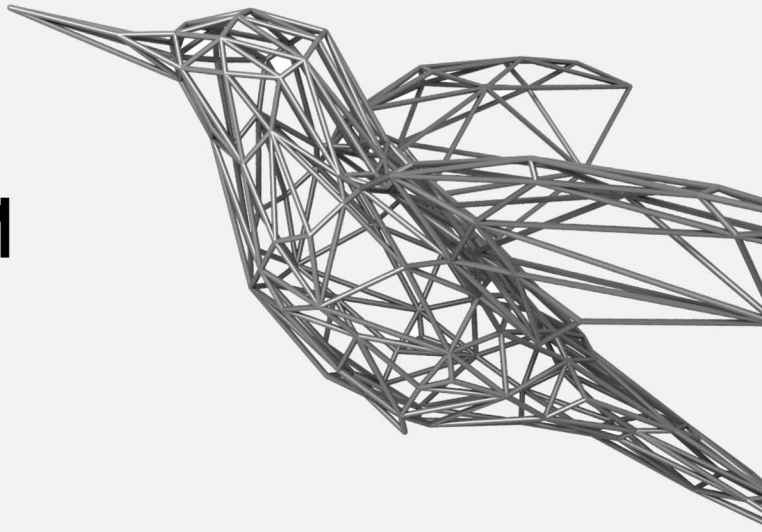
Васильев А.Н.

ПРОГРАММИРОВАНИЕ ДЛЯ НАЧИНАЮЩИХ НА C#



ОСОБЕННОСТИ ЯЗЫКА

- Важные знания об особенностях языка
- От абстрактных классов до указателей и коллекций
- Разбор примеров и задачи для проверки пройденного
- Подходит для студентов и самостоятельного изучения

РОССИЙСКИЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Васильев А.Н.

ПРОГРАММИРОВАНИЕ НА C#

ОСОБЕННОСТИ ЯЗЫКА

РОССИЙСКИЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Москва
2019

УДК 004.43
ББК 32.973.26-018.1
В19

Васильев, Алексей Николаевич.

В19 Программирование на C# для начинающих. Особенности языка / Алексей Васильев. — Москва : Эксмо, 2019. — 528 с. — (Российский компьютерный бестселлер).

ISBN 978-5-04-092520-9

Вторая книга известного российского автора самоучителей по программированию, посвященная особенностям языка C# и его практическому применению. Из этой книги вы узнаете, какие основные структурные единицы языка существуют, научитесь писать программы, используя все основные методы и интерфейсы, и овладеете одним из самых востребованных и популярных языков семейства C.

УДК 004.43
ББК 32.973.26-018.1

ISBN 978-5-04-092520-9

ОГЛАВЛЕНИЕ

Введение
РАСШИРЕННЫЕ ВОЗМОЖНОСТИ C#

Я мечтал об этом всю свою сознательную жизнь.

из к/ф «Ирония судьбы или с легким паром»

Вниманию читателя предлагается вторая часть книги о языке программирования C#. В первой части книги рассмотрены базовые синтаксические конструкции языка, основные типы данных, управляющие инструкции (условный оператор, оператор выбора, операторы цикла) и массивы. Также в ней описывались способы создания классов и объектов, методы их использования. Еще в первой части состоялось знакомство с индексаторами и свойствами, операторными методами. Наследование, перегрузка и переопределение методов также относятся к первой части книги. Предполагается, что читатель знаком (хотя бы на начальном уровне) с этими темами. В противном случае, перед изучением материала из второй части книги, следует освежить в памяти материал из первой части (или обратиться к другому аналогичному учебному пособию).

О чем пойдет речь

Я бы на вашем месте за докторскую диссертацию немедленно сел.

из к/ф «Иван Васильевич меняет профессию»

Далее мы рассмотрим различные темы, связанные с наиболее актуальными и перспективными механизмами языка C#:

- Сначала мы познакомимся с абстрактными классами. Узнаем, в чем особенность абстрактных классов, зачем они нужны и как используются.
- Мы рассмотрим способы создания и использования интерфейсов.

- Важные механизмы связаны с использованием делегатов. Мы узнаем, как объявляются делегаты, как на их основе создаются экземпляры и какую роль при этом играют ссылки на методы. Также состоится знакомство с анонимными методами и лямбда-выражениями.
- Кроме полей, методов, свойств и индексаторов, в классах могут быть такие члены, как события. Зачем они нужны и как используются, описывается в этой книге.
- Мы познакомимся с перечислениями — специальными типами, возможный диапазон значений которых определяется набором констант.
- Кроме классов, в языке C# широко используются структуры. Работе со структурами в книге уделяется отдельное внимание.
- Мощный механизм, связанный с выполнением операций с памятью, базируется на использовании указателей. Мы рассмотрим и эту тему.
- Отдельная глава посвящена перехвату и обработке исключений (ошибок, которые возникают в процессе выполнения программы).
- Язык C# имеет средства поддержки многопоточного программирования, позволяющие одновременно выполняться нескольким частям программы. Способы создания многопоточных приложений описываются в этой книге.
- Обобщенные типы — элегантный механизм, позволяющий создавать красивый и эффективный программный код. Данной теме в книге уделено достаточно внимания.
- Одна из глав книги посвящена вопросам создания приложений с графическим интерфейсом.
- Также у нас состоится краткое знакомство с коллекциями, а еще мы научимся выполнять различные операции с файлами.

Будет и кое-что еще.

Необходимые навыки и ресурсы

> Наконец-то все закончится. И я смогу спокойно поиграть в шахматы. И вообще, пора на пенсию.
>
> *из к/ф «Гостья из будущего»*

Если читатель предварительно ознакомился с содержанием первой части этой книги, то полученных знаний и навыков будет вполне достаточно для того, чтобы разобраться с представленным далее материалом. Вообще же для успешной и эффективной работы со второй частью книги нужно иметь представление о структуре программы в языке C#, базовых типах данных, операторах и управляющих инструкциях. Понадобятся знания в плане создания и использования массивов, описания классов и создания на их основе объектов. Мы будем использовать свойства и индексаторы, операторные методы, прибегать к наследованию, перегрузке и переопределению методов.

Предполагается, что читатель знаком со способами создания приложений на языке C# и обладает необходимыми навыками для работы со средой разработки. В качестве последней предлагается использовать приложение Microsoft Visual Studio (или некоммерческую версию Microsoft Visual Studio Express). Примеры из книги тестировались именно в этой среде разработки. Минимальная, но вполне достаточная для успешной работы информация о среде разработки есть в первой части книги. В этой, второй части в некоторых случаях (когда в этом есть необходимость) также даются пояснения по поводу особенностей работы со средой разработки.

Обратная связь с автором

Автор книги — *Васильев Алексей Николаевич*, доктор физико-математических наук, профессор кафедры теоретической физики физического факультета Киевского национального университета имени Тараса Шевченко. Информацию об этой и других книгах автора можно найти на сайте `www.vasilev.kiev.ua`. Вопросы, замечания и предложения можно отправлять автору по адресу электронной почты `alex@vasilev.kiev.ua`.

Глава 1
АБСТРАКТНЫЕ КЛАССЫ И ИНТЕРФЕЙСЫ

Мерзавец, а, мерзавец, ты, значит, здесь вместо работы латынь изучаешь?

из к/ф «Формула любви»

В этой главе мы в некотором смысле продолжим тему наследования, рассмотренную в первой части книги. Но сейчас речь пойдет скорее о «сопутствующих» технологиях. Нам предстоит познакомиться с абстрактными классами и интерфейсами. Мы узнаем:

- что такое абстрактный класс и зачем он нужен;
- что такое интерфейс, как он описывается и как используется;
- в чем особенность интерфейсных переменных;
- как реализуется расширение интерфейсов;
- что такое явная реализация интерфейса и в чем ее специфика.

Также мы рассмотрим дополнительные аспекты, имеющие отношение к использованию абстрактных классов и интерфейсов.

Знакомство с абстрактными классами

— Астронавты! Которая тут цаппа?
— Там... ржавая гайка, родной.

из к/ф «Кин-дза-дза»

Есть такое понятие, как *абстрактный метод*. Абстрактным называется метод, который в классе только объявлен, но не описан. Под объявлением метода подразумевается ситуация, когда в теле класса указан тип результата метода, его название и приведен список аргументов (после

закрывающей круглой скобки ставится точка с запятой), а тела метода нет. То есть блока из фигурных скобок с командами, выполняемыми при вызове метода, нет вообще. Но чтобы метод был абстрактным, недостаточно описать его без тела с командами. В описании метода необходимо явно указать, что метод абстрактный. Для этого используется ключевое слово `abstract`. Шаблон описания абстрактного метода представлен ниже (жирным шрифтом выделены ключевые элементы шаблона):

```
доступ abstract тип имя(аргументы);
```

Сначала указывается спецификатор уровня доступа, затем ключевое слово `abstract`, после него — идентификатор типа результат метода, название и список аргументов. При этом аргументы описываются, как и в обычном (не абстрактном) методе, с указанием типа аргумента и его формального названия. В конце ставится точка с запятой.

Если в классе есть хотя бы один абстрактный метод, то такой класс также считается *абстрактным*. Абстрактный класс, как и абстрактный метод, описывается с ключевым словом `abstract`.

Поскольку у абстрактного метода нет тела и не известно, какие команды должны выполняться при вызове метода, то такой метод вызвать нельзя. Поэтому, как вы могли догадаться, на основе абстрактного класса нельзя создать объект. В этом случае все логично: если бы такой объект можно было создать, то у него был бы метод, который нельзя вызвать. А это, как минимум, странно и непоследовательно. Но тогда возникает вопрос: а зачем вообще нужны абстрактные классы? Ответ состоит в том, что абстрактный класс может (и должен) использоваться как базовый при наследовании. Это его главное и наиболее важное назначение — быть базовым классом. Общая схема такова: создается абстрактный класс, а затем путем наследования на основе абстрактного класса создаются обычные (не абстрактные) производные классы. При этом в производных классах абстрактные методы переопределяются: описывается полная версия метода, и, как при переопределении обычных методов, в описании указывается ключевое слово `override`. В производном классе должны быть переопределены (описаны) все абстрактные методы из абстрактного базового класса.

НА ЗАМЕТКУ

Абстрактный метод по определению является виртуальным. При этом ключевое слово `virtual` в объявлении абстрактного метода не используется. Также следует учитывать, что абстрактный метод не может быть статическим.

Преимущество использования абстрактного класса как базового состоит в том, что, описывая базовый класс, мы фактически создаем некий шаблон для производных классов. Все производные классы, созданные на основе одного и того же абстрактного класса, будут иметь определенный набор методов. В каждом из классов эти методы реализуются по-своему, но они есть, и мы в этом можем быть уверены.

Конечно, никто не запрещает нам описать обычный (не абстрактный) класс и затем на его основе создавать производные классы, переопределяя в них методы их базового класса. Но это не самый лучший подход, поскольку если забыть переопределить в производном классе метод (который надо переопределить), то в производном классе будет использована унаследованная версия метода из базового класса и формальной ошибки в этом не будет. А такие ситуации сложно отслеживать. Делая же методы абстрактными в базовом абстрактном классе, мы с необходимостью должны будем описать их в производном классе. Если этого не сделать, программа просто не скомпилируется.

ПОДРОБНОСТИ

Если класс описать с ключевым словом `abstract`, то он будет абстрактным, даже если в классе нет ни одного абстрактного метода. При наследовании абстрактного класса в производном классе можно переопределять не все абстрактные методы из базового класса. Но в таком случае производный класс также будет абстрактным и должен быть описан с ключевым словом `abstract`.

Также следует заметить, что хотя создать объект абстрактного класса нельзя, но можно объявить объектную переменную для абстрактного класса. Эта переменная не может ссылаться на объект абстрактного класса (поскольку такой объект создать нельзя), зато она может ссылаться на объект производного класса. А поскольку абстрактные методы по умолчанию являются виртуальными и переопределяются в производном классе, то через объектную переменную базового абстрактного класса будут вызываться именно те версии методов, что определены в производном классе. Далее перейдем к рассмотрению примеров.

ПОДРОБНОСТИ

Абстрактными могут быть также свойства и индексаторы. При описании абстрактного свойства или индексатора используется ключевое слово `abstract`. В теле свойства или метода аксессоры

не описываются. Указываются только ключевые слова `get` и `set` или только одно из них, если у свойства или индексатора только один аксессор.

В производном классе свойство или индексатор описываются с ключевым словом `override`, как при переопределении метода. При этом должны быть описаны все аксессоры, объявленные в абстрактном классе для данного свойства или индексатора.

Использование абстрактных классов

— Дядя Вова. Цаппу надо крутить, цаппу.
— На! Сам делай!
— Мне нельзя, я чатланин.

из к/ф «Кин-дза-дза»

Для начала мы рассмотрим очень простой пример, в котором описывается абстрактный класс, а затем на основе этого класса создаются производные классы. Рассмотрим программу, представленную в листинге 1.1.

