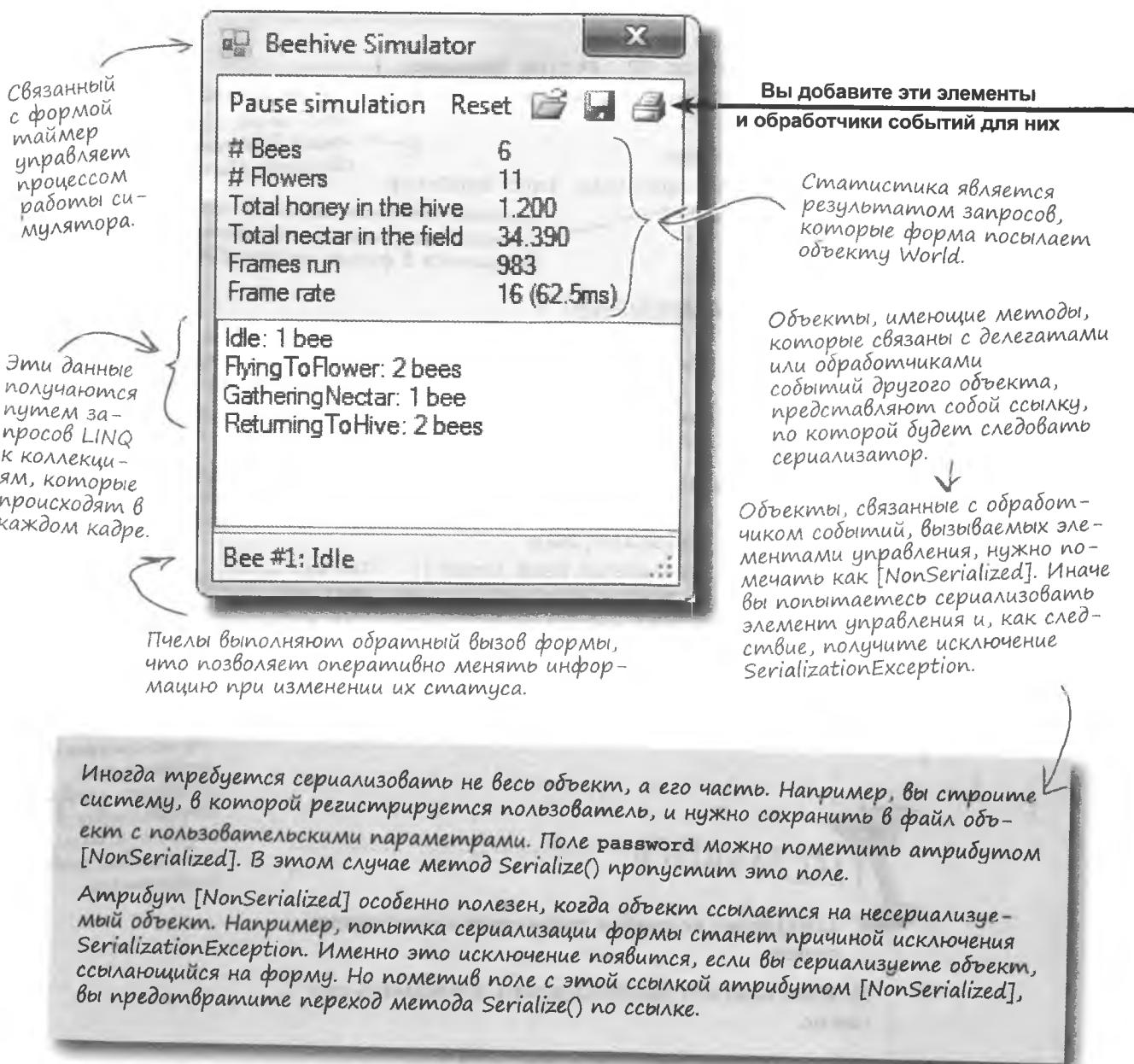
РАССЛАБЬТЕСЬ

LINQ будет подробно рассмотрен в следующих главах.

Поэтому пока не старайтесь понять и запомнить синтаксис.

## Тестирование (вторая попытка)

Скомпилируйте код и запустите проект. В случае ошибок проверьте синтаксис, особенно новый код с LINQ. Посмотрим, как работает наш симулятор!





Вот код, заставляющий работать кнопки Save и Open.

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

Не забудьте про дополнительные операторы using.

Классы World, Hive, Flower и Bee нужно сделать сериализуемыми. В процессе сериализации объекта World .NET обнаружит ссылки на объекты Hive, Flower и Bee и сериализует их.

```
[Serializable]
class World { }
```

```
[Serializable]
class Hive { }
```

```
[Serializable]
class Flower { }
```

```
[Serializable]
class Bee { }
```

```
[NonSerialized]
public BeeMessage MessageSender;
```

Убедимся, что поле MessageSender в классах Hive и Bee помечено атрибутом [NonSerialized].

Код для кнопки Save.

```
private void saveToolStripButton_Click(object sender, EventArgs e) {
    bool enabled = timer1.Enabled;
    if (enabled)
        timer1.Stop();
```

```
SaveFileDialog saveDialog = new SaveFileDialog();
saveDialog.Filter = "Simulator File (*.bees)|*.bees";
saveDialog.CheckPathExists = true;
saveDialog.Title = "Choose a file to save the current simulation";
if (saveDialog.ShowDialog() == DialogResult.OK)
```

```
try {
    BinaryFormatter bf = new BinaryFormatter();
    using (Stream output = File.OpenWrite(saveDialog.FileName)) {
        bf.Serialize(output, world);
        bf.Serialize(output, framesRun);
    }
}
```

```
catch (Exception ex) {
    MessageBox.Show("Unable to save the simulator file\r\n" + ex.Message,
        "Bee Simulator Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

```
}
```

После сохранения файла можно перезапустить таймер (если он был остановлен).

Для файлов, в которые записывается информация из симулятора, мы используем расширение .bees.

Помните, что вместе с объектом World сериализации подвергаются и все объекты, на которые он ссылается... пчелы, цветы и улей.

Именно здесь мир сохраняется в файл.

## Последние штрихи: Open и Save

Все уже практически готово для перехода к следующему этапу — добавлению в наш симулятор графики. Осталось предусмотреть возможность загружать данные, сохранять их и выводить статистику.

О кнопке **Print**  
разговор пойдет  
в следующей главе.

### 1 Значки Open, Save и Print

Элемент ToolStrip умеет автоматически вставлять стандартные значки: new, open, save, print, cut, copy, paste и help. Щелкните на элементе ToolStrip правой кнопкой мыши и выберите команду **Insert Standard Items**. В верхней части формы появится панель с кнопками. Щелкните на первой из них — это кнопка new — и удалите ее. Три следующие кнопки (open, save и print) нам нужны. Разделитель и остальные кнопки также требуется удалить. Свойство **CanOverflow** элемента ToolStrip должно иметь значение false (чтобы кнопки с правой стороны панели инструментов не добавлялись в меню переполнения), а свойство **GripStyle** — значение Hidden (чтобы скрыть расположенный слева дескриптор перемещения).

### 2 Обработчики событий для кнопок

Новые кнопки называются `openToolStripButton`, `saveToolStripButton` и `printToolStripButton`. Двойным щелчком добавьте к ним обработчики событий.



Добавьте код, заставляющий работать кнопки Open и Save.

#### Упражнение

1. Кнопка **Save** должна сериализовать объект **world** в файл. Она должна останавливать таймер (если симулятор все еще работает, таймер может перезапуститься после сохранения) и вызывать окно диалога Save. Сериализации должен подвергаться не только объект **World**, но и количество кадров, прожитых симулятором.

Сначала вы получите исключение `SerializationException` с текстом Type 'Form1' is not marked as `Serializable`. Ведь делегат `BeeMessage` связан с формой, значит, ее пытаются сериализовать.

Добавьте полям `MessageSender` в классах `Hive` и `Bee` атрибут `[NonSerialized]`.

2. Кнопка **Open** десериализует мир из файла. Таймер ведет себя так же, как при щелчке на кнопке **Save**. Должно открываться окно диалога **Open** и происходить десериализация указанного пользователем файла. После чего можно снова связать делегат `MessageSender` с формой и при необходимости перезапустить таймер.

3. Не забудьте про обработку исключений! Убедитесь, что проблемы с чтением и записью в файл не влияют на состояние мира. Создайте всплывающие окна с сообщениями об ошибке.

Код для кнопки Open.

```

private void openToolStripButton_Click(object sender, EventArgs e) {
    World currentWorld = world; ← Перед тем как открыть и прочитать
    int currentFramesRun = framesRun; ← файл, сделаем копию текущего состояния
                                         мира и параметра framesRun. В случае
                                         проблем мир будет восстановлен из этой
                                         копии.

    bool enabled = timer1.Enabled;
    if (enabled)
        timer1.Stop();

    OpenFileDialog openFileDialog = new OpenFileDialog();
    openFileDialog.Filter = "Simulator File (*.bees)|*.bees";
    openFileDialog.CheckPathExists = true;
    openFileDialog.CheckFileExists = true;
    openFileDialog.Title = "Choose a file with a simulation to load";
    if (openDialog.ShowDialog() == DialogResult.OK) {
        try {
            BinaryFormatter bf = new BinaryFormatter();
            using (Stream input = File.OpenRead(openDialog.FileName)) {
                world = (World)bf.Deserialize(input); ← Здесь происходит
                framesRun = (int)bf.Deserialize(input); ← десериализация мира
                                                 и количества кадров.

Оператор ← При появлении исключения мы
using за- → восстанавливаем последнюю
рантируем ← сохраненную версию мира
закрытие но- → и параметра framesRun.
тока.      }
        }
        catch (Exception ex) {
            MessageBox.Show("Unable to read the simulator file\r\n" + ex.Message,
                "Bee Simulator Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
            world = currentWorld; ←
            framesRun = currentFramesRun; ←
        }
    }

    world.Hive.MessageSender = new BeeMessage(SendMessage);
    foreach (Bee bee in world.Bees)
        bee.MessageSender = new BeeMessage(SendMessage);
    if (enabled)
        timer1.Start();
}

```

Настстройки и вызов окна диалога Open File.

Здесь происходит десериализация мира и количества кадров.

После загрузки мы подсоединяем делегат и перезапускаем таймер.

**Рабочую версию этого файла можно получить на сайте Head First Labs: [www.headfirstlabs.com / books / hfsharp /](http://www.headfirstlabs.com/books/hfsharp/)**

## 13 Элементы управления и графические фрагменты

# Наводим красоту



**Иногда управление графикой приходится брать в свои руки.**

До этого момента визуальный аспект наших приложений был отдан на откуп элементам управления. Но иногда этого недостаточно, например, если вы хотите **анимировать изображение**. Вам предстоит научиться создавать **свои элементы управления** для .NET, применять **двойную буферизацию** и даже рисовать непосредственно на формах.

## Элементы управления как средство взаимодействия с программой

Элементы TextBox, PictureBox, Label... вы уже давно имеете представление о работе с ними. Но что *именно* вы о них знаете? (Кроме того, что их можно перетаскивать из окна ToolBox на форму).



### Создание пользовательских элементов управления

Элементы ToolBox бесценны при построении форм и приложений, но ничего таинственного в них нет. Это всего лишь классы, а любой класс вы можете написать самостоятельно. В C# создать пользовательский элемент управления крайне легко. Достаточно нужного базового класса и функции наследования.



### Ваши элементы управления в окне ToolBox

Ничего таинственного нет и в окне ToolBox. Оно просматривает классы проекта и встроенные классы .NET в поисках элементов управления. При обнаружении класса, реализующего нужный интерфейс, отображается значок. Ваши элементы также отобразятся в этом окне.



Можно создать класс, наследующий от любого из существующих классов элементов управления, и новый элемент автоматически появится в окне toolbox.



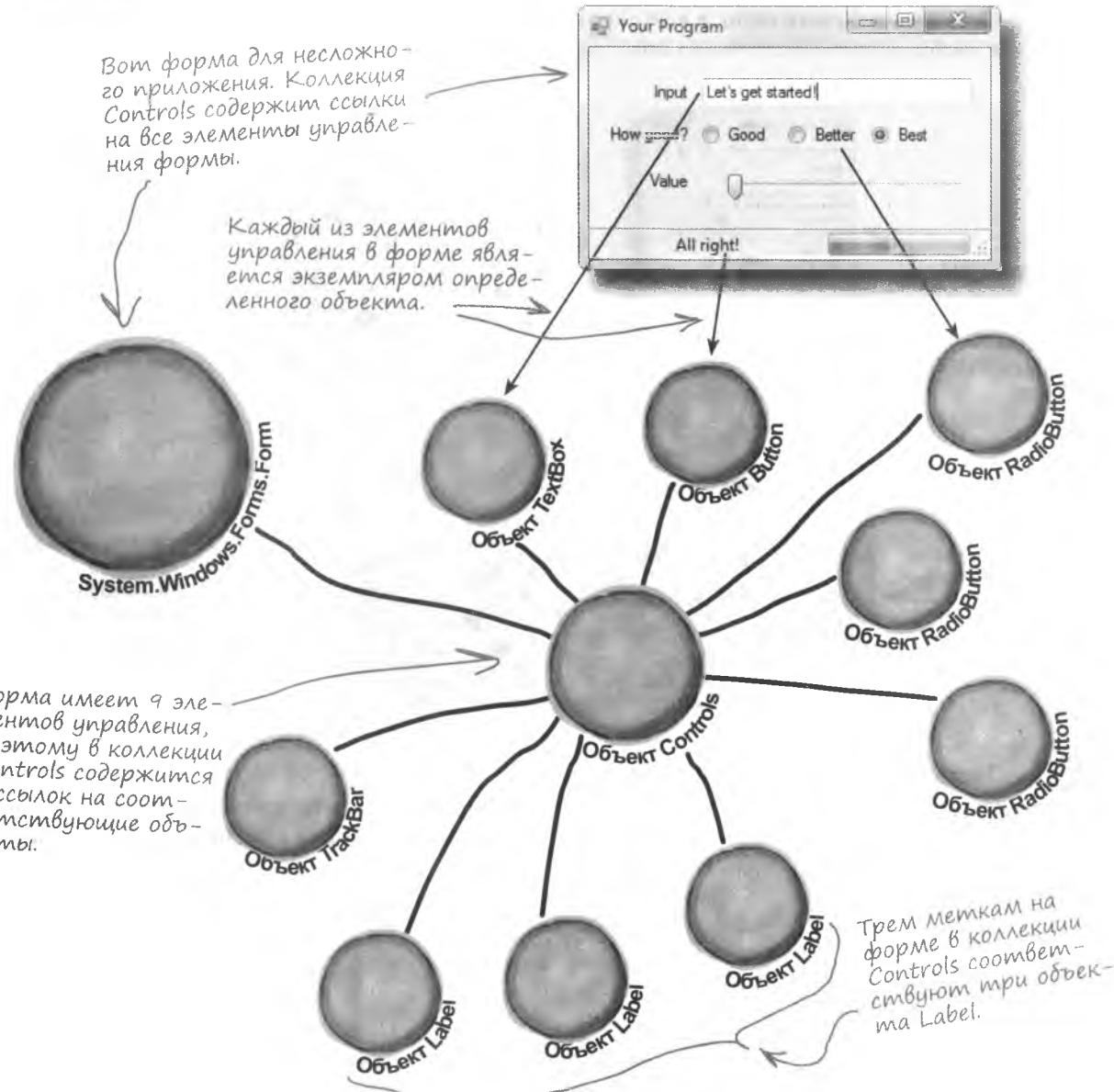
### Код, добавляющий к форме элементы управления и даже удаляющий их в процессе работы программы

Форме в конструкторе можно придать и другой вид. Вы уже работали с элементами PictureBox. Их можно также добавлять и удалять. В конце концов, при построении вами формы, ИСР добавляет к ней элементы управления... это означает, что вы можете написать аналогичный код и использовать его по мере необходимости.

## Элементы управления — это тоже объекты

Вы уже осведомлены о важности элементов управления. Вы пользовались кнопками, текстовыми полями, флажками, метками и другими элементами начиная с главы 1. И вот пришла пора узнать, что это такие же объекты, как и все остальные.

Элементы управления просто умеют рисовать сами себя. Объект `Form` следит за их состоянием при помощи специальной коллекции `Controls`. Эта коллекция дает вам возможность добавлять элементы управления в ваш код и удалять их.



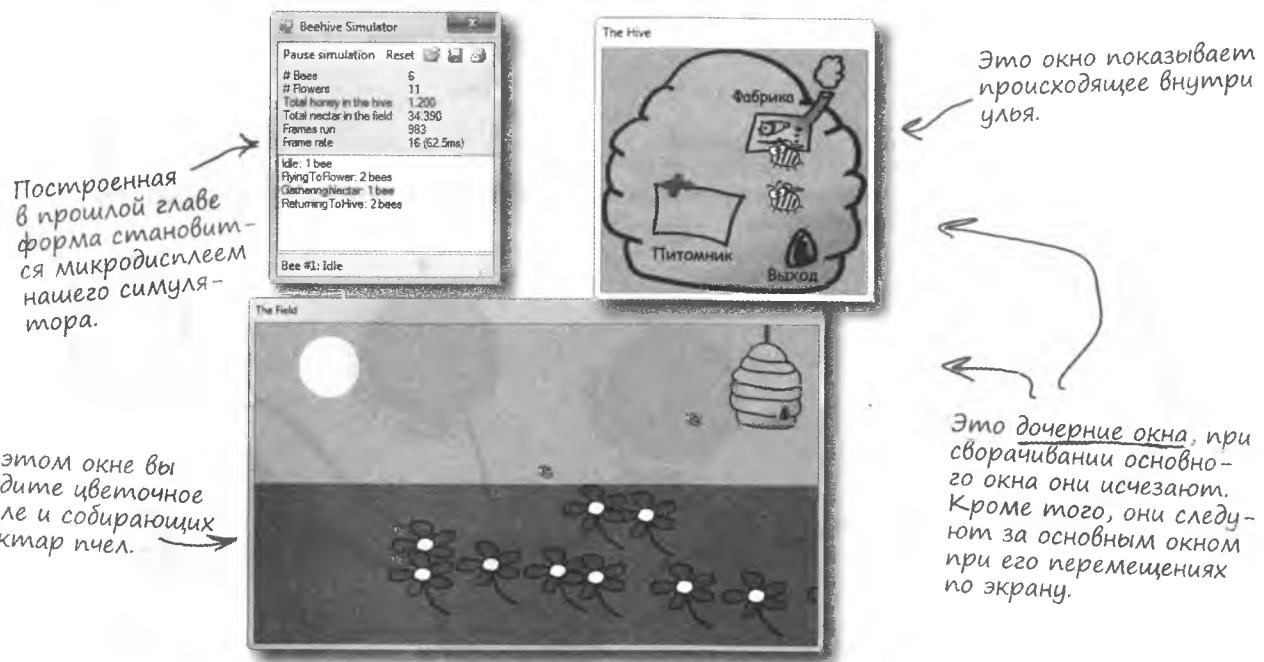
## Анимируем симулятор улья

Наш замечательный симулятор не радует глаз. Пришло время создать визуализацию, демонстрирующую пчел в действии. Анимируем улей при помощи элементов управления.

1

### Происходящее показывает пользовательский интерфейс

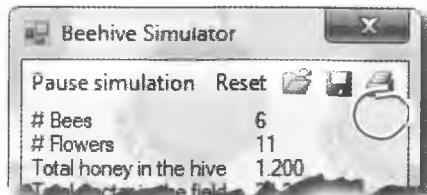
Симулятор будет состоять из трех окон. Основное, отображающее статистику, уже готово. Осталось добавить окно, в котором можно наблюдать за происходящим внутри улья, и окно с изображением цветочного поля, на котором пчелы собирают нектар.



2

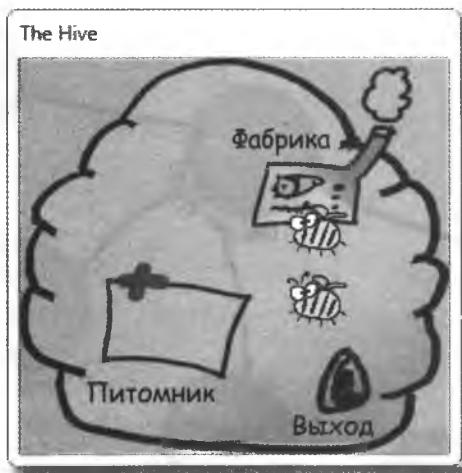
### Заставим работать кнопку Print

Окно статистики имеет работающие кнопки Open и Save, в то время как кнопка Print пока не функционирует. Сделаем так, чтобы щелчок на ней приводил к распечатке действий симулятора.



**3****Окно Hive**

Пчелы летают по всему пространству симулятора. Когда они залетают в улей, происходящее отображается в этом окне.



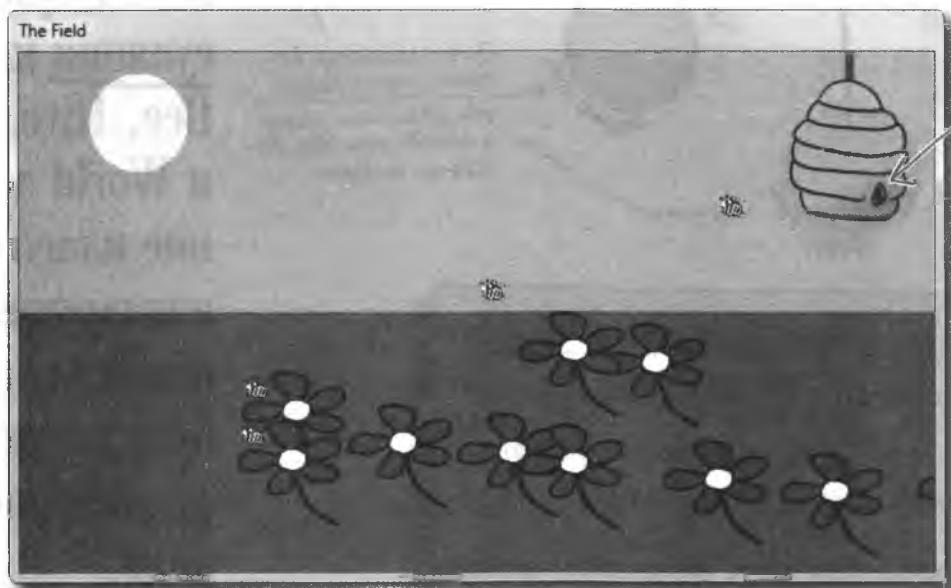
В улье выделены три точки. Пчелы появляются в питомнике, пользуются выходом, когда приходит время лететь на цветочное поле, затем они возвращаются и отправляются на фабрику по переработке нектара.

Выход из улья находится в форме *hive*, а вход — в форме *field*. (Поэтому оба значения были помещены в словарь.)

Это вход в улей. Достигнув этой точки, пчелы исчезают с поля и появляются у выхода в окне *hive*.

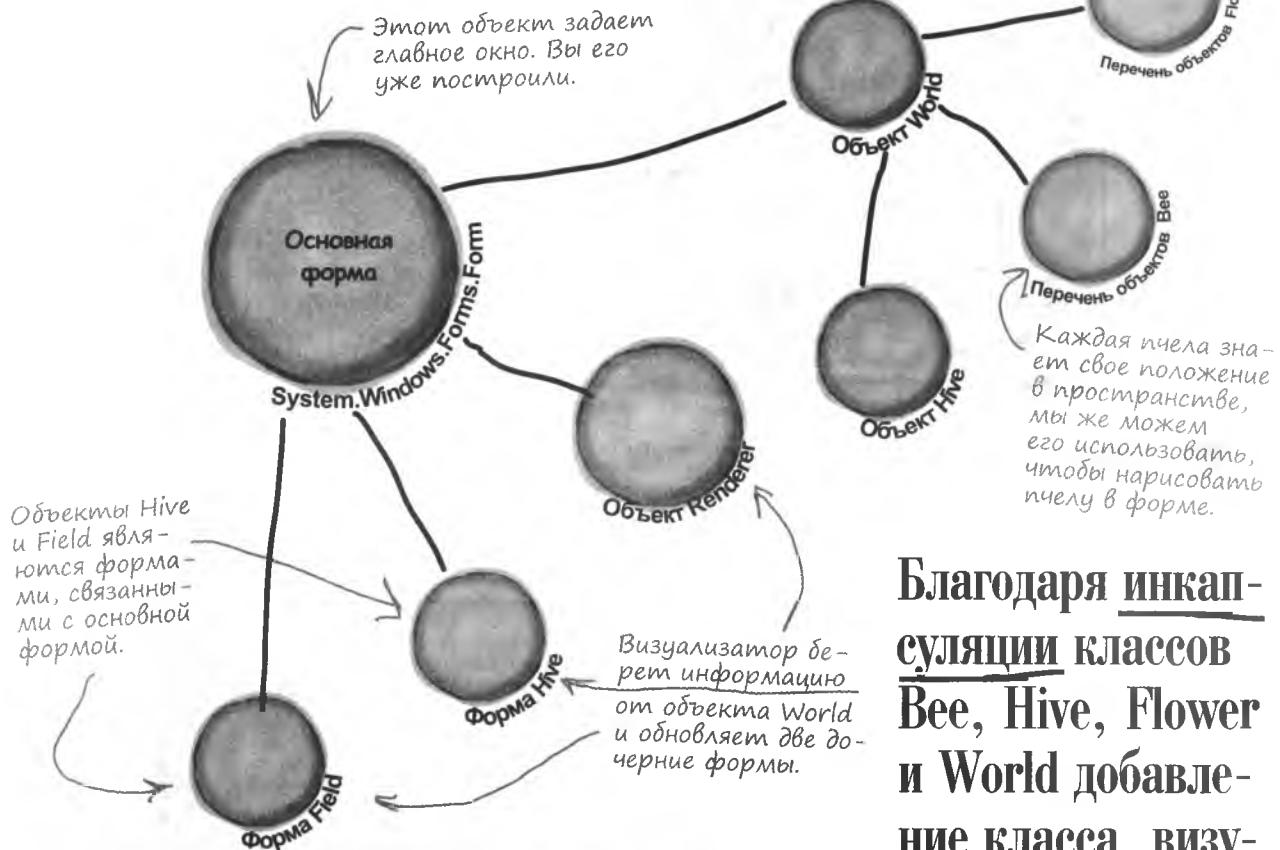
**4****Сбор нектара происходит в окне Field**

Основным занятием пчел является сбор нектара, который будет переработан в мед. Поедание этого меда дает пчелам энергию для сбора нового нектара.



## Визуализатор

Нам требуется класс, который на основе информации из нашего мира будет рисовать в двух новых формах улей пчел и цветы. С этой задачей отлично справится класс `Renderer`. Благодаря инкапсуляции уже имеющихся классов нам не потребуется сильно менять код.



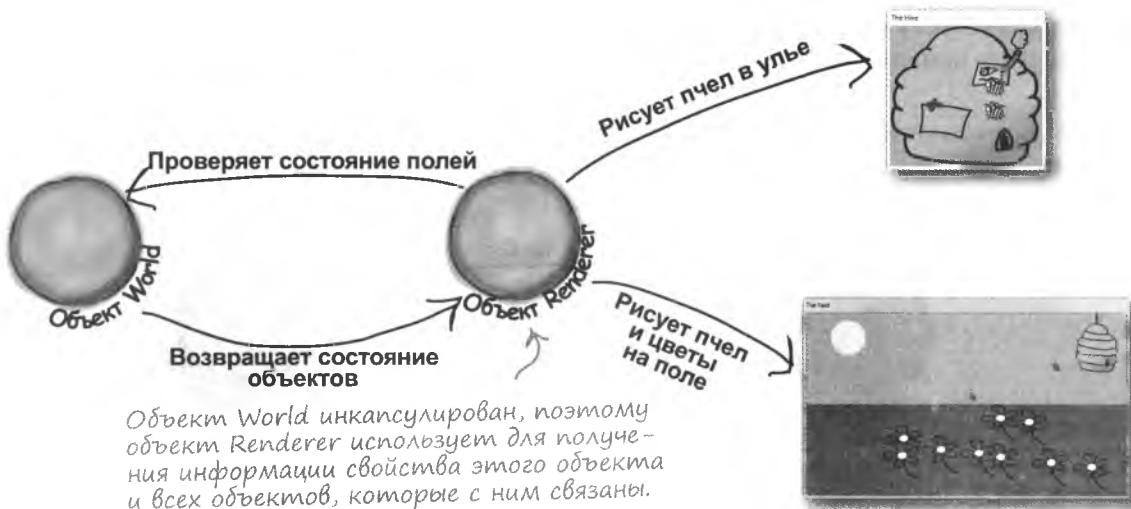
**ви-зу-а-ли-зи-ро-вать**, глаг.  
представлять или изображать.

Учитель попросил **визуализировать**  
линии модели на листе бумаги.

Благодаря инкапсуляции классов `Bee`, `Hive`, `Flower` и `World` добавление класса, визуализирующего эти объекты, не требует значительного редактирования кода.

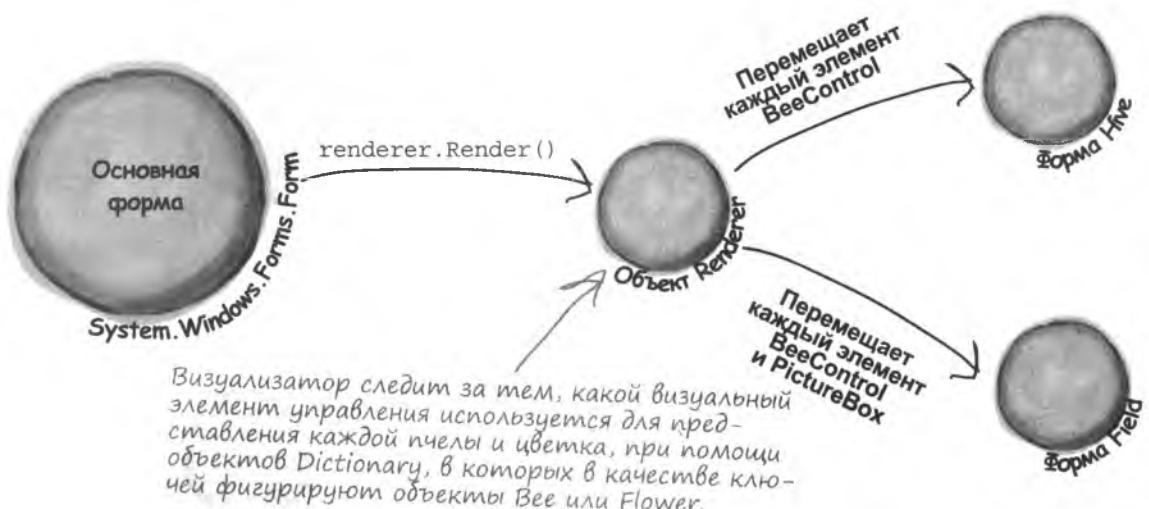
## Формы для визуализации

Объект World отслеживает все происходящее в симуляторе, но не в состоянии наглядно показать результат. Эту работу выполняет объект Renderer. Он считывает информацию с объектов World, Hive, Bee и Flower и представляет ее в графическом виде.



### Визуализируется каждый кагр

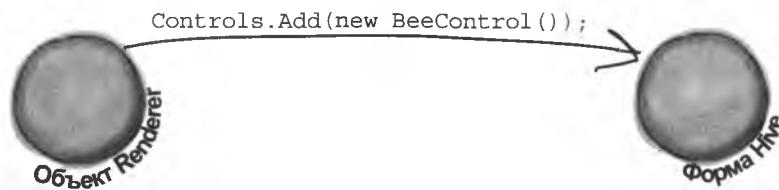
Вызванный основной формой метод Go() объекта World должен вызывать метод Render() объекта Renderer для обновления графической информации. Цветы отображаются при помощи элемента PictureBox. Для пчел мы создадим анимированный элемент управления BeeControl.



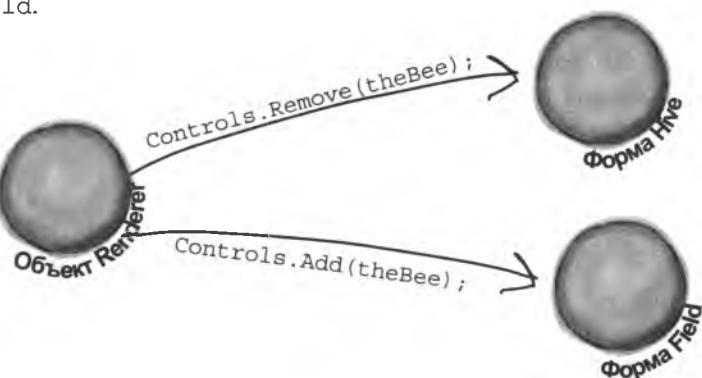
## Элементы управления прекрасно подходят для визуализации

Создание новой пчелы должно сопровождаться появлением нового элемента BeeControl в форме объекта Hive. Этот элемент будет менять положение при перемещениях пчелы. Вылет из улья означает удаление элемента из формы Hive и добавление его в форму Field. Возвращение в улей сопровождается обратным процессом. И все это время крылья пчелы должны двигаться. Эту задачу легко решить при помощи элементов управления.

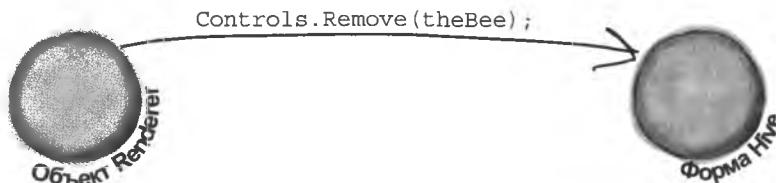
- 1 При появлении новой пчелы создается BeeControl, который добавляется в коллекцию Controls формы Hive.



- 2 При перемещении пчелы из улья на поле элемент удаляется из коллекции Controls улья и добавляется в одноименную коллекцию формы Field.



- 3 Состарившись, пчелы выходят на пенсию. Если при проверке объектом Renderer списком Bees обнаруживается отсутствие пчелы, он удаляет ее элемент управления из формы Hive.



# Возьми в руку карандаш



Определите назначение каждого из фрагментов кода. Все эти фрагменты принадлежат форме.