Листинг 1.1. Знакомство с абстрактными классами

```
using System;
// Абстрактный класс:
abstract class Base{
   // Защищенное целочисленное поле:
   protected int num;
   // Конструктор:
   public Base(int n){
      // Вызов метода:
      set(n);
   }
   // Абстрактные методы:
   public abstract void show();
   public abstract void set(int n);
   public abstract int get();
```

```
}
// Производный класс на основе абстрактного класса:
class Alpha:Base{
   // Защищенное целочисленное поле:
   protected int val;
   // Конструктор:
   public Alpha(int n):base(n){
      // Вызов метода:
      show();
   }
   // Переопределение абстрактного метода:
   public override void show(){
      // Отображение сообщения:
      Console.WriteLine("Alpha: {0}, {1} и {2}",num,val,get());
   }
   // Переопределение абстрактного метода:
   public override void set(int n){
      // Присваивание значений полям:
      num=n;
      val=n%10;
   }
   // Переопределение абстрактного метода:
   public override int get(){
      return num/10;
   }
}
// Производный класс на основе абстрактного класса:
class Bravo:Base{
   // Защищенное целочисленное поле:
   protected int val;
   // Конструктор:
   public Bravo(int n):base(n){
```

```
      // Вызов метода:
      show();
   }
   // Переопределение абстрактного метода:
   public override void show(){
      // Отображение сообщения:
      Console.WriteLine("Bravo: {0}, {1} и {2}",num,val,get());
   }
   // Переопределение абстрактного метода:
   public override void set(int n){
      // Присваивание значений полям:
      num=n;
      val=n%100;
   }
   // Переопределение абстрактного метода:
   public override int get(){
      return num/100;
   }
}
// Класс с главным методом:
class AbstractDemo{
   // Главный метод:
   static void Main(){
      // Объектная переменная абстрактного класса:
      Base obj;
      // Создание объектов производных классов:
      Alpha A=new Alpha(123);
      Bravo B=new Bravo(321);
      // Объектной переменной базового класса присваивается
      // ссылка на объект производного класса:
      obj=A;
      Console.WriteLine("После выполнения команды obj=A");
```

```
      // Вызов методов через объектную переменную
      // базового класса:
      obj.set(456);
      obj.show();
      // Объектной переменной базового класса присваивается
      // ссылка на объект производного класса:
      obj=B;
      Console.WriteLine("После выполнения команды obj=B");
      // Вызов методов через объектную переменную
      // базового класса:
      obj.set(654);
      obj.show();
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 1.1)

```
Alpha: 123, 3 и 12
Bravo: 321, 21 и 3
После выполнения команды obj=A
Alpha: 456, 6 и 45
После выполнения команды obj=B
Bravo: 654, 54 и 6
```

В программе описан абстрактный класс `Base`. Он описан с ключевым словом `abstract`. В классе объявлены три абстрактных метода (все описаны с ключевым словом `abstract`): метод `show()` без аргументов и не возвращающий результат, метод `set()` с целочисленным аргументом и не возвращающий результат, метод `get()` без аргументов и возвращающий целочисленный результат. Все эти методы должны быть описаны в производном классе, создаваемом на основе данного абстрактного класса. Но не все «содержимое» абстрактного класса является «абстрактным». В нем описано закрытое целочисленное поле `num`, а также конструктор с одним целочисленным аргументом. В теле

конструктора содержится команда `set(n)`, которой метод `set()` вызывается с аргументом `n`, переданным конструктору. Интересно здесь то, что метод `set()` абстрактный и в классе `Base` не описан, а только объявлен.

ПОДРОБНОСТИ

Хотя на основе абстрактного класса объект создать нельзя, но можно описать конструктор для абстрактного класса. Этот конструктор будет вызываться при создании объекта производного класса, поскольку в этом случае сначала вызывается конструктор базового класса. В нашем примере в теле конструктора класса `Base` вызывается абстрактный метод `set()`, в классе не описанный. Но проблемы при этом не возникает, поскольку выполняться конструктор базового класса будет при создании объекта производного класса, в котором метод `set()` должен быть описан. Проще говоря, на момент, когда метод `set()` будет вызываться, он уже будет описан.

На основе класса `Base` создается два класса: `Alpha` и `Bravo`. Эти классы очень похожи, но описываются по-разному. Начнем с общих моментов для обоих классов. И в классе `Alpha`, и в классе `Bravo` появляется дополнительное целочисленное поле `val`. В каждом из классов описан конструктор с целочисленным аргументом, который передается конструктору базового класса. В теле конструктора вызывается метод `show()`. Но описывается метод `show()` в каждом классе со своими особенностями. В классе `Alpha` при вызове метода `show()` отображаются название класса `Alpha`, значения полей `num` и `val` и результат вызова метода `get()`. Метод `show()` для класса `Bravo` описан так же, но название класса отображается другое. Метод `get()` в каждом классе также свой. В классе `Alpha` метод `get()` описан так, что результатом возвращается значение `num/10`. Это значение поля `num`, если в нем отбросить разряд единиц. В классе `Bravo` результатом метода `get()` является значение `num/100`, которое получается отбрасыванием разрядов единиц и десятков в значении поля `num`.

Метод `set()` в классе `Alpha` переопределен таким образом, что при целочисленном аргументе `n` выполняются команды `num=n` и `val=n%10`. То есть полю `num` присваивается значение аргумента, а полю `val` в качестве значения присваивается остаток от деления значения аргумента на `10` (это последняя цифра в десятичном представлении числового значения аргумента `n`).

В классе `Bravo` метод `set()` описан похожим образом, но полю `val` значение присваивается командой `val=n%100` (остаток от деления

значения аргумента на 100, или число из двух последних цифр в десятичном представлении значения аргумента n).

НА ЗАМЕТКУ

Обращаем внимание, что все абстрактные методы из базового класса в производном классе описываются с ключевым словом override.

В главном методе программы мы командой Base obj объявляем объектную переменную obj абстрактного класса Base. Командами Alpha A=new Alpha(123) и Bravo B=new Bravo(321) создаются объекты производных классов. При этом в консольном окне появляются сообщения, содержащие название класса для созданного объекта, значения полей и результат вызова метода get() из созданного объекта.

ПОДРОБНОСТИ

В сообщении, кроме имени класса, отображается еще три числа. Первое — это значение поля num. Второе — это значение поля val. Для объекта класса Alpha это последняя цифра в значении поля num, а для объекта класса Bravo это две последние цифры в значении поля num. Третье число — значение, возвращаемое методом get(). Для объекта класса Alpha это значение поля num без последней цифры, а для объекта класса Bravo это значение поля num без двух последних цифр.

После выполнения команды obj=A объектной переменной базового абстрактного класса присваивается ссылка на объект производного класса Alpha. Далее командой obj.set(456) меняются значения полей объекта, после чего командой obj.show() проверяются значения полей и результат вызова метода get().

Затем выполняется команда obj=B, при помощи которой объектной переменной базового абстрактного класса присваивается ссылка на объект производного класса Bravo. Командой obj.set(654) вносятся изменения в значения полей объекта, а командой obj.show() проверяется результат.

НА ЗАМЕТКУ

Стоит заметить, что если мы вызываем через объектную переменную абстрактного базового класса переопределенные методы из объекта производного класса, то версия метода определяется на основе класса объекта, из которого вызывается метод.

Еще один пример, который мы рассмотрим далее, дает представление о том, как описывается и используется абстрактный класс, в котором есть абстрактные свойства и индексаторы.

Листинг 1.2. Абстрактные свойства и индексаторы

```
using System;
// Абстрактный класс:
abstract class Base{
   // Абстрактное текстовое свойство:
   public abstract string text{
      get;
      set;
   }
   // Абстрактный индексатор с целочисленным индексом:
   public abstract char this[int k]{
      get;
   }
   // Абстрактное целочисленное свойство:
   public abstract int length{
      get;
   }
}
// Производный класс на основе абстрактного:
class Alpha:Base{
   // Закрытое поле, являющееся ссылкой на массив:
   private char[] symbs;
   // Конструктор:
   public Alpha(string t):base(){
      // Текстовому свойству присваивается значение:
      text=t;
   }
   // Переопределение текстового свойства:
   public override string text{
```

```
      get{
         // Результатом является текстовая строка:
         return new string(symbs);
      }
      set{
         // Создание символьного массива и присваивание
         // значения полю:
         symbs=value.ToCharArray();
      }
   }
   // Переопределение целочисленного свойства:
   public override int length{
      get{
         // Размер массива:
         return symbs.Length;
      }
   }
   // Переопределение индексатора:
   public override char this[int k]{
      get{
         // Значение элемента символьного массива:
         return symbs[k];
      }
   }
}
// Производный класс на основе абстрактного:
class Bravo:Base{
   // Закрытое текстовое поле:
   private string txt;
   // Конструктор:
   public Bravo(string t):base(){
      // Текстовому свойству присваивается значение:
```

```
      text=t;
   }
   // Переопределение текстового свойства:
   public override string text{
      get{
         // Значение поля:
         return txt;
      }
      set{
         // Присваивание значения полю:
         txt=value;
      }
   }
   // Переопределение целочисленного свойства:
   public override int length{
      get{
         // Количество символов в тексте:
         return txt.Length;
      }
   }
   // Переопределение индексатора:
   public override char this[int k]{
      get{
         // Символ в тексте:
         return txt[k];
      }
   }
}
// Класс с главным методом:
class AbstrPropAndIndexDemo{
   // Главный метод:
   static void Main(){
```

```
      // Ссылка на объект производного класса записывается
      // в объектную переменную базового класса:
      Base obj=new Alpha("Alpha");
      // Отображение значения текстового свойства:
      Console.WriteLine(obj.text);
      // Новое значение текстового свойства:
      obj.text="Base";
      // Индексирование объекта:
      for(int k=0;k<obj.length;k++){
         Console.Write("|"+obj[k]);
      }
      Console.WriteLine("|");
      // Ссылка на объект производного класса записывается
      // в объектную переменную базового класса:
      obj=new Bravo("Bravo");
      // Индексирование объекта:
      for(int k=0;k<obj.length;k++){
         Console.Write("|"+obj[k]);
      }
      Console.WriteLine("|");
      // Новое значение текстового свойства:
      obj.text="Base";
      // Отображение значения текстового свойства:
      Console.WriteLine(obj.text);
   }
}
```

Ниже показано, как выглядит результат выполнения программы:

Результат выполнения программы (из листинга 1.2)

```
Alpha
|B|a|s|e|
```

```
|B|r|a|v|o|
Base
```

В программе описан абстрактный класс `Base`, в котором объявлены два абстрактных свойства (текстовое `text` и целочисленное `length`) и индексатор с целочисленным индексом. Все эти члены класса описаны с ключевым словом `abstract`. В теле абстрактного текстового свойства `text` указаны ключевые слова `get` и `set` (после каждого ключевого слова ставится точка с запятой). Это означает, что при переопределении (по факту при описании) свойства должен быть описан и `get`-аксессор, и `set`-аксессор. В теле свойства `length` и в теле индексатора указано только ключевое слово `get`. Поэтому при переопределении свойства и индексатора в производном классе описывается только `get`-аксессор.

НА ЗАМЕТКУ

В производных классах свойства и индексатор описываются с ключевым словом `override`, как при переопределении методов.

На основе класса `Base` путем наследования создается класс `Alpha` и класс `Bravo`. Классы практически идентичны, но имеются отличия на «техническом» уровне. В классе `Alpha` используется закрытое поле `symbs`, являющееся ссылкой на символьный массив. В классе `Bravo` описано закрытое текстовое свойство `txt`. У каждого из классов есть конструктор с текстовым аргументом.

НА ЗАМЕТКУ

В базовом классе `Base` конструктор не описывался. В производных классах `Alpha` и `Bravo` описываются конструкторы. В них вызывается конструктор базового класса без аргументов (инструкция `base()`). Имеется в виду конструктор по умолчанию класса `Base`.

В каждом из конструкторов переданное аргументом текстовое значение присваивается свойству `text`. Но в классе `Alpha` процедура присваивания значения свойству `text` описана так, что присваиваемое текстовое значение с помощью библиотечного метода `ToCharArray()` преобразуется в массив и ссылка на этот массив записывается в поле `symbs`. В классе `Bravo` текстовое значение при присваивании свойству `text` в действительности записывается в поле `txt`. Значение поля `txt` возвращается в виде значения свойства `text` для объекта класса `Bravo`.

Для объекта класса `Alpha` в качестве значения свойства `text` возвращается текстовая строка, сформированная на основе символьного массива `symbs`. Чтобы сформировать текст на основе символьного массива, мы создаем анонимный объект класса `String` и передаем ему аргументом ссылку на символьный массив.

НА ЗАМЕТКУ

В команде `return new string(symbs)` в `get`-аксессоре свойства `text` класса `Alpha` мы использовали синоним `string` для инструкции `System.String`.

Свойство `length` описано таким образом, что для объекта класса `Alpha` оно возвращает длину символьного массива `symbs`, а для объекта класса `Bravo` значение свойства определяется количеством символов в текстовом поле `txt`. Наконец, индексатор описан так, что результатом возвращается символьное значение элемента массива (класс `Alpha`) или символ из текста (класс `Bravo`).

В главном методе программы сначала командой `Base obj=new Alpha("Alpha")` создается объект класса `Alpha`, а ссылка на объект записывается в объектную переменную `obj` абстрактного класса `Base`. Мы проверяем значение свойства `text` (команда `Console.WriteLine(obj.text)`), присваиваем свойству новое значение (команда `obj.text="Base"`) и с помощью оператора цикла, индексируя объектную переменную `obj`, посимвольно отображаем содержимое символьного массива из объекта, на который ссылается переменная `obj`. После этого командой `obj=new Bravo("Bravo")` создаем объект класса `Bravo` и записываем ссылку на него в переменную `obj`. В теле оператора цикла выполняется индексирование объекта, благодаря чему мы посимвольно отображаем содержимое текстового поля объекта, на который теперь ссылается переменная `obj`. Командой `obj.text="Base"` текстовому свойству объекта присваивается новое значение, после чего мы проверяем значение этого свойства (команда `Console.WriteLine(obj.text)`).

Что мы получили в данном случае? На основе абстрактного класса мы создали два класса с одинаковым набором характеристик (имеются в виду свойства и индексатор), но при этом «механизм» реализации производных классов разный. Получается, что абстрактный класс задал некоторый шаблон, в соответствии с которым реализованы производные классы. Такой подход на практике нередко оказывается полезным.

Знакомство с интерфейсами

> Статуя здесь ни при чем. Она тоже женщина несчастная. Она графа любит.
>
> *из к/ф «Формула любви»*

Выше мы познакомились с базовыми принципами использования абстрактных классов. Как уже несколько раз отмечалось, использование абстрактного класса в качестве базового позволяет создавать производные классы «по одному шаблону» — то есть с одинаковым набором свойств и методов. Вместе с тем здесь имеется один «тонкий момент». Дело в том, что в языке C# запрещено множественное наследование: мы не можем создать производный класс на основе сразу нескольких базовых.

> ⓘ **НА ЗАМЕТКУ**
>
> Множественное наследование есть в языке C++. В языке C++ у производного класса может быть несколько базовых классов. В языках Java и C# от множественного наследования отказались в целях безопасности.

Это не очень хорошо, поскольку часто возникает необходимость «объединить в одно целое» сразу несколько классов. Например, в языке C++, ставшем «прародителем» для языка C#, такая возможность существует. Это полезная возможность, но одновременно это и небезопасная возможность. Ведь разные классы описывались независимо друг от друга. Их объединение в один класс может привести к конфликтным ситуациям. Поэтому в языке C# от технологии множественного наследования отказались. Вместо множественного наследования используется другая технология, связанная с реализацией *интерфейсов*.

Главная опасность в попытке объединения классов связана с тем, что в них есть методы и эти методы каким-то образом определены. Когда метод описывался в классе, перспектива совместного использования этого метода с методами из иных классов, скорее всего, не рассматривалась. Отсюда и неприятные сюрпризы. Но если объединять классы с абстрактными методами, то данная проблема снимается автоматически, поскольку абстрактные методы не имеют тела, они только объявлены (но не описаны). Необходимо только обеспечить, чтобы все методы были абстрактными. В обычном абстрактном классе в общем случае это не так. Отсюда появляется потребность в *интерфейсах*.

Интерфейс представляет собой блок из абстрактных методов, свойств и индексаторов. Фактически это аналог абстрактного класса. Но, в отличие от абстрактного класса, в интерфейсе абсолютно все абстрактное. Описывается интерфейс специальным образом, хотя описание интерфейса и напоминает описание класса. Общий шаблон описания интерфейса представлен ниже (жирным шрифтом выделены ключевые элементы шаблона):