```
this.Controls.Add(new Button()); .....  
.....  
.....
```

```
Form2 childWindow = new Form2();  
childWindow.BackgroundImage =  
    Properties.Resources.Mosaic;  
childWindow.BackgroundImageLayout =  
    ImageLayout.Tile;  
childWindow.Show();  
.....
```

Если форма снабжена элементом `ListBox`,  
для добавления элементов в список можно  
использовать метод `AddRange()`.

```
Label myLabel = new Label(); ↓  
myLabel.Text = "Твое любимое животное";  
myLabel.Location = new Point(10, 10);  
ListBox myList = new ListBox();  
myList.Items.AddRange( new object[]  
    { "Кот", "Пес", "Рыбка", "Нет" } );  
myList.Location = new Point(10, 40);  
Controls.Add(myLabel);  
Controls.Add(myList);  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....
```

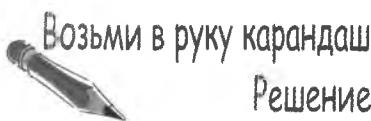
↑  
Объясняю назначение каждой строки не требуется, достаточно записать, какую функцию выполняет фрагмент.

```
Label controlToRemove = null;  
foreach (Control control in Controls) {  
    if (control is Label  
        && control.Text == "Bobby")  
        controlToRemove = control as Label;  
}  
Controls.Remove(controlToRemove);  
controlToRemove.Dispose();  
.....  
.....  
.....  
.....  
.....
```

Что получится, если так сделать, вы можете посмотреть сами. Попробуйте понять, почему получается именно такой результат.

**Дополнительный вопрос: Как вы думаете, почему оператор `Controls.Remove()` не был помещен в цикл `foreach`?**

.....  
.....  
.....



## Возьми в руку карандаш

### Решение

Вот какую функцию выполняет каждый из фрагментов кода:

```
this.Controls.Add(new Button());
```

Создает кнопку и добавляет ее к форме. Имеет значения по умолчанию (например, свойство Text будет пустым).

```
Form2 childWindow = new Form2();
childWindow.BackgroundImage =
    Properties.Resources.Mosaic;
childWindow.BackgroundImageLayout =
    ImageLayout.Tile;
childWindow.Show();
```

Создается вторая форма Form2, загружается фоновое изображение из файла Mosaic, которое заполняет всю форму. Затем полученное окно демонстрируется пользователю.

```
Label myLabel = new Label();
myLabel.Text = "Твое любимое животное";
myLabel.Location = new Point(10, 10);
ListBox myList = new ListBox();
myList.Items.AddRange( new object[]
    { "Кот", "Пес", "Рыбка", "Нет" } );
myList.Location = new Point(10, 40);
Controls.Add(myLabel);
Controls.Add(myList);
```

Создается метка, задается ее текст и положение. Затем создается текстовое поле, в него добавляются четыре элемента, и поле помещается под меткой. Метка и текстовое поле добавляются к форме и отображаются.

```
Label controlToRemove = null;
foreach (Control control in Controls) {
    if (control is Label
        && control.Text == "Bobby")
        controlToRemove = control as Label;
}
Controls.Remove(controlToRemove);
controlToRemove.Dispose();
```

Что произойдет при отсутствии элемента **Bobby** в коллекции **Controls**?

Цикл просматривает все элементы управления формы в поисках метки с текстом «**Bobby**». После обнаружения метка удаляется из формы.

**Дополнительный вопрос: Как вы думаете, почему оператор `Controls.Remove()` не был помещен в цикл `foreach`?**

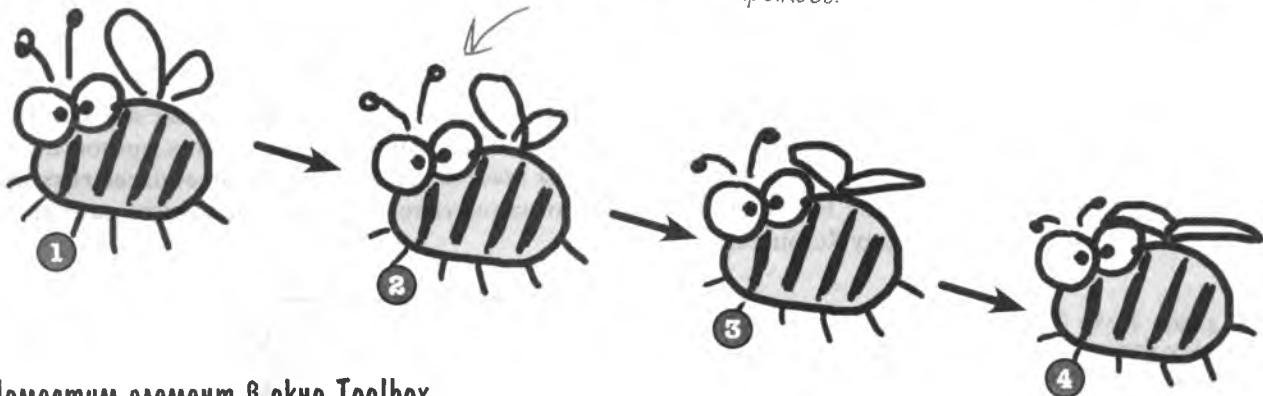
Попытавшись так сделать, вы получите исключение. Ведь нарушение целостности коллекции ведет к непредсказуемым результатам. Именно поэтому используется цикла `for`.

Коллекции нельзя редактировать в середине цикла `foreach`.

## Первый анимированный элемент управления

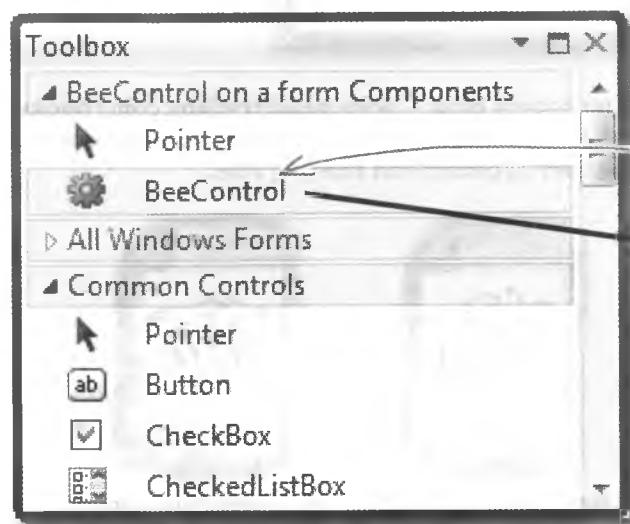
Мы собираемся построить **собственный** элемент управления с анимированным изображением пчелы. Получить анимацию не так тяжело, как кажется: рисуется ряд изображений, который при последовательном показе создает иллюзию движения. Это несложно сделать средствами C# и .NET.

Скачайте четыре изображения пчелы (от Bee animation 1.png до Bee animation 4.png) с сайта Head First Labs и добавьте их к ресурсам своего проекта. При быстром показе изображений возникает иллюзия движения крыльев.



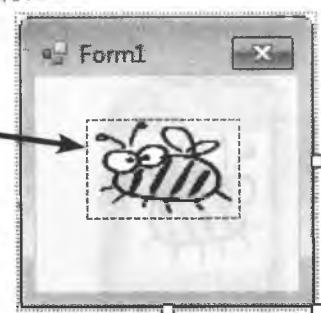
### Поместим элемент в окно Toolbox

Если вы корректно построите элемент управления BeeControl, то сможете перетаскивать его на форму из окна Toolbox. Он выглядит? как уже знакомый вам элемент PictureBox, демонстрировавший изображение пчелы, просто на этот раз картинка анимирована.



**Скачать изображения для этой главы можно по адресу:**  
[www.headfirstlabs.com/books/hfsharp/](http://www.headfirstlabs.com/books/hfsharp/)

Если правильно выбрали базовый класс для нового элемента, .NET позаботится о том, чтобы он появился в окне Toolbox.



Это напоминает PictureBox, просто в изображении встроена анимация. Угадайте, по отношению к какому классу класс BeeControl является производным.

## Базовый класс PictureBox

Так как все элементы в окне toolbox являются объектами, получить новый элемент легко. Достаточно добавить к проекту класс, наследующий от одного из существующих классов, задав затем поведение нового элемента.

Присвоим новому элементу управления имя **BeeControl**. Сначала свяжем с ним статичную картинку, к которой потом будет добавлена анимация, то есть начнем с уже знакомого элемента PictureBox.



- 1 Создайте новый проект и добавьте к его ресурсам четыре анимированные ячейки. Помните, как вы добавляли логотип объективильской компании в главе 1? Но в данном случае вас интересуют не ресурсы *формы*, а ресурсы *проекта*. Найдите файл **Resources.resx** проекта в окне Solution Explorer (как показано на рисунке) и дважды щелкните на нем для перехода на вкладку Resources.

В главе 1 мы добавляли логотип к файлу формы *Resources*. На этот раз изображения будут добавлены в общую коллекцию ресурсов, благодаря чему они станут доступны для всех классов проекта.

Вернитесь к главе 1, чтобы вспомнить, каким образом вы это сделали.



Изображения будут доступны для всего проекта, а не только для формы.

Дважды щелкните на строчке *Resources.resx* для перехода на вкладку *Resources*.

- 2 Изображения пчелы вы можете скачать по адресу <http://www.headfirstlabs.com/books/hfcsharp/>. Затем в верхней части вкладки *Resources* выберите в первом раскрывающемся списке вариант *Images*, а в списке *Add Resource* вариант *Add Existing File...*



Bee animation 1.png



Bee animation 2.png



Bee animation 3.png

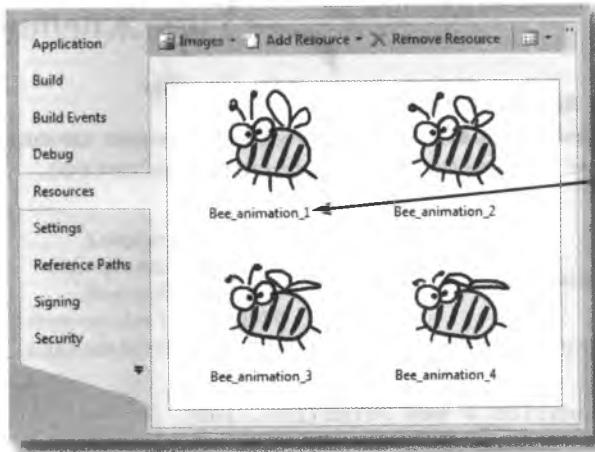


Bee animation 4.png

Импортируйте эти изображения и добавьте их к ресурсам своего проекта.

**3**

Доступ к добавленным ресурсам осуществляется посредством класса `Properties.Resources`. Введите в произвольном месте кода `Properties.Resources.` и окно IntelliSense покажет раскрывающийся список со всеми импортированными изображениями.



Обратите внимание на точку. Именно она указывает ИСР на необходимость вызывать окно с перечнем свойств и методов введенного вами класса.

В процессе работы программы изображение хранятся в памяти как объекты Bitmap.

`pictureBox1.Image = Properties.Resources.Bee_animation_1;`  
Здесь указывается изображение, связанное с элементом PictureBox (в данном случае наше начальное изображение).

Нужно добавить строку `<using System.Windows.Forms>`, так как в проекте присутствуют элементы PictureBox и Timer.

**4**

#### Добавим элемент BeeControl!

```
class BeeControl : PictureBox {
    private Timer animationTimer = new Timer();
    public BeeControl() {
        animationTimer.Tick += new EventHandler(animationTimer_Tick);
        animationTimer.Interval = 150;
        animationTimer.Start();
        BackColor = System.Drawing.Color.Transparent;
        BackgroundImageLayout = ImageLayout.Stretch;
    }
}
```

Убедитесь, что в верхней части класса находится строкка `<using System.Windows.Forms>`.

Здесь таймеру присваиваются начальные значения путем создания его экземпляра, задания свойства Interval и добавления обработчика событий.

Каждый отсчет таймера вызывает событие, оно увеличивает значение переменной cell на 1, и при помощи оператора switch назначает картинку свойству Image.

После возвращения к кадру #1, значение параметра cell становится равным 0.

```
private int cell = 0;
void animationTimer_Tick(object sender, EventArgs e) {
    cell++;
    switch (cell) {
        case 1: BackgroundImage = Properties.Resources.Bee_animation_1; break;
        case 2: BackgroundImage = Properties.Resources.Bee_animation_2; break;
        case 3: BackgroundImage = Properties.Resources.Bee_animation_3; break;
        case 4: BackgroundImage = Properties.Resources.Bee_animation_4; break;
        case 5: BackgroundImage = Properties.Resources.Bee_animation_3; break;
        default: BackgroundImage = Properties.Resources.Bee_animation_2;
        cell = 0; break;
    }
}
```

Введя код элемента управления, перестройте программу для отображения изменений в конструкторе.

**Постройте программу.** Элемент управления BeeControl должен появиться в окне toolbox. Перетащите его на форму, и вы получите анимированную пчелу!

## Кнопка добавления элемента BeeControl

Добавить элемент управления к форме легко, достаточно добавить его в коллекцию Controls. Также легко он удаляется из формы. Но элементы управления реализуют интерфейс `IDisposable`, поэтому следите за **высвобождением ресурсов** после удаления элементов.



1

### Удалите элемент BeeControl с формы и добавьте кнопку

Откройте конструктор форм и удалите элемент `BeeControl`. Добавьте к форме кнопку. Сделаем так, чтобы она управляла появлением и исчезновением элемента `BeeControl`.

2

### Код управляющей кнопки

Вот как выглядит обработчик событий для кнопки:

```
BeeControl control = null;  
private void button1_Click(object sender, EventArgs e) {  
    if (control == null) {  
        control = new BeeControl() { Location = new Point(100, 100) };  
        Controls.Add(control);  
    } else {  
        using (control) {  
            Controls.Remove(control);  
        }  
        control = null;  
    }  
}
```

Добавленный  
в коллек-  
цию `Controls`  
элемент  
немедленно  
появляется  
на форме.

Вы используете  
инициализатор для  
задания свойств  
`BeeControl` после соз-  
дания его экземпляра.

Оператор `using` гарантирует вы-  
свобождение ресурсов после удаления  
элемента из коллекции `Controls`.

Запустите программу. Первый щелчок на кнопке должен добавлять элемент `BeeControl`. При втором щелчке элемент удаляется. Ссылка на него хранится в закрытом поле `control`. При исчезновении элемента она указывает на значение `null`.

Чтобы добавить элемент в окно toolbox, создайте класс, производный от класса `Control`.

За сценой

Все визуальные элементы в окне Toolbox наследуют от `System.Windows.Forms.Control`. Члены этого класса вам уже знакомы: `Visible`, `Width`, `Height`, `Text`, `Location`, `BackColor`, `BackgroundImage`... вы их видите в окне Properties для всех элементов управления.

## Удаление дочерних элементов управления

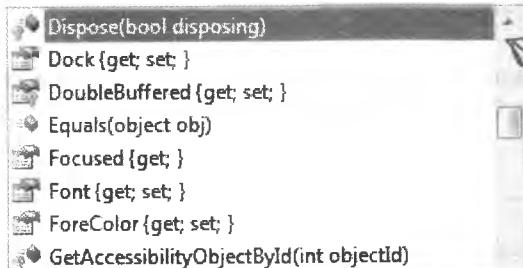
Отработав, элементы управления должны высвобождать ресурсы. Но элемент BeeControl создает экземпляр элемента Timer ... который остается неудаленным! К счастью, эту проблему легко решить – достаточно перекрыть метод `Dispose()`.

Класс элементов управления реализует интерфейс `IDisposable`, поэтому нужно проследить, чтобы все элементы удалялись, когда в них пропадает надобность.

### 3 Перекройте метод `Dispose()` и удалите таймер

Наш класс `BeeControl` должен обладать унаследованным методом `Dispose()`. Достаточно перекрыть и расширить этот метод. Введите в код класса ключевое слово `override`:

```
class BeeControl : PictureBox {
    override
```



После введения ключевого слова `override` появится окно IntelliSense с перечнем доступных для перекрытия методов. Выберите метод `Dispose()`!

ИСР введет строчку `base.Dispose()`, которой будет вызываться данный метод:

```
protected override void Dispose(bool disposing) {
    base.Dispose(disposing);
}
```

### 4 Код удаления таймера

Добавьте в конец нового метода `Dispose()` код, вызывающий метод `animationTimer.Dispose()`, когда аргумент `disposing` имеет значение `true`.

```
protected override void Dispose(bool disposing) {
    base.Dispose(disposing);
    if (disposing) {
        animationTimer.Dispose();
    }
}
```

Мы перекрываем помеченный ключевым словом `protected` метод `Dispose()`, вызываемый реализацией метода `IDisposable.Dispose()` элемента управления.

Теперь элемент `BeeControl` убирает за собой!

Проверьте сами. Добавьте точку останова в вставленную строчку и запустите программу. При каждом удалении элемента `BeeControl` из коллекции `Controls` будет вызываться метод `Dispose()`.

**Любой созданный вами элемент управления должен удалять все порожденные им элементы и другие допускающие удаление объекты.**

Если вы собираетесь создавать свои элементы управления, вам будет полезно прочитать материалы с сайта <http://msdn.microsoft.com/ru-ru/library/system.idisposable.dispose.aspx>

## Класс UserControl

Есть и более простой способ создания элементов управления для окна toolbox. Вместо того чтобы пользоваться классом, наследующим от существующего элемента управления, вам нужно добавить в свой проект **класс UserControl**. Вы работаете с ним, как с формой, перетаскивая на него элементы из окна toolbox. И точно так же, как в случае с формой вы пользуетесь событиями. Поэтому заново создадим элемент BeeControl, но уже с помощью класса UserControl.

### Упражнение!

- 1 Откройте новый проект Windows Forms Application и добавьте к его ресурсам четыре изображения. Перетащите на форму кнопку и введите известный вам код, добавляющий и удаляющий элемент BeeControl.
- 2 Щелкните правой кнопкой мыши на имени проекта в окне Solution Explorer и выберите команду Add >> User Control... Добавьте элемент BeeControl. Он будет открыт в конструкторе форм.

Используйте метод animationTimer\_Tick() и поле cell field из старой версии элемента управления.
- 3 Перетащите на ваш элемент управления элемент Timer. Назовите его **animationTimer**, свойству **Interval** присвойте значение 150, свойству **Enabled** – значение true. Дважды щелкните на таймере, чтобы добавить обработчик события Tick. Используйте для него тот же код, что и раньше.
- 4 Обновите конструктор элемента BeeControl:

```
public BeeControl() {  
    InitializeComponent();  
    BackColor = System.Drawing.Color.Transparent;  
    BackgroundImageLayout = ImageLayout.Stretch;  
}
```

Эти изменения можно внести и посредством окна Properties.
- 5 Запустите программу, кнопка должна работать так же, как и раньше, просто теперь ею управляет класс UserControl. Именно он отвечает за появление и удаление элемента BeeControl.

**Класс UserControl позволяет легко добавить элемент управления в окно Toolbox. Как и при работе с формой, вы можете перетаскивать на него другие элементы управления и пользоваться событиями.**



Сколько я ни пользовалась элементами управления, мне ни разу не приходилось их удалять!  
Зачем это вообще делать?

**Вам не требовалось удалять элементы управления, так как эту работу выполняла форма.**

Впрочем, не верьте нам на слово. Воспользуйтесь функцией поиска и найдите в коде слово `Dispose`, вы обнаружите, что ИСР добавила метод в файл `Form1.Designer.cs` для перекрытия метода `Dispose()` и вызова своего собственного метода `base.Dispose()`. При удалении формы **автоматически удаляется вся связанная с ней коллекция Controls**. Но если вы вручную удаляете элементы управления с формы или создаете новые экземпляры, не принадлежащие коллекции `Controls`, высвобождать ресурсы предстоит уже вам.

### часто Задаваемые Вопросы

**В:** Почему код формы элементов `BeeControl`, один из которых получен из элемента `PictureBox`, а другой на базе класса `UserControl`, работает совершенно одинаково?

**О:** Код безразлично, каким именно образом реализуется объект `BeeControl`. Ему важна лишь возможность добавить объект методу `Controls`.

**В:** После двойного щелчка на строчке `OldBeeControl` в окне `Solution Explorer` появилось сообщение о добавлении в мой класс компонентов. Что это значит?

**О:** Когда вы создаете элемент управления, добавляя к проекту класс, наследующий от элемента `PictureBox`, ИСР позволяет вам работать с компонентами. К ним относятся невизуальные элементы управления, например, `Timer`

и `OpenFileDialog`. Их значки появляются под формой.

Проверьте сами. Создайте пустой класс, наследующий от класса `PictureBox`. Перестройте проект и дважды щелкните на его название в окне `Solution Explorer`. Появится сообщение:

*«Чтобы добавить компоненты в класс, перетащите их из окна Toolbox и воспользуйтесь окном Properties для задания их свойств».*

Перетащите из окна `toolbox` элемент `OpenFileDialog`. Он появится в виде значка. Вы можете выделите его и изменить его свойства. Проделайте эту операцию, а затем вернитесь к коду класса. Посмотрите на конструктор, там вы найдете код создания экземпляра `OpenFileDialog` и задания его свойств.

**В:** Редактируя свойства элемента `OpenFileDialog`, я заметил сообщение об ошибке: «Вы должны перестроить проект, чтобы изменения отобразились в любом открытом конструкторе». Почему оно появилось?

**О:** Элементы управления отображаются конструктором. И для отображения последней версии элемента программу требуется перестроить.

Помните, как двигались крылья пчелы на элементе `BeeControl`, даже в процессе перетаскивания его из окна `Toolbox`? Программа еще не была запущена, но написанный вами код уже выполнялся. Таймер запускал событие `Tick`, а его обработчик менял картинку. Подобное поведение возможно только после компиляции кода, когда программа работает в памяти. Именно поэтому вам напомнили о необходимости регулярно обновлять код, чтобы элементы управления отображались корректно.

## Поместим на форму анимированных пчел

Все готово для анимации симулятора. У вас есть класс BeeControl и две формы и нужно расположить пчел в пространстве и перемещать их с одного объекта на другой, поддерживая это движение. Требуется также расположить цветы в форме FieldForm. Впрочем, это простая задача, так как цветы статичны. Этот новый код мы поместим в класс Renderer. Вот что он будет делать:

1

### Форма со статистикой станет базовой

Добавим к нашему проекту две недостающие формы. Форма HiveForm будет отображать события в улье, а форма FieldForm – происходящее на поле. Поместите в конструктор основной формы строчки отображения двух дочерних форм. Передайте ссылку в основную форму, указывая, что именно она является их владельцем:

```
public Form1() {  
    // остальной код конструктора Form1  
    hiveForm.Show(this); }  
    fieldForm.Show(this); }
```

Нужно связать формы, отображающие улей и поле, с формой статистики, в результате свертка последней формы будет приводить к свертке первых двух. Для этого форму со статистикой нужно объявить владельцем.



Перед тем как приступить к построению визуализатора, подумаем, как именно должен работать класс Renderer...

Каждая форма обладает методом Show(). Если вы хотите сделать одну форму владельцем другой, передайте ссылку на нее в метод Show().

2

### Класс Renderer ссылается на объект world и на обе формы

Класс Renderer должен знать положение каждой пчелы и каждого цветка, поэтому ему нужна ссылка на объект World. А при добавлении, перемещении и удалении элементов управления с форм не обойтись без ссылок на эти формы:

```
class Renderer {  
    private World world;  
    private HiveForm hiveForm;  
    private FieldForm fieldForm; }
```

Начните класс Renderer с этих строк.



3

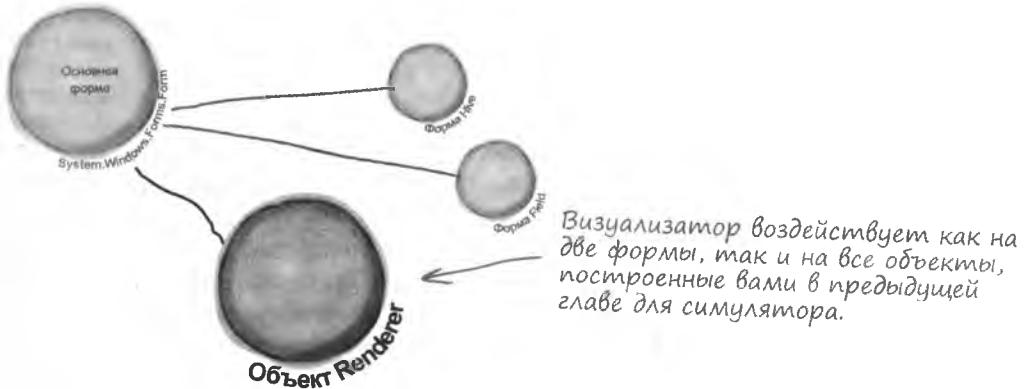
### Состояние элементов отслеживают словари

Класс World следит за пчелами и цветами при помощи списков List<Bee> и List<Flower>. Визуализатору нужен доступ к объектам Bee и Flower, чтобы понять, какие элементы BeeControl и PictureBox им соответствуют. Если он не находит элемента управления, то должен создать его. В этом вам помогут словари. В класс Renderer нужно добавить два закрытых поля:

```
private Dictionary<Flower, PictureBox> flowerLookup =  
    new Dictionary<Flower, PictureBox>();  
private Dictionary<Bee, BeeControl> beeLookup =  
    new Dictionary<Bee, BeeControl>();
```

Эти словари позволяют визуализатору хранить элементы управления для каждой пчелы и каждого цветка из нашего мира.

Словари задают однозначное соответствие между пчелой или цветком и их элементами управления.

**4****Пчелы и цветы знают свое место**

Класс Point хранит информацию о положении цветов и пчел. Для любого объекта Bee можно легко посмотреть BeeControl и задать положение.

```
beeControl = beeLookup[bee];
beeControl.Location = bee.Location;
```

Мы можем увидеть элементы управления для любой пчелы и любого цветка. После чего положение элемента совмещается с положением объекта.

**5****Визуализатор добавляет элементы управления новым пчелам**

Если метод словаря ContainsKey() для выделенного объекта Bee возвращает значение false, значит, для данной пчелы отсутствует элемент управления. Класс Renderer должен создать BeeControl и добавить его в словарь, затем на форму. (Не забудьте вызвать метод BringToFront() (Поместить вперед) элемента управления, чтобы пчела не оказалась закрыта цветами.)

```
if (!beeLookup.ContainsKey(bee)) {
    beeControl = new BeeControl() { Width = 40, Height = 40 };
    beeLookup.Add(bee, beeControl);
    hiveForm.Controls.Add(beeControl);
    beeControl.BringToFront();
} else
    beeControl = beeLookup[bee];
```

Метод ContainsKey() показывает наличие пчелы в словаре. Если она там отсутствует, значит, ее нужно добавить вместе с ее элементом управления.

Помните, что в качестве ключа словари могут использовать что угодно? В данном случае ключом выступает объект Bee. Визуализатору нужно знать, какой элемент BeeControl на форме соответствует той или иной пчеле. Поэтому он находит пчелу в словаре, определяет ее элементы управления и таким образом получает возможность перемещать ее по форме.

Метод BringToFront() гарантирует, что пчелы всегда будут появляться поверх цветов. Соответственно, для формы Hive этот метод обеспечивает появление пчел поверх фона.

начнем!

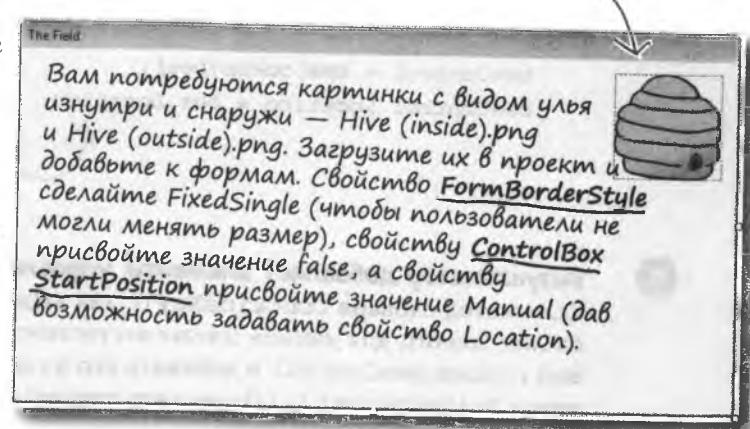
## Формы Hive и Field

Добавим недостающие формы. Возьмите существующий проект симулятора и при помощи команды Add >> Existing Item... добавьте пользовательский элемент управления BeeControl. В классе UserControl три файла — .cs, .designer.cs и .resx — добавьте их все. В файлах .cs и .designer.cs измените пространство имен в соответствии с вашим проектом. Перестройте проект; элемент BeeControl должен появиться в окне Toolbox. Добавьте графические ресурсы. Затем щелкните правой кнопкой мыши на имени проекта в окне Solution Explorer и выберите команду Windows Form... в меню Add. Назовите файлы HiveForm.cs и FieldForm.cs, и ИСР автоматически присвоит их свойствам Name значения HiveForm и FieldForm. Вы только что создали два новых класса.

Это элемент управления PictureBox, свойство BackgroundImage которого связано с внешним изображением улья. Свойству BackgroundImageLayout присвоено значение Stretch. При загрузке изображений улья в конструктор ресурсов они начинают отображаться в списке, вызываемом щелчком на кнопке ... рядом со свойством BackgroundImage в окне Properties.



Свойству форме BackgroundImage присвойте изображение улья, а BackgroundImageLayout — Stretch.



### Определим местоположение объектов

Нужно понять, где на форме FieldForm находится улей. В окне Properties создайте обработчик для события **MouseClick** формы Hive и добавьте код:

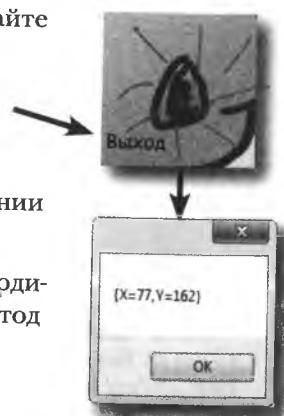
```
private void HiveForm_MouseClick(object sender, MouseEventArgs e) {  
    MessageBox.Show(e.Location.ToString());  
}
```

Мы скоро запустим форму. В этот момент вам нужно будет щелкнуть на изображении выхода, и обработчик событий покажет точную координату этого места.

Добавьте аналогичный обработчик к форме Field. Щелчком получите координаты выхода, питомника и фабрики. Эти координаты позволяют обновлять метод InitializeLocations(), написанный в предыдущей главе для класса Hive:

Запустив симулятор, вы сможете заполнить коллекцию координат внутри улья.

```
private void InitializeLocations()  
  
    locations = new Dictionary<string, Point>();  
    locations.Add("Entrance", new Point(626, 110));  
    locations.Add("Nursery", new Point(77, 162));  
    locations.Add("HoneyFactory", new Point(157, 78));  
    locations.Add("Exit", new Point(175, 180));
```



Закончив, уберите обработчик щелчка мышки... он требовался только для определения координат ваших объектов.

Это наш вариант координат, для ваших форм координаты могут отличаться.

# Построение визуализатора

Вот код класса Renderer. Основная форма вызывает метод Render() этого класса после метода World.Go(), чтобы отобразить на формах пчел и цветы. Убедитесь, что изображение цветка (Flower.png) также загружено в ваш проект.

```

class Renderer {
    private World world;
    private HiveForm hiveForm;
    private FieldForm fieldForm;

    private Dictionary<Flower, PictureBox> flowerLookup =
        new Dictionary<Flower, PictureBox>();
    private List<Flower> deadFlowers = new List<Flower>();

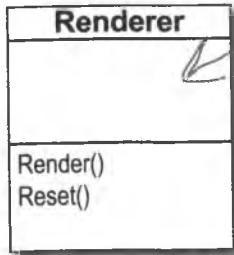
    private Dictionary<Bee, BeeControl> beeLookup =
        new Dictionary<Bee, BeeControl>();
    private List<Bee> retiredBees = new List<Bee>();

    public Renderer(World world, HiveForm hiveForm, FieldForm fieldForm) {
        this.world = world;
        this.hiveForm = hiveForm;
        this.fieldForm = fieldForm;
    }

    public void Render() {
        DrawBees();
        DrawFlowers();
        RemoveRetiredBeesAndDeadFlowers();
    }

    public void Reset() {
        foreach (PictureBox flower in flowerLookup.Values) {
            fieldForm.Controls.Remove(flower);
            flower.Dispose();
        }
        foreach (BeeControl bee in beeLookup.Values) {
            hiveForm.Controls.Remove(bee);
            fieldForm.Controls.Remove(bee);
            bee.Dispose();
        }
        flowerLookup.Clear();
        beeLookup.Clear();
    }
}

```



Все поля визуализатора закрыты, так как другие классы не должны иметь доступа к его свойствам. Класс world просто вызывает методы Render() и Reset(). Первый рисует графику на формах, а второй при перезагрузке форм удаляет с них элементы управления.

Отслеживание состояния пчел и цветов осуществляется через объекты Bee и Flower. Для отображения цветов используется элемент PictureBox, а для отображения пчел — элемент BeeControl. При помощи словарей визуализатор связывает каждую пчелу и каждый цветок с их элементами BeeControl и PictureBox.

Когда цветок вянет, а пчела уходит на заслуженный отдох, информация о них удаляется из словарей при помощи списков deadFlowers и retiredBees.

При перезагрузке симулятора для каждой формы вызывается метод Controls.Remove(), удаляющий все элементы управления. Он находит каждый элемент в словаре и вызывает для него метод Dispose(). После чего очищает и словари также.

Цветы рисуются при помощи двух циклов foreach. Первый добавляем элемент PictureBoxes для новых цветов, а второй удаляем этот элемент для цветов, которые завяли.

```
private void DrawFlowers() {
    foreach (Flower flower in world.Flowers)
        if (!flowerLookup.ContainsKey(flower))
            PictureBox flowerControl = new PictureBox() {
                Width = 45,
                Height = 55,
                Image = Properties.Resources.Flower,
               SizeMode = PictureBoxSizeMode.StretchImage,
                Location = flower.Location
            };
    flowerLookup.Add(flower, flowerControl);
    fieldForm.Controls.Add(flowerControl);
}
```

Метод DrawFlowers()  
задает положение  
элемента PictureBox  
на форме при помощи  
свойства Location объ-  
екта Flower.

```
foreach (Flower flower in flowerLookup.Keys) {
    if (!world.Flowers.Contains(flower))
        PictureBox flowerControlToRemove = flowerLookup[flower];
    fieldForm.Controls.Remove(flowerControlToRemove);
    flowerControlToRemove.Dispose();
    deadFlowers.Add(flower);
}
```

Первый цикл foreach  
использует словарь  
flowerLookup для про-  
верки наличия элемента  
управления у выделенного  
цветка. Не обнаружив  
такового, он создает но-  
вый элемент PictureBox  
с помощью инициализа-  
тора объектов, добавля-  
ет его к форме, а затем  
добавляет его в словарь  
flowerLookup.

Второй цикл foreach  
ищет в словаре  
flowerLookup элементы  
PictureBox, которые уже  
не принадлежат форме,  
и удаляет их.

Удалив элемент PictureBox, он  
вызывает его метод Dispose().  
После чего объект Flower добав-  
ляется в список deadFlowers.

```
private void DrawBees() {
    BeeControl beeControl;
    foreach (Bee bee in world.Bees) {
        beeControl = GetBeeControl(bee);
        if (bee.InsideHive) {
            if (fieldForm.Controls.Contains(beeControl))
                MoveBeeFromFieldToHive(beeControl);
        } else if (hiveForm.Controls.Contains(beeControl))
            MoveBeeFromHiveToField(beeControl);
        beeControl.Location = bee.Location;
    }
}
```

После удаления эле-  
мента BeeControl  
нужно вызывать его  
метод Dispose(),  
при этом будет  
удален и связанный  
с ним таймер.

```
foreach (Bee bee in beeLookup.Keys) {
    if (!world.Bees.Contains(bee))
        beeControl = beeLookup[bee];
    if (fieldForm.Controls.Contains(beeControl))
        fieldForm.Controls.Remove(beeControl);
    if (hiveForm.Controls.Contains(beeControl))
        hiveForm.Controls.Remove(beeControl);
    beeControl.Dispose();
    retiredBees.Add(bee);
}
```

Метод DrawBees() так-  
же использует два цикла  
foreach. Он по сути де-  
лает то же самое, что  
и метод DrawFlowers(). Но  
в данном случае функция  
сложнее, поэтому мы рас-  
пределили его функции по  
отдельным методам, об-  
легчив понимание кода.

Метод DrawBees() проверя-  
ет, не возникла ли ситуация,  
когда пчела находится в улье,  
а ее элемент управления —  
на форме FieldForm или  
наоборот. Для перемещения  
элемента BeeControls между  
формами используются два  
дополнительных метода.

Второй цикл foreach ра-  
ботает в основном как в  
методе DrawFlowers(), но  
ему еще приходится уда-  
лять элементы BeeControl  
из правой формы.

**В верхней части класса Renderer должны находиться строки using System.Drawing и using System.Windows.Forms.**

```
private BeeControl GetBeeControl(Bee bee) {
    BeeControl beeControl;
    if (!beeLookup.ContainsKey(bee)) {
        beeControl = new BeeControl() { Width = 40, Height = 40 };
        beeLookup.Add(bee, beeControl);
        hiveForm.Controls.Add(beeControl);
        beeControl.BringToFront();
    }
    else
        beeControl = beeLookup[bee];
    return beeControl;
}
```

Не забудьте:  
! означает НЕ!

Метод GetBeeControl() ищет пчелу в словаре beeLookup. При отсутствии нужной пчелы он создает элемент управления BeeControl размером 40 x 40 и добавляет его к форме hive (ведь именно здесь появляются новые пчелы).

```
private void MoveBeeFromHiveToField(BeeControl beeControl) {
    hiveForm.Controls.Remove(beeControl);
    beeControl.Size = new Size(20, 20);
    fieldForm.Controls.Add(beeControl);
    beeControl.BringToFront();
}
```

Метод MoveBeeFromHiveToField() берет указанный элемент BeeControl из коллекции Controls улья и добавляет его в коллекцию Controls поля.

Пчелы на поле имеют меньший размер, чем пчелы в улье, соответственно, метод должен менять свойство Size элемента BeeControl.

```
private void MoveBeeFromFieldToHive(BeeControl beeControl) {
    fieldForm.Controls.Remove(beeControl);
    beeControl.Size = new Size(40, 40);
    hiveForm.Controls.Add(beeControl);
    beeControl.BringToFront();
}
```

Метод MoveBeeFromFieldToHive() возвращает элемент управления BeeControl обратно в улей. Соответственно, он должен увеличить его размер.

```
private void RemoveRetiredBeesAndDeadFlowers() {
    foreach (Bee bee in retiredBees)
        beeLookup.Remove(bee);
    retiredBees.Clear();
    foreach (Flower flower in deadFlowers)
        flowerLookup.Remove(flower);
    deadFlowers.Clear();
}
```

Как только методы DrawBees() и DrawFlowers() обнаруживают, что цветка или пчелы большие нет, они добавляют эти объекты в списки deadFlowers и retiredBees, чтобы в конце кадра произошло их удаление.

Напоследок визуализатор вызывает метод для удаления из словарей всех увядших цветков и прекративших работу пчел.

## Соединим основную форму с формами HiveForm и FieldForm

Мы создали визуализатор, но у него нет форм, в которые можно было бы выводить изображение. Чтобы исправить ситуацию, вернемся к классу Form основной формы (скорее всего, он называется Form1) и внесем изменения в код:

Код перезагрузки мира помещен в метод ResetSimulator().

```
public partial class Form1 : Form {
    private HiveForm hiveForm = new HiveForm();
    private FieldForm fieldForm = new FieldForm();
    private Renderer renderer;
```

// остальные поля

```
public Form1() {
    InitializeComponent();
```

Код, создающий экземпляр объекта World, поместите в метод ResetSimulator().

```
MoveChildForms();
hiveForm.Show(this);
fieldForm.Show(this);
ResetSimulator();
```

Форма передает ссылку на себя методу Form.Show(), становясь таким способом родительской формой.

```
timer1.Interval = 50;
timer1.Tick += new EventHandler(RunFrame);
timer1.Enabled = false;
UpdateStats(new TimeSpan());
```

Конструктор основной формы помещает на место две дочерние формы, а потом отображает их. Затем он вызывает метод ResetSimulator(), создающий экземпляр объекта Renderer.

```
} private void MoveChildForms()
```

Так как свойство StartPosition обеих дочерних форм имеет значение Manual, основная форма может перемещать их при помощи свойства Location.

```
hiveForm.Location = new Point(Location.X + Width + 10, Location.Y);
fieldForm.Location = new Point(Location.X,
    Location.Y + Math.Max(Height, hiveForm.Height) + 10);
```

Этот код заставляет дочерние формы перемещаться. При этом форма Hive расположена рядом с основной формой, а форма Field — под ними.

```
public void RunFrame(object sender, EventArgs e) {
    framesRun++;
    world.Go(random);
    renderer.Render();
    // предыдущий код
}
```

Эта строка в методе RunFrame заставляет симулятор обновлять изображение при каждом вызове метода Go().

```
private void Form1_Move(object sender, EventArgs e) {
    MoveChildForms();
```

Кнопка Events

В окне Properties позволяет добавить обработчик события Move.

Свойство StartPosition обеих дочерних форм должно иметь значение Manual, иначе метод MoveChildForms() работать не будет.

Событие Move возникает при каждом перемещении основной формы. Метод MoveChildForms() гарантирует, что дочерние формы будут перемещаться следом.

Основная форма, загружаясь, создает экземпляры двух других форм. После этого они становятся всего лишь объектами в куче, и не отображаются, пока не вызван их метод Show().

Здесь мы создаем экземпляры  
классов World и Renderer, которые  
перезагружают симулятор.

```
private void ResetSimulator() {
    framesRun = 0;
    world = new World(new BeeMessage(SendMessage));
    renderer = new Renderer(world, hiveForm, fieldForm);
}
```

```
private void reset_Click(object sender, EventArgs e) {
    renderer.Reset();
    ResetSimulator();
    if (!timer1.Enabled)
        toolStrip1.Items[0].Text = "Start simulation";
}
```

Кнопка Reset вызывает метод Reset() для удаления всех элементов BeeControls и PictureBox и перегрузки симулятора.

```
private void openToolStripButton_Click(object sender, EventArgs e) {
    // Остаток кода для кнопки остается без изменений.
```

```
    renderer.Reset();
    renderer = new Renderer(world, hiveForm, fieldForm);
}
```

Напоследок добавим на элемент ToolStrip код для кнопки Open. Он будет использовать метод Reset() для удаления всех ичел и цветов из коллекций Controls обеих форм, а затем создавать новый визуализатор на основе загруженного пользователем мира.

## часто Задаваемые Вопросы

**В:** Для отображения формы мы использовали метод Show(), но зачем мы передавали в качестве параметра ключевое слово this?

**О:** Дело в том, что форма это всего лишь очередной класс. При ее отображении вы создаете экземпляр этого класса и вызываете его метод Show(). Существует перегруженная версия этого метода, которая в качестве параметра использует родительское окно. Между родительским и дочерним окном существует особая связь, например, сворачивание родительского окна ведет к автоматическому сворачиванию всех дочерних окон.

**В:** Можно ли редактировать предустановленные элементы управления и вносить изменения в их код?

**О:** Нет, доступ к коду встроенных элементов управления Visual Studio отсутствует. Но каждый из этих элементов является классом, от которого вы можете наследовать. Вспомните, как наследованием от элемента PictureBox вы создали элемент BeeControl.

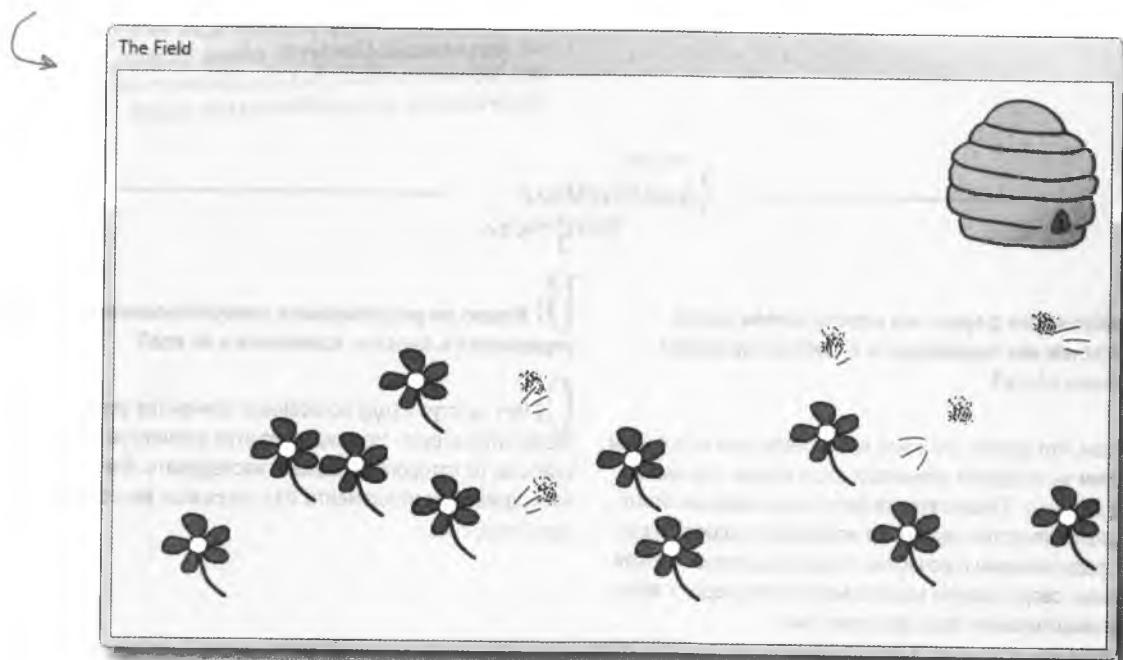
что-то не так

## Тестирование... жжжж...

Скомпилируйте код, исправьте любые обнаруженные ошибки и запустите симулятор.



Поменяйте константы и посмотрите, как визуализатор реагирует на изменение количества цветов и пчел.



## Выглядит красиво, но чего-то не хватает...

Присмотревшись к пчелам, жужжащим вокруг улья и цветов, вы заметите, что их визуализация происходит не без проблем. Помните, что свойству BackColor элемента BeeControl было присвоено значение Color.Transparent? К сожалению, этого недостаточно, чтобы избежать проблем.

1

### Проблемы с производительностью

Вы заметили, как замедляется симулятор, когда все пчелы залетают в улей? Особенно это заметно, если добавить пчел, увеличив константу в классе Hive. Обратите внимание, как при этом меняется частота кадров.

2

### Неполная прозрачность фона

И другая, совсем *отдельная* проблема. Графический файл с цветком имеет прозрачный фон. Это позволяет поместить цветы на фон формы, но не очень красиво выглядит, когда цветы начинают перекрываться.

Прозрачный фоновый цвет элемента PictureBox означает, что на изображении в этом месте будут выводиться прозрачные пиксели, сквозь которые будет проходить фон формы... но это не всегда то, что нам нужно.



Когда один элемент PictureBox оказывается поверх другого, C# рисует прозрачные пиксели, удаляя часть пикселов с нижнего элемента.

3

### Фон пчел также недостаточно прозрачен

Кажется, параметр Color.Transparent имеет ряд ограничений. Когда пчелы садятся на цветок, появляется вырезанный прямоугольный фрагмент. Прозрачность лучше проявляется внутри улья, но при перекрытии изображений пчел возникает та же проблема. Кроме того, если присмотреться к пчелам, перемещающимся по улью, можно заметить, как искается изображение.



## Проблемы с производительностью

Загруженные изображения пчел имеют большой размер. Действительно большой. Откройте одно из них в программе просмотра и убедитесь. Это означает, что элемент PictureBox ужимает изображение при каждом изменении. А ведь изменение масштаба изображения занимает время. Именно поэтому движение пчел замедляется, когда они залетают в улей. Прозрачный фон элемента BeeControl удваивает работу: сначала требуется уменьшить изображение пчелы, а затем участок фона, на котором будут выводиться прозрачные пиксели.

Bee animation 1.png



Изображение пчелы велико, и элементу PictureBox в каждом кадре требуется время, чтобы его уменьшить.

Элемент PictureBox уменьшает картинку с изображением пчелы. Это занимает время...

Картина с изображением внутреннего устройства улья тоже огромна. Каждый раз, когда перед ней оказывается пчела, элемент PictureBox уменьшает ее до нужного размера.

Hive (Inside).png



...поэтому для увеличения производительности симулятора картинку нужно уменьшить до ее отображения.

Для увеличения производительности нужно добавить визуализатору метод, меняющий размер изображения в момент его загрузки, чтобы для отображения пчел и фонового рисунка улья пользоваться уже отмасштабированными версиями.



## 1 Метод ResizeImage для визуализатора

Все изображения проекта хранятся как объекты Bitmap. Вот статический метод, меняющий их размер, который нужно добавить в класс Renderer:

```
public static Bitmap ResizeImage(Bitmap picture, int width, int height) {  
    Bitmap resizedPicture = new Bitmap(width, height);  
    using (Graphics graphics = Graphics.FromImage(resizedPicture)) {  
        graphics.DrawImage(picture, 0, 0, width, height);  
    }  
    return resizedPicture;  
}
```

Что такое объект Graphics и как работает  
данний метод, мы поговорим чуть позже.

## 2 Метод ResizeCells для элемента BeeControl

Элемент BeeControl хранит собственные объекты Bitmap в виде массива. Вот код, заполняющий этот массив и дающий каждому элементу правильный размер:

```
private Bitmap[] cells = new Bitmap[4]; Объекты Bitmap, хранящие картинки с изображением пчел,  
масштабируются при помощи метода ResizeImage().  
private void ResizeCells() {  
    cells[0] = Renderer.ResizeImage(Properties.Resources.Bee_animation_1, Width, Height);  
    cells[1] = Renderer.ResizeImage(Properties.Resources.Bee_animation_2, Width, Height);  
    cells[2] = Renderer.ResizeImage(Properties.Resources.Bee_animation_3, Width, Height);  
    cells[3] = Renderer.ResizeImage(Properties.Resources.Bee_animation_4, Width, Height);  
}
```

## 3 Пусть оператор switch работает с массивом cells, а не с ресурсами

Оператор switch обработчика событий Tick задает свойство BackgroundImage:

```
BackgroundImage = Properties.Resources.Bee_animation_1;
```

Замените Properties.Resources.Bee\_animation\_1 на cells[0]. Измените и остальные строки case, введя на вторую позицию cells[1], на третью – cells[2], на четвертую – cells[3], на пятую – cells[2], на позицию default – cells[1], чтобы отображались только отмасштабированные рисунки.

## 4 Добавить вызов метода ResizeCells() для элемента BeeControl

Новый метод ResizeCells() должен вызываться в нижней части конструктора. Затем дважды щелкните на строчке BeeControl в окне Properties. Откройте в этом окне страницу Events (щелкнув на значке с изображением молнии) и дважды щелкните на строчке Resize, чтобы добавить обработчик этого события. Новый обработчик также должен вызывать метод ResizeCells(), чтобы масштабирование формы сопровождалось масштабированием анимированных картинок.

## 5 Вручную выберите фоновое изображение формы

В окне Properties выберите для фонового изображения улья вариант (none). Затем в конструкторе загрузите вместо него отмасштабированное изображение.

```
public partial class HiveForm : Form {  
    public HiveForm() {  
        InitializeComponent();  
        BackgroundImage = Renderer.ResizeImage(  
            Properties.Resources.Hive_inside_,  
            ClientRectangle.Width, ClientRectangle.Height);  
    }  
}
```

Свойство ClientRectangle формы включает  
свойство Rectangle, содержащее информа-  
цию об отображаемой области.

Запустите симулятор, он работает намного быстрее!

## Объект Graphics

Посмотрим на добавленный к визуализатору метод `ResizeImage()`. Он начинает с создания объекта `Bitmap` требуемого размера. Затем при помощи метода `Graphics.FromImage()` **создается объект `Graphics`** и используется метод `DrawImage()` этого объекта для отображения картинки на объекте `Bitmap`. Обратите внимание, каким образом методу `DrawImage()` передаются параметры `width` и `height`. Именно в этот момент осуществляется масштабирование. Наконец вам возвращается только что созданный объект `Bitmap`, который можно использовать в качестве фонового изображения улья или анимированного рисунка пчелы.

```
public static Bitmap ResizeImage(Bitmap picture, int width, int height) {  
    Bitmap resizedPicture = new Bitmap(width, height);  
    using (Graphics graphics = Graphics.FromImage(resizedPicture)) {  
        graphics.DrawImage(picture, 0, 0, width, height);  
    }  
    return resizedPicture;  
}
```

### Масштабирование изображения

Перетащите кнопку на форму `Field` и добавьте к ней этот код, создающий элемент `PictureBox` размером  $100 \times 100$  пикселов с черной линией по краю, что позволяет оценить его размер. Затем метод `ResizeImage()` масштабирует картинку пчелы до  $80 \times 40$  пикселов и назначает свойству `Image`. После добавления к форме элемента `PictureBox` появляется пчела.