```
interface имя{
   // Тело интерфейса
}
```

Начинается описание интерфейса с ключевого слова `interface`, после которого указывается название интерфейса, а в блоке из фигурных скобок объявляются методы, индексаторы и свойства.

НА ЗАМЕТКУ

Помимо методов, индексаторов и свойств, интерфейс также может содержать объявление событий. Мы рассмотрим события в следующей главе.

Для методов указывается только сигнатура: тип результата, название метода и список аргументов. Ключевое слово `abstract` не указывается, как и ключевое слово `virtual`. По умолчанию объявленные в интерфейсе методы (а также свойства и индексаторы) считаются абстрактными и виртуальными. Спецификатор уровня доступа также не указывается. Все методы (свойства, индексаторы), объявленные в интерфейсе, являются открытыми (то есть будто бы описанными с ключевым словом `public`, хотя оно явно не используется).

ПОДРОБНОСТИ

Свойства и индексаторы в интерфейсе объявляются с пустым телом, в котором указываются ключевые слова `get` и `set`. Наличие обоих ключевых слов (после каждого ставится точка с запятой) означает, что при описании у свойства и индексатора должно быть два аксессора. Можно указывать только одно ключевое слово, соответствующее аксессору, который должен быть у свойства или индексатора.

Интерфейс нужен для того, чтобы на его основе создавать классы. Если класс создается на основе интерфейса, то говорят, что класс *реализует*

интерфейс. Реализация интерфейса в классе подразумевает, что в этом классе описаны все методы, свойства и индексаторы, которые объявлены в интерфейсе. Причем при описании методов, свойств и индексаторов в классе ключевое слово `override` не используется.

ПОДРОБНОСТИ

При наследовании абстрактного класса мы можем объявить производный класс как абстрактный и не описывать в нем некоторые или все абстрактные методы из абстрактного класса. При реализации интерфейса класс, реализующий интерфейс, должен содержать описание всех методов (свойств, индексаторов) из интерфейса. Описать только часть методов и на этом основании объявить класс абстрактным не получится.

Имя интерфейса, реализуемого в классе, указывается в описании класса через двоеточие после имени класса (то есть так же, как указывается имя базового класса при наследовании). Шаблон описания класса, реализующего интерфейс, следующий:

```
class имя:интерфейс{
   // Тело класса
}
```

Реализация интерфейса напоминает наследование абстрактного класса. Но базовый класс может быть только один, а вот что касается реализации интерфейсов, то в одном классе может реализоваться больше одного интерфейса. Если класс реализует несколько интерфейсов, то эти интерфейсы перечисляются через запятую (после двоеточия) в описании класса:

```
class имя:интерфейс,интерфейс,...,интерфейс{
   // Тело класса
}
```

Наследование базового класса (абстрактного или обычного) и реализация интерфейсов могут использоваться одновременно. В этом случае в описании класса после имени класса и двоеточия сначала указывается имя базового класса, а затем через запятую перечисляются реализуемые в классе интерфейсы:

```
class имя:базовый_класс,интерфейс,интерфейс,...,интерфейс{
```

```
   // Тело класса
}
```

Если так, то в классе должны быть описаны все методы, свойства и индексаторы, объявленные в реализуемых интерфейсах, а если наследуемый базовый класс абстрактный — то и все абстрактные методы из базового класса.

Для начала мы рассмотрим небольшой пример, в котором описывается интерфейс, а затем этот интерфейс реализуется в классе. Обратимся к программе в листинге 1.3.

Листинг 1.3. Знакомство с интерфейсами

```
using System;
// Интерфейс:
interface MyInterface{
   // Объявление методов:
   void show();
   void setNum(int n);
   int getNum();
   // Объявление свойства:
   int number{
      get;
      set;
   }
   // Объявление индексатора:
   int this[int k]{
      get;
   }
}
// Класс реализует интерфейс:
class MyClass:MyInterface{
   // Закрытое целочисленное поле:
   private int num;
   // Конструктор с одним аргументом:
```

```
public MyClass(int n){
   // Присваивание значения свойству:
   number=n;
   // Вызывается метод из интерфейса:
   show();
}
// Описание метода из интерфейса:
public void show(){
   // Отображение значения свойства:
   Console.WriteLine("Свойство number="+number);
}
// Описание метода из интерфейса:
public void setNum(int n){
   // Присваивание значения полю:
   num=n;
}
// Описание метода из интерфейса:
public int getNum(){
   // Значение поля:
   return num;
}
// Описание свойства из интерфейса:
public int number{
   // Аксессор для считывания значения:
   get{
      // Вызывается метод из интерфейса:
      return getNum();
   }
   // Аксессор для присваивания значения:
   set{
      // Вызывается метод из интерфейса:
      setNum(value);
```

```
      }
   }
   // Описание индексатора из интерфейса:
   public int this[int k]{
      // Аксессор для считывания значения:
      get{
         // Локальная переменная:
         int r=number;
         // "Отбрасывание" цифр в десятичном представлении числа:
         for(int i=0;i<k;i++){
            r/=10;
         }
         // Результат:
         return r%10;
      }
   }
}
// Класс с главным методом:
class InterfaceDemo{
   // Главный метод:
   static void Main(){
      // Целочисленная переменная:
      int m=9;
      // Создание объекта:
      MyClass obj=new MyClass(12345);
      // Индексирование объекта:
      for(int i=0;i<=m;i++){
         Console.Write("|"+obj[m-i]);
      }
      Console.WriteLine("|");
   }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 1.3)

```
Свойство number=12345
|0|0|0|0|0|1|2|3|4|5|
```

В программе мы описали интерфейс с названием `MyInterface`. В этом интерфейсе объявлены три метода (метод `show()` без аргументов и не возвращающий результат; метод `getNum()` без аргументов с целочисленным результатом; и метод `setNum()` с целочисленным аргументом и не возвращающий результат), целочисленное свойство `number` (доступное для считывания и записи) и индексатор с целочисленным индексом (доступен для считывания).

ПОДРОБНОСТИ

То, что мы объявили в интерфейсе индексатор только с `get`-аксессором, означает, что при реализации индексатора в классе для индексатора должен быть описан `get`-аксессор. Но мы можем описать для индексатора и `set`-аксессор. А вот если бы мы определяли индексатор, унаследованный из базового абстрактного класса, то описывать нужно было бы только те аксессоры, которые «заявлены» для индексатора в абстрактном классе.

Класс `MyClass` реализует интерфейс `MyInterface`. В классе есть закрытое целочисленное поле `num`. Метод `setNum()` описан так, что переданный методу аргумент присваивается значением полю `num`. Метод `getNum()` результатом возвращает значение `num`.

НА ЗАМЕТКУ

Методы, объявленные в интерфейсе, по умолчанию считаются открытыми. Поэтому в классе они описываются с ключевым словом `public`. Это же замечание относится к свойствам и индексаторам.

Целочисленное свойство `number` описано так, что при считывании значения свойства вызывается метод `getNum()`, который в свою очередь возвращает результатом значение поля `num`. При присваивании значения свойству `number` вызывается метод `setNum()`, аргументом которому передается присваиваемое свойству значение (определяется параметром `value`). В результате это значение присваивается полю `num`.

Метод `show()` описан таким образом, что при его вызове отображается значение свойства `number`.

В конструкторе класса сначала командой `number=n` свойству `number` присваивается в качестве значения аргумент `n` конструктора. После этого вызывается метод `show()`. Таким образом, при создании объекта класса `MyClass` в консольном окне появляется сообщение с информацией о значении свойства `number`.

Индексатор в классе описан таким образом, что при индексировании объекта целочисленным индексом результатом возвращается цифра в десятичном представлении числа. Число в данном случае — это значение свойства `number`, и оно же — значение поля `num`. Индекс определяет позицию (разряд), на которой находится возвращаемая цифра в числе. Для вычисления результата в теле `get`-аксессора целочисленной переменной `r` присваивается значение свойства `number`, после чего запускается оператор цикла, в котором на каждой итерации командой `r/=10` в десятичном представлении числа, записанного в переменную `r`, отбрасывается последняя цифра. Количество итераций цикла определяется значением индекса `k`. В итоге цифра, которая нас интересует, окажется на последней позиции. Эту цифру мы вычисляем командой `r%10` как остаток от деления на `10`.

В главном методе программы командой `MyClass obj=new MyClass(12345)` создается объект `obj` класса `MyClass`. При этом через цепочки вызовов задействованы практически все методы и свойство `number`, которые описаны в классе `MyClass`. Пример индексирования объекта представлен в операторе цикла, с помощью которого отображается десятичное представление числа, являющегося значением поля `num` объекта `obj`. Количество отображаемых разрядов превышает разрядную длину числа, поэтому старшие разряды заполнены нулями.

Еще один пример, который мы рассмотрим далее, показывает, как реализовать в классе несколько интерфейсов при условии наследования базового класса (абстрактного). Интересующий нас программный код представлен в листинге 1.4.

Листинг 1.4. Реализация интерфейсов и наследование базового класса

```
using System;
// Базовый абстрактный класс:
abstract class Base{
```

```
   // Объявление абстрактного свойства:
   public abstract int number{
      get;
      set;
   }
   // Конструктор с одним аргументом:
   public Base(int n){
      // Присваивание значения свойству:
      number=n;
      // Вызывается обычный (не абстрактный) метод:
      show();
   }
   // Описание обычного (не абстрактного) метода:
   public void show(){
      // Отображение значения свойства:
      Console.WriteLine("Свойство number="+number);
   }
}
// Первый интерфейс:
interface First{
   // Объявление методов:
   void setNum(int n);
   int getNum();
}
// Второй интерфейс:
interface Second{
   // Объявление индексатора:
   int this[int k]{
      get;
   }
}
// Класс реализует интерфейс:
```

```
class MyClass:Base,First,Second{
   // Закрытое целочисленное поле:
   private int num;
   // Конструктор с одним аргументом:
   public MyClass(int n):base(n){}
   // Описание метода из интерфейса First:
   public void setNum(int n){
      // Присваивание значения полю:
      num=n;
   }
   // Описание метода из интерфейса First:
   public int getNum(){
      // Значение поля:
      return num;
   }
   // Описание свойства из класса Base:
   public override int number{
      // Аксессор для считывания значения:
      get{
         // Вызывается метод из интерфейса First:
         return getNum();
      }
      // Аксессор для присваивания значения:
      set{
         // Вызывается метод из интерфейса First:
         setNum(value);
      }
   }
   // Описание индексатора из интерфейса Second:
   public int this[int k]{
      // Аксессор для считывания значения:
      get{
```

```
        // Локальная переменная:
        int r=number;
        // "Отбрасывание" цифр в десятичном
        // представлении числа:
        for(int i=0;i<k;i++){
           r/=10;
        }
        // Результат:
        return r%10;
      }
   }
}
// Класс с главным методом:
class MoreInterfaceDemo{
   static void Main(){
      int m=9;
      MyClass obj=new MyClass(12345);
      for(int i=0;i<=m;i++){
         Console.Write("|"+obj[m-i]);
      }
      Console.WriteLine("|");
   }
}
```

Ниже показано, как выглядит результат выполнения программы:

Результат выполнения программы (из листинга 1.4)

```
Свойство number=12345
|0|0|0|0|0|1|2|3|4|5|
```

Результат выполнения программы такой же, как и в предыдущем случае. Это неудивительно, поскольку представленный выше пример является

вариацией предыдущего. Мы просто немного иначе организовали программный код. Если раньше у нас был один интерфейс `MyInterface`, на основе которого создавался класс `MyClass`, то теперь имеется абстрактный класс `Base`, на основе которого путем наследования создается класс `MyClass`. При этом класс `MyClass` реализует интерфейсы `First` и `Second`.

В классе `Base` объявляется абстрактное свойство `number`, причем поскольку теперь свойство объявляется не в интерфейсе, а в абстрактном классе, то мы используем в объявлении ключевое слово `abstract`. Также мы явно указали спецификатор уровня доступа `public` для свойства. При описании свойства в классе `MyClass` использовано ключевое слово `override`.

В классе `Base` описан конструктор с целочисленным аргументом. Этот аргумент присваивается значением свойству `number`. После присваивания вызывается метод `show()`. Последний также описан в классе `Base`, причем хотя метод обычный (не абстрактный), но в теле метода (как и в теле конструктора) выполняется обращение к абстрактному свойству `number`.

В интерфейсе `First` объявлены методы `setNum()` и `getNum()`. В интерфейсе `Second` объявлен индексатор с целочисленным индексом. Индексатор должен возвращать целое число, и у него должен быть `get`-аксессор.

Как отмечалось выше, класс `MyClass` наследует абстрактный класс `Base` и реализует интерфейсы `First` и `Second`. Поэтому в классе `MyClass` описываются свойство `number` из класса `Base`, методы `getNum()` и `setNum()` из интерфейса `First`, а также индексатор из интерфейса `Second`. Соответствующий код должен быть понятен читателю.

ПОДРОБНОСТИ

Как и в предыдущем примере, класс `MyClass` содержит закрытое целочисленное поле `num`. Конструктор класса `MyClass` описан таким образом, что переданный конструктору целочисленный аргумент далее передается конструктору базового класса. Никаких дополнительных действий не выполняется, поэтому тело конструктора представляет собой пустой блок.

В главном методе содержатся те же команды, что и в предыдущем примере.

Наследование интерфейсов

— Но позвольте узнать, как же я вас пропишу? У вас же нет ни имени, ни фамилии.
— Это вы несправедливо. Имя я себе совершенно спокойно могу избрать. Пропечатал в газете — и шабаш.

из к/ф «Собачье сердце»

Один интерфейс может наследовать другой интерфейс или даже несколько интерфейсов. Технически наследование интерфейсов реализуется так: в описании интерфейса, наследующего другие интерфейсы, после его имени через двоеточие указываются наследуемые интерфейсы. Общий шаблон наследования интерфейсов выглядит следующим образом:

```
interface имя:интерфейс,интерфейс,...,интерфейс{
   // Тело интерфейса
}
```

Если интерфейс `Alpha` наследует интерфейс `Bravo`, то все, что объявлено в интерфейсе `Bravo`, автоматически включается в состав интерфейса `Alpha`. Поэтому если некоторый класс `MyClass` реализует интерфейс `Alpha`, который наследует интерфейс `Bravo`, то фактически в этом классе должно быть описано все, что явно объявлено в интерфейсе `Alpha`, и все, что описано в интерфейсе `Bravo` (поскольку содержимое интерфейса `Bravo` по факту содержится и в интерфейсе `Alpha`).

НА ЗАМЕТКУ

Нередко вместо термина «наследование интерфейсов» для обозначения ситуации, когда интерфейс создается на основе одного или более уже существующих интерфейсов, используют термин «расширение интерфейсов».

Программа, в которой используется наследование интерфейсов, представлена в листинге 1.5. Это еще одна вариация на тему предыдущих двух примеров (для сокращения объема программного кода некоторые комментарии удалены).

Листинг 1.5. Наследование интерфейсов

```
using System;
// Первый интерфейс:
interface First{
   // Свойство:
   int number{
      get;
      set;
   }
   // Индексатор:
   int this[int k]{
      get;
   }
}
// Второй интерфейс:
interface Second{
   // Методы:
   void setNum(int n);
   int getNum();
}
// Интерфейс наследует другие интерфейсы:
interface MyInterface{
   // Метод:
   void show();
}
// Класс реализует интерфейс:
class MyClass:MyInterface{
   private int num;
   public MyClass(int n){
      number=n;
      show();
```

```
   }
   public void show(){
      Console.WriteLine("Свойство number="+number);
   }
   public void setNum(int n){
      num=n;
   }
   public int getNum(){
      return num;
   }
   public int number{
      get{
         return getNum();
      }
      set{
         setNum(value);
      }
   }
   public int this[int k]{
      get{
         int r=number;
         for(int i=0;i<k;i++){
            r/=10;
         }
         return r%10;
      }
   }
}
// Класс с главным методом:
class InterfaceDemo{
   static void Main(){
      int m=9;
```

```
      MyClass obj=new MyClass(12345);
      for(int i=0;i<=m;i++){
         Console.Write("|"+obj[m-i]);
      }
      Console.WriteLine("|");
   }
}
```

Результат выполнения программы такой же, как и в предыдущих случаях:

Результат выполнения программы (из листинга 1.5)

```
Свойство number=12345
|0|0|0|0|0|1|2|3|4|5|
```

Описание класса `MyClass` не изменилось (как и команды в главном методе). Класс `MyClass` реализует интерфейс `MyInterface`. Но только теперь сам интерфейс `MyInterface` наследует интерфейсы `First` и `Second`. В интерфейсе `First` объявлено свойство `number` и индексатор. В интерфейсе `Second` объявлены методы `setNum()` и `getNum()`. Непосредственно в интерфейсе `MyInterface` объявлен метод `show()`. Но поскольку интерфейс `MyInterface` наследует интерфейсы `First` и `Second`, то содержимое этих интерфейсов «незримо присутствует» в интерфейсе `MyInterface`. Поэтому класс, реализующий интерфейс `MyInterface`, должен содержать описание не только тех методов, свойств и индексаторов, которые объявлены непосредственно в интерфейсе `MyInterface`, но и описание всего, что объявлено в интерфейсах, наследуемых в интерфейсе `MyInterface`.

Интерфейсные переменные

Так вам и надо. Ведь знали ж, кто он такой.

из к/ф «Собачье сердце»

Хотя интерфейс напоминает класс, но на основе интерфейса объект создать нельзя (хочется верить, что не нужно объяснять, почему этого нельзя сделать). Но существует такое понятие, как *интерфейсная*

переменная. Интерфейсная переменная — это переменная, типом которой указано название интерфейса. Особенность и «сила» интерфейсной переменной в том, что интерфейсная переменная может ссылаться на объект любого класса, реализующего данный интерфейс (указанный типом интерфейсной переменной). То есть если имеется класс, который реализует некоторый интерфейс, и мы создали объект такого класса, то ссылку на данный объект можно записать не только в объектную переменную, типом которой указан интерфейс. Правда здесь имеется важное ограничение: через интерфейсную переменную можно получить доступ только к тем методам, свойствам и индексаторам (через индексирование объекта), которые объявлены в интерфейсе.

НА ЗАМЕТКУ

Напомним, что, кроме методов, свойств и индексаторов, в интерфейсе могут объявляться события. С событиями мы познакомимся позже, но то, что мы обсуждаем для методов, свойств и индексаторов как содержимого интерфейсов, справедливо и для событий.

Специфика интерфейсных переменных несколько напоминает работу с объектными переменными базовых классов: объектная переменная базового класса может ссылаться на объект производного класса, и через такую переменную можно получить доступ только к тем членам, которые объявлены в базовом классе.

Рассмотрим пример, в котором используется указанная особенность интерфейсных переменных. Проанализируем программу, представленную в листинге 1.6.

Листинг 1.6. Интерфейсные переменные

```
using System;
// Интерфейс:
interface MyInterface{
   // Объявление метода:
   char getChar(int n);
   // Объявление индексатора:
   char this[int k]{
      get;
   }
```

```
}
// Первый класс, реализующий интерфейс:
class Alpha:MyInterface{
   // Закрытое символьное поле:
   private char symb;
   // Конструктор с символьным аргументом:
   public Alpha(char s){
      // Полю присваивается значение:
      symb=s;
   }
   // Описание метода:
   public char getChar(int n){
      // Результат:
      return (char)(symb+n);
   }
   // Описание индексатора:
   public char this[int k]{
      // Аксессор для считывания значения:
      get{
         // Результат:
         return getChar(k);
      }
   }
}
// Второй класс, реализующий интерфейс:
class Bravo:MyInterface{
   // Закрытое текстовое поле:
   private string text;
   // Конструктор с текстовым аргументом:
   public Bravo(string t){
      // Полю присваивается значение:
      text=t;
```

```
   }
   // Описание метода:
   public char getChar(int k){
      return text[k%text.Length];
   }
   // Описание индексатора:
   public char this[int k]{
      // Аксессор для считывания значения:
      get{
         // Результат:
         return getChar(k);
      }
   }
}
// Класс с главным методом:
class InterfaceVarDemo{
   // Главный метод:
   static void Main(){
      // Целочисленная переменная:
      int m=5;
      // Интерфейсная переменная:
      MyInterface R;
      // Создается объект класса Alpha и ссылка на него
      // записывается в интерфейсную переменную:
      R=new Alpha('F');
      // Вызов метода через интерфейсную переменную:
      Console.WriteLine("Символы от \'{0}\' до \'{1}\':",R.getChar(-m),R.getChar(m));
      // Индексирование объекта через
      // интерфейсную переменную:
      for(int i=-m;i<=m;i++){
         Console.Write("|"+R[i]);
      }
```

```
      Console.WriteLine("|");
      // Создается объект класса Bravo и ссылка на него
      // записывается в интерфейсную переменную:
      R=new Bravo("bravo");
      // Вызов метода через интерфейсную переменную:
      Console.WriteLine("Символы от \'{0}\' до \'{1}\':",R.getChar(0),R.getChar(2*m+1));
      // Индексирование объекта через
      // интерфейсную переменную:
      for(int i=0;i<=2*m+1;i++){
         Console.Write("|"+R[i]);
      }
      Console.WriteLine("|");
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 1.6)

```
Символы от 'A' до 'K':
|A|B|C|D|E|F|G|H|I|J|K|
Символы от 'b' до 'r':
|b|r|a|v|o|b|r|a|v|o|b|r|
```

В программе описан интерфейс `MyInterface`. В этом интерфейсе объявляется метод `getChar()` с целочисленным аргументом и символьным результатом, а также объявлен индексатор с целочисленным индексом и символьным значением. Интерфейс `MyInterface` реализуется в классах `Alpha` и `Bravo`. В классе `Alpha` имеется закрытое символьное поле `symb` и конструктор с одним аргументом (определяет значение символьного поля). Метод `getChar()` описан в классе так, что результатом возвращается значение выражения `(char)(symb+n)`. Это значение вычисляется следующим образом: к коду символа из поля `symb` прибавляется целочисленное значение аргумента метода `n`, и полученное целое число преобразуется к символьному типу. Индексатор определен

так, что при заданном индексе k результатом возвращается значение выражения getChar(k). То есть при индексировании объекта с индексом k и вызове метода getChar() с аргументом k по определению получаем один и тот же результат. Стоит сразу отметить, что в классе Bravo индексатор определен так же — на основе метода getChar(). Этот класс тоже реализует интерфейс MyInterface. В классе есть закрытое текстовое поле text и конструктор с текстовым аргументом, определяющим значение текстового поля. Метод getChar() определен таким образом, что при аргументе k в качестве результата возвращается значение выражения text[k%text.Length]. Это символ из текстового поля text с индексом k (с учетом циклической перестановки индекса, если он выходит за верхнюю допустимую границу).

ПОДРОБНОСТИ

Значением выражения k%text.Length является значение k, если значение k меньше значения выражения text.Length (количество символов в тексте). Вообще же значение выражения k%text.Length — это остаток от деления значения k на значение text.Length.

Как отмечалось выше, для индексатора с индексом k результатом возвращается значение getChar(k).

НА ЗАМЕТКУ

Метод getChar() в классе Alpha определен так, что объект этого класса можно индексировать, помимо прочего, и отрицательными индексами. В классе Bravo метод getChar() определен таким образом, что индексировать объект класса можно только неотрицательными индексами.

В главном методе командой MyInterface R объявляется интерфейсная переменная R типа MyInterface. Командой R=new Alpha('F') создается объект класса Alpha, и ссылка на него записывается в интерфейсную переменную R. Так можно делать, поскольку класс Alpha реализует интерфейс MyInterface. При значении -5 целочисленной переменной m выполняется оператор цикла, в котором индексная переменная i пробегает значения от -m до m, отображаются значение, выражения R[i] с проиндексированным объектом. Получаем последовательность символов, начиная с символа за 5 позиций до символа 'F' (аргумент конструктора класса Alpha при создании объекта) до символа через

5 позиций после 'F'. Первый и последний символы в этой последовательности можно вычислить с помощью выражений R.getChar(-m) и R.getChar(m), в которых через интерфейсную переменную вызывается метод объекта, на который переменная ссылается.

Похожие операции выполняются с объектом класса Bravo. Этот объект создается командой R=new Bravo("bravo"), а ссылка на него записывается в интерфейсную переменную R. Как и в предыдущем случае, когда переменная R ссылалась на объект класса Alpha, мы можем вызывать через переменную метод getChar() (команды R.getChar(0) и R.getChar(2*m+1)), а также можем индексировать объект, на который ссылается переменная (команда вида R[i]). При этом необходимо учитывать, что перебор символов выполняется в тексте "bravo" (аргумент, переданный конструктору класса Bravo при создании объекта) с использованием принципа циклической перестановки индекса, если он выходит за верхнюю допустимую границу.

НА ЗАМЕТКУ

При вызове метода через интерфейсную переменную версия метода определяется по объекту, из которого вызывается метод. Собственно, других, даже теоретических, вариантов здесь нет, поскольку в интерфейсе метод только объявляется, но не описывается.

Явная реализация членов интерфейса

— Вот у вас, на Земле, как вы определяете — кто перед кем сколько должен присесть?
— Ну, это на глаз...

из к/ф «Кин-дза-дза»

Если класс реализует несколько интерфейсов, то может сложиться ситуация, когда в нескольких интерфейсах объявлен один и тот же метод, свойство или индексатор. В таком случае в классе, реализующем интерфейсы, достаточно один раз описать соответствующий член. В следующем примере класс реализует два интерфейса, которые содержат объявление одинаковых методов. Код упрощен, чтобы легче было понять суть. Программа представлена в листинге 1.7.

Листинг 1.7. Интерфейсы и неявная реализация членов

```
using System;
// Первый интерфейс:
interface First{
   void show();
}
// Второй интерфейс:
interface Second{
   void show();
}
// Класс реализует интерфейсы:
class MyClass:First,Second{
   // Описание метода:
   public void show(){
      Console.WriteLine("Объект класса MyClass");
   }
}
// Класс с главным методом:
class ImplInterfaceDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
      MyClass obj=new MyClass();
      // Ссылка на объект записывается
      // в интерфейсные переменные:
      First A=obj;
      Second B=obj;
      // Вызовы метода через разные переменные:
      obj.show();
      A.show();
```

```
        B.show();
    }
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 1.7)

```
Объект класса MyClass
Объект класса MyClass
Объект класса MyClass
```

В программе описаны интерфейсы `First` и `Second`, в каждом из которых объявлен метод `show()` без аргументов, не возвращающий результат. Класс `MyClass` реализует оба эти интерфейса, но метод `show()` в классе описан только один раз (при вызове метод выводит сообщение в консоль). В главном методе программы создается объект класса `MyClass` и в три переменные (объектная `obj` класса `MyClass`, интерфейсная `A` типа `First` и интерфейсная переменная `B` типа `Second`) записывается ссылка на него. Затем через каждую из трех переменных вызывается метод `show()`. Вполне ожидаемо результат во всех трех случаях один и тот же.

ПОДРОБНОСТИ

Если бы в рассмотренном выше примере кроме интерфейсов `First` и `Second` был еще и абстрактный базовый класс `Base`, в котором объявлен абстрактный метод `show()`, то ситуация разрешалась бы похожим образом. В классе `MyClass`, наследующем класс `Base` и реализующем интерфейсы `First` и `Second`, следовало бы описать метод `show()`, но в описании метода необходимо использовать ключевое слово `override`.

Но у нас есть и альтернатива. Мы можем воспользоваться *явной реализацией методов, свойств и индексаторов интерфейса*. При явной реализации, когда соответствующий член описывается в классе, перед его именем (через точку) указывается название интерфейса, а спецификатор уровня доступа не указывается. Доступ к такому члену класса можно получить через переменную соответствующего интерфейса. Дальше обратимся к примеру. Программа представлена в листинге 1.8.

Листинг 1.8. Интерфейсы и явная реализация членов

```
using System;
// Базовый класс:
abstract class Base{
   // Абстрактное свойство:
   public abstract char symbol{
      get;
   }
   // Абстрактный индексатор:
   public abstract int this[int k]{
      get;
   }
   // Абстрактный метод:
   public abstract void show();
}
// Первый интерфейс:
interface First{
   // Свойство:
   char symbol{
      get;
   }
   // Индексатор:
   int this[int k]{
      get;
   }
   // Метод:
   void show();
}
// Второй интерфейс:
interface Second{
   // Свойство:
   char symbol{
```

```
      get;
   }
   // Индексатор:
   int this[int k]{
      get;
   }
   // Метод:
   void show();
}
// Производный класс наследует абстрактный базовый класс
// и реализует интерфейсы:
class MyClass:Base,First,Second{
   // Закрытое символьное поле:
   private char smb;
   // Конструктор с символьным аргументом:
   public MyClass(char s):base(){
      smb=s;
   }
   // Описание свойства из абстрактного класса:
   public override char symbol{
      get{
         return smb;
      }
   }
   // Явная реализация свойства из интерфейса First:
   char First.symbol{
      get{
         return (char)(smb+1);
      }
   }
   // Описание индексатора из базового класса:
   public override int this[int k]{
```

```
      get{
         return smb+k;
      }
   }
   // Явная реализация индексатора из интерфейса Second:
   int Second.this[int k]{
      get{
         return smb-k;
      }
   }
   // Описание метода из базового класса:
   public override void show(){
      Console.WriteLine("Базовый класс Base:\t\'{0}\'",symbol);
   }
   // Явная реализация метода из интерфейса First:
   void First.show(){
      Console.WriteLine("Интерфейс First:\t\'{0}\'",symbol);
   }
   // Явная реализация метода из интерфейса Second:
   void Second.show(){
      Console.WriteLine("Интерфейс Second:\t\'{0}\'",symbol);
   }
}
// Класс с главным методом:
class ExplInterfaceDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
      MyClass obj=new MyClass('A');
      // Интерфейсные переменные:
      First A=obj;
      Second B=obj;
```

```
      // Вызов метода через переменные:
      obj.show();
      A.show();
      B.show();
      // Считывание значения свойства:
      Console.WriteLine("obj.symbol=\'{0}\'",obj.symbol);
      Console.WriteLine("  A.symbol=\'{0}\'",A.symbol);
      Console.WriteLine("  B.symbol=\'{0}\'",B.symbol);
      // Индексирование объекта:
      Console.WriteLine("obj[10]={0}",obj[10]);
      Console.WriteLine("  A[10]={0}",A[10]);
      Console.WriteLine("  B[10]={0}",B[10]);
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 1.8)