```
private void button1_Click(object sender, EventArgs e)  
{  
    PictureBox beePicture = new PictureBox();  
    beePicture.Location = new Point(10, 10);  
    beePicture.Size = new Size(100, 100);  
    beePicture.BorderStyle = BorderStyle.FixedSingle;  
    beePicture.Image = Renderer.ResizeImage(  
        Properties.Resources.Bee_animation_1, 80, 40);  
    Controls.Add(beePicture);  
}
```

Вы наблюдаете процесс масштабирования изображения — сжатая картинка намного меньше элемента `PictureBox`. И метод `ResizeImage()` уменьшил размер элемента.

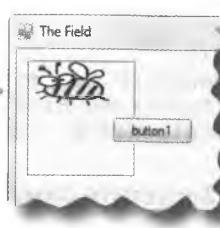
Формы и элементы управления снабжены методом `CreateGraphics()`, который возвращает новый объект `Graphics`. Впрочем, скоро вы все увидите собственными глазами.

Вы передаете методу изображение с данными о том, какую ширину и высоту оно должно иметь.

Метод `FromImage()` возвращает новый объект `Graphics`, позволяющий вставлять графические фрагменты в изображение. Воспользуйтесь функцией `IntelliSense` для изучения методов, принадлежащих классу `Graphics`. Например, метод `DrawImage()` копирует изображение в поле `resizedPicture` с координатой  $(0, 0)$  и масштабирует в соответствии с заданными параметрами `width` и `height`.

Это пример. Закончив его, удалите кнопку и код.

**Метод `ResizeImage()` создает объект `Graphics`, рисующий на невидимом объекте `Bitmap`. Он возвращает объект `Bitmap`, который и отображается на форме или элементе `PictureBox`.**



## Объект Bitmap

Что происходит с графическими файлами, которые импортируются в ресурсы проекта? Вы уже знаете способ доступа к ним — `Properties.Resources`. Но как именно поступает с ними приложение?

.NET превращает изображения в объект `Bitmap`:



Bee animation 1.png

```
Bitmap bee = new Bitmap("Bee animation 1.png")
```



Если проблем с производительностью нет, добавляйте пчел, пока программа не замедлит работу!

Класс `Bitmap` имеет несколько перегруженных конструкторов. Этот загружает графический файл с диска. Если передать ему целые числа, указывающие желаемые ширину и высоту, результатом станет объект `Bitmap`, которому пока не сопоставлено никакого рисунка.

## Отображение объекта Bitmap

Загруженные в объекты `Bitmap` изображения форма показывает на экране при помощи следующего кода:

```
using (Graphics g = CreateGraphics()) {
    g.DrawImage(myBitmap, 30, 30, 150, 150);
```

} Метод `DrawImage()` использует объект `Bitmap` в качестве изображения, которое нужно нарисовать...

...начальные координаты X и Y...

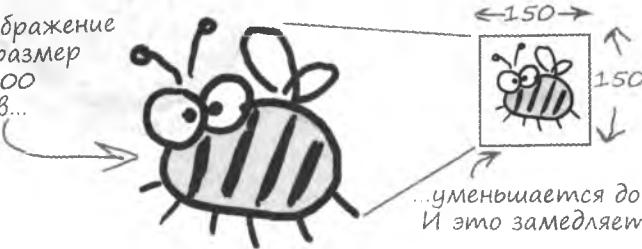
Эта строка заставляет объект `Graphics` рисовать на форме. Оператор `using` гарантирует удаление объекта `Graphics` после того, как в нем пропадет надобность.

...и размер, 150 x 150 пикселов.

## Чем они больше...

Вы обратили внимание на два последних параметра метода `DrawImage()`? Что, если изображение, помещаемое в объект `Bitmap`, имеет размер 175 x 175? Его требуется уменьшить до размера 150 x 150. А как быть, если объект `Bitmap` содержит изображение размером 1500 x 2025? Масштабирование произойдет намного медленнее....

Это изображение имеет размер 300 x 300 пикселов...



...уменьшается до размера 150 x 150 пикселов. И это замедляет работу симулятора!

**Масштабирование изображений требует большой производительности!** Ничего страшного, если эта процедура является однократной. Но когда ее приходится выполнять в **КАЖДОМ КАДРЕ**, работа программы замедляется. Именно со слишком большим размером загруженных вами изображений связана слишком медленная работа симулятора.

## Пространство имен System.Drawing

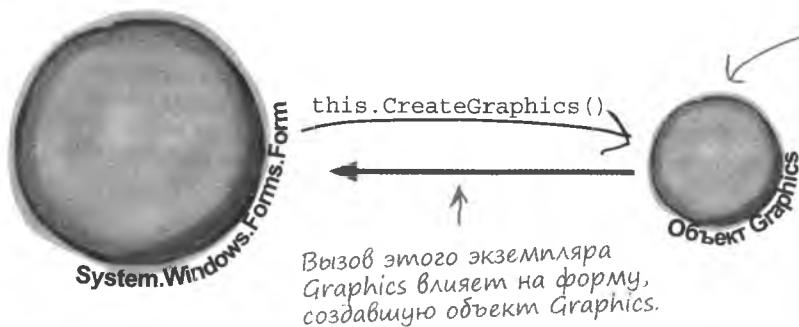
Объект `Graphics` принадлежит пространству имен `System.Drawing`. В .NET Framework имеется набор инструментов для работы с графикой, намного превосходящий по возможностям знакомый вам `PictureBox`. Вы можете рисовать формы, работать со шрифтами и сложной графикой... благодаря объекту `Graphics`. Он создается каждый раз, когда требуется добавить или отредактировать графический фрагмент. Затем вам остается воспользоваться его методами.

Графические методы в пространстве имен `System.Drawing` иногда называют GDI+, что означает `Graphics Device Interface` (интерфейс для представления графических объектов). Создавая с его помощью графику, вы начинаете с объекта `Graphics`, с которым связан объект `Bitmap`, форма, элемент управления или другой объект, который вы хотите отобразить.

1

### Объект, на котором вы собираете рисовать

Например, возьмем форму. Ее метод `CreateGraphics()` возвращает экземпляр `Graphics`, позволяющий рисовать на самом себе.



Метод `CreateGraphics()` может быть вызван формой, которой он принадлежит, или другим объектом. В обоих случаях он возвращает ссылку на объект `Graphics`, методы которого и создают рисунок.



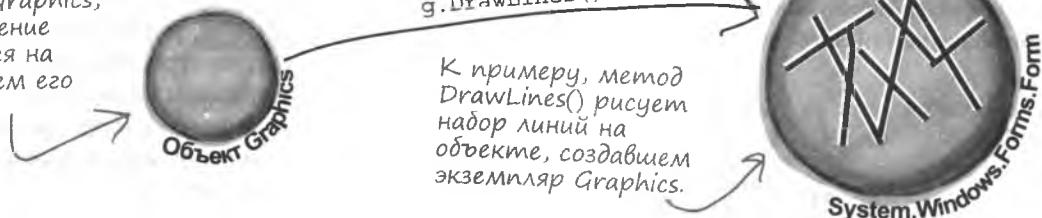
Вы ничего не рисуете на объекте `Graphics`. Он используется для рисования на других объектах.

2

### Методы объекта `Graphics`

Каждый объект `Graphics` снабжен методами, позволяющими рисовать на породившем его объекте. Создаваемые им фигуры и изображения появляются на форме.

Вы вызываете методы объекта `Graphics`, а изображение появляется на породившем его объекте.



## Знакомство с GDI+

Объект `Graphics` позволяет рисовать любые формы и изображения. Вы вызываете его методы, и на создавшем его объекте появляется нужный рисунок.

Для доступа к этим методам в верхней части файла класса должна располагаться строка `using System.Drawing;`. В противном случае при добавлении к проекту формы ИСР автоматически добавит эту строчку в класс этой формы.



- 1** Начнем с создания объекта `Graphics`. Используйте метод `CreateGraphics()` формы. Помните, что этот объект реализует интерфейс `IDisposable()`, поэтому при создании нового экземпляра пользуйтесь оператором `using`:

```
using (Graphics g = this.CreateGraphics()) {
```

Помните, что изображение появляется на объекте, создавшем этот экземпляр.

- 2** Чтобы нарисовать линию, вызовите метод `DrawLine()` и укажите координаты X и Y начальной и конечной точек:

```
g.DrawLine(Pens.Blue, 30, 10, 100, 45);
```

Координата начальной точки...

...координата конечной точки.

то же самое можно сделать при помощи класса `Point`:

```
g.DrawLine(Pens.Blue, new Point(30, 45), new Point(100, 10));
```

Вы можете выбирать цвета, достаточно ввести `<<Color>>`, `<<Pens>>` или `<<Brushes>>` поставить точку, и в окне IntelliSense появится перечень цветов.

- 3** Этот код рисует заливый серым цветом прямоугольник с голубой рамкой. Его размер задает объект `Rectangle`. В нашем случае верхний левый угол находится в точке (150, 15), ширина 140 пикселов, а высота 90 пикселов.

```
g.FillRectangle(Brushes.SlateGray, new Rectangle(150, 15, 140, 90));
g.DrawRectangle(Pens.SkyBlue, new Rectangle(150, 15, 140, 90));
```

- 4** Для рисования окружностей и эллипсов пользуйтесь методами `DrawCircle()` или `FillCircle()`. Размер формы задает объект `Rectangle`. Следующий код рисует сдвинутые друг относительно друга для создания эффекта тени эллипсы:

```
g.FillEllipse(Brushes.DarkGray, new Rectangle(45, 65, 200, 100));
g.FillEllipse(Brushes.Silver, new Rectangle(40, 60, 200, 100));
```

- 5** Метод `DrawString()` вводит текст. Но сначала вам нужно создать объект `Font`, определяющий шрифт. Этот объект реализует интерфейс `IDisposable`:

```
using (Font arial24Bold = new Font("Arial", 24, FontStyle.Bold)) {
    g.DrawString("Hi there!", arial24Bold, Brushes.Red, 50, 75);
}
```

Выполнив операторы по порядку, вы получите такую форму. Верхний левый угол имеет координату (0, 0).



На первом шаге мы создавали объект `Graphics`, поэтому на форме отсутствует цифра 1.

## Рисуем на форме

Создадим новое приложение Windows, которое рисует картинку после щелчка на форме.



### 1 Добавим к форме событие Click

На вкладке Events окна Properties найдите событие Click и дважды щелкните на нем.

Так как нам требуется объект Graphics, начните код обработчика события с оператора using. При работе с GDI+ вы используете объекты, реализующие интерфейс IDisposable. Если не удалить их, они будут потреблять ресурсы до выхода из программы. Поэтому вам потребуется множество операторов using:

```
using (Graphics g = CreateGraphics()) {
```

Это первая строчка метода обработчика события Form1\_Click(). Постепенно мы предоставим вам и остальные строчки.

### 2 Порядок рисования на форме

Нам потребуется голубой фон, поэтому начнем с голубого прямоугольника, все остальное будет появляться **поверх него**. Вы воспользуетесь свойством формы ClientRectangle. Это объект Rectangle, определяющий границы доступной для рисования области. Для создания этого объекта нужно указать координаты верхнего левого угла объекта Point, а также ширину и высоту. После этого появятся свойства Top, Left, Right и Bottom. И такой полезный метод как Contains(), возвращающий значение true, если точка попадает внутрь прямоугольника.

```
g.FillRectangle(Brushes.SkyBlue, ClientRectangle);
```

Чуть позже этот метод вам пригодится! Как вы думаете, что вам предстоит делать с методом Contains()?

### 3 Нарисуем цветок и пчелу

Вы уже знаете, как работает метод DrawImage().

```
g.DrawImage(Properties.Resources.Bee_animation_1, 50, 20, 75, 75);  
g.DrawImage(Properties.Resources.Flower, 10, 130, 100, 150);
```

Ручки предназначены для рисования линий и имеют толщину. Чтобы нарисовать заполненную форму или текст используется объект Brush.

### 4 Добавим ручку для рисования

Рисование линий осуществляется при помощи объекта Pen, задающего цвет и толщину. Встроенный класс Pens дает вам множество вариантов (например, Pens.Red – это тонкая красная ручка). Конструктор класса Pen позволяет создавать собственные ручки. Для рисования заливых форм применяются кисти. Класс Brushes предоставляет кисти различных цветов.

```
using (Pen thickBlackPen = new Pen(Brushes.Black, 3.0F)) {
```

Это мы добавим внутрь  
оператора using, созданного  
объектом Pen.

## 5 Добавим указывающую на цветок стрелку

Некоторые методы объекта Graphics берут массив объектов Point и соединяют их линиями или кривыми. Методом DrawLines() нарисуем наконечник стрелы, а методом DrawCurve() – ее основу. Есть и другие методы, работающие с массивами точек (например, метод DrawPolygon() рисует замкнутые формы, а метод FillPolygon() заполняет их).

```
g.DrawLines(thickBlackPen, new Point[] {  
    new Point(130, 110), new Point(120, 160), new Point(155, 163)});  
g.DrawCurve(thickBlackPen, new Point[] {  
    new Point(120, 160), new Point(175, 120), new Point(215, 70)});
```

}

Здесь заканчивается блок  
using, объект thickBlackPen  
удаляется, так как он нам  
больше не нужен.

На основе массива точек метод  
DrawCurve() рисует гладкую,  
соединяющую их друг с другом  
кривую.

## 6 Добавим шрифт

Первое, что требуется для написания текста, – создать объект Font. Снова напишем оператор using, так как он реализует интерфейс IDisposable. Создать шрифт просто. Существует несколько перегруженных конструкторов, самый простой использует имя шрифта, его размер и перечисление FontStyle.

```
using (Font font = new Font("Arial", 16, FontStyle.Italic)) {
```

## 7 Добавим текст «Nectar here»

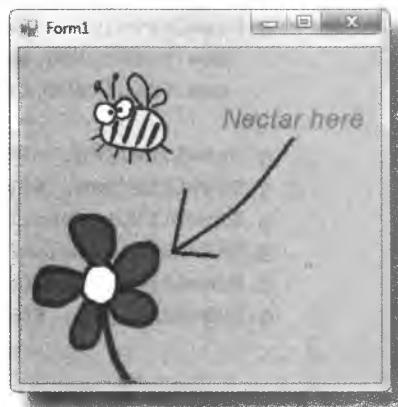
Определим место для строки, выяснив, какой размер она будет иметь на форме при помощи метода MeasureString(), возвращающего параметр SizeF. (SizeF – это версия float параметра Size). Так как мы знаем, где находится конец стрелки, поместим над ним центральную точку надписи.

```
SizeF size = g.MeasureString("Nectar here", font);  
g.DrawString("Nectar here", font, Brushes.Red, new Point(  
    215 - (int)size.Width / 2, 70 - (int)size.Height));  
}  
}
```

}

Не забудьте закрыть оба блока using.

Для создания объекта Rectangle требуется  
точка и параметр Size (то есть ширина  
и высота). Затем вы можете опреде-  
лить его границы и проверить методом  
Contains(), есть ли внутри объект Point.



как это выглядит?

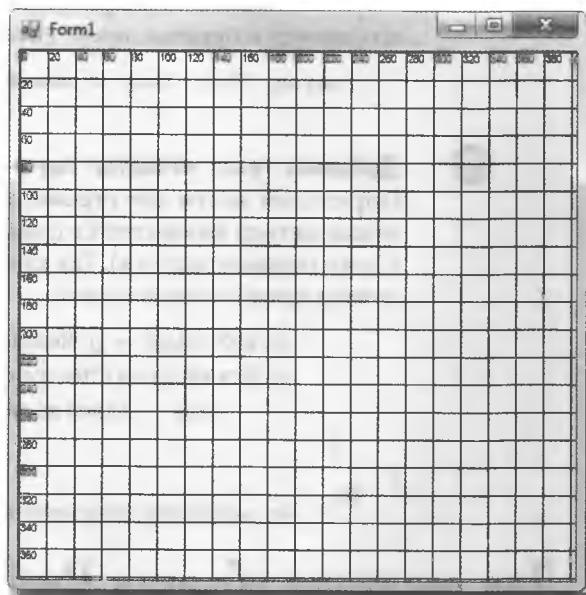
## Возьми в руку карандаш

1. Работа с объектами **Graphics** включает в себя представление ваших объектов в координатах X и Y. Вот код для построения показанной снизу сетки; вам нужно заполнить отсутствующие части.

```
using (Graphics g = this.CreateGraphics())
    using (Font f = new Font("Arial", 6, FontStyle.Regular)) {
        for (int x = 0; x < this.Width; x += 20) {
            .....
        }
        for (int y = 0; y < this.Height; y += 20) {
            .....
        }
    }
```

2. Что произойдет при запуске приведенного ниже кода? Нарисуйте результат на форме, используя для размещения отдельных точек только что визуализированную сетку.

```
using (Pen pen =
    new Pen(Brushes.Black, 3.0F)) {
    g.DrawCurve(pen, new Point[] {
        new Point(80, 60),
        new Point(200, 40),
        new Point(180, 60),
        new Point(300, 40),
    });
    g.DrawCurve(pen, new Point[] {
        new Point(300, 180), new Point(180, 200),
        new Point(200, 180), new Point(80, 200),
    });
    g.DrawLine(pen, 300, 40, 300, 180);
    g.DrawLine(pen, 80, 60, 80, 200);
    g.DrawEllipse(pen, 40, 40, 20, 20);
    g.DrawRectangle(pen, 40, 60, 20, 300);
    g.DrawLine(pen, 60, 60, 80, 60);
    g.DrawLine(pen, 60, 200, 80, 200);
}
```



3. Этот код работает с неправильными формами.

Определите, какой рисунок создает этот код,  
и воспроизведите его на сетке.

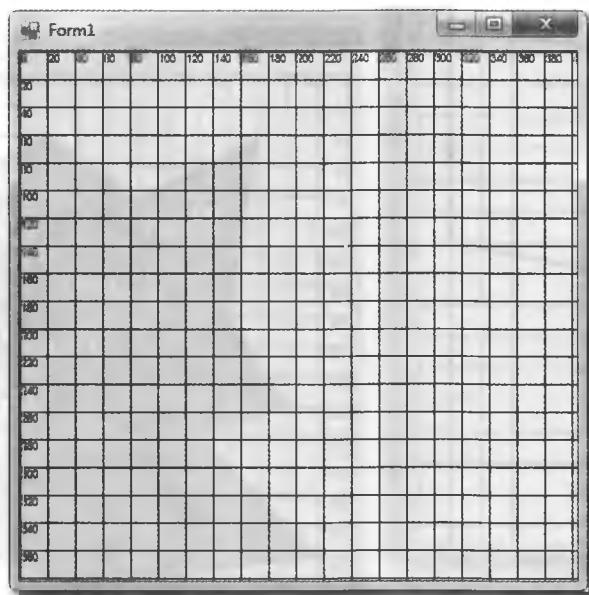
Конструктор методов FillPolygon()  
и DrawLines() берет массив объектов  
Point и рисует вершины, которые  
затем соединяются линиями.

```

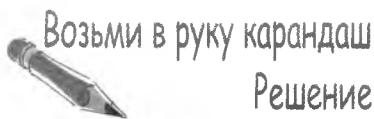
g.FillPolygon(Brushes.Black, new Point[] {
    new Point(60,40), new Point(140,80), new Point(200,40),
    new Point(300,80), new Point(380,60), new Point(340,140),
    new Point(320,180), new Point(380,240), new Point(320,300),
    new Point(340,340), new Point(240,320), new Point(180,340),
    new Point(20,320), new Point(60, 280), new Point(100, 240),
    new Point(40, 220), new Point(80,160),
});

using (Font big = new Font("Times New Roman", 24, FontStyle.Italic)) {
    g.DrawString("Pow!", big, Brushes.White, new Point(80, 80));
    g.DrawString("Pow!", big, Brushes.White, new Point(120, 120));
    g.DrawString("Pow!", big, Brushes.White, new Point(160, 160));
    g.DrawString("Pow!", big, Brushes.White, new Point(200, 200));
    g.DrawString("Pow!", big, Brushes.White, new Point(240, 240));
}

```



выглядит прекрасно, но...



## Решение

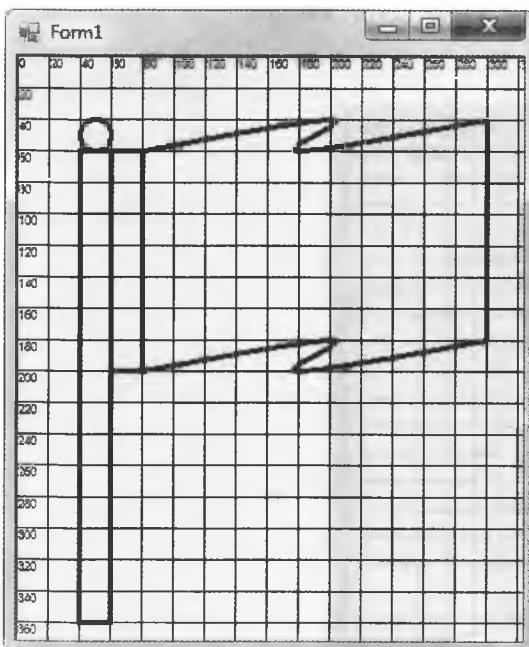
Вам нужно было вписать недостающий код и нарисовать фигуры, создаваемые двумя фрагментами кода.

```
using (Graphics g = this.CreateGraphics())
using (Font f = new Font("Arial", 6, FontStyle.Regular)) {
    for (int x = 0; x < this.Width; x += 20) {
        g.DrawLine(Pens.Black, x, 0, x, this.Height);
        g.DrawString(x.ToString(), f, Brushes.Black, x, 0);
    }
    for (int y = 0; y < this.Height; y += 20) {
        g.DrawLine(Pens.Black, 0, y, this.Width, y);
        g.DrawString(y.ToString(), f, Brushes.Black, 0, y);
    }
}
```

Сначала мы нарисовали вертикальные линии и числа вдоль оси Y. Вертикальные линии располагаются вдоль оси X через каждые 20 пикселов.

Мы использовали операторы `using`, гарантировав тем самым, что объекты `Graphics` и `Font` будут уничтожены после завершения формы.

Затем мы нарисовали горизонтальные линии и числа вдоль оси X. Вы выбрали значение Y и начали линию с координаты (0, y) в левой части формы по направлению к координате (0, this.Width) в правой части формы.



Нарисованные нашим визуализатором перекрывающиеся пчелы имели странный вид.

## Решение проблемы с прозрачностью

Давайте устраним артефакты изображения в местах, где маленькие картинки перекрываются! Решить проблему поможет метод `DrawImage()`. Мы вернемся к нашему приложению Windows и отредактируем его таким образом, чтобы перекрывающиеся изображения не удаляли фрагменты друг друга.



- Добавьте метод `DrawBee()`, рисующий пчел на объектах `Graphics`. Он использует перегруженный конструктор `DrawImage()`, который определяет место и размер рисунка при помощи структуры `Rectangle`.

```
public void DrawBee(Graphics g, Rectangle rect) {  
    g.DrawImage(Properties.Resources.Bee_animation_1, rect);  
}
```

Перекрывающиеся изображения пчел выглядят намного лучше.

- Это новый обработчик события `Click`. Он располагает верхний левый угол улья вне формы, в точке `(-Width, -Height)`, распространяя его на двойную ширину и высоту формы, и дает возможность менять ее размер. Затем методом `DrawBee()` добавляются четыре пчелы.

```
private void Form1_Click(object sender, EventArgs e) {  
    using (Graphics g = CreateGraphics()) {  
        g.DrawImage(Properties.Resources.Hive_inside_,  
                    -Width, -Height, Width * 2, Height * 2);  
        Size size = new Size(Width / 5, Height / 5);  
        DrawBee(g, new Rectangle(  
            new Point(Width / 2 - 50, Height / 2 - 40), size));  
        DrawBee(g, new Rectangle(  
            new Point(Width / 2 - 20, Height / 2 - 60), size));  
        DrawBee(g, new Rectangle(  
            new Point(Width / 2 - 80, Height / 2 - 30), size));  
        DrawBee(g, new Rectangle(  
            new Point(Width / 2 - 90, Height / 2 - 80), size));  
    }  
}
```

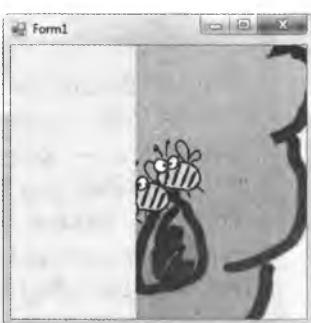
Сначала рисуется фон улья, который выходит за границы формы. Затем мы рисуем четырех перекрывающихся пчел, если они не перекрываются, увеличьте форму и снова щелкните на ней.



Но посмотрите, что произойдет, если перетащить пчел за границу формы и обратно!

### ...но остается проблема

- Запустите программу и щелкните на форме, чтобы нарисовать пчел. Но если перетащить форму за границу экрана, а потом вернуть, изображение исчезнет! Сделайте то же самое с картинкой «Nectar here», программу для которой вы написали несколько страниц назад, — та же проблема!



Как вы думаете, что происходит?

## Событие Paint

Наша графика исчезает, стоит закрыть ее, например, другим окном. К счастью, этот недочет можно исправить при помощи обработчика события **Paint**. Это событие вызывается при перерисовке формы. Одним из свойств параметра **PaintEventArgs** является объект **Graphics**, который собственно и рисует на элементах управления.

1

### Добавим обработчик события Paint

Дважды щелкните на строке **Paint** на вкладке **Events** окна **Properties**. Событие **Paint** возникает всякий раз, когда появляется необходимость перерисовать изображение.

Дважды щелкните на строке **Paint**, чтобы добавить к этому событию обработчик. Его параметр **PaintEventArgs** обладает свойством **Graphics**, поэтому все ваши нарисованное будет «приклеено» к форме.

2

### Воспользуйтесь объектом **Graphics**

В данном случае обработчик событий должен начинаться не с оператора **using**, а вот так:

```
private void Form1_Paint(object sender, PaintEventArgs e) {  
    Graphics g = e.Graphics;
```

Объект был создан не вами, поэтому вы не должны его уничтожать.

1

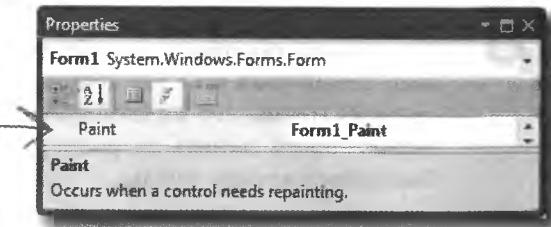
### Скопируйте код, рисующий перекрывающихся пчел и цветы

Добавьте метод **DrawBee()** с предыдущей страницы к новому пользовательскому элементу управления. Затем скопируйте код события **Click** в событие **Paint** кроме первой строчки с оператором **using**, ведь у вас уже есть объект **Graphics**, который называется **g**. Теперь запустите программу. Графика остается на месте!

Проделайте аналогичную операцию для рисунка «Nectar here».

Скорее всего, вы не обращали на это внимание, но формам все время приходится себя перерисовывать. Все элементы управления на формах являются с **X**. Каждый раз, когда форма оказывается за границами экрана или под другой формой, часть, которая была скрыта, становится НЕДЕЙСТВИТЕЛЬНОЙ, то есть перестает отображать графику. В этот момент .NET инициирует перерисовку формы, вызвав событие **Paint**. Вы можете вручную инициировать перерисовку форм и элементов управления, вызвав метод **Invalidate()**.

С формами и элементами управления связано событие **Paint**, дающее доступ к объекту **Graphics**, который автоматически перерисовывает все изображения.





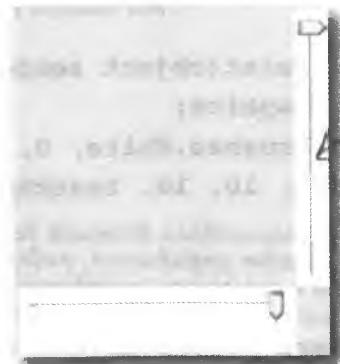
Используйте полученные знания о формах и элементах управления, чтобы попрактиковаться в использовании объектов `Bitmap` и метода `DrawImage()`. Создайте пользовательский элемент управления, который с помощью элемента `TrackBars` меняет масштаб изображения.

1

### Добавим два элемента `TrackBar`

Откройте новый проект Windows Application. Добавьте элемент `UserControl` – присвойте ему имя `Zoomer`, а свойству `Size` – значение `(300, 300)`. Создайте два элемента `TrackBar` – первый внизу, а второй справа. Свойству `Orientation` второго элемента присвойте значение `Vertical`. Свойство `Minimum` обоих элементов должно равняться `1`, `Maximum` – `175`, `Value` – `175`, а `TickStyle` – `None`. Для фона выберите белый цвет. Добавьте элементам обработчик события `Scroll` и заставьте оба обработчика вызывать метод `Invalidate()`.

Ваш пользовательский элемент управления связан с событием `Paint` и работает совершенно аналогично элементу, который вы только что применяли в форме. Воспользуйтесь параметром в свойства `PaintEventArgs`. С ним связано свойство `Graphics`, и все нарисованное при помощи объекта `Graphics` окажется на экземпляре элемента управления, который вы перетащили из окна `Toolbox`.

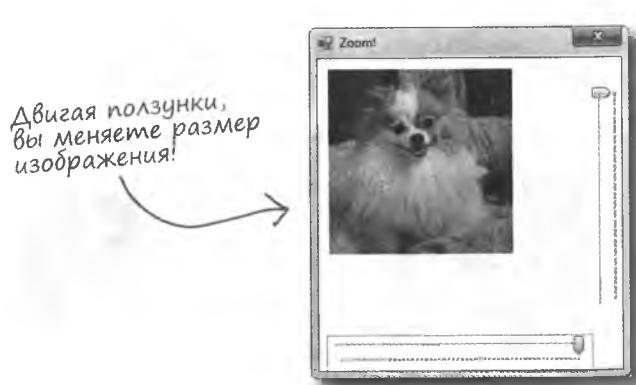


Поместите ползунки на белый фон. В этом случае они сольются с белым прямоугольником, который вы нарисуете в качестве фонового изображения.

2

### Загрузите изображение для элемента управления в объект `Bitmap`

Добавьте поле `photo` объекта `Bitmap` к элементу `Zoomer`. Используйте конструктор экземпляра `Bitmap` для загрузки любимого изображения. Добавьте к элементу событие `Paint`. Обработчик сформирует графический объект, заполнит фон белым цветом и затем с помощью метода `DrawImage()` нарисует содержимое поля `photo`, поместив его верхний левый угол в точку `(10, 10)`. Его ширина определяется параметром `trackBar1.Value`, а высота – `trackBar2.Value`. Перетащите элемент на форму.



Когда вы перемещаете ползунки, элементы `TrackBar` вызывают метод `Invalidate()`. Ваш элемент управления вызывает событие `Paint` и меняет размер фотографии.



Еще немного попрактикуемся в использовании объектов Bitmap и метода DrawImage(). Построим форму, которая загружает изображение из файла, а потом меняет его масштаб.

```
public partial class Zoomer : UserControl {
```

Конструктор Bitmap загружает изображение из файла. Он связан с другими переопределеными конструкторами, включая том, который позволяет вам указать ширину и высоту. Конструктор создает пустой объект bitmap.

```
    Bitmap photo = new Bitmap(@"c:\Graphics\fluffy_dog.jpg");
```

Выберите собственный файл — конструктор Bitmap работает с множеством форматов. Попробуйте воспользоваться методом OpenFileDialog для выбора подходящего изображения.

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

```
    private void Zoomer_Paint(object sender, PaintEventArgs e) {
        Graphics g = e.Graphics;
        g.FillRectangle(Brushes.White, 0, 0, Width, Height);
        g.DrawImage(photo, 10, 10, trackBar1.Value, trackBar2.Value);
    }
```

Мы нарисовали большой белый прямоугольник, заполнивши весь элемент управления, поверх него было добавлено фото. Последние два параметра определяют размер изображения: trackBar1 задает ширину, а trackBar2 — высоту.

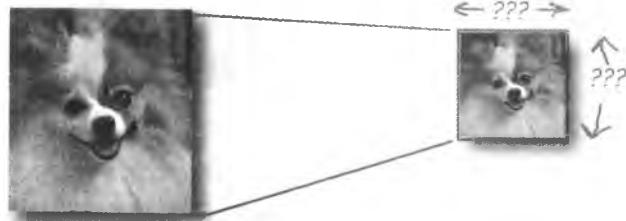
```
    private void trackBar1_Scroll(object sender, EventArgs e) {
        Invalidate();
    }
    private void trackBar2_Scroll(object sender, EventArgs e) {
        Invalidate();
    }
```

Каждый раз, когда пользователь начинает перемещать ползунок, вызывается событие Scroll. Обработчики этого события вызывают метод Invalidate() элемента управления, что приводит к перерисовке содержимого формы... при этом новая копия изображения имеет другой размер.



Перемаскивание ползунка вызывает изменение размера изображения при помощи метода DrawImage().

```
g.DrawImage(myBitmap, 30, 30, 150, 150);
```



# Перерисовка форм и элементов управления



Как уже было сказано, начав работать с объектами `Graphics`, вы получили контроль над графикой. Вы как бы говорите .NET «Эй, я отвечаю за свои действия». А что делать, если вы не хотите перерисовки формы при ее свертывании и развертывании... или наоборот, хотите, чтобы форма обновлялась чаще? Разобравшись, что происходит при перерисовке, вы сможете управлять процессом.

## 1 Каждая форма связана с событием Paint

Откройте список событий для любой формы, и вы обнаружите в нем событие `Paint`. Оно вызывается, если нужно перерисовать форму. Но каким образом? При помощи метода `OnPaint`, унаследованного от класса `Control`. (`On` в начале имени метода означает, что метод вызывает событие.) Перекройте метод `OnPaint`:

Перекройте  
метод  
`OnPaint` для  
любой формы  
и добавьте  
этую строку.