```
Базовый класс Base:     'A'
Интерфейс First:        'A'
Интерфейс Second:       'A'
obj.symbol='A'
  A.symbol='B'
  B.symbol='A'
obj[10]=75
  A[10]=75
  B[10]=55
```

Интерфейсы `First` и `Second` описаны одинаково. Отличает их только название. В каждом из интерфейсов объявлено символьное свойство `symbol` с `get`-аксессором, индексатор с целочисленным индексом и `get`-аксессором и метод `show()` — без аргументов и не возвращающий результат. Такая же «начинка» и у абстрактного класса `Base`, но только с поправкой на использование в описании соответствующих членов

ключевых слов public и abstract. Класс MyClass наследует класс Base и реализует интерфейсы First и Second. В этом классе появляется закрытое символьное поле smb и конструктор с одним символьным аргументом, который определяет значение поля. Но нас, конечно, в первую очередь интересует тот способ, которым описывается метод show(), свойство symbol и индексатор.

Свойство symbol описано дважды. Одно из описаний свойства — это обычное описание унаследованного из абстрактного класса свойства. При запросе значения свойства результатом возвращается значение поля smb. Также в классе есть явная специализация для свойства из интерфейса First. Данная версия описана без ключевого слова public, а название свойства указано в виде First.symbol. Значение свойства вычисляется как (char)(smb+1). Это следующий символ после символа, записанного в поле smb (но значение самого поля smb при этом не меняется). Такая версия свойства будет задействована, если мы будем обращаться к объекту класса MyClass через интерфейсную переменную типа First. Если мы будем получать доступ к объекту через объектную переменную класса MyClass или интерфейсную переменную типа Second, то используется первая версия свойства.

У индексатора также две версии. Одна описана как свойство, переопределяемое в производном классе. При запросе значения выражения с проиндексированным объектом при заданном индексе k результатом возвращается значение smb+k (сумма кода символа из поля smb и индекса k). Явная реализация индексатора выполняется для интерфейса Second. Версия описывается без ключевого слова public, а вместо ключевого слова this используется конструкция Second.this. Запрос значения выражения с проиндексированным объектом вычисляется как разность кода символа из поля smb и индекса k (выражение smb-k). Явная реализация индексатора будет задействована, если мы станем индексировать объект через интерфейсную переменную типа Second. При индексировании объекта через объектную переменную класса MyClass или через интерфейсную переменную типа First в игру вступает первая версия индексатора.

У метода show() в классе MyClass есть три версии. Версия с ключевым словом override вызывается через объектную переменную базового класса. Явная реализация метода из интерфейса First задействуется при обращении к объекту через переменную интерфейсного типа

First. При обращении к объекту через интерфейсную переменную типа Second используется явная реализация метода для интерфейса Second. Каждая версия метода отображает значение свойства symbol объекта и название класса или интерфейса, для которого выполнена реализация метода.

ПОДРОБНОСТИ

Вообще общая концепция использования явных реализаций методов, свойств и индексаторов интерфейса немного другая. Мы можем для интерфейса выполнить явную реализацию метода, свойства, индексатора. И если обращение к объекту выполняется через интерфейсную переменную соответствующего типа, то используется данная явная реализация для метода, свойства, индексатора. Во всех прочих случаях используются «обычные» реализации (или неявные реализации — для них не указывается имя интерфейса). Что касается метода show(), то для него определены явные реализации для интерфейсов First и Second. Первая используется, если доступ к объекту выполняется через ссылку типа First, а вторая — при использовании переменной типа Second. Во всех прочих случаях используется версия, переопределяющая абстрактный метод из класса Base. Для нас «все прочие случаи» — это обращение к объекту через объектную переменную класса MyClass.

В главном методе командой MyClass obj=new MyClass('A') создается объект класса MyClass. Командами First A=obj и Second B=obj ссылка на этот объект записывается в интерфейсные переменные A и B. Переменные A, B и obj мы используем для вызова метода show() (команды obj.show(), A.show() и B.show()), считывания значения свойства symbol (инструкции obj.symbol, A.symbol и B.symbol) и индексирования объектов (инструкции obj[10], A[10] и B[10]). Верится, что особых пояснений результат выполнения программы не требует. Но есть одно обстоятельство, связанное с методом show(). В этом методе выполняется обращение к свойству symbol. Но через какую бы переменную мы ни вызывали метод, всегда используется «общая» версия свойства, поэтому во всех трех случаях значение свойства равно 'A' (значение аргумента, переданного конструктору класса MyClass при создании объекта). А вот когда мы обращаемся к свойству через переменную A, то получаем значение 'B', поскольку здесь задействована явная реализация для данного свойства.

Резюме

Не надо булочной. Не надо справочной.

из к/ф «Кин-дза-дза»

- Абстрактным называется метод, который не содержит описания (у него нет тела с командами, выполняемыми при вызове метода). Абстрактный метод описывается с ключевым словом `abstract`. Если в классе есть хотя бы один абстрактный метод, то такой класс является абстрактным. Абстрактный класс также описывается с ключевым словом `abstract`.
- На основе абстрактного класса нельзя создать объект (но можно объявить объектную переменную абстрактного класса). Абстрактный класс используется как базовый для создания производных классов. В производном классе (если он не является абстрактным) должны быть описаны (переопределены) все абстрактные методы из абстрактного базового класса.
- Абстрактные методы по умолчанию являются виртуальными. Ключевое слово `virtual` в объявлении абстрактных методов не используется. При описании (переопределении) абстрактных методов в производном классе используется ключевое слово `override`. Абстрактный метод не может быть статическим.
- Свойства и индексаторы также могут быть абстрактными. При объявлении абстрактного свойства или индексатора используется ключевое слово `abstract`. В теле абстрактного свойства или индексатора указываются ключевые слова `get` и `set` (без тела с выполняемыми командами). Если у свойства или индексатора должен быть только один аксессор, то указывается только одно из ключевых слов `get` или `set` (в зависимости от того, какой аксессор должен быть у свойства или индексатора). При описании (переопределении) таких свойств и индексаторов указывается ключевое слово `override`.
- Интерфейс представляет собой блок из объявления методов, индексаторов, свойств (и событий). Описание интерфейса начинается с ключевого слова `interface`, после которого указывается название интерфейса. В блоке из фигурных скобок выполняется объявление методов, свойств и индексаторов. Объявление выполняется так же, как и объявление соответствующих абстрактных членов абстрактного класса, но при этом ключевые слова `public` и `abstract`

не используются. Все, что описано в интерфейсе, имеет уровень доступа `public`.

- Класс может реализовать один или несколько интерфейсов. Интерфейсы, реализуемые в классе, указываются в описании класса после его имени (через двоеточие). Названия интерфейсов разделяются запятыми. Если при этом класс еще и наследует базовый класс, то перед списком реализуемых интерфейсов указывается имя базового класса.
- Если класс реализует интерфейс, то в классе должны быть описаны все методы, свойства и индексаторы, объявленные в интерфейсе. Соответствующие члены класса описываются с ключевыми словами `public` и `override`.
- На основе интерфейса нельзя создать объект, но можно объявить переменную интерфейсного типа. Интерфейсная переменная может ссылаться на объект любого класса, реализующего данный интерфейс. Доступ через такую переменную возможен только к тем членам объекта, которые объявлены в интерфейсе.
- Для методов, свойств и индексаторов из интерфейса в классе, реализующем интерфейс, можно выполнять явную реализацию. Соответствующий член класса описывается без ключевого слова `public`, а перед его названием через точку указывается имя интерфейса. Доступ к такому члену можно получить через интерфейсную переменную соответствующего интерфейса.

Задания для самостоятельной работы

> Дядя Степан, помог бы ты им, а? Ну грех смеяться над убогими.
>
> *из к/ф «Формула любви»*

1. Напишите программу, содержащую абстрактный базовый класс с защищенным полем, являющимся ссылкой на целочисленный массив. У класса должен быть конструктор с одним аргументом (определяет размер массива и создает его), целочисленное свойство (значение — размер массива), абстрактный метод (без аргументов, не возвращает результат) и индексатор с целочисленным индексом (доступен для чтения

и записи). В производном классе описать абстрактный метод из базового класса, чтобы он отображал в консоли содержимое массива. Индексатор определить так, чтобы с его помощью можно было прочитать значение элемента массива и присвоить значение элементу массива.

2. Напишите программу, содержащую базовый класс с защищенным текстовым полем. У класса должен быть конструктор с текстовым аргументом, доступное для чтения абстрактное целочисленное свойство, доступный для чтения абстрактный индексатор с целочисленным индексом, не возвращающий результат абстрактный метод с текстовым аргументом, не возвращающий результат абстрактный метод без аргументов. В производном классе: свойство результатом возвращает количество символов в текстовом поле, значением выражения с проиндексированным объектом является код символа в тексте, метод с текстовым аргументом присваивает новое значение полю, а метод без аргументов отображает значение поля в консольном окне.

3. Напишите программу, в которой будет описан интерфейс с методом без аргументов, который возвращает результатом целое число. На основе интерфейса создайте два класса. У каждого класса должно быть целочисленное поле. В первом классе метод результатом возвращает сумму четных чисел, во втором классе метод возвращает результатом сумму нечетных чисел. Количество слагаемых в сумме определяется полем объекта, из которого вызывается метод. Проверьте работу метода, получив доступ к объекту класса через объектную переменную и через интерфейсную переменную.

4. Напишите программу, содержащую абстрактный класс с двумя защищенными целочисленными полями и конструктор с двумя целочисленными аргументами. В классе должен быть объявлен абстрактный индексатор с целочисленным индексом. Опишите интерфейс, в котором есть метод с целочисленным аргументом и целочисленным результатом. Опишите класс, который наследует абстрактный базовый класс и реализует интерфейс. В этом классе опишите индексатор так, чтобы при четном индексе выполнялось обращение к первому полю, а при нечетном индексе обращение выполнялось ко второму полю. Метод следует описать таким образом, чтобы он результатом возвращал сумму значений полей, умноженную на аргумент метода.

5. Напишите программу, содержащую два интерфейса. В первом интерфейсе опишите метод с символьным аргументом и целочисленным результатом, а во втором интерфейсе — метод с целочисленным

аргументом и символьным результатом. Опишите класс, реализующий оба интерфейса. Проверьте работу методов, вызвав их через объектную переменную и через интерфейсные переменные (там, где это возможно).

6. Напишите программу, в которой на основе двух интерфейсов создается класс. В одном интерфейсе объявлен индексатор с символьным индексом, возвращающий целочисленное значение. В другом интерфейсе объявлен индексатор с целочисленным индексом, возвращающий символьное значение.

7. Напишите программу, в которой класс создается на основе двух интерфейсов. В первом интерфейсе есть целочисленное свойство, доступное для чтения и записи. Во втором индексаторе есть текстовое свойство, доступное для чтения и записи. В каждом из интерфейсов объявлены одинаковые (с одинаковыми именами) методы, без аргументов, не возвращающие результат. В классе описать соответствующий метод, который в консольном окне отображает значения свойств.

8. Напишите программу, в которой класс создается на основе двух интерфейсов. В каждом из интерфейсов объявлено текстовое свойство с одним и тем же названием, доступное только для чтения. В классе выполнить общую (неявную) реализацию свойства, а также явную специализацию свойства для каждого из интерфейсов. Проверьте значение свойства для объекта класса, выполнив ссылку на объект через объектную переменную и интерфейсные переменные.

9. Напишите программу, в которой класс создается на основе абстрактного базового класса и интерфейса. В абстрактном классе есть поле, являющееся ссылкой на защищенный символьный массив. Конструктору класса передается текстовый аргумент, на основе которого создается и заполняется символьный массив. В абстрактном классе опишите метод, который по целочисленному аргументу возвращает значение символа с соответствующим индексом в массиве. Также в абстрактном классе должен быть объявлен абстрактный метод с двумя аргументами (целое число и символ), не возвращающий результат. В интерфейсе должен быть объявлен метод с таким же именем, но с одним текстовым аргументом. Также в интерфейсе должен быть объявлен индексатор (с двумя аксессорами) с символьным результатом и целочисленным индексом. На основе абстрактного класса и интерфейса нужно создать класс. Абстрактный метод из базового класса переопределить таким образом, чтобы он присваивал значение элементу массива, метод из интерфейса

должен создавать новый массив, а индексатор должен предоставлять доступ к элементам массива.

10. Напишите программу, содержащую абстрактный класс и два интерфейса. Класс должен содержать объявление абстрактного свойства (с двумя аксессорами), абстрактного индексатора (с двумя аксессорами) и абстрактного метода. Такое же свойство, индексатор и метод должны быть в интерфейсах. На основе абстрактного класса и интерфейсов необходимо создать класс. В этом классе необходимо выполнить явную реализацию для свойства, индексатора и метода для каждого из интерфейсов. Проверьте работу свойства, индексатора и метода, получив доступ к объекту класса через объектную переменную и через интерфейсные переменные.

Глава 2
ДЕЛЕГАТЫ И СОБЫТИЯ

Я красавец. Быть может, неизвестный собачий принц. Инкогнито.

из к/ф «Собачье сердце»

В этой главе обсуждаются достаточно важные и не совсем обычные темы. Наш ближайший план в изучении языка C# включает такие пункты:

- знакомство с делегатами;
- ссылки на методы и создание экземпляров делегата;
- анонимные методы;
- лямбда-выражения;
- знакомство с событиями.

Мы также коснемся других тем, которые связаны с использованием делегатов и событий. Следует отметить, что вопросы, рассматриваемые далее, относятся к фундаментальным технологиям, используемым в том числе и при разработке приложений с графическим интерфейсом.

Знакомство с делегатами

Мы к вам, профессор, и вот по какому делу.

из к/ф «Собачье сердце»

Ранее мы рассматривали различные классы, на основе которых создавались объекты. Мы явно или неявно исходили из того, что класс представляет собой некоторый аналог «типа данных», с поправкой на то обстоятельство, что в классе «спрятаны» не только данные (или средства для их реализации), но и некоторая «функциональность» в виде кода для обработки этих данных. Теперь нам предстоит познакомиться с еще

более специфическим «типом» — речь идет о *делегатах*. Чтобы понять, что же такое делегат, удобно представлять делегат как некий аналог класса. Так же как на основе класса создается объект, на основе делегата тоже создается объект. Мы будем называть такой объект *экземпляром делегата*. Экземпляр делегата — это такой специфический объект, который может ссылаться на методы.

> **НА ЗАМЕТКУ**
>
> Существует определенная неоднозначность, связанная с терминологией. Иногда делегатом называют и собственно делегат, и экземпляр делегата. Здесь нет ничего страшного — главное, чтобы это не приводило к недоразумениям. Иногда делегат называют типом делегата. Мы будем придерживаться принципа, согласно которому на основе делегата создается экземпляр делегата, подобно тому как на основе класса создается объект (экземпляр класса).

Правда мы еще не знаем, что такое *ссылка на метод*. Но в этом случае можно отталкиваться от позитивистской концепции: есть некий объект (экземпляр делегата), который связан с некоторым методом: объект «знает», что это за метод и где его «искать». Объект можно вызывать, как метод. При вызове объекта на самом деле вызывается метод, с которым связан объект.

> **НА ЗАМЕТКУ**
>
> Вообще, здесь ничего удивительного нет. Переменная массива ссылается на массив. Объектная переменная ссылается на объект. Когда мы обращаемся к переменной массива, получаем доступ к массиву. Когда обращаемся к объектной переменной, получаем доступ к объекту. Экземпляр делегата ссылается на метод. Когда мы обращаемся к экземпляру делегата, то получаем доступ к методу. Просто для метода «получение доступа» означает, что метод вызывается.

Как же описывается делегат и как на его основе создавать экземпляры (и, самое главное, что с ними потом делать)? Начнем с описания делегата. Чтобы понять логику того, как описывается делегат, следует еще раз вспомнить, зачем он нужен. Нужен делегат для создания объектов, которые будут ссылаться на методы. А что важно при работе с методом? Какие параметры или характеристики могут использоваться для «классификации» методов? Несложно сообразить, что это тип результата и количество и тип аргументов. Поэтому когда мы описываем делегат,

то должны явно указать, на какие методы смогут ссылаться экземпляры делегата. Другими словами, нам необходимо указать, какого типа результат и какого типа аргументы должны быть у метода, чтобы экземпляр данного делегата мог ссылаться на метод.

НА ЗАМЕТКУ

Вообще стоит заметить, что тип аргументов в методе может быть базовым для типа аргументов в делегате. Также разрешается, чтобы тип результата в делегате был базовым для типа результата метода.

Именно эти характеристики указываются при описании делегата. Начинается оно с ключевого слова `delegate`. Далее указывается идентификатор типа — это идентификатор типа результата метода, на который сможет ссылаться экземпляр делегата. Например, если указано ключевое слово `int`, то это означает, что экземпляр делегата сможет ссылаться на методы, которые результатом возвращают `int`-значение. Далее, после идентификатора типа указывается название делегата. Это имя, которое мы даем делегату, сродни имени класса. После имени делегата в круглых скобках описываются аргументы (указывается тип и формальное название аргумента). Это аргументы, которые должны быть у метода для того, чтобы ссылку на метод можно было присвоить экземпляру делегата. Общий шаблон объявления делегата такой:

```
delegate тип имя(аргументы);
```

Фактически, если взять сигнатуру (тип результата, имя и список аргументов) метода, на который может ссылаться экземпляр делегата, заменить в сигнатуре название метода на название делегата и указать в самом начале ключевое слово `delegate`, мы получим объявление делегата. Например, мы хотим описать делегат с названием `MyDelegate`, экземпляру которого можно было бы присвоить ссылку на метод. У метода два аргумента (типа `int` и `string`) и результат типа `char`. Такой делегат объявлялся бы следующей инструкцией:

```
delegate char MyDelegate(int k,string txt);
```

Это объявление делегата. Но делегат, напомним, нужен для того, чтобы создать объект (экземпляр делегата). Схема здесь достаточно простая, и внешне все напоминает создание объекта класса, где роль класса играет делегат. Мы можем использовать следующий шаблон:

```
делегат переменная=new делегат(метод);
```

В данном случае инструкцией вида new делегат(метод) создается экземпляр делегата, а ссылка на этот экземпляр записывается в переменную. Экземпляр делегата, на который ссылается переменная, сам ссылается на метод, указанный в инструкции new делегат(метод).

ПОДРОБНОСТИ

Вообще, цепочка ссылок получается нетривиальная. Есть переменная (аналог объектной переменной), которая ссылается на экземпляр делегата (объект). Этот экземпляр делегата ссылается на метод.

Возвращаясь к примеру с делегатом MyDelegate, мы могли бы создать экземпляр этого делегата командой такого вида:

```
MyDelegate meth=new MyDelegate(метод);
```

Данной командой объявляется объектная переменная meth типа MyDelegate (фактически объектная переменная), в эту переменную записывается ссылка на созданный объект (экземпляр делегата MyDelegate), а сам этот объект ссылается на метод, указанный в круглых скобках в инструкции new MyDelegate(метод). Осталось осветить вопрос с тем, как выполняется ссылка на метод. Здесь все очень просто: для выполнения ссылки на метод достаточно указать имя этого метода (без круглых скобок и аргументов). Если речь идет о выполнении ссылки на нестатический метод, то указывается имя объекта и название метода (разделенные точкой), то есть ссылка на нестатический метод выполняется в формате объект.метод. Если метод статический, то ссылка на него выполняется в формате класс.метод, то есть указывается имя класса и после него, через точку — имя метода. Например, если нужно выполнить ссылку на метод method(), который должен вызываться из объекта obj, то она будет выглядеть как obj.method. Команда создания экземпляра делегата, ссылающегося на метод method() объекта obj, выглядит так:

```
MyDelegate meth=new MyDelegate(obj.method);
```

Если метод method() статический в классе MyClass, то ссылка на такой метод выполняется как MyClass.method. Команда создания экземпляра делегата, ссылающегося на статический метод method() класса MyClass, выглядит следующим образом:

```
MyDelegate meth=new MyDelegate(MyClass.obj);
```

После того как мы создали экземпляр делегата, с его помощью можно вызывать метод, на который экземпляр ссылается. Для этого достаточно «вызвать» экземпляр делегата. Например, если экземпляр делегата `meth` ссылается на метод `method()` объекта `obj` и мы хотим вызвать этот метод с аргументами `number` (целое число типа `int`) и `text` (значение типа `string`), то мы можем воспользоваться командой `meth(number,text)`, которая будет означать выполнение инструкции `obj.method(number,text)`. Если экземпляр делегата `meth` ссылается на статический метод `method()` класса `MyClass`, то для выполнения инструкции `MyClass.method(number,text)` можно воспользоваться командой `meth(number,text)`.

ПОДРОБНОСТИ

Выше мы назвали переменную `meth` экземпляром делегата, что не совсем точно. Переменная `meth` ссылается на экземпляр делегата (который ссылается на метод). Обычно, если это не будет приводить к недоразумениям, мы будем отождествлять переменную, ссылающуюся на экземпляр делегата, с этим экземпляром делегата.

Количество и тип аргументов, которые следует передать экземпляру делегата при вызове, определяются тем, как описан соответствующий делегат. Скажем, если делегат `MyDelegate` описан так, как указано выше, то экземпляр делегата `MyDelegate` следует вызывать с двумя аргументами: целочисленным и текстовым, — в соответствии с тем, какие аргументы указаны после названия делегата `MyDelegate` в его объявлении.

Описанный выше способ создания экземпляра делегата — не единственный и не самый простой. Можно пойти более радикальным путем: объявить переменную, тип которой является делегатом, и присвоить такой переменной ссылку на метод. Шаблон команды следующий:

```
делегат переменная=метод;
```

Вернемся к делегату `MyDelegate` и методу `method()`: для нестатического метода все могло бы выглядеть так:

```
MyDelegate meth=obj.method;
```

Если метод статический, то экземпляр делегата можно создать таким образом:

```
MyDelegate meth=MyClasss.method;
```

При присваивании ссылки на метод переменной типа делегата происходит следующее: создается экземпляр делегата, который ссылается на данный метод, а ссылка на созданный экземпляр делегата присваивается переменной.

Перейдем к практическим моментам. В листинге 2.1 представлена программа, в которой объявляются делегаты, создаются экземпляры делегатов и затем эти экземпляры делегатов используются для вызова методов.

Листинг 2.1. Знакомство с делегатами

```
using System;
// Объявление делегата:
delegate char MyDelegate(int k,string txt);
// Класс:
class MyClass{
   // Целочисленное поле:
   public int code;
   // Конструктор:
   public MyClass(int n){
      code=n;
   }
   // Нестатический метод с двумя аргументами:
   public char getChar(int k,string txt){
      return (char)(txt[k]+code);
   }
   // Статический метод с двумя аргументами:
   public static char getFirst(int k,string txt){
      return txt[k];
   }
}
// Класс с главным методом:
class DelegateDemo{
   // Статический метод с двумя аргументами:
   static char getLast(int k,string txt){
```

```
        return txt[txt.Length-k-1];
    }
    // Главный метод:
    static void Main(){
        // Создание объекта:
        MyClass obj=new MyClass(5);
        // Создание экземпляра делегата:
        MyDelegate meth=new MyDelegate(obj.getChar);
        // Вызов экземпляра делегата:
        Console.WriteLine("Символ \'{0}\'",meth(4,"Alpha"));
        // Присваивание значения полю объекта:
        obj.code=1;
        // Вызов экземпляра делегата:
        Console.WriteLine("Символ \'{0}\'",meth(4,"Alpha"));
        // Присваивание нового значения переменной делегата:
        meth=MyClass.getFirst;
        // Вызов экземпляра делегата:
        Console.WriteLine("Символ \'{0}\'",meth(2,"Alpha"));
        // Присваивание нового значения переменной делегата:
        meth=getLast;
        // Вызов экземпляра делегата:
        Console.WriteLine("Символ \'{0}\'",meth(1,"Alpha"));
    }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 2.1)

```
Символ 'f'
Символ 'b'
Символ 'p'
Символ 'h'
```

Делегат MyDelegate в программе объявляется инструкцией delegate char MyDelegate(int k,string txt). Такое объявление означает, что экземпляр делегата может ссылаться на метод, который результатом возвращает значение типа char и имеет два аргумента: первый типа int и второй типа string.

В программе описывается класс MyClass. В классе описано открытое целочисленное поле code и конструктор с одним аргументом (аргумент задает значение поля объекта). Еще в классе описано два метода: нестатический и статический. У каждого из них два аргумента (целое число и текст), и каждый из них возвращает результатом символьное значение. Оба метода соответствуют характеристикам делегата MyDelegate, и поэтому экземпляр делегата сможет ссылаться на эти методы.

Нестатический метод getChar() имеет два аргумента: k (целое число) и txt (текст). Результатом возвращается символ, вычисляемый выражением (char)(txt[k]+code).

ПОДРОБНОСТИ

Из текста txt берется символ с индексом k (выражение txt[k]), к коду полученного символа прибавляется значение поля code (выражение txt[k]+code), и полученное число преобразуется к текстовому формату (выражение (char)(txt[k]+code)). Получается, что результатом является символ, смещенный к символу txt[k] на количество позиций, определяемое значением поля code.

Статический метод getFirst() с целочисленным аргументом k и текстовым аргументом txt возвращает результатом символ из текста txt с индексом k (выражение txt[k]).

В классе DelegateDemo, в котором описан главный метод Main(), есть описание еще одного статического метода getLast(). Метод для заданного текста txt (второй аргумент) возвращает символ из этого текста с индексом k (первый аргумент), если индекс отсчитывать с конца текста (результат вычисляется как txt[txt.Length-k-1]).

В методе Main() командой MyClass obj=new MyClass(5) создается объект obj класса MyClass, значение поля code объекта obj равно 5.

Экземпляр делегата MyDelegate создается командой MyDelegate met h=new MyDelegate(obj.getChar). В итоге мы получаем переменную

meth делегата MyDelegate, которая ссылается на экземпляр делегата, ссылающийся на метод getChar() объекта obj. Поэтому при вызове экземпляра делегата инструкцией meth(4,"Alpha") из объекта obj вызывается метод getChar() с аргументами 4 и "Alpha". В результате из текста "Alpha" берется символ с индексом 4 (это символ 'a') и выполняется смещение на 5 позиций. В результате получаем символ 'f'.

Далее мы командой obj.code=1 меняем значение поля code объекта obj и снова вызываем экземпляр делегата с теми же аргументами (команда meth(4,"Alpha")). Теперь результатом является символ 'b', поскольку к символу 'a' применяется смещение на 1 позицию (новое значение поля code), а не на 5 (старое значение поля code), как было ранее.

Командой meth=MyClass.getFirst переменной meth делегата MyDelegate присваивается новое значение: ссылка на статический метод getFirst() класса MyClass. После этого вызов экземпляра делегата командой meth(2,"Alpha") означает вызов статического метода getFirst() с аргументами 2 и "Alpha". Результатом является символ 'p' (символ с индексом 2 в тексте "Alpha").

Командой meth=getLast экземпляру делегата meth в качестве значения присваивается ссылка на статический метод getLast(). Поскольку команда присваивания находится (в главном методе) в том же классе, в котором описан статический метод getLast(), то имя класса в ссылке на метод можно не указывать. При вызове экземпляра делегата командой meth(1,"Alpha") вызывается метод getLast() с аргументами 1 и "Alpha". Результатом является символ 'h': второй индекс с конца текста "Alpha".

Многократная адресация

— Так что передать мой король?
— Передай твой король мой пламенный привет!

из к/ф «Иван Васильевич меняет профессию»

Делегаты поддерживают многократную адресацию. Это означает, что экземпляр делегата может ссылаться не на один, а сразу на несколько методов. Если так, то при вызове делегата выполняется цепочка вызовов: последовательно вызываются методы, на которые ссылается

вызываемый экземпляр делегата. Методы вызываются в той последовательности, в которой ссылки на методы добавлялись экземпляру делегата. Для добавления ссылки на метод экземпляру делегата используется оператор += (команда вида `экземпляр+=метод`) или полная версия операции присваивания вида `экземпляр=экземпляр+метод`.

Таким образом, мы можем связать с экземпляром делегата не только отдельный метод, но и целый список методов. Если впоследствии нам понадобится удалить ссылку на метод из списка методов, на которые ссылается экземпляр делегата, можно воспользоваться оператором `-=` (команда вида `экземпляр-=метод`) или командой вида `экземпляр=экземпляр-метод`. В листинге 2.2 представлена программа, в которой используется многократная адресация для экземпляра делегата.