```
protected override void OnPaint (PaintEventArgs e) {
    →Console.WriteLine("OnPaint {0} {1}", DateTime.Now, e.ClipRectangle);
    base.OnPaint (e);
}
```

Вы уже проделывали такую операцию для метода `Dispose()`.

Подвигайте форму за границы экрана, сверните ее, спрячьте под другим окном и посмотрите на окно вывода. Вы увидите, что метод `OnPaint` вызывает событие `Paint` каждый раз, когда форма становится недействительной и нуждается в перерисовке. Параметр `ClipRectangle` – это прямоугольник, описывающий ставшую недействительной часть формы. Он передается свойству `PaintEventArgs` события `Paint` и увеличивает производительность, так как перерисовке подвергается только та часть формы, которой это действительно нужно.

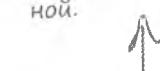
## 2 Используйте метод `Invalidate()`

Если скрыть, а потом снова сделать видимой часть формы, .NET вызывает событие `Paint`, которое вызывает метод `Invalidate()`, и передает ему параметр `Rectangle`, указывающий, какая часть формы должна быть перерисована. После чего .NET вызывает метод `OnPaint`, инициирующий событие `Paint` формы, и недействительная область перерисовывается.

Метод `Invalidate()`  
по сути объявляет  
часть формы  
недействитель-  
ной.



Вызвав этот метод,  
вы фактически гово-  
рите .NET, что ваша  
форма или элемент  
управления недей-  
ствительны и должны  
быть перерисованы.  
Методу можно пере-  
дать прямоугольную  
область – она будет  
передана в параметру  
`PaintEventArgs` собы-  
тия `Paint`.



## 3 Метод `Update()` помещает запрос `Invalidate` наверх

Форма все время получает сообщения. В системе, вызывающей метод `OnPaint`, когда поверх формы оказывается другой объект, существуют и другие сообщения. Напечатайте `override` и посмотрите список методов, начинающихся с `On`, каждый из них сообщает что-то форме. Метод `Update()` помещает сообщение метода `Invalidate` в самый верх списка.

## 4 Метод `Refresh()` как сумма методов `Invalidate()` и `Update()`

Формы и элементы управления обладают методом `Refresh()`, который сначала вызывает метод `Invalidate()`, объявляющий недействительной всю занятую графикой область, а затем `Update()`, гарантирующий сообщению метода `Invalidate` самый высокий приоритет.

часто  
Задаваемые  
Вопросы

**В:** Мне кажется, что менять размер изображения лучше в Paint или Photoshop. Я неправ?

**О:** Вы можете так поступить, при условии, что именно вы контролируете изображение и что оно больше не будет менять размер. Но в большинстве случаев вы получаете графический фрагмент из стороннего источника, например, от коллеги-дизайнера. В этом случае его размер приходится менять внутри кода.

**В:** Но не лучше ли менять размер изображения вне .NET?

**О:** Да, если вы уверены, что вам никогда не потребуется рисунок большего размера. Уменьшать изображение намного проще, чем увеличивать его. В большинстве случаев лучше менять размер графического фрагмента программными средствами. В этом случае вам не придется сталкиваться с ограничениями, накладываемыми внешними программами, например, получая файлы, предназначенные только для чтения.

**В:** Метод `CreateGraphics()` использует объект `Graphics` для рисования на форме, а зачем метод `FromImage()` вызывает метод `ResizeImage()`?

**О:** `FromImage()` восстанавливает объект `Graphics` для объекта `Bitmap`. И так же как вызванный для формы метод `CreateGraphics()`, возвращает рисующий на этой форме объект `Graphics`, метод `FromImage()` восстанавливает объект `Graphics` для рисования на объекте `Bitmap`.

**В:** То есть объект `Graphics` умеет не только рисовать на форме?

**О:** Этот объект рисует на любой структуре, которая в состоянии его породить. Объект `Bitmap` дает вам объект `Graphics`, при помощи которого вы рисуете на невидимом фоне. Этот фон может располагаться не только на формах. Перетащите на форму кнопку, перейдите к коду и введите имя и точку. В окне IntelliSense вы увидите метод `CreateGraphics()`, возвращающий объект `Graphics`. Все, что вы нарисуете на этом объекте, окажется на кнопке! Аналогично для элементов `Label`, `PictureBox`, `StatusStrip`... практически все элементы в окне `Toolbox` обладают объектом `Graphics`.

**В:** Я думал, что оператор `using` используется при работе с потоками. Зачем он нужен при работе с графикой?

**О:** Ключевое слово `using` применяется не только для потоков, но и для любого класса, реализующего интерфейс `IDisposable`. Создав экземпляр такого класса, вы должны вызвать метод `Dispose()`, как только работа с объектом завершится. В случае потоков метод `Dispose()` гарантирует закрытие всех открытых файлов.

Объекты `Graphics`, `Pen` и `Brush` также подлежат удалению. Они требуют места в памяти и других ресурсов и не всегда освобождают их. Когда вы рисуете что-то один раз, это не имеет особого значения. Но в большинстве случаев код, работающий с графикой, вызывается снова и снова, например обработчиком события `Paint` он может вызываться много раз в секунду. В таких случаях не обойтись

без метода `Dispose()`. Его использование гарантирует оператор `using`. В итоге высвобождение ресурсов осуществляется средствами .NET. Для всех объектов, созданных при помощи оператора `using`, метод `Dispose()` автоматически вызывается в конце блока. Именно это гарантирует, что ваша программа по мере работы не начнет занимать в памяти все больше и больше места.

**В:** Для создания элементов управления лучше пользоваться классом `UserControl` или наследованием от одного из встроенных элементов `Toolbox`?

**О:** Это зависит от назначения вашего элемента управления. Если его функции сходны с функцией уже существующего в `Toolbox` элемента, используйте наследование. Но в большинстве случаев приходится иметь дело с полностью пользовательскими элементами управления. Зато вы можете **перетаскивать на них элементы `Toolbox`**, созданный вами элемент прекрасно справляется с ролью контейнера.

**Пользовательский  
элемент управле-  
ния может служить  
контейнером для  
других элементов.**

Я заметил какое-то мерцание вокруг моего элемента управления Zoomer. Вы столько говорили о контроле над графикой, что я уверен, вы знаете, как решить эту маленькую проблему.

**Рисование изображения на форме занимает время, даже если размер изображения не меняется.**

А представьте, что в симуляторе меняется размер каждого изображения. Рисование всех этих пчел, цветов и самого улья требует времени. И когда вы видите конец процесса визуализации, вы воспринимаете его как мерцание.

Проблема состоит в слишком большом объеме рисуемой графики. Такая же проблема встречается и в некоторых любительских компьютерных играх.



### МОЗГОВОЙ ШТУРМ

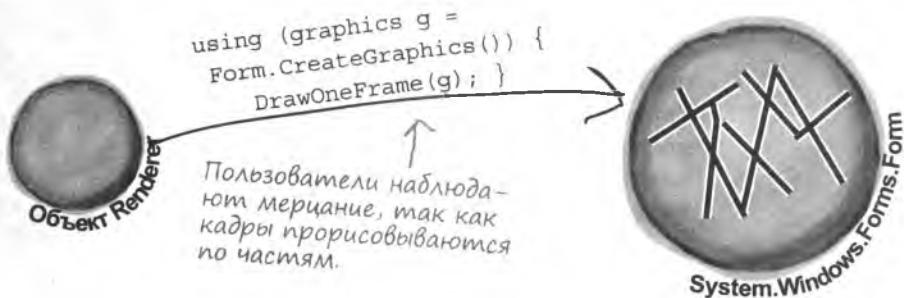
Как избавиться от этого мерцания? Что делать, если вы должны нарисовать на форме множество графических фрагментов? Ведь именно это является причиной.

**Отмасштабированная графика будет выглядеть лучше, если перед вызовом метода DrawImage () объекта Graphics свойству InterpolationMode этого объекта присвоить значение InterpolationMode.HighQualityBicubic. (В верхнюю часть кода при этом нужно добавить строку using System.Drawing.2D;)**

## Двойная буферизация

Вернитесь к масштабируемому изображению и подвигайте ползунки. Обратили внимание на появляющееся мерцание? Дело в том, что при каждом перемещении ползунка обработчик события Paint рисует белый прямоугольник, а уже на нем — изображение. Эта процедура происходит несколько раз в секунду, и именно она интерпретируется человеческим глазом как мерцание. Избежать мерцания можно с помощью **двойной буферизации (double buffering)**. Каждый кадр или ячейка анимации сначала рисуются в буфер, и новый кадр отображается только после его полной прорисовки. Рассмотрим принцип работы этой техники на примере объекта Bitmap.

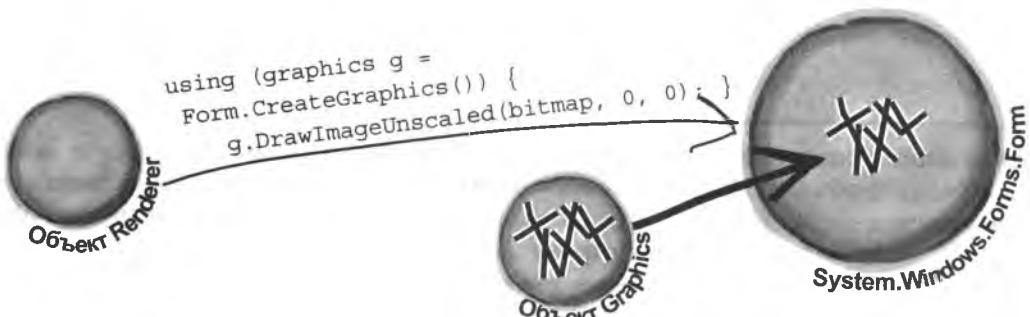
- 1 Вот типичная программа, рисующая на форме при помощи объекта Graphics.



- 2 Добавим в программу объект Bitmap, который сыграет роль буфера. Теперь именно на нем, а не непосредственно на форме будут рисоваться графические фрагменты.



- 3 Метод DrawImageUnscaled() скопирует рисунок с объекта Bitmap на объект Graphics формы. Копирование произойдет мгновенно, что избавит нас от проблемы мерцания.



## Двойная буферизация встроена в формы и элементы управления

Включить двойную буферизацию можно вручную с помощью объекта `Bitmap`, но в C# и .NET существует встроенная поддержка этой функции. Достаточно присвоить свойству `DoubleBuffered` значение `true`. Выделите ваш элемент управления `Zoomer`, в окне Properties присвойте свойству `DoubleBuffered` значение `true`, и мерцание прекратится! Проделайте это и для элемента `BeeControl`. Всех проблем это не решит, но разницу вы увидите сразу.

*Для решения проблем с графикой в нашем симуляторе все готово!*

### Капитальный ремонт симулятора

В следующем упражнении вы полностью перестройте симулятор. Наверное, имеет смысл создать новый проект и воспользоваться командой `Add >> Existing Item...` для добавления всех необходимых файлов. (Не забудьте отредактировать пространство имен!)

Вот что мы будем делать:

1

#### Удалим пользовательский элемент управления `BeeControl`

На поле и в улье не должно оставаться элементов управления. Пчел, цветы и улей мы нарисуем средствами GDI+. Поэтому щелкните правой кнопкой мыши на строчке `BeeControl.cs` в окне Solution Explorer и выберите команду `Delete`.

2

#### Для управления пчелиными крыльями вам потребуется таймер

Пчелы двигают крыльями намного медленнее, чем обновляются кадры симулятора, поэтому вам потребуется более медленный счетчик. Это неудивительно, ведь элемент `BeeControl` был снабжен для этой цели встроенным таймером.

3

#### Реконструкция визуализатора

Вам больше не понадобятся словари, так как элементов `PictureBox` и `BeeControl` уже нет. Вместо них фигурирует метод `DrawHive(g)`, рисующий форму `Hive` на объекте `Graphics`, и метод `DrawField(g)`, рисующий форму `Field`.

4

#### И напоследок, подключим новый визуализатор

Формам `Hive` и `Field` нужны обработчики события `Paint`. Каждый из них вызывает метод `DrawField(g)` или `DrawHive(g)` объекта `Renderer`. Таймеры же вызывают метод `Invalidate()`, обновляющий содержимое форм. При этом связанные с ними обработчики события `Paint` будут визуализировать кадр.

Начнем! →



Устраним проблемы с графикой. Воспользуемся объектами *Graphics* и двойной буферизацией.

1

### Отредактируем метод `RunFrame()` основной формы

Нужно убрать вызов метода `Renderer.Render()` и добавить два оператора `Invalidate()`.

```
public void RunFrame(object sender, EventArgs e) {  
    framesRun++;  
    world.Go(random);  
    end = DateTime.Now;  
    TimeSpan frameDuration = end - start;  
    start = end;  
    UpdateStats(frameDuration);  
    hiveForm.Invalidate(); ← Уберем вызов renderer.Render(),  
    fieldForm.Invalidate(); так как этот метод нам  
}                                большие не нужен.
```

Вы регулярно обновляете мир, а обе формы имеют ссылку на визуализатор, значит, нужно всего лишь анимировать их, вызывая соответствующие методы `Invalidate()`. Об остальном позаботятся их обработчики события `Paint`.

2

### Добавим к основной форме второй таймер

Присвойте свойству `Interval` нового таймера значение 150ms, а свойству `Enabled` – значение true. Двойным щелчком добавьте обработчик событий:

```
private void timer2_Tick(object sender, EventArgs e) {  
    renderer.AnimateBees();  
}
```

Добавьте метод `AnimateBees()`, управляющий движением крыльев пчел:

```
private int cell = 0;  
private int frame = 0;  
public void AnimateBees() {  
    frame++; ← Нужно задать значения поля  
    if (frame >= 6)           Cell, чтобы пользоваться им для  
    frame = 0;                рисования пчел в визуализаторе.  
    switch (frame) {          BeeAnimationLarge[Cell] должно  
        case 0: cell = 0; break;  отображаться в форме Hive,  
        case 1: cell = 1; break;  а BeeAnimationSmall[Cell] –  
        case 2: cell = 2; break;  в форме Field. Таймер будет  
        case 3: cell = 3; break;  все время вызывать метод  
        case 4: cell = 2; break;  AnimateBees(), меняющий  
        case 5: cell = 1; break;  состояние поля Cell. В результате  
        default: cell = 0; break;  крылья пчел начнут двигаться.  
    }  
    hiveForm.Invalidate();  
    fieldForm.Invalidate();  
}
```

Нужно задать значения поля `Cell`, чтобы пользоваться им для рисования пчел в визуализаторе. `BeeAnimationLarge[Cell]` должно отображаться в форме `Hive`, а `BeeAnimationSmall[Cell]` – в форме `Field`. Таймер будет все время вызывать метод `AnimateBees()`, меняющий состояние поля `Cell`. В результате крылья пчел начнут двигаться.

Если ваши пчелы летят не туда, проверьте правильность ввода координат! Для определения координат пользуйтесь событием `MouseClick` из предыдущей главы.

**3****Формам Hive и Field требуется открытое свойство Renderer**

Добавьте свойство общего доступа Renderer к обеим формам:

```
public Renderer Renderer { get; set; } ← Добавьте это свойство к обеим формам.
```

Не забывайте добавлять эти модификаторы для ступна!

**Отредактируйте объявление** класса Renderer следующим образом: public class Renderer. **То же самое** нужно сделать для классов World, Hive, Bee и Flower и для перечисления BeeState.

Экземпляр Renderer() создается в коде кнопки Open и в методе ResetSimulator(). Удалите все вызовы метода renderer.Reset(). Затем обновите конструктор объекта Renderer, чтобы задать свойство Renderer для обеих форм:

```
hiveForm.Renderer = this;  
fieldForm.Renderer = this;
```

Метод Reset() удалял с формы элементы управления, но теперь ему нечего удалять.

**4****Настройте двойную буферизацию форм Hive и Field**

Удалите из конструктора формы Hive код задания фонового изображения. Затем удалите все элементы управления из обеих форм и присвойте свойству DoubleBuffered значение true. Добавьте к формам обработчик события Paint. Он записан для формы Hive, для формы Field он отличается вызовом Renderer.PaintField() вместо Renderer.PaintHive():

```
private void HiveForm_Paint(object sender, PaintEventArgs e) {  
    Renderer.PaintHive(e.Graphics);  
}
```

Без двойной буферизации ваши формы будут мерцать!

**5****Удалим из визуализатора код на основе элементов управления и добавим графику**

Вот что вам нужно сделать:

- ★ Удалите словари. Также вам больше не понадобятся элемент BeeControl и методы Render(), DrawBees() и DrawFlowers().
- ★ Для хранения изображений добавьте поля Bitmap с именами HiveInside, HiveOutside и Flower. Создайте два массива Bitmap[] BeeAnimationLarge и BeeAnimationSmall. В каждом из них будут храниться по четыре изображения пчелы: большие 40x40 и маленькие 20x20. Создайте метод InitializeImages(), масштабирующий ресурсы, помещающий их в эти поля и вызывающий их из конструктора класса Renderer.
- ★ Добавьте метод PaintHive(), который берет в качестве параметра объект Graphics и рисует на нем улей. Начните с голубого прямоугольника, затем методом DrawImageUnscaled() нарисуйте улей изнутри, а методом DrawImageUnscaled() – находящихся внутри улья пчел.
- ★ Добавьте метод PaintField(), рисующий голубой прямоугольник в верхней части формы и зеленый – в нижней. Свойства ClientSize и ClientRectangle указут размер каждой области. Методом FillEllipse() нарисуйте желтое солнце, методом DrawLine() – ветку, с которой свисает улей, а методом DrawImageUnscaled() – вид улья снаружи. Нарисуйте цветы, а перед ними – пчел (используйте для этого маленькие картинки).
- ★ Помните, что метод AnimateBees() задает значение поля cell.



Избавимся от артефактов графики в нашем симуляторе улья. Используйте двойную буферизацию, чтобы сделать изображение четким.

## Решение

```
using System.Drawing;

public class Renderer {
    private World world;
    private HiveForm hiveForm;
    private FieldForm fieldForm;

    public Renderer(World TheWorld, HiveForm hiveForm, FieldForm fieldForm) {
        this.world = TheWorld;
        this.hiveForm = hiveForm;
        this.fieldForm = fieldForm;
        fieldForm.Renderer = this;
        hiveForm.Renderer = this;
        InitializeImages();
    }
}
```

```
public static Bitmap ResizeImage(Image ImageToResize, int Width, int Height) {
    Bitmap bitmap = new Bitmap(Width, Height);
    using (Graphics graphics = Graphics.FromImage(bitmap)) {
        graphics.DrawImage(ImageToResize, 0, 0, Width, Height);
    }
    return bitmap;
}
```

```
Bitmap HiveInside;
Bitmap HiveOutside;
Bitmap Flower;
Bitmap[] BeeAnimationSmall;
Bitmap[] BeeAnimationLarge;
private void InitializeImages() {
    HiveOutside = ResizeImage(Properties.Resources.Hive_outside_, 85, 100);
    Flower = ResizeImage(Properties.Resources.Flower, 75, 75);
    HiveInside = ResizeImage(Properties.Resources.Hive_inside_,
        hiveForm.ClientRectangle.Width, hiveForm.ClientRectangle.Height);
    BeeAnimationLarge = new Bitmap[4];
    BeeAnimationLarge[0] = ResizeImage(Properties.Resources.Bee_animation_1, 40, 40);
    BeeAnimationLarge[1] = ResizeImage(Properties.Resources.Bee_animation_2, 40, 40);
    BeeAnimationLarge[2] = ResizeImage(Properties.Resources.Bee_animation_3, 40, 40);
    BeeAnimationLarge[3] = ResizeImage(Properties.Resources.Bee_animation_4, 40, 40);
    BeeAnimationSmall = new Bitmap[4];
    BeeAnimationSmall[0] = ResizeImage(Properties.Resources.Bee_animation_1, 20, 20);
    BeeAnimationSmall[1] = ResizeImage(Properties.Resources.Bee_animation_2, 20, 20);
    BeeAnimationSmall[2] = ResizeImage(Properties.Resources.Bee_animation_3, 20, 20);
    BeeAnimationSmall[3] = ResizeImage(Properties.Resources.Bee_animation_4, 20, 20);
}
```

Это готовый класс Renderer, включающий метод AnimateBees(). Убедитесь, что вы отредактировали все три формы, особенное внимание уделите обработчикам события Paint для форм Hive и Field. Эти обработчики вызывают методы PaintHive() и PaintField() визуализатора, которые и выполняют всю анимацию.

\* Не забудьте внести изменения в файл Renderer.cs, объявив класс Renderer, как public. Проделайте аналогичную операцию для классов World, Hive, Flower и Bee, в противном случае вы получите ошибку построения.

Метод InitializeImages() масштабирует все изображения и сохраняет их в полях Bitmap объекта Renderer. В результате методы PaintHive() и PaintForm() получают возможность рисовать немасштабированные изображения при помощи методов DrawImageUnscaled() объекта Graphics формы.

```

public void PaintHive(Graphics g) {
    g.FillRectangle(Brushes.SkyBlue, hiveForm.ClientRectangle);
    g.DrawImageUnscaled(HiveInside, 0, 0);
    foreach (Bee bee in world.Bees) {
        if (bee.InsideHive)
            g.DrawImageUnscaled(BeeAnimationLarge[cell],
                bee.Location.X, bee.Location.Y);
    }
}

public void PaintField(Graphics g) {
    using (Pen brownPen = new Pen(Color.Brown, 6.0F)) {
        g.FillRectangle(Brushes.SkyBlue, 0, 0,
            fieldForm.ClientSize.Width, fieldForm.ClientSize.Height / 2);
        g.FillEllipse(Brushes.Yellow, new RectangleF(50, 15, 70, 70));
        g.FillRectangle(Brushes.Green, 0, fieldForm.ClientSize.Height / 2,
            fieldForm.ClientSize.Width, fieldForm.ClientSize.Height / 2);
        g.DrawLine(brownPen, new Point(593, 0), new Point(593, 30));
        g.DrawImageUnscaled(HiveOutside, 550, 20);
        foreach (Flower flower in world.Flowers) {
            g.DrawImageUnscaled(Flower, flower.Location.X, flower.Location.Y);
        }
        foreach (Bee bee in world.Bees) {
            if (!bee.InsideHive)
                g.DrawImageUnscaled(BeeAnimationSmall[cell],
                    bee.Location.X, bee.Location.Y);
        }
    }
}

private int cell = 0;
private int frame = 0;
public void AnimateBees() {
    frame++;
    if (frame >= 6)
        frame = 0;
    switch (frame) {
        case 0: cell = 0; break;
        case 1: cell = 1; break;
        case 2: cell = 2; break;
        case 3: cell = 3; break;
        case 4: cell = 2; break;
        case 5: cell = 1; break;
        default: cell = 0; break;
    }
    hiveForm.Invalidate();
    fieldForm.Invalidate();
}
}

```

Свойство формы ClientSize — это объект Rectangle, определяющий размер предназначенный для рисования области.

Метод PaintField() ищет пчел и цветы в классе World и по их координатам рисует поле. Сначала рисуется небо и земная поверхность, затем солнце, а после этого появляются цветы и улей. Напоследок появляются пчелы. Порядок рисования объектов имеет значение, например, если нарисовать пчел первыми, они окажутся за цветами.

Это уже знакомый вам метод AnimateBees(). Он циклически показывает изображения, используя для этого поле Frame, сначала демонстрируется cell 0, затем cell 1, затем 2, затем 3, и затем снова 2, и снова 1. В этом случае анимация пчелиных крыльев получится плавной.

## Вывод на печать

Методы объекта `Graphics`, которыми вы пользовались для рисования на форме, подходят и для вывода на печать. Инструменты .NET для этой процедуры находятся в пространстве имен `System.Drawing.Printing`. Вам нужно только создать объект `PrintDocument`. С ним связано событие `PrintPage`, которое применяется так же, как и событие `Tick` таймера. Затем требуется вызвать метод `Print()` этого объекта и распечатать документ. Вот как это выглядит.



- 1 Создайте приложение Windows и добавьте к форме кнопку. В верхнюю часть кода формы добавьте строку `using System.Drawing.Printing;` Дважды щелкните на кнопке и введите код. Вот что появится, когда вы введете `+=`:

```
private void button1_Click(object sender, EventArgs e) {  
    PrintDocument document = new PrintDocument();  
    document.PrintPage +=  
        new PrintPageEventHandler(document_PrintPage); (Press TAB to insert)
```

- 2 Нажмите Tab, и нужный код появится автоматически. Именно так вы добавляли обработчики событий в главе 11:

```
private void button1_Click(object sender, EventArgs e) {  
    PrintDocument document = new PrintDocument();  
    document.PrintPage += new PrintPageEventHandler(document_PrintPage);  
Press TAB to generate handler 'document_PrintPage' in this class
```

- 3 ИСР генерирует метод обработки событий и добавит его к форме.

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

Вместо исключения  
вы можете ввести  
**ЛЮБОЙ** код обработки  
графики... что именно  
напечатать в данном  
случае, вы увидите  
далее.

Параметр `e` свойства `PrintPageEventArgs` обладает свойством `Graphics`. Замените последнюю строку кодом, вызывающим методы объекта `e.Graphics`.

- 4 Завершите обработчик события `button1_Click`, вызвав метод `document.Print()`. При этом объект `PrintDocument` создает объект `Graphics` и, взяв его в качестве параметра, вызывает событие `PrintPage`. Все, что обработчик события рисует на объекте `Graphics`, передается на принтер.

```
private void button1_Click(object sender, EventArgs e) {  
    PrintDocument document = new PrintDocument();  
    document.PrintPage += new PrintPageEventHandler(document_PrintPage);  
    document.Print();  
}
```

## Окно предварительного просмотра и окно диалога Print

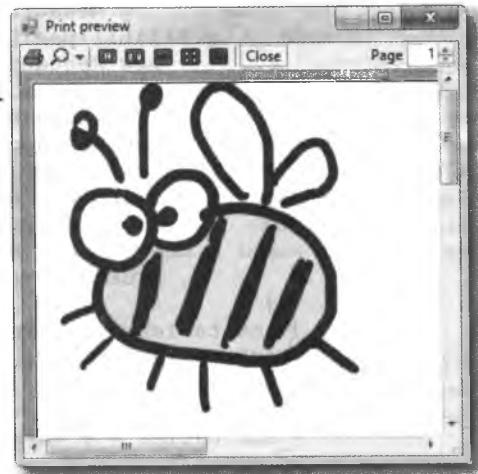
Добавление окна предварительного просмотра или окна диалога Print напоминает процедуру добавления окон диалога Open и Save. Вам нужно создать объект PrintDialog или PrintPreviewDialog, присвоить его свойству Document ваш объект Document и вызвать метод Show(). Документ на печать отправит окно диалога, вам не потребуется вызывать метод Print(). Добавим его к кнопке, созданной вами на шаге 1:

```
private void button1_Click(object sender, EventArgs e) {
    5 PrintDocument document = new PrintDocument();
    document.PrintPage += new PrintPageEventHandler(document_PrintPage);
    PrintPreviewDialog preview = new PrintPreviewDialog();
    preview.Document = document;
    preview.ShowDialog(this);
}

void document_PrintPage(object sender,
    PrintPageEventArgs e) {
    DrawBee(e.Graphics, new Rectangle(0, 0, 300, 300));
}
```

Мы воспользуемся уже написанным методом DrawBee().

При наличии объекта PrintDocument и соответствующего обработчика событий для вызова окна предварительного просмотра достаточно создать объект PrintPreviewDialog.



### Печать многостраничного документа

Чтобы распечатать несколько страниц, нужно присвоить свойству e.HasMorePages обработчика событий PrintPage значение true. В результате объект Document узнает о дополнительных страницах. Обработчик событий будет вызываться снова и снова, до тех пор пока он присваивает свойству e.HasMorePages значение true. Отредактируем его таким образом, чтобы распечатать две страницы:

```
bool firstPage = true;
void document_PrintPage(object sender, PrintPageEventArgs e) {
    6 DrawBee(e.Graphics, new Rectangle(0, 0, 300, 300));
    using (Font font = new Font("Arial", 36, FontStyle.Bold)) {
        if (firstPage) {
            e.Graphics.DrawString("Первая страница", font, Brushes.Black, 0, 0);
            e.HasMorePages = true;
            firstPage = false;
        } else {
            e.Graphics.DrawString("Вторая страница", font, Brushes.Black, 0, 0);
            firstPage = true;
        }
    }
}
```

Если присвоить свойству e.HasMorePages значение true, объект Document снова вызовет обработчик событий и отправит на печать следующую страницу.

Запустите программу и убедитесь, что в окне предварительного просмотра появляются две страницы.



Напишите код для кнопки Print, которая вызывает окно предварительного просмотра со статистикой пчел и изображениями улья и поля.

1

### Пусть кнопка вызывает окно предварительного просмотра

Добавьте к событию Click кнопки обработчик, прерывающий работу симулятора, он будет вызывать окно предварительного просмотра и после его закрытия снова запускать симулятор.

2

### Создайте обработчик события PrintPage

Он должен создавать показанное на следующей странице окно:

```
private void document_PrintPage(object sender, PrintPageEventArgs e) {
    Graphics g = e.Graphics;
    Size textSize;
    using (Font arial24bold = new Font("Arial", 24, FontStyle.Bold)) {
        textSize = Size.Ceiling(
            g.MeasureString("Bee Simulator", arial24bold));
        g.FillEllipse(Brushes.Gray,
            new Rectangle(e.MarginBounds.X + 2, e.MarginBounds.Y + 2,
            textSize.Width + 30, textSize.Height + 30));
        g.FillEllipse(Brushes.Black,
            new Rectangle(e.MarginBounds.X, e.MarginBounds.Y,
            textSize.Width + 30, textSize.Height + 30));
        g.DrawString("Bee Simulator", arial24bold,
            Brushes.Gray, e.MarginBounds.X + 17, e.MarginBounds.Y + 17);
        g.DrawString("Bee Simulator", arial24bold,
            Brushes.White, e.MarginBounds.X + 15, e.MarginBounds.Y + 15);
    }
    int tableX = e.MarginBounds.X + (int)textSize.Width + 50;
    int tableWidth = e.MarginBounds.X + e.MarginBounds.Width - tableX - 20;
    int firstColumnX = tableX + 2;
    int secondColumnX = tableX + (tableWidth / 2) + 5;
    int tableY = e.MarginBounds.Y;
    // Вам нужно написать остальную часть метода, распечатывающего окно
}
```

Овал с текстом был создан при помощи метода MeasureString(), возвращающего значение Size, то есть размер строки. Овал и текст были нарисованы два раза для создания эффекта тени.

Процедура построения таблицы.

3

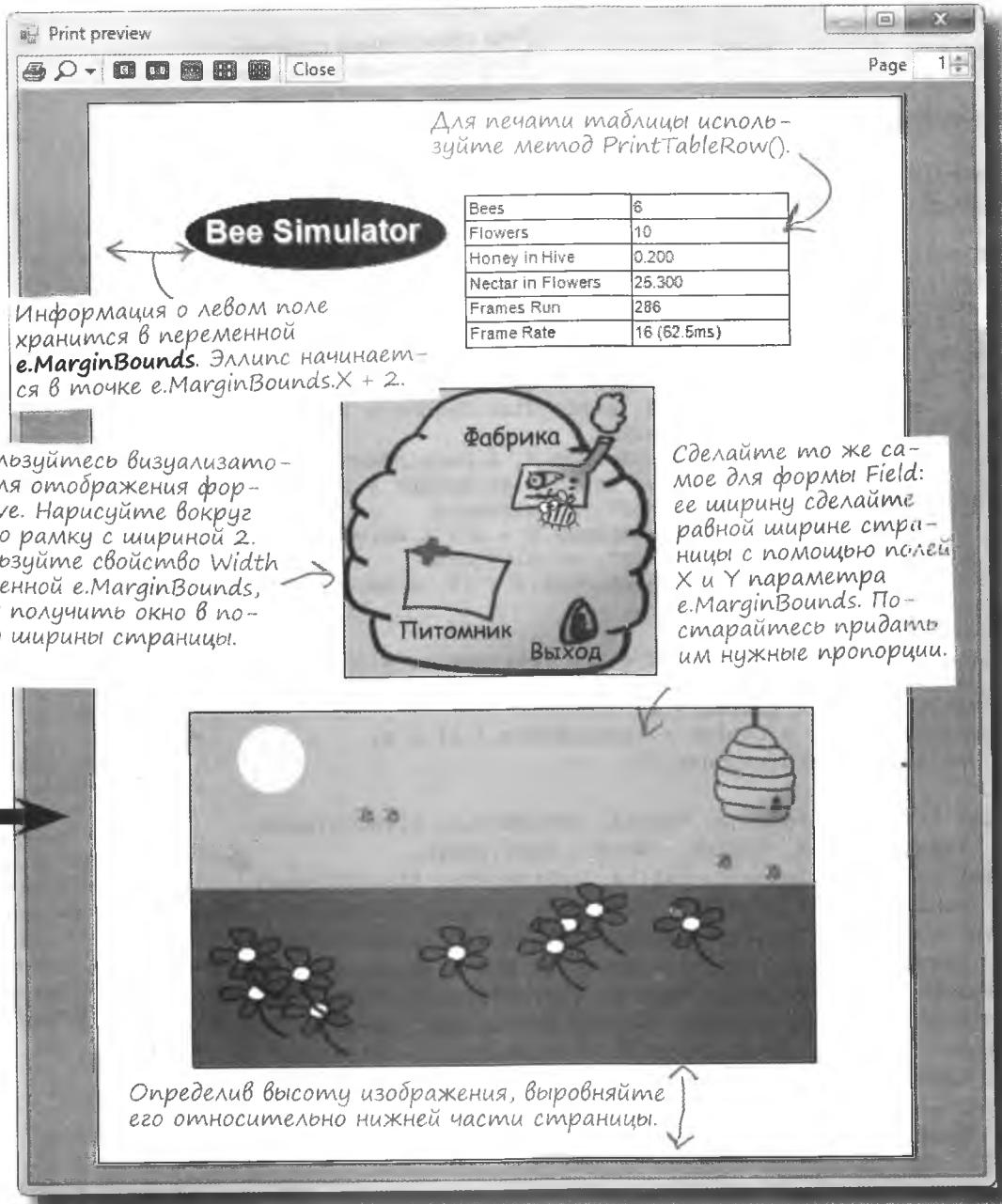
### Используйте метод PrintTableRow()

Этот метод пригодится для создания таблицы со статистикой в верхней части окна.

```
private int PrintTableRow(Graphics printGraphics, int tableX,
    int tableWidth, int firstColumnX, int secondColumnX,
    int tableY, string firstColumn, string secondColumn) {
    Font arial12 = new Font("Arial", 12);
    Size textSize = Size.Ceiling(printGraphics.MeasureString(firstColumn, arial12));
    tableY += 2;
    printGraphics.DrawString(firstColumn, arial12, Brushes.Black,
        firstColumnX, tableY);
    printGraphics.DrawString(secondColumn, arial12, Brushes.Black,
        secondColumnX, tableY);
    tableY += (int)textSize.Height + 2;
    printGraphics.DrawLine(Pens.Black, tableX, tableY, tableX + tableWidth, tableY);
    arial12.Dispose();
    return tableY;
}
```

При каждом вызове метод PrintTableRow() добавляем высоту распечатанной строки в переменную tableY и возвращаем новое значение.

Внимательно прочтайте примечания, которыми мы снабдили этот скриншот!



**Подсказка:** Чтобы вычислить высоту каждой формы в окне предварительного просмотра, умножьте соотношение высоты и ширины на итоговую ширину формы. Координата верхней части формы Field рассчитывается по формуле: (e.MarginBounds.Y + e.MarginBounds.Height - fieldHeight).

## решение упражнения



```
using System.Drawing.Printing;
```

```
private void document_PrintPage(object sender, PrintPageEventArgs e) {  
    Graphics g = e.Graphics;
```

```
    Size stringSize;  
    using (Font arial24bold = new Font("Arial", 24, FontStyle.Bold)) {  
        stringSize = Size.Ceiling(  
            g.MeasureString("Bee Simulator", arial24bold));  
        g.FillEllipse(Brushes.Gray,  
            new Rectangle(e.MarginBounds.X + 2, e.MarginBounds.Y + 2,  
            stringSize.Width + 30, stringSize.Height + 30));  
        g.FillEllipse(Brushes.Black,  
            new Rectangle(e.MarginBounds.X, e.MarginBounds.Y,  
            stringSize.Width + 30, stringSize.Height + 30));  
        g.DrawString("Bee Simulator", arial24bold,  
            Brushes.Gray, e.MarginBounds.X + 17, e.MarginBounds.Y + 17);  
        g.DrawString("Bee Simulator", arial24bold,  
            Brushes.White, e.MarginBounds.X + 15, e.MarginBounds.Y + 15);  
    }
```

```
    int tableX = e.MarginBounds.X + (int)stringSize.Width + 50;  
    int tableWidth = e.MarginBounds.X + e.MarginBounds.Width - tableX - 20;  
    int firstColumnX = tableX + 2;  
    int secondColumnX = tableX + (tableWidth / 2) + 5;  
    int tableY = e.MarginBounds.Y;
```

```
    tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,  
        secondColumnX, tableY, "Bees", Bees.Text);  
    tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,  
        secondColumnX, tableY, "Flowers", Flowers.Text);  
    tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,  
        secondColumnX, tableY, "Honey in Hive", HoneyInHive.Text);  
    tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,  
        secondColumnX, tableY, "Nectar in Flowers", NectarInFlowers.Text);  
    tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,  
        secondColumnX, tableY, "Frames Run", FramesRun.Text);  
    tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,  
        secondColumnX, tableY, "Frame Rate", FrameRate.Text);
```

```
    g.DrawRectangle(Pens.Black, tableX, e.MarginBounds.Y,  
        tableWidth, tableY - e.MarginBounds.Y);  
    g.DrawLine(Pens.Black, secondColumnX, e.MarginBounds.Y,  
        secondColumnX, tableY);
```

Это обработчик события  
PrintPage нашего документа.  
Он входит в код формы.

Эту часть  
кода мы вам  
уже давали.  
Здесь рисует-  
ся заголовок  
в овале и зада-  
ются параметры таблицы  
со статисти-  
кой.

Вы уже поняли, как  
работает метод  
PrintTableRow()? Вам  
нужно вызывать его  
для каждой строки,  
и требуемый текст  
будет распечаты-  
ваться в два столбца.  
Метод возвращает  
новое значение tableY  
для следующей строки.

Не забудьте нарисовать  
границу таблицы и линию,  
разделяющую столбцы.

Для рисования рамки вокруг снимков экрана Вам потребуется черная ручка с толщиной линии 2 пикселя.

Так как объекты Pen и Bitmap после применения нужно будет удалить, мы поместили их в блок using.

```
using (Pen blackPen = new Pen(Brushes.Black, 2))
using (Bitmap hiveBitmap = new Bitmap(hiveForm.ClientSize.Width,
                                         hiveForm.ClientSize.Height))
using (Bitmap fieldBitmap = new Bitmap(fieldForm.ClientSize.Width,
                                         fieldForm.ClientSize.Height))
{
    using (Graphics hiveGraphics = Graphics.FromImage(hiveBitmap))
    {
        renderer.PaintHive(hiveGraphics);
    }
    int hiveWidth = e.MarginBounds.Width / 2;
    float ratio = (float)hiveBitmap.Height / (float)hiveBitmap.Width;
    int hiveHeight = (int)(hiveWidth * ratio);
    int hiveX = e.MarginBounds.X + (e.MarginBounds.Width - hiveWidth) / 2;
    int hiveY = e.MarginBounds.Height / 3;
    g.DrawImage(hiveBitmap, hiveX, hiveY, hiveWidth, hiveHeight);
    g.DrawRectangle(blackPen, hiveX, hiveY, hiveWidth, hiveHeight);

    using (Graphics fieldGraphics = Graphics.FromImage(fieldBitmap))
    {
        renderer.PaintField(fieldGraphics);
    }
    int fieldWidth = e.MarginBounds.Width;
    float ratio = (float)fieldBitmap.Height / (float)fieldBitmap.Width;
    int fieldHeight = (int)(fieldWidth * ratio);
    int fieldX = e.MarginBounds.X;
    int fieldY = e.MarginBounds.Y + e.MarginBounds.Height - fieldHeight;
    g.DrawImage(fieldBitmap, fieldX, fieldY, fieldWidth, fieldHeight);
    g.DrawRectangle(blackPen, fieldX, fieldY, fieldWidth, fieldHeight);
}
}

private void printToolStripButton1_Click(object sender, EventArgs e)
{
    bool stoppedTimer = false;
    if (timer1.Enabled)
    {
        timer1.Stop();
        stoppedTimer = true;
    }
    PrintPreviewDialog preview = new PrintPreviewDialog();
    PrintDocument document = new PrintDocument();
    preview.Document = document;
    document.PrintPage += new PrintPageEventHandler(document_PrintPage);
    preview.ShowDialog(this);
    if (stoppedTimer)
        timer1.Start();
}
```

Размер объектов Bitmap должен совпадать с областью формы, предназначенной для рисования, поэтому Вам придется обект ClientSize.

Метод PaintHive() рисует при помощи объекта Graphics, поэтому код создает и передает ему пустой объект Bitmap.

e.MarginBounds.Width указывает ширину области, которая будет выведена на печать. Именно такую ширину должны иметь ваши снимки экрана.

Из соотношения высоты и ширины форм мы вычисляем высоту снимка экрана.

Код кнопки Print прерывает работу симулятора, создает объект PrintDocument, связывает его с обработчиком события PrintPage, вызывает окно диалога и возобновляет работу симулятора.

## А еще можно было бы...

Вы построили небольшой, но симпатичный симулятор, так зачем останавливаться? Вот идеи, которые при желании вы можете реализовать самостоятельно.

### Добавьте панель управления

Превратите константы классов World и Hive в свойства. Затем добавьте новую форму с панелью управления и ползунками.

### Создайте врагов

Пусть улей атакуют враги. Чем больше цветов, тем больше должно быть врагов. Вам потребуются Sting Patrol для борьбы с врагами и Hive Maintenance для восстановления улья. И все эти пчелы потребляют мед.

### Усовершенствуйте улей

При наличии достаточного количества меда улей растет. Чем больше улей, тем больше в нем пчел. Но при этом растет потребление меда и учащаются атаки врагов. При значительном ущербе улей снова должен стать меньше.

### Пусть пчелиная матка откладывает яйца

Для заботы о яйцах вам потребуются рабочие пчелы Baby Bee Care. Чем больше меда в улье, тем больше яиц откладывает матка. А значит, требуются дополнительные пчелы-няньки, которые едят мед.

### Добавьте анимацию

Анимируйте фоновое изображение формы Hive, заставив солнце двигаться по небу. Пусть ночью становится темно и появляются луна и звезды. Добавьте перспективу: пусть пчелы, сидящие на ближних цветах, имеют больший размер, чем пчелы, расположенные дальше.

### Используйте свое воображение!

Придумайте собственные способы сделать симулятор интерактивным и более интересным.

Хороший симулятор позволяет пользователю выбирать, каким способом пойдет развитие системы.

Вы создали удачную версию симулятора? Покажите ее нам, загрузите код на форму Head First C# [www.headfirstlabs.com/books/hfcsharp/](http://www.headfirstlabs.com/books/hfcsharp/)

КАПИТАН  
ВЕЛИКОЛЕПНЫЙ  
СМЕРТЬ  
ОБЪЕКТА

Head First

\$2.98    глава  
14



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 *Polyorphism*

POW!!

Я УНИЧТОЖУ ССЫЛКИ  
НА КЛОНЫ ОДНУ ЗА  
ОДНОЙ.



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); 1  
            captainAmazing.Think("Я уничтожу ссылки на клонов,  
одну за одной");  
            captainAmazing.IdentifyTheClones(Cloness);  
            captainAmazing.RemoveTheClones(Cloness);  
            swindler.Think("Через несколько минут моя армия станет мусором");  
            swindler.Think("будет собрана!");  
            escapePlane = new SwindlersEscapePlane(swindler); 2  
            swindler.TrapCaptainAmazing(Factory);  
            MessageBox.Show("Жулик убежал");  
        }  
    }  
}
```

Можно предположить, что клоны были созданы при помощи инициализатора коллекции.

Первый рисунок сделайте для этой точки. Покажите, что же происходит на фабрике.

[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);  
    }  
}
```

3 Как будет выглядеть куча после запуска конструктора FinalBattle (Финальная битва)?

Нарисуйте, что происходит в момент создания экземпляра объекта SwindlersEscapePlane (План побега Жулика).

```
...  
...  
}  
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 = 600;  
        factory.SelfDestruct.Start();  
    }  
    private void SelfDestruct_Tick(object sender, EventArgs e) {  
        finalBattle.Factory = null;  
    }  
}
```

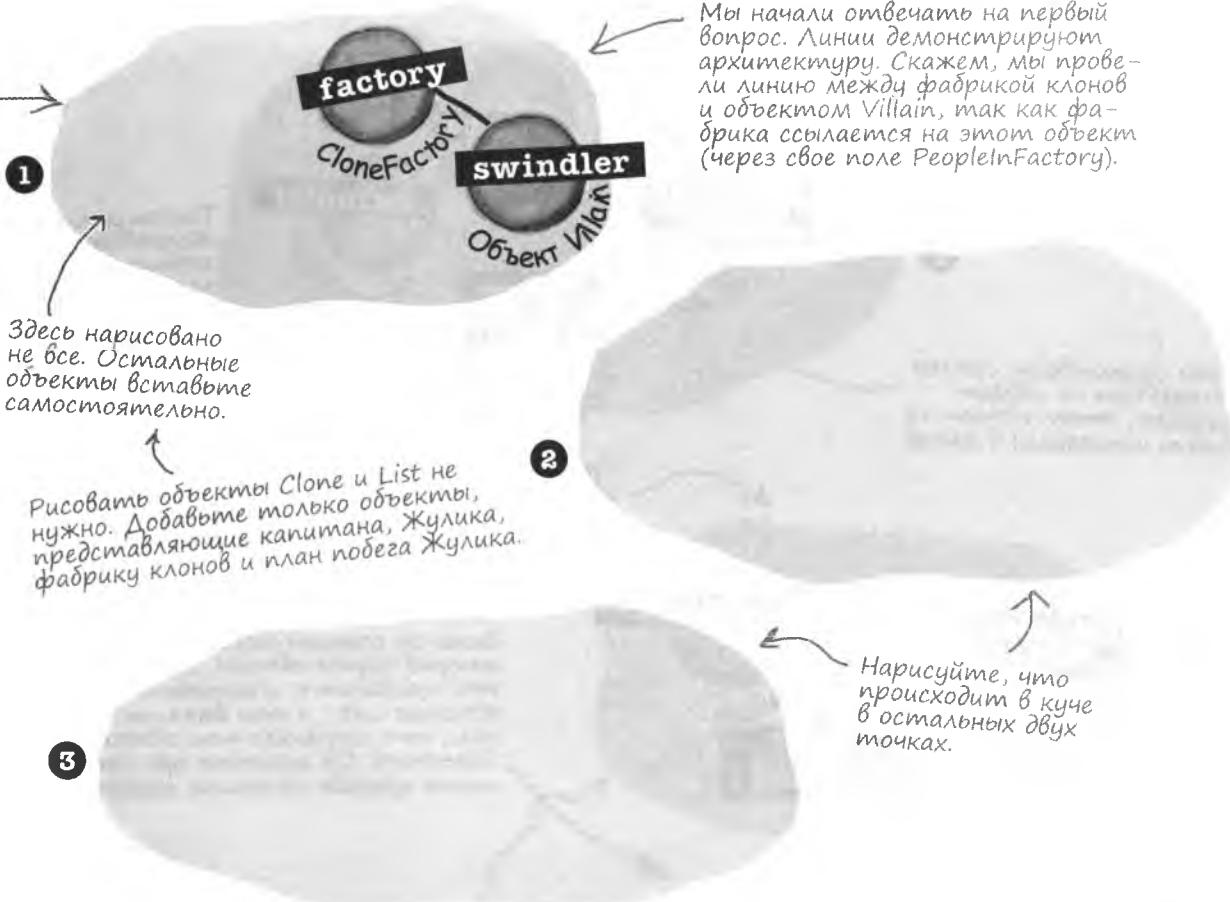
Здесь должен быть дополнительный код (включаящий метод Dispose(), реализующий интерфейс IDisposable). Но он скрыт, так как не имеет значения для решения задачи.

```
class SwindlersEscapePlane {
    public Villain PilotsSeat;
    public SwindlersEscapePlane(Villain escapee) {
        PilotsSeat = escapee;
    }
}
```

```
class CloneFactory {
    public Timer SelfDestruct = new Timer();
    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.



В каком месте кода умирает капитан Великолепный?

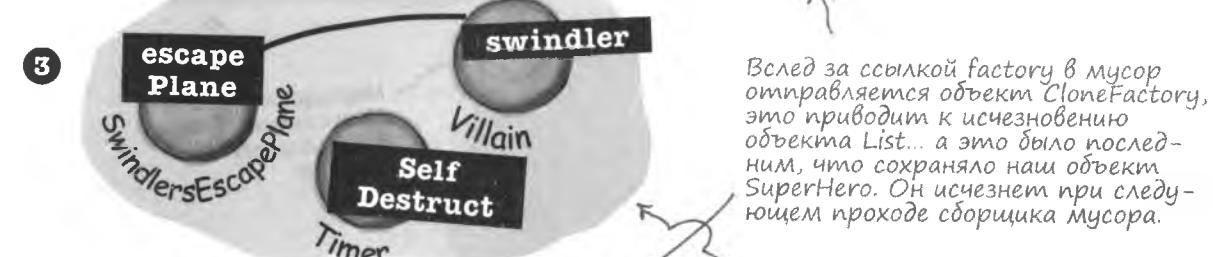
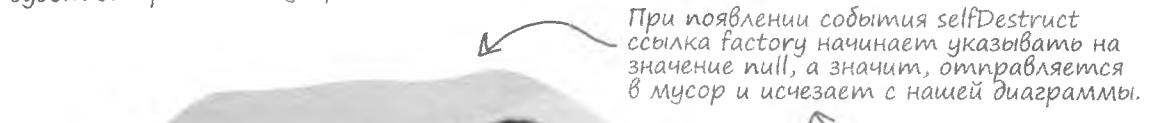
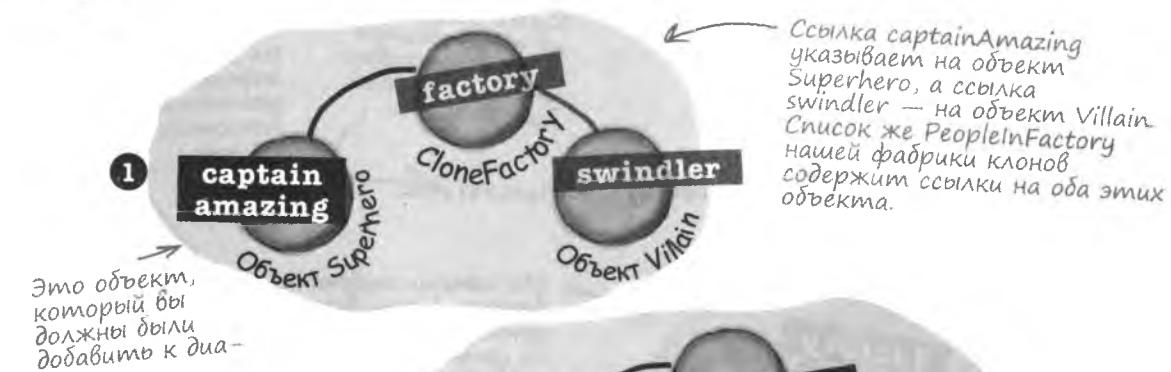
.....  
.....  
.....

Оставьте соответствующий комментарий на диаграмме.

что бы могли означать эти цифры

## Возьми в руку карандаш Решение

Вот как выглядит куча, по мере выполнения программы FinalBattle.



Вот момент исчезновения нашего супергероя:

```
void SelfDestruct_Tick(object sender, EventArgs e) {  
    finalBattle.factory = null;
```

Ссылка finalBattleFactory стала указывать на null, и это привело к исчезновению последней ссылки на капитана!

Герой исчезнет после запуска конструктора FinalBattle.

Так как экземпляр Superhero не имеет клонов, на которые мог бы сослаться объект factory, он предназначается для сборщика мусора.

это твое последнее слово?

## Метод завершения объекта

Иногда нужно, чтобы некие действия произошли *до того*, как объект отправится в мусор, например **высвобождение неуправляемых ресурсов**.

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 объ-  
екта Clone.

Знак ~ указывает, что код в  
этом блоке выполняется, когда  
объект отправляется в мусор.

Это метод завер-  
шения объекта. Он  
отправляет не-  
годяю 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 в **прямо перед отправкой** объекта в мусор.

Этот метод не имеет параметров, так как .NET должна уничтожить объект.

Вам не придется писать метод заверше-  
ния для объектов, обладающих управ-  
ляемыми ресурсами. Все, с чем вы  
сталкивались в этой книге было управ-  
ляемым — оно управлялось CLR. Но  
бывают случаи, когда программистам  
требуется доступ к базовым ресурсам  
Windows, которые не являются частью  
.NET Framework. Скажем, код с атрибу-  
том **[DllImport]** может использо-  
вать неуправляемые ресурсы. Если их  
вовремя не удалить (например, соот-  
ветствующим методом), эти ресурсы могут  
повлиять на стабильность системы.  
Именно для такой цели требуется метод  
завершения объекта.



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 вы-  
зывающий *MessageBox* в методе завершения объектов.  
Эти вещи затрагивают «сердце» CLR. Мы показываем их,  
чтобы объяснить принцип сборки мусора. **Никогда не ис-  
пользуйте их в рабочих программах.**

Позднее на панихиде...

ГРОБ КАПИТАНА  
ПУСТ...НО ЧТО ЭТО?



ПОХОЖЕ НА КАКОЙ-ТО  
СЕКРЕТНЫЙ КОД. НЕУЖЕЛИ  
ЭТО ПОСЛАНИЕ ОТ  
КАПИТАНА?

6e 61 6d 65 73 70 61 63 65 20 51 7b 0d 0a 5b 53  
65 72 69 61 6c 69 7a 61 62 6c 65 5d 70 75 62 6c  
69 63 20 63 6c 61 73 73 20 4d 73 67 7b 0d 0a 70  
75 62 6c 69 63 20 73 74 72 69 6e 67 20 61 3b 70  
75 62 6c 69 63 20 73 74 72 69 6e 67 20 62 3b 70  
75 62 6c 69 63 20 76 6f 69 64 20 53 68 6f 77 28 29  
62 6c 69 73 73 61 67 65 42 6f 78 2e 53 68 6f 77  
7b 4d 65 73 75 62 73 74 72 69 6e 67 28 31 2c 32  
28 63 2e 53 75 62 73 74 72 69 6e 67 28 31 2c 32  
29 2b 69 2b 22 40 22 2b 61 2b 63 2b 22 2e 22 2b  
62 29 3b 7d 7d 00 00 00 00 00 00 00 00 ff ff ff 01  
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ff ff ff  
56 65 72 73 69 6f 6e 3d 31 2e 30 2e 30 2e 30 2c  
20 43 75 6c 74 75 72 65 3d 6e 65 75 74 72 61 6c  
2c 20 50 75 62 6c 69 63 4b 65 79 54 6f 6b 65 6e  
3d 6e 75 6c 6c 05 01 00 00 00 00 00 00 00 00 00 00 00  
04 00 00 00 01 61 01 62 01 63 01 69 01 01 01 00  
08 02 00 00 00 06 03 00 00 00 05 51 2e 4d 73 67  
04 00 00 00 03 6e 65 74 06 05 00 00 04 6f 62 6a 65 06  
76 69 6c 65 17 00 00 00 00 0b

## Когда запускает 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, ссылающийся на *ваш* объект, удаляется. Доступ к нему пропадает. По сути объект **умер**.

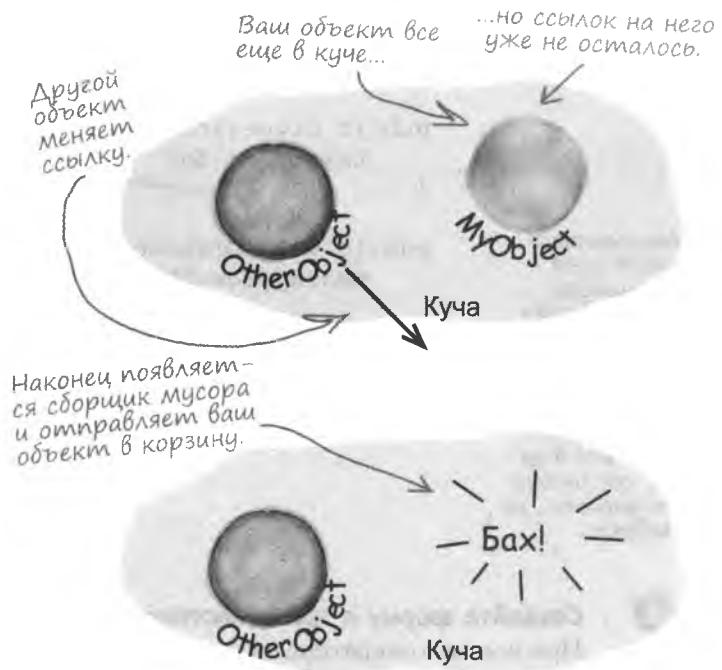
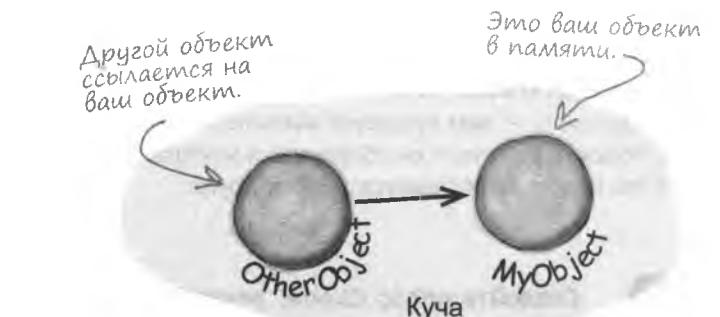
Но дело в том, что *сборкой мусора управляет .NET*, а не объекты. Поэтому до запуска процедуры сборки объект живет в памяти. Его невозможно использовать, но он есть. И его метод завершения объекта еще не начал свою работу.

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 еще раз не подчеркнуть, 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

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:

Как вы уже видели, метод `Dispose()` работает и без оператора `using`. Его многократное применение не имеет побочных эффектов.

### Упражнение!

1

#### Создайте класс `Clone`, реализующий интерфейс `IDisposable`

Класс должен иметь автоматическое свойство `Id` типа `int`, а также конструктор, метод `Dispose()` и метод завершения объекта:

```
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...");
    }
}
```

Напоминаем:  
Вызов окна  
`MessageBox`  
в методе за-  
вершения обь-  
екта может  
внести пусты-  
ницу в работу  
CLR. Приме-  
нить его в це-  
лях, отличных  
от учебных, не  
следует.

Так как класс реализует  
интерфейс `IDisposable`, в нем  
должен присутствовать  
метод `Dispose()`.

Это метод завершения обьекта. Он запускается  
перед отправкой обьекта в сборку мусора.

2

#### Создайте форму с тремя кнопками

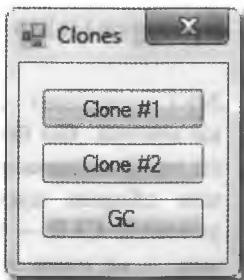
При помощи оператора `using` создайте экземпляр `Clone` в обработ-  
чике события `Click`. Это первая часть кода кнопки:

Метод  
создает  
экземпляр  
`Clone` и не-  
медленно  
убивает  
его, убирая  
все ссылки.

```
private void clone1_Click(object sender, EventArgs e) {
    using (Clone clone1 = new Clone(1)) {
        // Ничего не делайте!
    }
}
```

Так как `clone1` был объяв-  
лен при помощи оператора  
`using`, запускается его метод  
`Dispose()`.

Вот так вы-  
глядит ваша  
форма.



После завершения работы  
блока `using` вызывается  
метод `Dispose()` объекта  
`Clone`. Ссылок на обьект 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 строчка 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().

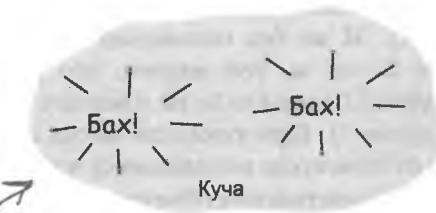


Мусор **в итоге** собран. В большинстве случаев вы **не** увидите окна с сообщением об этом, так как между присвоением объекту значения null и сборкой мусора проходит некоторое время.

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.

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 щелкайте 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.

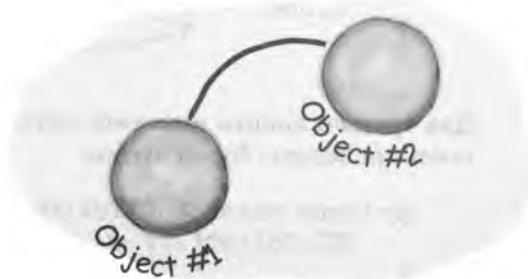
Что это означает на практике? Представим, что у вас есть ссылающиеся друг на друга объекты. Если объект #1 будет удален первым, ссылка объекта #2 начнет указывать в никуда. И наоборот. Другими словами, *вы не можете полагаться на ссылки в методе завершения объекта*. То есть помещать в метод завершения 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`.