Листинг 2.2. Многократная адресация для экземпляров делегата

```
using System;
// Объявление делегата:
delegate void MyDelegate();
// Класс:
class MyClass{
   // Текстовое поле:
   public string name;
   // Конструктор с текстовым аргументом:
   public MyClass(string txt){
      name=txt;
   }
   // Метод без аргументов:
   public void show(){
      Console.WriteLine(name);
   }
}
// Класс с главным методом:
class MulticastDemo{
   // Статический метод без аргументов:
   static void makeLine(){
```

```
        Console.WriteLine("--------");
    }
    // Главный метод:
    static void Main(){
        // Создание объектов:
        MyClass A=new MyClass("Объект A");
        MyClass B=new MyClass("Объект B");
        MyClass C=new MyClass("Объект C");
        // Объявление переменной делегата:
        MyDelegate meth;
        // Присваивание переменной делегата ссылки на метод:
        meth=A.show;
        // Вызов экземпляра делегата:
        meth();
        // Присваивание переменной делегата нового значения:
        meth=makeLine;
        // Добавление методов в список вызова:
        meth+=A.show;
        meth+=B.show;
        meth=meth+C.show;
        // Вызов экземпляра делегата:
        meth();
        // Удаление метода из списка вызова:
        meth-=B.show;
        // Вызов экземпляра делегата:
        meth();
        // Удаление метода из списка вызова:
        meth=meth-A.show;
        // Вызов экземпляра делегата:
        meth();
    }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 2.2)

```
Объект A
--------
Объект A
Объект B
Объект C
--------
Объект A
Объект C
--------
Объект C
```

Делегат `MyDelegate` объявлен командой `delegate void MyDelegate()`. Экземпляры делегата могут ссылаться на методы, не имеющие аргументов и не возвращающие результат. В программе описывается класс `MyClass` с открытым текстовым полем `name`. У класса есть конструктор с текстовым аргументом, который присваивается в качестве значения полю `name` создаваемого объекта. Также в классе описан метод `show()`, при вызове отображающий в консольном окне значение текстового поля `name`. У метода нет аргументов, и он не возвращает результат.

В классе `MulticastDemo`, кроме главного метода `Main()`, есть еще и статический метод `makeLine()`. У метода `makeLine()` нет аргументов, и он не возвращает результат. При вызове метода в консольном окне отображается импровизированная «линия» из дефисов.

НА ЗАМЕТКУ

Таким образом, и статический метод `makeLine()`, и нестатический метод `show()` из класса `MyClass` соответствуют «требованиям», которые «задекларированы» при объявлении делегата `MyDelegate`. Поэтому экземпляр делегата может ссылаться на оба эти метода.

В методе `Main()` последовательно создаются три объекта `A`, `B` и `C` класса `MyClass`. Поле `name` каждого из объектов получает уникальное текстовое значение. Командой `MyDelegate meth` объявляется переменная

meth делегата MyDelegate. Пока эта переменная не ссылается на экземпляр делегата — мы его еще не создали. Экземпляр делегата создается (и ссылка на него записывается в переменную meth) при выполнении команды meth=A.show, которой ссылка на метод show() объекта A присваивается переменной meth. Команда выполняется так: создается экземпляр делегата, который ссылается на метод show() объекта A, и ссылка на этот экземпляр записывается в переменную meth.

ПОДРОБНОСТИ

Уместно напомнить, что доступ к экземпляру делегата мы получаем через переменную делегата (аналог объектной переменной). Если такой переменной в качестве значения присвоить ссылку на метод, то автоматически будет создан экземпляр делегата, ссылающегося на метод, а в переменную будет записана ссылка на экземпляр делегата. Командой MyDelegate meth мы только объявили переменную. Экземпляр делегата создается, когда переменной meth присваивается значение (ссылка на метод). В принципе, можно создавать экземпляр делегата и с помощью выражения на основе new-инструкции.

При выполнении команды meth() из объекта A вызывается метод show(). Затем выполняется команда meth=makeLine. Формально здесь переменной meth присваивается новое значение. Но в действительности происходит следующее. Создается новый экземпляр делегата, который ссылается на статический метод makeLine(). Ссылка на экземпляр делегата записывается в переменную meth, а ее связь с прежним экземпляром делегата (который ссылался на метод show() объекта A) теряется. Внешне все выглядит так, как если бы экземпляр делегата, отождествляемый с переменной meth, получал ссылку на метод makeLine(). А вот выполнение команды meth+=A.show приводит к тому, что в экземпляр делегата добавляется еще и ссылка на метод show() объекта A. При этом метод makeLine() также остается в списке вызовов экземпляра делегата. Командами meth+=B.show и meth=meth+C.show в список вызовов последовательно добавляются методы show() объектов B и C. Поэтому при вызове экземпляра делегата командой meth() будут выполнены вызовы makeLine(),A.show(), B.show() и C.show() — строго в той последовательности, в которой методы добавлялись в список вызовов.

После выполнения команды meth-=B.show из списка вызовов экземпляра делегата meth удаляется метод show() объекта B. При выполнении

вызова экземпляра делегата командой `meth()` теперь последовательно выполняются вызовы `makeLine()`, `A.show()` и `C.show()`. Если после этого выполнить команду `meth=meth-A.show`, то из списка вызовов будет удален еще и метод `show()` объекта `A`. Вызов экземпляра делегата командой `meth()` приведет к выполнению вызовов `makeLine()` и `C.show()`.

ПОДРОБНОСТИ

В список вызовов экземпляра делегата можно добавлять несколько раз один и тот же метод (подряд или нет). Если из списка вызовов удаляется метод, который представлен в этом списке несколько раз, то удаляется последняя добавленная ссылка на этот метод. Если попытаться удалить метод, которого в списке вызовов нет, то не произойдет ничего.

Методы в списке вызовов могут возвращать результат. Если так, то при вызове экземпляра делегата, ссылающегося на список вызовов, результатом возвращается значение, возвращаемое последним методом в списке вызовов.

Использование делегатов

> Он бы прямо на митингах мог деньги зарабатывать. Первоклассный деляга.
>
> *из к/ф «Собачье сердце»*

В этом разделе мы рассмотрим несколько примеров, которые иллюстрируют красоту и мощь делегатов. Начнем с примера из листинга 2.3, в котором экземпляр делегата (точнее, переменная, которая может ссылаться на экземпляр делегата) является полем в классе.

Листинг 2.3. Экземпляр делегата как поле класса

```
using System;
// Объявление делегата:
delegate void MyDelegate(string txt);
// Класс с полем, являющимся ссылкой на экземпляр делегата:
class MyClass{
   // Поле является ссылкой на экземпляр делегата:
```

```
   public MyDelegate apply;
   // Конструктор:
   public MyClass(MyDelegate md){
      apply=md;
   }
}
// Класс:
class Alpha{
   // Закрытое текстовое поле:
   private string name;
   // Метод для присваивания значения полю:
   public void set(string t){
      name=t;
   }
   // Переопределение метода ToString():
   public override string ToString(){
      return name;
   }
}
// Класс с главным методом:
class DelegateAsFieldDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
      Alpha A=new Alpha();
      // Создание объекта
      // (аргумент конструктора – ссылка на метод):
      MyClass obj=new MyClass(A.set);
      // Вызов экземпляра делегата:
      obj.apply("Объект A");
      // Проверка поля объекта:
      Console.WriteLine(A);
```

```
      // Создание объекта:
      Alpha B=new Alpha();
      // Полю значением присваивается ссылка на метод:
      obj.apply=B.set;
      // Вызов экземпляра делегата:
      obj.apply("Объект B");
      // Проверка поля объекта:
      Console.WriteLine(B);
      // Добавление метода в список вызовов экземпляра
      // делегата:
      obj.apply+=A.set;
      // Вызов экземпляра делегата:
      obj.apply("Объект X");
      // Проверка полей объектов:
      Console.WriteLine(A+" и "+B);
      // Удаление метода из списка вызовов экземпляра
      // делегата:
      obj.apply-=B.set;
      // Вызов экземпляра делегата:
      obj.apply("Объект A");
      // Проверка полей объектов:
      Console.WriteLine(A+" и "+B);
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 2.3)

```
Объект A
Объект B
Объект X и Объект X
Объект A и Объект X
```

Делегат `MyDelegate` объявляется командой `delegate void MyDelegate(string txt)`. Экземпляры делегата смогут ссылаться на методы, имеющие один текстовый аргумент и не возвращающие результат. В классе `MyClass` описано поле `apply`, типом которого указан делегат `MyDelegate`. Это означает, что поле `apply` может ссылаться на метод (или список методов). Но самое важное — мы можем «вызывать» это поле (передав ему один текстовый аргумент). Также в классе описан конструктор с одним аргументом. Аргумент конструктора тоже примечательный — это переменная `md` типа `MyDelegate`. В теле конструктора командой `apply=md` аргумент, переданный конструктору, присваивается значением полю `apply`.

ПОДРОБНОСТИ

Если в качестве типа переменной указан делегат, то такая переменная, по своей сути, является ссылкой на метод (который соответствует параметрам делегата). Во всяком случае, значением переменной может присваиваться ссылка на метод. Правда происходит все немного сложнее: когда переменной типа делегата присваивается ссылка на метод, создается экземпляр делегата, переменная ссылается на этот экземпляр делегата, а экземпляр делегата ссылается на метод. Фактически экземпляр делегата является посредником между переменной и методом. Но в практическом плане, для понимания происходящего, обычно можно отождествлять переменную типа делегата (как поле `apply` или аргумент конструктора класса `MyClass`) со ссылкой на метод.

Еще один класс `Alpha` имеет закрытое текстовое поле `name`. Для присваивания значения полю предусмотрен открытый метод `set()`, имеющий текстовый аргумент и не возвращающий результат. Для считывания значения поля `name` мы переопределяем метод `ToString()` (метод результатом возвращает значение поля `name`).

В методе `Main()` командой `Alpha A=new Alpha()` создается объект класса `Alpha`. После этого командой `MyClass obj=new MyClass (A.set)` мы создаем объект класса `MyClass`. Пикантность ситуации в том, что аргументом конструктору класса `MyClass` передается ссылка `A.set` на метод `set()` объекта `A`.

ПОДРОБНОСТИ

В несколько упрощенном виде общая последовательность действий такая. Аргумент конструктора класса `MyClass` объявлен как

переменная типа MyDelegate. При вызове конструктора для записи аргумента выделяется место в памяти — то есть создается техническая переменная типа MyDelegate. Аргументом конструктору передается ссылка A.set, которая и присваивается технической переменной. В результате создается экземпляр делегата, который ссылается на метод set() объекта A, а техническая переменная (аргумент конструктора) ссылается на этот экземпляр. В теле конструктора, при выполнении команды apply=md, значение технической переменной копируется в поле apply. В результате поле apply ссылается на экземпляр делегата, который ссылается на метод set() объекта A.

Поэтому при выполнении команды obj.apply("Объект A"), вызывающей экземпляр делегата, на который ссылается поле apply, с аргументом "Объект A" вызывается метод set() объекта A. Полю name объекта A присваивается значение, что подтверждается при выполнении команды Console.WriteLine(A). Затем командой Alpha B=new Alpha() создается еще один объект класса Alpha. С помощью команды obj.apply=B.set поле apply связывается с методом set() объекта B. Поэтому после выполнения команды obj.apply("Объект B") присваивается значение полю name объекта B. Это значение проверяем командой Console.WriteLine(B).

Командой obj.apply+=A.set в список вызовов экземпляра делегата apply добавляется метод set() объекта A. И когда выполняется команда obj.apply("Объект X"), то из объектов B и A последовательно вызывается метод set() с аргументом "Объект X". В результате поле name каждого из объектов получает значение "Объект X", в чем мы и убеждаемся с помощью команды Console.WriteLine(A+" и "+B).

Наконец, выполнение команды obj.apply-=B.set приводит к удалению метода set() объекта B из списка вызовов экземпляра делегата apply. Поэтому при выполнении команды obj.apply("Объект A") меняется значение поля name только для объекта A.

В следующем примере мы используем делегат для того, чтобы передавать ссылки на методы аргументом другому методу. Программа достаточно простая: описывается статический метод, который используется для того, чтобы создать таблицу значений другого метода. Таблица формируется так. Берется некоторый метод, у которого целочисленный аргумент и который возвращает целочисленный результат. Аргумент метода изменяется в заданных пределах. Для каждого значения аргумента отображается значение метода. Этот простой алгоритм не зависит

от того, для какого именно метода строится таблица значений. Главное, чтобы у метода был целочисленный аргумент, и он должен возвращать целочисленный результат. Именно та ситуация, когда разумно использовать делегаты. Рассмотрим программный код, представленный в листинге 2.4.

Листинг 2.4. Передача метода в качестве аргумента

```
using System;
// Объявление делегата:
delegate int MyDelegate(int n);
// Класс с главным методом:
class DelegateAsArgDemo{
   // Статический метод для вычисления нечетных чисел:
   static int f(int n){
      return 2*n+1;
   }
   // Статический метод для вычисления четных чисел:
   static int g(int n){
      return 2*n;
   }
   // Статический метод для вычисления квадратов чисел:
   static int h(int n){
      return n*n;
   }
   // Статический метод, которому аргументом
   // передается ссылка на метод:
   static void display(MyDelegate F,int a,int b){
      Console.WriteLine("{0,-4}|{1,4}","x"," F(x)");
      Console.WriteLine("----------");
      for(int k=a;k<=b;k++){
         // Команда с вызовом экземпляра делегата:
         Console.WriteLine("{0,-4}|{1,4}",k,F(k));
      }
```

```
      Console.WriteLine();
   }
   // Главный метод:
   static void Main(){
      // Диапазон изменения аргумента:
      int a=0,b=5;
      Console.WriteLine("Нечетные числа:");
      // Передача аргументом ссылки на метод:
      display(f,a,b);
      Console.WriteLine("Четные числа:");
      // Передача аргументом ссылки на метод:
      display(g,a,b);
      Console.WriteLine("Число в квадрате:");
      // Передача аргументом ссылки на метод:
      display(h,a,b);
   }
}
```

В результате выполнения программы получаем следующее:

Результат выполнения программы (из листинга 2.4)

```
Нечетные числа:
x   |  F(x)
----------
0   |   1
1   |   3
2   |   5
3   |   7
4   |   9
5   |  11
```

```
Четные числа:
x   |  F(x)
----------
0   |   0
1   |   2
2   |   4
3   |   6
4   |   8
5   |  10

Число в квадрате:
x   |  F(x)
----------
0   |   0
1   |   1
2   |   4
3   |   9
4   |  16
5   |  25
```

Программа небольшая и простая. В программе командой `delegate int MyDelegate(int n)` объявляется делегат, экземпляр которого может ссылаться на метод, имеющий целочисленный аргумент и возвращающий целочисленный результат. В классе с главным методом программы описано еще несколько статических методов. Методы `f()`, `g()` и `h()` соответствуют параметрам делегата `MyDelegate`: у каждого из этих методов один целочисленный аргумент, и каждый метод результатом возвращает целое число. При заданном целочисленном аргументе `n` метод `f()` возвращает значение `2*n+1` (нечетное число), метод `g()` возвращает значение `2*n` (четное число), а метод `h()` возвращает значение `n*n` (число в квадрате). Еще один статический метод `display()` описан с тремя аргументами. Первый аргумент `F` имеет тип `MyDelegate` — то есть это переменная, которая может ссылаться на экземпляр делегата. Мы уже знаем, что такой переменной можно присваивать ссылку на метод. Поэтому при вызове метода первым аргументом

можно передавать ссылку на метод (с целочисленным аргументом и целочисленным результатом). Другими словами, под аргументом `F` мы можем подразумевать название метода, который можно вызывать, передав ему целочисленный аргумент. Еще два целочисленных аргумента (`a` и `b`) определяют диапазон, на котором табулируется переданный первым аргументом метод. Табуляция выполняется просто: с помощью оператора цикла в консоль выводится значение аргумента `k` и значение метода `F(k)`.

ПОДРОБНОСТИ

В командах `Console.WriteLine("{0,-4}|{1,4}","x","  F(x)")` и `Console.WriteLine("{0,-4}|{1,4}",k,F(k))` инструкция `{0,-4}` означает, что под отображаемое значение выделяется `4` позиции и выравнивание выполняется по левому краю. Инструкция `{1,4}` означает, что под отображаемое значение выделяется `4` позиции и выравнивание выполняется по правому краю.

В главном методе программы при заданных значениях переменных `a` и `b` командами `display(f,a,b)`, `display(g,a,b)` и `display(h,a,b)` табулируются методы `f()`, `g()` и `h()` соответственно. Чтобы передать метод аргументом методу `display()`, достаточно указать имя метода первым аргументом метода `display()`. Например, при выполнении команды `display(f,a,b)` выполняется код метода `display()`, но только вместо аргумента `F` в программном коде следует подразумевать `f`. То же касается и других команд с вызовом `display()`.

Еще один пример, который мы рассмотрим, касается ситуации, когда с помощью делегатов мы создаем иллюзию (хотя это и не совсем иллюзия), что результатом выражения является метод. Рассмотрим программный код, представленный в листинге 2.5.