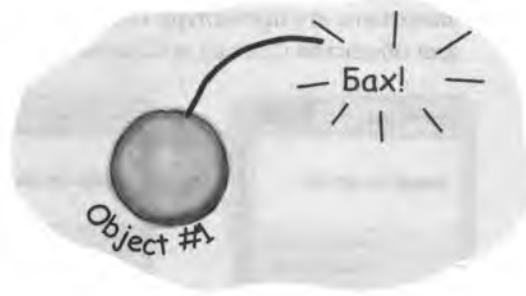
Предположим, у вас есть два объекта, ссылающиеся друг на друга...



...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.



### 1 Сделаем сериализуемым класс Clone (с. 640)

Просто добавьте в верхнюю часть класса атрибут `Serializable`.

`[Serializable]`

```
class Clone : IDisposable
```

### 2 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
```

Для доступа к используемым нами классам I/O вам потребуются несколько директив `using`.

```
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...");
}
```

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 эти проблемы и видите ли пути их решения?

### 3 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, по-вашему, выглядит полный код объекта `SuperHero`? Часть его показана на с. 634. Вы можете написать остальное?

Правда ли, что метод `Dispose()` не имеет побочных эффектов? Что произойдет, если его вызвать более одного раза? Реализация интерфейса `IDisposable`, имеет смысл заранее думать о таких вещах.

*что случилось с капитаном?*

## Беседа у камина



### 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?

Интерфейс существует **именно потому**, что я очень важен. Более того, я единственный метод этого интерфейса!

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.

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 `Dispose()` и метод завершения объекта спорят о том, кто из них ценнее.**

### 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.

Конечно, конечно... продолжай угешать себя. А что произойдет, если пользователь при создании экземпляра забудет оператор `using`? Тебя даже никто не найдет!

Обработчики используются программами для непосредственного взаимодействия с Windows. Так как .NET о них не знает, она не может удалить их за вас.

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.

Ты слишком много мнишь о себе, приятель.



## 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 ссылается на другие объекты. Можно также вызывать другие методы для утилизируемых объектов.

**В:** Что случается с исключениями, появившимися в методе завершения объекта?

**О:** Действительно, 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.

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...

Позднее...

Мяу!

Еще позднее...

БАХ... БАХ... ОХ!  
Я ИСТОЩЕН!

Captain Amazing's  
Hideout Collection  
**TOP SECRET**

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() { ... }
}
```

## ...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 исчерпывается.

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.

Структуры могут реализовывать интерфейсы, но не могут быть унаследованы.

Структуры могут обладать свойствами и полями...

...и определять методы.

**Достоинством объектов является их умение имитировать поведение реальных существ 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 **структуре 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 стороны – **объекты**, такие как `List`, `Stream` и `Exception`. И они работают по-разному.

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**.

Вспомним, чем отличаются значимые типы от объектов.



### Объявление переменных и операция присваивания одинаково

Помните, мы вы для обоих типов:

говорили, что

методы и опе-

раторы ВСЕГДА

находятся внутри

классов? Это не

совсем так, ведь

они могут нахо-

диться и внутри

структур.

```
int howMany = 25;  
bool Scary = true;
```

```
List<double> temperatures = new List<double>();  
Exception ex = new Exception("Does not compute");
```

`int` и `bool` – это значимые типы,  
в то время как `List` и `Exception`  
принадлежат типу `object`.

Присвоение начальных значений  
осуществляется стандартным  
способом.



Различия начинаются при присвоении значений. Для значимых типов эта операция осуществляется методом копирования:

Изменение зна-  
чения перемен-  
ной `fifteenMore`  
никак не влияет  
на переменную  
`howMany`.

```
int fifteenMore = howMany;  
fifteenMore += 15;  
Console.WriteLine("howMany has {0}, fifteenMore has {1}",  
    howMany, fifteenMore);
```

Значение переменной  
`fifteenMore` копируется в пе-  
ременную `howMany`, затем  
к нему прибавляется 15.

Результат демонстрирует, что переменные `fifteenMore` и `howMany` **ни-  
как не связаны**:

```
howMany has 25, fifteenMore has 40
```



В случае объектов присваиваются ссылки, а не значения:

В этой строке  
ке ссылка  
differentList  
запись на том  
же объекте,  
что и ссылка  
`temperatures`.

```
temperatures.Add(56.5D);  
temperatures.Add(27.4D);  
List<float> differentList = temperatures;  
differentList.Add(62.9D);
```

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()` до-  
бавляет новую температуру  
к объекту, на который ука-  
зывают ссылки `differentList`  
и `temperatures`.

## Структуры — это значимые, а объекты — ссылачные 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, таковыми 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

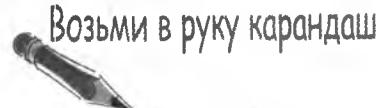
Вот код этого метода:

```
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;
bette.Name = "Betty";
bette.Breed = "pit bull";
jake.Speak();

Console.ReadKey();
```

Вы уже работали со структурами. Помните Point из глав 12 и 13 и DateTime из главы 9? Это были структуры!



### 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.

`Canine spot = new Canine("Spot", "pug"); ①`

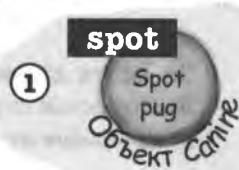
`Canine bob = spot; ②`

`bob.Name = "Spike";`

`bob.Breed = "beagle";`

`spot.Speak(); ③`

Создан объект `Canine`, на который теперь указывает ссылка `spot`.



Была создана новая ссылочная переменная `bob`, но нового объекта в куче не появилось — переменные `bob` и `spot` указывают на один и тот же объект.



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 beagle`.



`Dog jake = new Dog("Jake", "poodle"); ④`

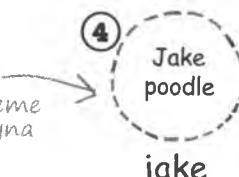
`Dog betty = jake; ⑤`

`betty.Name = "Betty";`

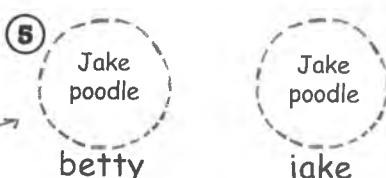
`betty.Breed = "pit bull";`

`jake.Speak(); ⑥`

Создание структуры напоминает создание объекта — вы получаете переменную для доступа к полям и методам.



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`.



## Операция присваивания в случае структур приводит к появлению независимой копии данных. Ведь структура — это значимый тип.

## 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.

Несмотря на возможность назначить структуре переменной типа `object`, структуры и объекты отличаются.

### Kog

Это код, который вы можете обнаружить в программе.

```
Canine spot = new Canine("Spot", "pug");
Dog jake = new Dog("Jake", "poodle");
```

Вот как выглядят стек после выполнения этих двух строк кода.

### Стек

А здесь находятся структуры и локальные переменные.



```
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 наверх стека. И удаляет их после завершения работы метода.



Помните, что в процессе работы вашей программы CLR управляет памятью, выделяя место в куче и собирая мусор.



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) { ... }
```

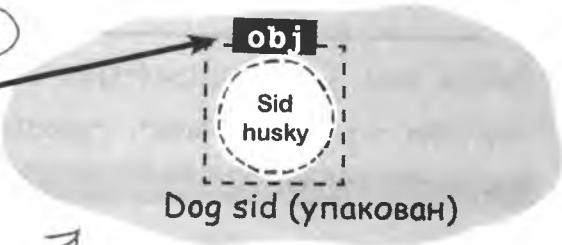
Переданная этому методу структура **упаковывается** в специальную «оболочку», позволяющую ей находиться в куче. В это время работать со структурой невозможно. Ее требуется сначала «распаковать». К счастью, это происходит *автоматически* при передаче методу вместо объекта значимого типа.

Чтобы определить, структура ли 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");  
Object obj = sid;
```



После упаковки структуры появляются две копии данных: в стеке и в куче.

2

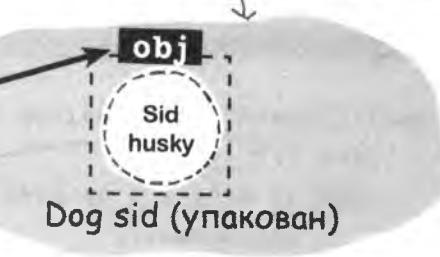
Объект достаточно привести к правильному типу, и он будет распакован автоматически. Со значимыми типами использовать ключевое слово as нельзя, поэтому приведем его к Dog.

```
Dog happy = (Dog) obj;
```

Эти структуры помещаются в кучу только после упаковки.



После этой строчки вы получаете третью копию данных в структуре happy, которая получает свое собственное место в стеке.



За сценой



### 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()`.

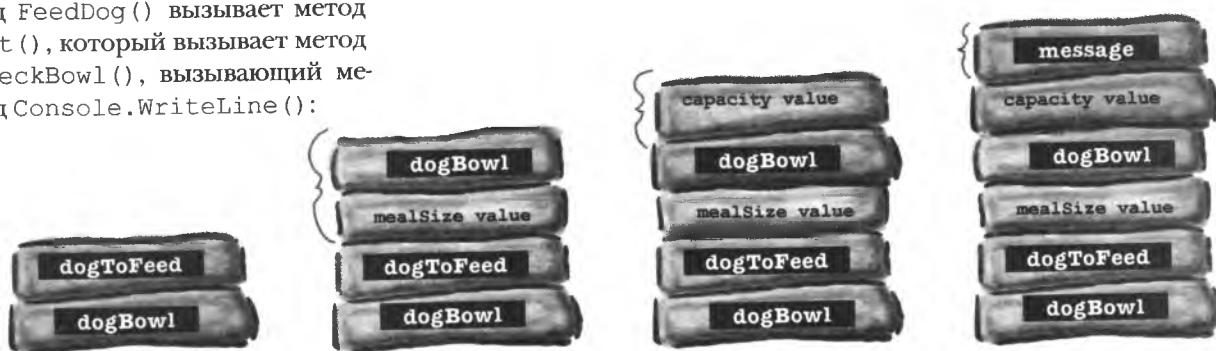
Напомним терминологию: параметр — это часть объявления метода, определяющая, какие значения вам требуются; аргумент — это реальное значение или ссылка, передаваемые вызываемому методу.

Вот как выглядит стек, когда метод `FeedDog()` вызывает метод `Eat()`, который вызывает метод `CheckBowl()`, вызывающий метод `Console.WriteLine()`:

```
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);
    }
}
```



- 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;
}
```

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* должен задавать параметр до возврата 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.

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); ← 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.
    Console.WriteLine("value = {0}, half = {1}, double = {2}", a, b, c);
}
```

**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`, `a` – значение, возвращаемое параметром `twice`.

Упражнение!

Параметр `out`  
дает методу  
возможность  
вернуть бо-  
лее одного  
значения.

## 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.

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;    }   Задавая значение и параметры кнопки,
    button = button1; }   метод меняет переменные q и b
                           в вызвавшем его методе button2_Click().
```

В отличие от аргумента, помеченного модификатором **ref**, аргумент, помеченный модификатором **out**, не требует инициализации перед его передачей.

Добавим кнопку с обработчиком события для вызова метода:

```
private void button2_Click(object sender, EventArgs e) {
    int q = 100;
    Button b = button3;
    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 поменялось.

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!");
}
```



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  
используй-  
те необяза-  
тельные па-  
раметры и  
именованные  
аргументы.**



## Типы, допускающие значение null

На минуту вернемся к карточкам с контактной информацией, которые в главе 1 стали основой для базы данных. Помните, каким образом выбирались параметры таблицы, чтобы позволить присвоение в каждом столбце значения null на случай, если информация не будет введена или будет введена некорректно. К сожалению, структурам и другим значимым типам невозможно присвоить значение null. Вот такие операторы:

```
bool myBool = null;
DateTime myDate = null;
```

на стадии компиляции станут причиной сообщения об ошибке!

Предположим, вам нужные 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? myNullableInt = 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;
```

Если свойство HasValue имеет значение false, свойство Value вызывает исключение InvalidOperationException, как и операция приведения (ведь она производится с использованием свойства Value).

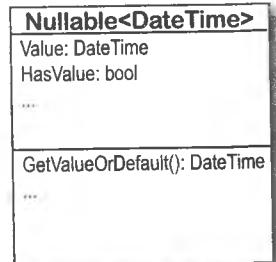
После добавления к любому значимому типу знака вопроса (например, int? или decimal?) компилятор начнет воспринимать полученный результат как структуру Nullable<T> (Nullable<int> или Nullable<decimal>). Добавьте к программе переменную Nullable<DateTime>, поместите на нее точку останова и создайте контрольное 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.Parse() и int.TryParse() – члены этой структуры

```
int value;
struct System.Int32
Represents a 32-bit signed integer.
```

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.

Вам не показалось странным, что даже в столбце Client мы допустили присутствие пустых значений? А ведь человек может или быть клиентом или не быть им, не так ли? Но нет никакой гарантии, что поле Client заполнено на всех карточках. Соответственно, значение null требуется на случай, когда мы просто не знаем, какие данные вводить.



Структура Nullable<T> позволяет хранить как значимые типы, так и значение null. На диаграмме вы видите методы и свойства структуры Nullable<DateTime>.

## 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:

```
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 != null)
            description = "Я родился " + Birthday.Value.ToString("yyyy-MM-dd");
        else
            description = "Я не знаю дату своего рождения";
        if (Height != null)
            description += ", во мне " + Height + " дюймов роста";
        else
            description += ", и я не знаю свой рост";
        return description;
    }
}
```

При вводе некорректных данных метод HasValue вернет значение false.

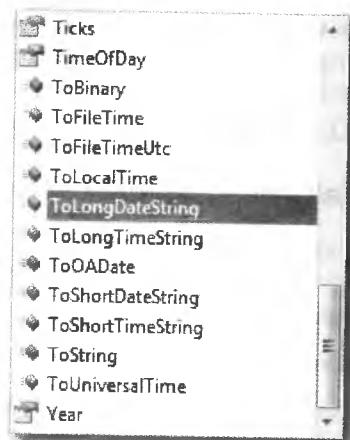
А вот код метода Main(). Он использует метод **Console.ReadLine()** для ввода данных:

```
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.

Добавив метод **RobustGuy.ToString()**, обратите внимание на окно IntelliSense. Вы увидите перечень членов для типа **DateTime**.

Метод **ToLongDateString()** преобразует это в читаемую строку.



Попробуйте экспериментировать с другими методами, связанными с типом **DateTime**, которые начинаются с **To**, чтобы понять, как они влияют на конечный результат.

**Запустите программу и попробуйте вводить различные данные. Многие из них будут распознаны методом **DateTime.TryParse()**. Если сделать это не удастся, свойство **Birthday** класса **RobustGuy** не будет иметь 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.

Фрагменты можно использовать более одного раза.

Brush  
Lamp  
bulb  
Table  
stairs

public  
private  
class  
new  
abstract  
interface

if  
or  
is  
on  
as  
oop

garden  
floor  
Window  
Door  
Hinge

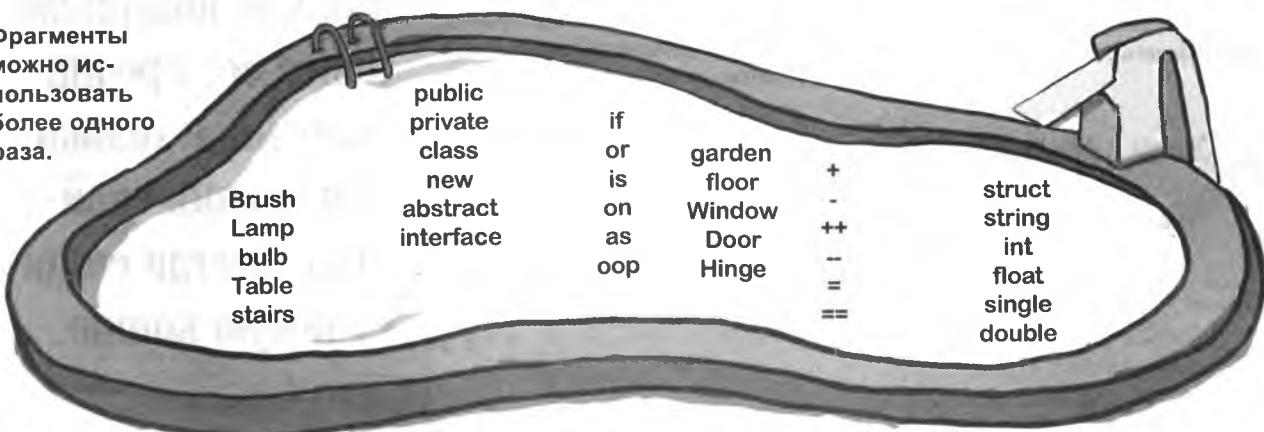
+  
-  
++  
--  
=

struct  
string  
int  
float  
single  
double

```
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.



→ Ответ на с. 668.

далее ▶

## надежность структур

**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, как это делал, к примеру, симулятор улья. 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?

Возьми в руку карандаш

Предполагалось, что этот метод убьет объект `Clone`, но он не работает. Почему?

```
private void SetCloneToNull(Clone clone) {  
    clone = null;  
}
```

## часто Задаваемые Вопросы

**В:** Это поможет вам, к примеру, при **инкапсуляции**. Посмотрите на уже знакомый вам код класса, определяющего местоположение:

```
private 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, предназначеннное 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`.

**В:** Если `Point` — это структура, то, может быть, я уже работал и с другими структурами, сам того не зная?

**О:** Да! При изучении графики вы сталкивались со структурой `Rectangle`. Она снабжена полезными методами, позволяющими указать границы области, и проверит, попадает ли в них выбранная точка. Достаточно указать местоположение и размер структуры, и программа автоматически рассчитает ее остальные параметры.

Также вы сталкивались с такой полезной структурой как `Size`. С ее помощью вы определяли размер строки в методе `MeasureString()`.

**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.

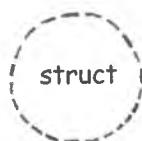
Структуры же полезны при повторяющейся работе с ограниченными типами данных. Хорошим примером служат прямоугольники и точки — они применяются только в определенных ситуациях, зато с удивительной регулярностью. При наличии у вас небольших групп разнородных данных, которые требуется сохранить в поле или передать методу в качестве параметра, скорее всего, имеет смысл воспользоваться структурой.

**Структура позволяет улучшить инкапсуляцию класса, так как возвращающее ее, пред назначенное только для чтения свойство, всегда создает новую копию.**

← Ответ на с. 662.

## 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



### Сравнение

- 1** Структуры не наследуют от классов и не реализуют интерфейсов  
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.

Возможность легко получать копии является большим преимуществом структур и других значимых типов.



- 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.



расширь это

## 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?

Добавим метод расширения `SuperSoldierSerum` (Суперсолдат):

```
static class SuperSoldierSerum {  
    public static string BreakWalls(this OrdinaryHuman h, double wallDensity) {  
        return ("Я сломал стену плотностью" + wallDensity + ".");  
    }  
}
```

Вы используете метод расширения, указывая первый параметр при помощи ключевого слова `this`.

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

Так как параметр `Clone` находится в стеке, присвоение ему значения `null` никак не скажется на состоянии кучи.

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`.

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 наследования. 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.



Я понял! Методы расширения позволяют отредактировать поведение встроенных классов .NET Framework.

Нужно помнить, что метод расширения не дает доступа к внутреннему коду класса.

**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**. Помните код LINQ, добавленный к симулятору в главе 12? LINQ был создан путем расширения интерфейса **IEnumerable<T>**. (Но об этом мы поговорим в следующей главе.)

Надеюсь, вы помните, что обычным образом наследовать от интерфейса невозможно?



## 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.

Упражнение!

1

### Поместим методы расширения в отдельное пространство имен

Сохранить расширения отдельного от основного кода – хорошая идея. Это позволит легко обнаружить их при необходимости:

Отдельное пространство имен это хороший организационный инструмент.

```
namespace MyExtensions {  
    public static class HumanExtensions {
```

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.

```
public static bool IsDistressCall (this string s) {
```

this string указывает, что мы расширяем класс string.

3

### Поместите в метод код, оценивающий строку

```
public static class HumanExtensions {  
    public static bool IsDistressCall(this string s) {  
        Нам нужен доступ  
        к этому классу  
        из других про-  
        странств имен,  
        поэтому исполь-  
        зуйте модифи-  
        катор public!  
        }  
        if (s.Contains("Помогите!"))  
            return true;  
        else  
            return false;
```

Метод расширения также должен быть статическим.

Здесь проверяется, не содержит ли строка определенного значения... которое отсутствует в исходном классе string.

4

### Создайте форму и добавьте к ней строку

Впишите в верхнюю часть кода формы строчку using MyExtensions;. К форме добавьте кнопку. Мы проверим действие нашего метода расширения внутри обработчика события. Теперь при работе с классом string вы получите доступ к методу расширения. Введите имя строковой переменной и период и убедитесь сами:

```
string message = "Клоны разрушают фабрику. Помогите!";  
message.
```

Сразу после ввода точки появится окно с перечнем методов класса string... в их число включен и ваш метод расширения.



Превратите строчку using в комментарий, и метод расширения исчезнет из окна IntelliSense.

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()
```

```
using Upside;  
namespace Sideways {
```

```
class Program {
```

```
    public static string ToPrice
```

}

```
public static string Green
{
    if (n == 1)
        return "a buck ";
    else
        b.Green().SendIt();
}

static void Main(string[] args)
{
    string s = i.ToPrice();
    Console.WriteLine(s);
    Console.ReadKey();
}
```



## 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.

```
namespace Upside {
```

```
    public static class Margin {
```

```
        public static void SendIt (this string s) {
            Console.WriteLine(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";
        }
```

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.

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();
```

Метод Green расширяет класс bool — он возвращает строку be, если булевская переменная имеет значение true, и get — в случае значения false.

Здесь класс Margin расширяет булевые переменные путем добавления к ним класса Green().



МЫ ПЕРЕСТРОИЛИ КЛАСС  
SUPERHERO, НО КАК ВЕРНУТЬ  
КАПИТАНА НАЗАД?



Я ПРОАНАЛИЗИРОВАЛА  
КОД: ВЕЛИКОЛЕПНЫЙ  
ИСПОЛЬЗОВАЛ СВОЮ  
СМЕРТЬ, ЧТОБЫ  
СЕРИАЛИЗОВАТЬ СЕБЯ!

# 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 9. 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!

# Решение ребуса в бассейне



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 объектом. 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);
    }
}

Помните, что  
ключевое слово as  
работает только  
с классами?
```

Метод Lamp помещает переданную ему строку в поле Stairs.

```
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.

## Управляем данными

Если взять первое слово из статьи и второе слово из списка и добавить их к пятому слову вот здесь... мы получим секретное сообщение от правительства!



Этот мир управляет 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**.

## Простой проект...

Объектвильская компания по производству бумаги решила сделать совместную промо-акцию с кофейным магазином. Этот магазин имеет специальную программу, отслеживающую, какой кофе покупает каждый клиент и как часто он это делает. Бумажная компания хотела бы узнать, кто из ее клиентов **регулярно посещает кофейный магазин**, чтобы отправить им бесплатную кружку и купон на покупку их любимого кофе... Вам нужно скомбинировать данные и создать список клиентов для рассылки кружек и купонов.



## ...но сначала нужно собрать данные

Кофейный магазин Starbuzz хранит данные в классах, сгруппированных в большой список List. А вот бумажная компания имеет базу данных (созданную в главе 1). Нужно найти посетителей магазина, потративших более \$90, сравнить со списком контактов бумажной компании и сформировать свой список, в котором будут указаны следующие данные: имя человека, фирма, где он работает, его любимый сорт кофе.

### Данные магазина в коллекции List<T>

Программисты из Starbuzz написали программу, связанную с их сайтом, и поместили все данные в список List<StarbuzzData>.



Это класс и  
перечисление из  
программы ко-  
фейного магази-  
на Starbuzz.

Вам нужно получить  
данные от кофейно-  
го магазина и найти  
там клиентов, ко-  
торые также пользую-  
тся услугами ду-  
мажной компании.

```
class StarbuzzData
{
    public string Name { get; set; }
    public Drink FavoriteDrink { get; set; }
    public int MoneySpent { get; set; }
    public int Visits { get; set; }
}

enum Drink {
    BoringCoffee,
    ChocoRockoLatte,
    TripleEspresso,
    ZestyLemonChai,
    DoubleCappuccino,
    HalfCafAmericano,
    ChocoMacchiato,
    BananaSplitInACup,
}
```

### У вас уже есть данные о заказчиках

Список контактов для объектвильской бумажной компании вы создали еще в главе 1, он содержит часть нужных данных.



База данных  
ContactDB

Все данные о клиентах фирмы  
по производству бумаги вы  
найдете в базе.

### МОЗГОВОЙ ШТУРМ

Как бы вы скомбинировали данные  
от двух организаций для получения  
единого списка контактов?

## Сбор данных из разных источников

Вас спасет LINQ! Эта аббревиатура расшифровывается как *Language Integrated Query* (Встроенный язык запросов). Вы уже использовали эту технологию в симуляторе улья для отслеживания занятий групп пчел. Простые запросы предоставляли вам данные из коллекций. Аналогичным способом LINQ может работать с данными из кофейного магазина. Эту технологию можно применять к любым коллекциям, реализующим интерфейс `IEnumerable<T>`.

LINQ позволяет работать и с наборами коллекций. Те же самые запросы извлекут данные из базы или из документа XML.

В главе 12 вы воспользовались готовым вариантом кода для LINQ-запроса. Теперь же мы подробно рассмотрим, как это работает.

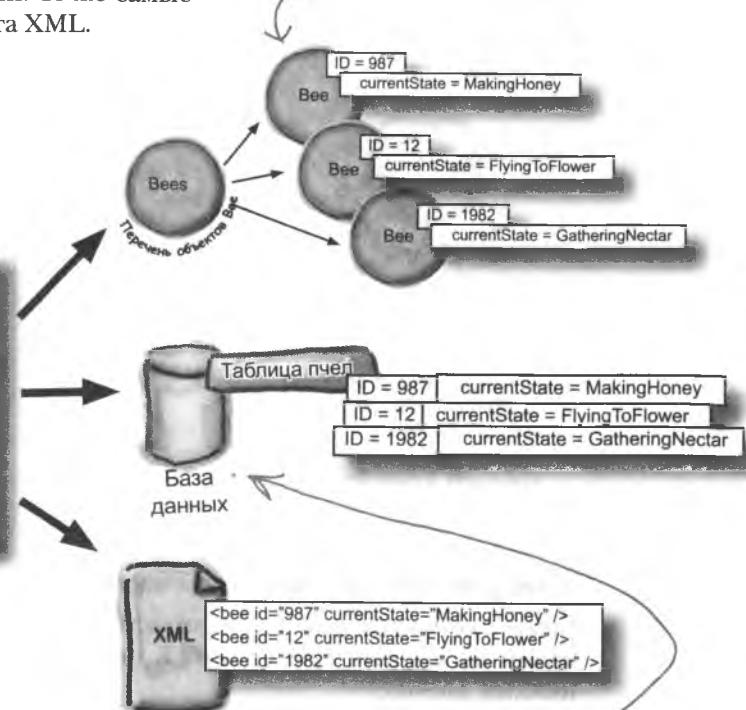
Вот запрос, при помощи которого в симуляторе улья мы группировали и сортировали пчел.

```
var beeGroups =
    from bee in world.Bees
    group bee by bee.CurrentState
    into beeGroup
    orderby beeGroup.Key
    select beeGroup;
```

# LINQ

Аналогичный запрос позволит нам получить данные о клиентах из коллекции кофейного магазина.

LINQ работает с любым источником данных в .NET. Достаточно вставить в верхнюю часть файла с кодом строчку `using System.Linq;`. Более того, ИСР автоматически помещает в верхнюю часть создаваемых файлов классов ссылку на LINQ.



Одни и те же запросы LINQ работают как с базами данных, так и с документами XML.

## 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()`:

```
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.
        I IEnumerator<T> GetEnumerator();
    }
}
```

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 посыпать коллекции запросы.

Обратили внимание, как интерфейс `IEnumerable<T>` расширяет интерфейс `IEnumerable`? Для получения детальной информации снова воспользуйтесь командой `Go to Definition`.

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.

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 строчку:

```
IEnumerable<int> result = from i in linqtest where i < 3 select i;
```

А теперь превратите в комментарий строчку `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 позволяют .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.

```
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;  
  
foreach(int i in result)  
    Console.WriteLine("{0}", i);  
  
Console.ReadKey();
```

Этот запрос состоит из четырех предложений: `from`, `where`, `orderby` и `select`.

Это предложение замещает букву `v` (переменную диапазона) значениями элементов массива. Сначала `v` равно 0, затем — 12, затем — 36... и т. д..

Это предложение выбирает все переменные диапазона `v`, не превышающие 37.

Затем эти значения упорядочиваются по возрастанию.

Пользователям, знакомым с SQL, может показаться странным конечное положение предложения `select`, но именно такой синтаксис используется в LINQ.

Теперь можно по одному перебирать возвращенную LINQ последовательность и вывести результат на консоль.

Результат:

0 8 12 13 36

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 про-сматривать 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 ему в поиске самых дорогих комиксов?

- 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 (limited edition)", Issue = 19 },
        new Comic { Name = "Woman's Work", Issue = 36 },
        new Comic { Name = "Hippie Madness (misprinted)", Issue = 57 },
        new Comic { Name = "Revenge of the New Wave Freak (damaged)", Issue = 68 },
        new Comic { Name = "Black Monday", Issue = 74 },
        new Comic { Name = "Tribal Tattoo Madness", Issue = 83 },
        new Comic { Name = "The Death of 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 Грэга. Джимми узнал, что выпуск #57 «Hippie Madness» из-за опечаток был практически полностью уничтожен издателем. Он обнаружил редкую копию, недавно проданную через каталог Грэга за \$13,525. 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. 674. 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();
```

Запрос LINQ извлекает объекты Comic из предложенного списка, на основе данных из словарного поля values.

```
var mostExpensive =
```

```
    from comic in comics
```

Первым в запросе идет предложение `from`. В качестве источника данных оно указывает на коллекцию `comics`, а переменной диапазона присваивает имя `comic`.

```
    where values[comic.Issue] > 500
```

```
    orderby values[comic.Issue] descending
```

В предложения `where` и `orderby` можно включить АЛЮБЫЕ операторы C#, поэтому мы используем словарные значения для выбора комиксов, цена которых превышает \$500. Затем мы сортируем результат по убыванию.

```
    select comic;
```

Появившееся в предложении `from` имя `comic` затем используется в предложениях `where` и `orderby`.

```
foreach (Comic comic in mostExpensive)
```

```
    Console.WriteLine("{0} стоит {1:c}",
```

Запись «`{1:c}`» в параметрах метода `WriteLine` означает, что второй параметр нужно вывести в формате локальной валюты.

```
        comic.Name, values[comic.Issue]);
```

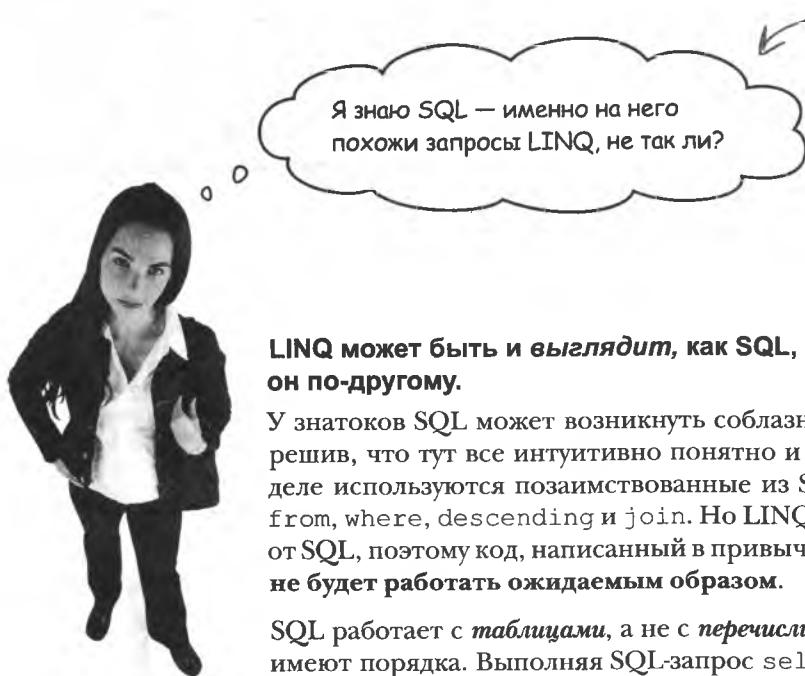
Результатом запроса возвращается в виде перечислителя `IEnumerable<T>`, который называется `mostExpensive`. Что именно вошло в результат, определило предложение `select` — запрос вернул набор объектов `Comic`.

### Результат:

Hippie Madness (с опечатками) стоит \$13,525.00

Johnny America vs. the Pinko стоит \$3,600.00

Woman's Work стоит \$650.00



Даже если вы ничего не знаете об SQL, поводов для беспокойства нет — для работы с LINQ вам не потребуются никакие соответствующие сведения.

### LINQ может быть и выглядит, как SQL, но работает он по-другому.

У знатоков SQL может возникнуть соблазн пропустить главу про LINQ, решив, 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 циклически, 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.*

## 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 принадлежности к 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

Добавим 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);
```

Добавив он rye в конец каждой строки, мы положили все элементы сэндвичей на ржаной хлеб.

Добавим строку «он rye» (на ржаном хлебе) ко всем элементам, возвращенным в результате запроса.

Изменения касаются результатов запроса... но никак не затрагивают элементы исходной коллекции или базы данных.

### Результат:

```
ham and cheese on rye
salami with mayo on rye
turkey and swiss on rye
chicken cutlet on rye
```

### ★ 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 все 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;
```

На этом раз числа сортируются по убыванию.

```
List<int> newList = under50sorted.ToList();
```

С помощью метода `Take()` можно сформировать подмножество результатов:

```
var firstFive = under50sorted.Take(6);
```

```
List<int> shortList = firstFive.ToList();
```

```
foreach (int n in shortList)  
    Console.WriteLine(n);
```



Запросы LINQ не выполняются заранее!

Будьте осторожны!

Это называется «отложенными вычислениями» — сначала должен сработать оператор, использующий результаты запроса. Поэтому так важен метод `ToList()` — он заставляет LINQ немедленно выполнить запрос.

Метод `ToList()` преобразует переменную типа `var` в объект `List<T>`, давая вам возможность сохранить результаты запроса. Аналогичную функцию выполняют методы `ToArray()` и `ToDictionary()`.

Метод `Take()` берет указанное число элементов из результатов запроса. Их можно поместить в другую переменную типа `var`, а потом преобразовать в список.

### Посетите официальную страницу 101 LINQ Samples от Microsoft

Здесь вы найдете дополнительные материалы по работе с LINQ:

<http://msdn2.microsoft.com/en-us/vcsharp/aa336746.aspx>

**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:

### Часто задаваемые вопросы

```
var under10 =  
    from number in numberArray  
    where number < 10  
    select number;
```

Несмотря на кажущуюся простоту, это довольно сложный кусок 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.**

## КЛЮЧЕВЫЕ МОМЕНТЫ



- **from** указывает `IEnumerable<T>`, что мы осуществляляем 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)`.

часто  
Задаваемые  
Вопросы

**В:** Как работает предложение `from`?

**О:** Оно напоминает первую строку цикла `foreach`. Запросы LINQ понячалу 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 строкка 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 элементов 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. Вся 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 поменять последнюю строку 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

select

orderby

select

var weasels =

int[] badgers =

var skunks =

var bears =

);

skunks

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.

LINQ начинает работу с некоторой последовательностью, коллекцией или массива — в данном случае это массив целых чисел.

```
int[] badgers = { 36, 5, 91, 3, 41, 69, 8 };
```

«from pigeon in badgers» выглядит как хорошая головоломка, но такого запроса LINQ не понимает. Запрос должен быть таким: «from badger in badgers».

```
var skunks =
    from pigeon in badgers
    where (pigeon != 36 && pigeon < 50)
    orderby pigeon descending
    select pigeon + 5;
```

После этого оператора последовательность skunks содержит четыре числа: 46, 13, 10 и 8.

Эти операторы LINQ выбирают из массива числа, которые меньше 50 и не равны 36. Затем к каждой из них прибавляется 5, последовательность сортируется по убыванию и помещается в новый объект, на который указывает ссылка skunks.

```
var bears =
    skunks .Take(3);
```

Здесь мы берем первые три элемента последовательности skunks и помещаем их в последовательность bears.

```
var weasels =
    from sparrow in bears
    select sparrow - 1;
```

После этого оператора последовательность weasels содержит три числа: 45, 12 и 9.

Этот оператор вычитает 1 из каждого элемента последовательности bears и помещает эти элементы в последовательность weasels.

```
Console.WriteLine("Get your kicks on route {0}",
    weasels .Sum());
```

Сумма чисел в последовательности weasels составляет 66.

$$45 + 12 + 9 = 66$$

**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.

```
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

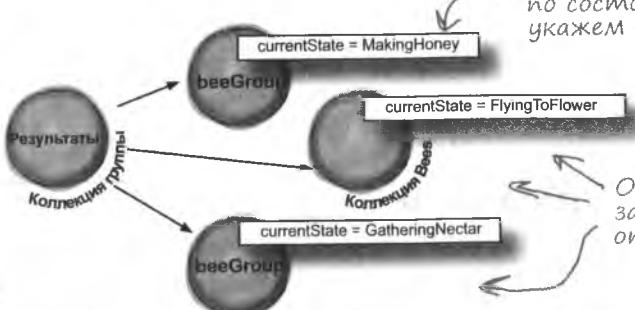
Следующая строка запроса содержит новое ключевое слово `group`. Оно приказывает запросу возвращать *группы* пчел. То есть вместо одной последовательности у нас будет целый набор. Запись `group bee by bee.CurrentState` говорит, что пчел нужно разбить на группы в соответствии со свойством `CurrentState`. 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.). Строкка `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`:



Так как пчелы группировались по состоянию, именно его мы укажем в качестве ключа.

Обратите внимание, что запрос возвращает не отдельных пчел, а их группы.

# 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 Грэга 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 \$1,000 и комиксы, стоящие дороже \$1,000. Мы создадим перечисление PriceRange, которое будет использоваться в качестве ключа, и метод EvaluatePrice(), определяющий цену и возвращающий перечисление PriceRange.

1

## 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;
}
```

2

## 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.WriteLine("I found {0} {1} comics: issues ", group.Count(), group.Key);
    foreach (var price in group)
        Console.Write(price.ToString() + " ");
    Console.WriteLine();
}
```

Каждая из групп является последовательностью, поэтому мы добавили внутренний цикл foreach для просмотра цен внутри группы.

Запрос определяет, к какой группе относится выбранный комикс, передавая его цену методу EvaluatePrice(). Метод же возвращает перечисление PriceRange, используемое в качестве ключа 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.

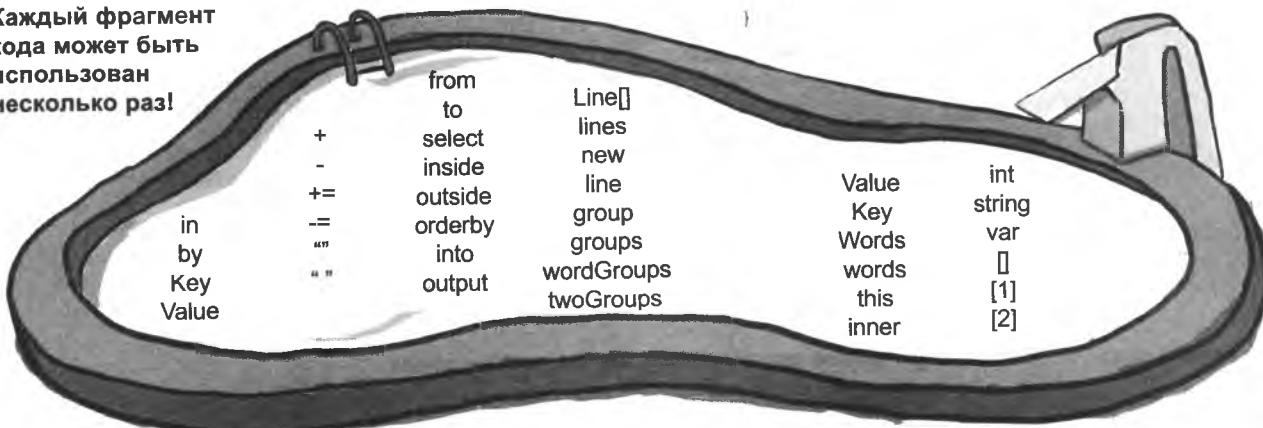
```
class Line {
    public string[] Words;
    public int Value;
    public Line(string[] Words, int Value) {
        this.Words = Words; this.Value = Value;
    }
}
Подсказка: LINQ сортирует строки в алфавитном порядке.
Line[] lines = {
    new Line( new string[] { "eating", "carrots",
        "but", "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 ) };
}
```

Каждый фрагмент кода может быть использован несколько раз!

+	from	Line[]
-	to	lines
+=	select	new
=	inside	line
in	outside	group
by	orderby	groups
Key	into	wordGroups
Value	output	twoGroups

```
var _____ =
    from _____ in _____
        _____ line by line._____
    into wordGroups
    orderby _____.
    select _____;

_____ = words._____();
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.WriteLine(word);
        }
    }
}
```



# 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;
```

Первый запрос делит объекты Line из массива lines[] на группы в соответствии со значением поля Value, располагая их по возрастанию.

```
var twoGroups = words.Take(2);
```

Первые две группы — это строчки со значениями Value 1 и 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);
        }
    }
}
```

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.

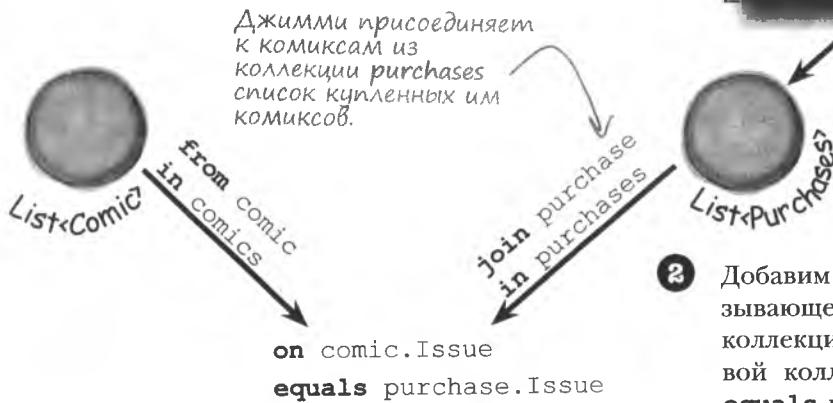
- После предложения `from` вместо критерия отбора результатов напишите:

```
join name in collection
```

Имя `name` назначается членам, которые извлекаются из объединенной коллекции на каждой итерации цикла. Затем вы используете его в предложении `where`.

Данные Джимми представлены в виде коллекции объектов `Purchase`, которая называется `purchases`.

```
class Purchase {
    public int Issue
    { get; set; }
    public decimal Price
    { get; set; }
}
```



- 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.

- Добавим предложение `on`, указывающее способ объединения коллекций. Затем укажем имя первой коллекции, ключевое слово `equals` и имя второй коллекции.

После предложения `select new` в фигурных скобках перечислены данные, которые следует включить в результат.

```
select new { comic.Name,
    comic.Issue, purchase.Price }
```

Issue = 6	name = "Johnny America"	Price = 3600
Issue = 19	name = "Rock and Roll"	Price = 375
Issue = 57	name = "Hippie Madness"	Price = 13215

## 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.

1

### Сначала Джимми создал дополнительную коллекцию

Он использовал свой старый метод BuildCatalog(). Джимми осталось только написать метод FindPurchases() и построить коллекцию классов Purchase.

```
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;
}
```

2

### Все готово для слияния!

Часть этого запроса вы уже видели. Осталось добавить к нему недостающий кусок.

```
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);
```

Джимми счастлив, что он знает LINQ, так как это позволило ему подсчитать, сколько же он сэкономил.

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.

### 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



Хорошо. Джимми использовал LINQ для запросов из коллекций... а как быть с промоакцией кофейного магазина? Я до сих пор не понимаю, как LINQ работает с базами данных.

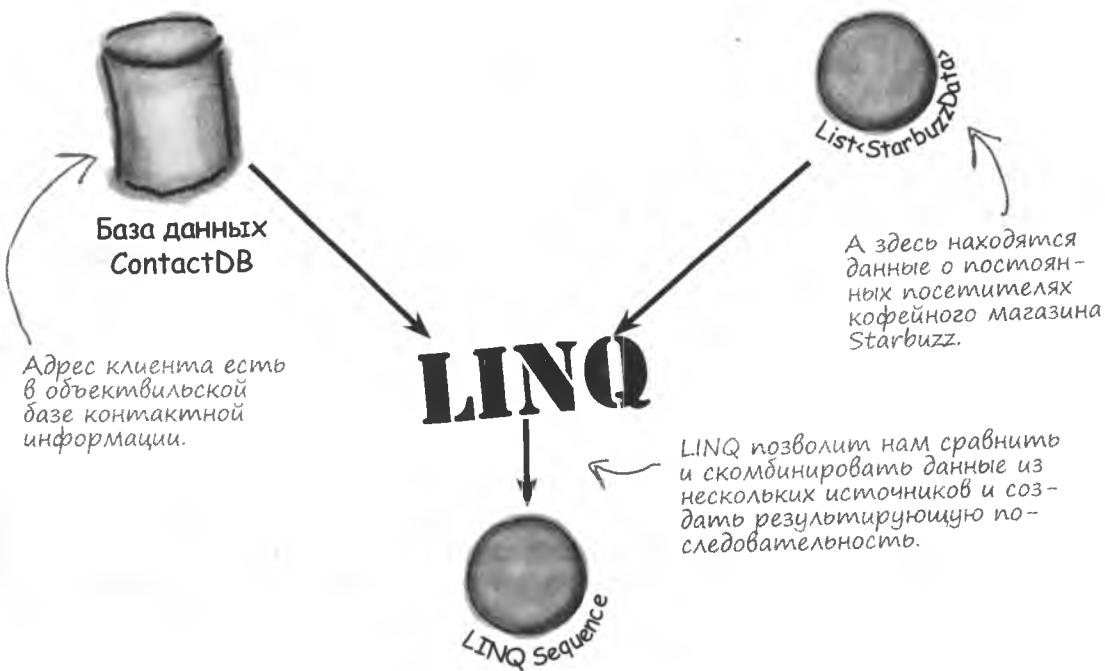
**Для работы с базами данных LINQ использует тот же самый синтаксис.**

В главе 1 вы уже видели, как легко .NET позволяет работать с базами данных. Вы можете быстро присоединить базу, добавить таблицы и даже привязать к форме данные из этих таблиц.

Сделаем запрос к этой, уже связанной с программой базе данных. LINQ без проблем скомбинирует данные из базы с данными, полученными от объектов.

Синтаксис запросов не изменится... вам потребуется всего лишь получить доступ к базе данных.

Несмотря на значительную разницу LINQ и SQL написанный вами код будет похож на остальные запросы LINQ.



## Соединение LINQ с базой данных SQL

LINQ работает с объектами, реализующими интерфейс `IEnumerable<T>`. Значит, для доступа к базе данных SQL нам требуется именно такой объект. Поэтому добавим его к нашему проекту.



Упражнение!

1

### Добавим базу данных к новому консольному приложению

В главе 1 с помощью SQL Server Compact вы создали базу с контактной информацией и сохранили ее в файл `ContactDB.sdf`. Начните новый проект, щелкните правой кнопкой мыши на его имени в окне Solution Explorer, выберите команду `Add Existing Item` и добавьте базу. В раскрывающемся списке типов объектов нужно будет выбрать вариант `Data Files`. (Появится мастер Data Source Configuration, но его можно просто закрыть.)

2

### Для сопоставления LINQ и SQL-платформы используйте `SqlMetal.exe`

Вам остался всего один шаг для присоединения к коду базы данных, и сделать его поможет программа `SqlMetal.exe`. Она устанавливается вместе с Visual Studio 2010 и находится в папке Microsoft SDKs, вложенной в папку Program Files. В командную строку введите команду, указывающую маршрут к папке Microsoft SDK. Для 64-битной версии Windows введите:

```
PATH=%PATH%;%ProgramFiles(x86)%\Microsoft SDKs\Windows\v7.0A\Bin\
```

Для 32-битной версии команда будет несколько отличаться:

```
PATH=%PATH%;%ProgramFiles%\Microsoft SDKs\Windows\v7.0A\Bin\
```

Теперь перейдите в папку с проектом (`cd folder-name`) и введите команду:

```
SqlMetal.exe ContactDB.sdf /dbml:ContactDB.dbml
```

Вот что должно получиться на выходе:

```
Microsoft (R) Database Mapping Generator 2008 version 1.00.30729
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.
```

После этого в вашей папке окажется файл `ContactDB.dbml`. Используйте команду `Add Existing Item`, чтобы добавить его к своему проекту. При этом ИСР создаст для вас еще два файла: `ContactDB.designer.cs` и `ContactDB.layout`.

Получить дополнительную информацию о `SqlMetal.exe` можно здесь:

<http://msdn.microsoft.com/ru-ru/library/bb386987.aspx>

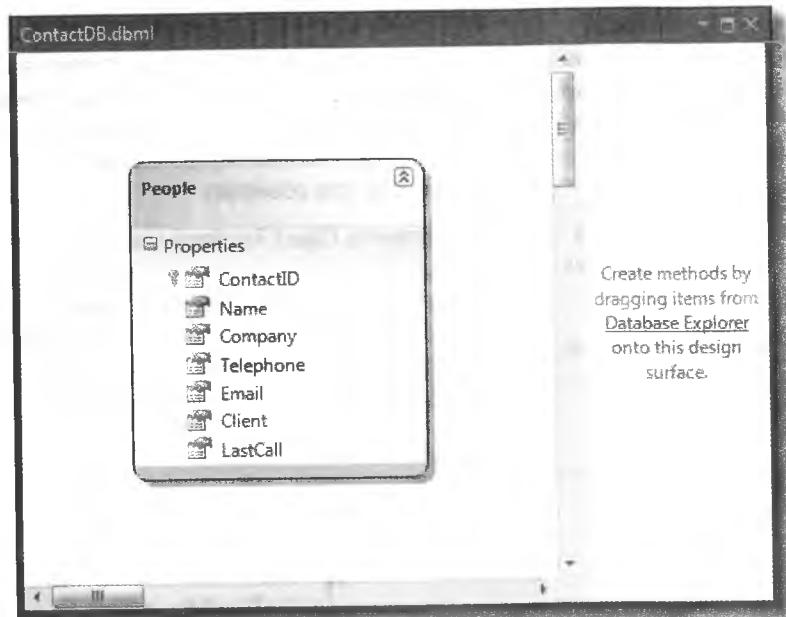
Это может быть  
и папка  
«`NETFX  
4.0 Tools»,  
вложенная  
в папку  
Bin\. В  
этом слу-  
чае в конце  
команды  
PATH=  
нужно на-  
печатать  
«NETFX  
4.0 Tools».`

3

### Откройте классы LINQ и SQL в конструкторе Object Relational

Программой SqlMetal.exe были созданы классы **LINQ to SQL**. Эти классы умеют посыпать запрос к таблицам вашей базы данных и при этом реализуют интерфейс **IEnumerable<T>** с функцией, возвращающей данные в таблицу.

ИСР оснащена таким замечательным инструментом как конструктор **Object Relational**. Здесь вы можете увидеть, какие классы были созданы программой SqlMetal.exe. Двойной щелчок на имени добавленного к проекту файла ContactDB.dbml открывает его в конструкторе **Object Relational**. Вот что вы при этом увидите:



В конструкторе **Object Relational** вы увидите класс **People**, созданный при помощи программы **SqlMetal.exe**. Он присоединяется к проекту таблицу **People** с базой данных и возвращает данные при помощи интерфейса **IEnumerable<T>**, давая возможность осуществлять запросы **LINQ**.

**Примечание:** В версиях Visual Studio, отличных от Express, можно перетащить источник данных SQL непосредственно в конструктор **Object Relational**.

4

### Все готово для написания запросов к базе данных

Добавьте этот код в метод **Main()**. Обратите внимание, как при помощи ключевых слов **select new** было указано, что результат должен содержать только данные из полей **Name** и **Company**.

Классу ContactDB было присвоено имя context. Используйте его свойство People для получения данных из таблицы People.

```
string connectionString = "Data Source=|DataDirectory|\\ContactDB.sdf";
ContactDB context = new ContactDB(connectionString);

var peopleData =
    from person in context.People
    select new { person.Name, person.Company };

foreach (var person in peopleData)
    Console.WriteLine("{0} works at {1}", person.Name, person.Company);
```

Попрактикуйтесь с применением ключевых слов **select new**. Именно они из всех данных базы выбирают только информацию об имени человека и фирме, в которой он работает.



## 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.
- Предложение `join` объединяет две коллекции в одном запросе. Члены обеих коллекций сравниваются друг с другом и из совпадающих пар формируется результат.
- Ключевые слова `on ... equals` задают критерий сравнения в предложении `join`.
- В результаты запроса `join` обычно требуется выборочно включить элементы из обеих коллекций. Это реализуется при помощи предложения `select`.
- Запрос к базе данных SQL осуществляется при помощи классов `LINQ to SQL`. Они обеспечивают программу объектами, работающими с LINQ (то есть дают вам непосредственный доступ к методам этих объектов).
- Конструктор `Object Relational` позволяет выбирать таблицы, к которым вы хотите обратиться средствами LINQ. После выбора таблиц к проекту добавляется класс `DataContext`. Члены его экземпляров добавляются к запросам LINQ и обеспечивают доступ к таблицам SQL.

**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;
```

## часто Задаваемые Вопросы

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».

**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.

**Предложение `select new` позволяет конструировать пользовательский запрос LINQ, в результаты которого включены только нужные вам элементы.**

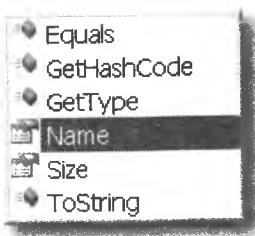
## часть Задаваемые Вопросы

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  
    };
```

ИСР 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:** Всегда ли нужно добавлять файл `.dbml`, создаваемый программой `SqlMetal.exe`? Я так и не понял, зачем он нужен.

**О:** Если вы собираетесь писать запросы к базе данных SQL, без этого файла не обойтись.

Помните, что LINQ требует от объектов реализации интерфейса `IEnumerable<T>`. Базы SQL не реализуют вообще никаких интерфейсов, так как не относятся к объектам. Поэтому для запросов LINQ источник данных требуется преобразовать в объект.

Вернитесь к только что написанному коду, щелкните правой кнопкой мыши на объекте `People` и выберите команду `Go to Definition`. Это приведет вас к методу доступа `ContactDB.designer.cs`, возвращающему объект `Table<People>`. Повторите процедуру для объекта `Table`. Вы увидите, что класс `Table< TEntity >` расширяет интерфейс `IQueryable< TEntity >`. Перейдя к определению этого интерфейса, вы обнаружите, что он реализует интерфейс `IEnumerable< T >`.

То есть файл `.dbml` (и создаваемый им файл класса `.cs`) обеспечивает нас объектом, реализующим интерфейс `IEnumerable`. ИСР точно знает, как поступать с файлом `.dbml`: генерировав его, добавив к проекту и открыв в конструкторе `Object Relational`, вы увидите члены класса `People`, совпадающие с таблицей `People` в базе данных. Этот класс присоединяется к SQL, автоматически читает данные из таблиц и преобразует их в форму, доступную для запросов LINQ.

## Соединим Starbuzz и Objectville

Теперь у вас есть всё, чтобы соединить данные кофейного магазина Starbuzz и объектвильской фирмы по производству бумаги.

Упражнение!

### 1 Добавим к проекту данные SQL

Если вы этого еще не сделали, откройте новое консольное приложение и добавьте к нему базу ContactDB. С помощью программы SqlMetal.exe создайте классы LINQ to SQL, добавьте их к проекту и напишите простой тестовый запрос, чтобы убедиться, что присоединение прошло успешно и все работает.

### 2 Построим объекты Starbuzz

Вот список с данными клиентов магазина Starbuzz. Добавьте его к проекту:

```
class StarbuzzData {  
    public string Name { get; set; }  
    public Drink FavoriteDrink { get; set; }  
    public int MoneySpent { get; set; }  
    public int Visits { get; set; }  
}  
  
enum Drink {  
    BoringCoffee, ChocoRockoLatte, TripleEspresso,  
    ZestyLemonChai, DoubleCappuccino, HalfCafAmericano,  
    ChocoMacchiato, BananaSplitInACup,  
}
```

Данные от кофейного магазина представлены в виде коллекции объектов StarbuzzData. Данных в ней слишком много — и большинство из них вам не нужны. Поэтому выберем только те из них, которые подходят для нашего запроса.

В кофейном магазине много замечательных напитков, и у каждого клиента есть свой любимый.

Вам потребуется метод, генерирующий образцы данных:

```
static IEnumerable<StarbuzzData> GetStarbuzzData() {  
    return new List<StarbuzzData> {  
        new StarbuzzData {  
            Name = "Janet Venutian", FavoriteDrink = Drink.ChocoMacchiato,  
            MoneySpent = 255, Visits = 50 },  
        new StarbuzzData {  
            Name = "Liz Nelson", FavoriteDrink = Drink.DoubleCappuccino,  
            MoneySpent = 150, Visits = 35 },  
        new StarbuzzData {  
            Name = "Matt Franks", FavoriteDrink = Drink.ZestyLemonChai,  
            MoneySpent = 75, Visits = 15 },  
        new StarbuzzData {  
            Name = "Joe Ng", FavoriteDrink = Drink.BananaSplitInACup,  
            MoneySpent = 60, Visits = 10 },  
        new StarbuzzData {  
            Name = "Sarah Kalter", FavoriteDrink = Drink.BoringCoffee,  
            MoneySpent = 110, Visits = 15 }  
    };  
}
```

Метод GetStarbuzzData() задает параметры объектов Starbuzz при помощи инициализатора коллекции и инициализатора объектов.

Напоминаем, что для коллекции и инициализаторов можно не использовать скобки () .

Метод содержит в числе прочего ряд имен из базы контактов бумажной компании. Если вы используете свои варианты имен, позаботьтесь о совпадениях в обоих списках.

### 3 Соединим базу SQL с коллекцией Starbuzz

Это код запроса. Поместите его в метод Main():

```
IEnumerable<StarbuzzData> starbuzzList = GetStarbuzzData();
```

```
string connectionString =
    "Data Source=|DataDirectory|\ContactDB.sdf";
ContactDB context = new ContactDB(connectionString);
```

```
var results =
```

```
from starbuzzCustomer in starbuzzList
```

```
where starbuzzCustomer.MoneySpent > 90
```

```
join person in context.People
```

```
on starbuzzCustomer.Name equals person.Name
```

```
select new { person.Name, person.Company,
```

```
starbuzzCustomer.FavoriteDrink };
```

```
foreach (var row in results)
```

```
Console.WriteLine("{0} at {1} likes {2}",
```

```
row.Name, row.Company, row.FavoriteDrink);
```

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

Предложение select выбирает из базы данные об имени и фирме, а из коллекции Starbuzz данные о люби- мом напитке и объединяет их в результатирующую последова- тельность.

Предложение join объеди- няет данные из коллекции Starbuzz с данными из таблицы People.

Член People класса DataContext — это коллекция, обеспечивающая вас доступом к таблице People в базе данных.

Проверьте результаты. Убедитесь, что все работает так, как вы ожидали.

Существует замечательный инструмент для изучения и применения LINQ. Он называется LINQPad и вы можете скачать его здесь:

<http://www.linqpad.net/>



Прекрасная работа... благодаря рекламной акции наш бизнес пойдет в гору. Мы обязательно позвоним вам снова.

*Эндрю Стиллмен, Дженифер Грин*

**Изучаем C#  
2-е издание**

*Перевела с английского И. Рузмайкина*

Заведующий редакцией

*А. Кривцов*

Руководитель проекта

*А. Юрченко*

Ведущий редактор

*Ю. Сергиенко*

Художественный редактор

*Л. Адуевская*

Корректор

*Л. Казарина*

Верстка

*Л. Родионова*

ООО «Мир книг», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29.  
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2;

95 3005 — литература учебная.

Подписано в печать 29.08.11. Формат 84x100/16. Усл. п. л. 72,24. Тираж 1500. Заказ 26278.

Отпечатано по технологии СоТ в ОАО «Первая Образцовая типография»,

обособленное подразделение «Печатный двор».

197110, Санкт-Петербург, Чкаловский пр., 15.