Листинг 2.5. Экземпляр делегата как значение выражения

```
using System;
// Объявление делегата:
delegate int Method(int n);
// Класс с индексатором:
class MyClass{
   // Закрытое целочисленное поле:
   private int num;
```

```
   // Конструктор с целочисленным аргументом:
   public MyClass(int n){
      num=n;
   }
   // Первый закрытый метод:
   private int first(int n){
      return n+num;
   }
   // Второй закрытый метод:
   private int second(int n){
      return n-num;
   }
   // Третий закрытый метод:
   private int third(int n){
      return n*num;
   }
   // Индексатор с целочисленным индексом.
   // Результат - ссылка на экземпляр делегата:
   public Method this[int k]{
      // Аксессор для считывания значения:
      get{
         switch(k){
            case 0:
               return first;
            case 1:
               return second;
            default:
               return third;
         }
      }
   }
}
```

```
// Класс с главным методом:
class DelegateAsResDemo{
   // Главный метод:
   static void Main(){
      // Переменная:
      int x=12;
      // Создание объекта:
      MyClass obj=new MyClass(4);
      // Индексирование объекта:
      for(int k=0;k<=2;k++){
         Console.WriteLine("obj[{0}]({1})={2}",k,x,obj[k](x));
      }
   }
}
```

При выполнении программы получим следующий результат:

Результат выполнения программы (из листинга 2.5)

```
obj[0](12)=16
obj[1](12)=8
obj[2](12)=48
```

Делегат `Method` объявляется командой `delegate int Method(int n)` и соответствует методам с одним целочисленным аргументом и целочисленным результатом. В классе `MyClass` мы описываем закрытое целочисленное поле `num`, значение которому присваивается при создании объекта. Еще в классе описаны три закрытых метода. Метод `first()` возвращает в качестве результата сумму значения своего аргумента и поля `num`. Метод `second()` результатом возвращает разность аргумента и поля `num`. Метод `third()` возвращает результатом произведение значений аргумента и поля `num`. Еще раз подчеркнем, что все три метода закрытые, поэтому прямого доступа к ним вне пределов класса нет. Но в классе описан индексатор с `get`-аксессором. Тип результата, возвращаемого аксессором, указан как `Method`. Это означает, что при индексировании объекта мы результатом получаем ссылку на экземпляр

делегата Method. Но фактически это ссылка на метод, который соответствует параметрам делегата Method.

Индексатор описан так, что если индекс равен 0, то результатом возвращается значение first (ссылка на метод first() индексируемого объекта). Если индекс равен 1, то результатом возвращается значение second (ссылка на метод second() индексируемого объекта). При всех других значениях индекса результатом возвращается ссылка third на метод third() индексируемого объекта.

ПОДРОБНОСТИ

В описании аксессора указано, что результатом он возвращает значение типа Method. Что это означает? Это значит, что результатом аксессора является ссылка на экземпляр делегата Method. При вызове аксессора для записи результата выделяется переменная (типа Method). Эта переменная может ссылаться на экземпляр делегата. Фактическим значением аксессор возвращает ссылку на метод. Получается, что переменной типа Method присваивается ссылка на метод. В этом случае, как мы помним, автоматически создается экземпляр делегата, который ссылается на метод. Ссылка на экземпляр делегата записывается в переменную, выделенную для запоминания результата аксессора.

В главном методе программы мы проверяем работу индексатора. Для этого командой MyClass obj=new MyClass(4) создаем объект obj класса MyClass. Целочисленное поле num этого объекта равно 4 (аргумент конструктора). Далее для переменной x со значением 12 в операторе цикла при фиксированном значении переменной k (изменяется в пределах от 0 до 2 включительно) вычисляется значение выражения obj[k](x). Несложно заметить, что в этом случае с аргументом x из объекта obj последовательно вызываются методы first(), second() и third().

НА ЗАМЕТКУ

Таким образом, выражения вида obj[k] мы можем отождествлять с именем метода.

Еще один небольшой пример иллюстрирует особенности реализации ссылок на методы объектов. Рассмотрим программный код, представленный в листинге 2.6.

Листинг 2.6. Ссылки на методы и ссылки на объекты

```
using System;
// Объявление делегата:
delegate void MyDelegate(int n);
// Класс:
class MyClass{
   // Закрытое целочисленное поле:
   private int number;
   // Конструктор с одним аргументом:
   public MyClass(int n){
      // Присваивание значения полю:
      set(n);
   }
   // Метод для присваивания значения полю:
   public void set(int n){
      // Присваивание значения полю:
      number=n;
   }
   // Переопределение метода ToString():
   public override string ToString(){
      return "Поле number="+number;
   }
}
// Класс с главным методом:
class MethAndObjRefsDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
      MyClass A=new MyClass(100);
      // Объектная переменная ссылается
      // на ранее созданный объект:
      MyClass B=A;
```

```
        // Переменной типа делегата значением присваивается
        // ссылка на метод объекта:
        MyDelegate apply=A.set;
        // Переменной присваивается ссылка на новый объект:
        A=new MyClass(200);
        // Проверка значений полей объектов:
        Console.WriteLine(A);
        Console.WriteLine(B);
        // Вызов экземпляра делегата:
        apply(300);
        Console.WriteLine();
        // Проверка значений полей объектов:
        Console.WriteLine(A);
        Console.WriteLine(B);
    }
}
```

Ниже показано, как выглядит результат выполнения программы:

Результат выполнения программы (из листинга 2.6)

```
Поле number=200
Поле number=100

Поле number=200
Поле number=300
```

В программе объявляется делегат `MyDelegate`, соответствующий методам, имеющим целочисленный аргумент и не возвращающим результат. Также мы используем класс `MyClass`. В классе описано закрытое целочисленное поле `number`. Значение полю присваивается при создании объекта. Не возвращающий результат метод `set()` получает при вызове целочисленный аргумент, значение которого присваивается полю `number`. Также в классе переопределен метод `ToString()`, возвращающий результатом текстовую строку со значением поля `number` объекта.

В главном методе мы с помощью команды MyClass A=new MyClass(100) создаем объект, значение поля number которого равно 100. После выполнения команды MyClass B=A переменные A и B ссылаются на один и тот же объект. При выполнении команды MyDelegate apply=A.set создается экземпляр делегата MyDelegate, он содержит ссылку на метод set() объекта A, а ссылка на экземпляр делегата записывается в переменную apply. После этого с помощью команды A=new MyClass(200) создается новый объект (значение поля number этого объекта равно 200) и ссылка на него записывается в переменную A. Получается такая ситуация: при присваивании значения переменной apply переменная A ссылалась на один объект, а теперь она ссылается на другой объект. Возникает естественный вопрос: на метод set() какого объекта будет ссылаться (через экземпляр делегата) переменная apply? Ответ следующий: переменная apply ссылается на метод того объекта, на который ссылалась переменная A на момент присваивания значения переменной apply. Другими словами, при выполнении команды apply=A.set создается ссылка на метод set() объекта, на который ссылается переменная A (напомним, что переменная B ссылается на тот же объект). И даже если затем переменная A получит новое значение и будет ссылаться на другой объект, переменная apply будет связана все с тем же объектом. Для нее ничего не меняется.

Проверить справедливость данного утверждения легко. У нас есть две переменные: переменная A ссылается на новый объект (значение поля 200), а переменная B ссылается на исходный объект (значение поля 100). Сначала проверяем значения полей объектов (команды Console.WriteLine(A) и Console.WriteLine(B)), а затем командой apply(300) вызываем экземпляр делегата, связанного с методом set() объекта, на который ссылается переменная B, и снова проверяем значения полей объектов. Как видим, изменилось поле того объекта, на который ссылается переменная B.

Знакомство с анонимными методами

Никто такого еще не видел. Понаблюдаем.

из к/ф «Собачье сердце»

Выше мы имели дело с примерами, в которых переменным типа делегата в качестве значения присваивались ссылки на методы. Методы были статические или нестатические, но в любом случае они описывались

заранее в соответствующем классе. Можно сказать, что эти методы имели самостоятельную ценность и теоретически мы могли бы их использовать так, как мы обычно использовали методы, не выполняя на них никакие ссылки с привлечением экземпляров делегата. С другой стороны, когда мы вызываем экземпляр делегата, то речь идет о вызове метода или методов — то есть о выполнении некоторой последовательности команд. Поэтому, в рамках концепции использования делегатов, логично было бы иметь возможность связывать экземпляры делегатов с некоторым блоком программного кода, который выполнялся бы при вызове экземпляра делегата и который бы не нужно было описывать в классе в виде метода. Такая возможность имеется. Реализуется она с помощью *анонимных методов*.

Анонимный метод — это оформленный особым образом блок программного кода, который можно присвоить в качестве значения переменной типа делегата. В результате такой операции создается экземпляр делегата (на который будет ссылаться переменная). Этот экземпляр делегата ссылается на метод, который будет вызываться при вызове экземпляра делегата. Причем под методом здесь понимается набор команд, содержащийся в описании анонимного метода.

Анонимный метод описывается практически так же, как и обычный метод. Но понятно, что имеются некоторые особенности. А именно, описание анонимного метода начинается с ключевого слова `delegate`. Это ключевое слово фактически является заменителем имени метода (у анонимного метода имени нет). Тип результата для анонимного метода не указывается. Аргументы описываются после ключевого слова `delegate`. Описываются они так же, как и у обычного метода: указывается тип аргумента и его название, объявления аргументов разделяются запятыми. Далее в фигурных скобках описывается тело анонимного метода — то есть те команды, которые выполняются при вызове метода. Общий шаблон описания анонимного метода следующий (жирным шрифтом выделены ключевые элементы шаблона):

```
delegate(аргументы){
    // Тело метода
}
```

Анонимный метод не может быть вызван сам по себе. Как отмечалось выше, анонимный метод (блок с описанием метода) присваивается в качестве значения переменной типа делегата. При этом анонимный метод может содержать ссылки на внешние (по отношению к методу) переменные.

ПОДРОБНОСТИ

Тип возвращаемого результата для анонимного метода не указывается. При этом анонимный метод может возвращать результат. И здесь нужно учесть, что анонимный метод существует не сам по себе, а присваивается в качестве значения переменной делегата. Но при описании соответствующего делегата был указан тип результата для метода, который может присваиваться в качестве значения экземпляру делегата. Получается, что тип результата анонимного метода неявно определяется описанием делегата (определяющего тип переменной, которой присваивается анонимный метод).

В листинге 2.7 представлена программа, в которой используются анонимные методы.

Листинг 2.7. Знакомство с анонимными методами

```
using System;
// Объявление первого делегата:
delegate int Alpha(int n);
// Объявление второго делегата:
delegate void Bravo(string t);
// Класс:
class MyClass{
   // Открытое целочисленное поле:
   public int number;
   // Конструктор:
   public MyClass(int n){
      number=n; // Значение поля
   }
   // Поле, являющееся ссылкой на экземпляр делегата:
   public Alpha method;
}
// Класс с главным методом:
class AnonymousMethDemo{
   // Главный метод:
```

```
static void Main(){
   // Создание объектов:
   MyClass A=new MyClass(100);
   MyClass B=new MyClass(200);
   // Полю объекта значением присваивается
   // анонимный метод:
   A.method=delegate(int n){
      return A.number+n;
   };
   // Полю объекта значением присваивается
   // анонимный метод:
   B.method=delegate(int n){
      return B.number-n;
   };
   // Целочисленная переменная:
   int x=80;
   // Вызов экземпляра делегата:
   Console.WriteLine("A.method({0})={1}",x,A.method(x));
   // Присваивание нового значения полю:
   A.number=300;
   // Вызов экземпляров делегата:
   Console.WriteLine("A.method({0})={1}",x,A.method(x));
   Console.WriteLine("B.method({0})={1}",x,B.method(x));
   // Объявление переменной типа делегата:
   Bravo show;
   // Присваивание переменной анонимного метода:
   show=delegate(string t){
      Console.WriteLine("Текст: \"{0}\"",t);
   };
   // Вызов экземпляра делегата:
   show("Bravo");
   // Присваивание переменной анонимного метода:
```

```
      show=delegate(string t){
         for(int k=0;k<t.Length;k++){
            Console.Write("|"+t[k]);
         }
         Console.WriteLine("|");
      };
      // Вызов экземпляра делегата:
      show("Bravo");
   }
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 2.7)

```
A.method(80)=180
A.method(80)=380
B.method(80)=120
Текст: "Bravo"
|B|r|a|v|o|
```

В программе мы объявляем два делегата: делегат `Alpha` соответствует методам, имеющим целочисленный аргумент и возвращающим целочисленный результат, а делегат `Bravo` подразумевает использование методов, имеющих текстовый аргумент и не возвращающих результат.

Класс `MyClass` имеет открытое целочисленное поле `number`, значение которому присваивается при вызове конструктора. Также в классе объявлено поле `method` типа `Alpha`. В это поле можно записать ссылку на экземпляр делегата, ссылающегося на метод, имеющий целочисленный аргумент и возвращающий целочисленный результат. Но на практике это означает, что значением полю можно присвоить ссылку на метод и, как следствие, такому полю можно присвоить в качестве значения анонимный метод.

НА ЗАМЕТКУ

Напомним, что делегаты поддерживают многократную адресацию, поэтому экземпляры делегата могут быть связаны не с одним

методом, а со списком методов, которые вызываются при вызове экземпляра делегата.

В главном методе программы командами `MyClass A=new MyClass(100)` и `MyClass B=new MyClass(200)` создаются объекты `A` и `B` класса `MyClass`. У объекта `A` поле `number` равно `100`, а у объекта `B` поле `number` равно `200`.

Далее полю `method` каждого из объектов значением присваивается анонимный метод. Для объекта `A` команда присваивания следующая:

```
A.method=delegate(int n){
   return A.number+n;
};
```

Полю `method` объекта `A` присваивается в качестве значения конструкция, которая начинается с ключевого слова `delegate` и заканчивается закрывающей фигурной скобкой, после которой стоит точка с запятой. То есть значением присваивается такое выражение:

```
delegate(int n){
   return A.number+n;
}
```

Это и есть описание анонимного метода. У него один целочисленный аргумент (обозначен как `n`), а результатом метод возвращает значение выражения `A.number+n` (сумма значения поля `number` объекта `A` и аргумента метода `n`). Именно такой код будет выполняться при вызове экземпляра делегата, на который ссылается поле `method` объекта `A`.

Полю `method` объекта `B` также присваивается анонимный метод, но немного другой. Соответствующая команда выглядит следующим образом:

```
B.method=delegate(int n){
   return B.number-n;
};
```

В данном случае анонимный метод определен таким образом, что результатом он возвращает разность значения поля `number` объекта `B` и аргумента метода `n`.

Работу анонимных методов проверяем, вызывая соответствующие экземпляры делегатов командами `A.method(x))` и `B.method(x)` (при значении `80` для переменной `x`). Легко убедиться, что результат вычисляется в соответствии с тем, как мы определили анонимные методы.

НА ЗАМЕТКУ

Обратите внимание на одно обстоятельство. Объекты `A` и `B` созданы на основе класса `MyClass`, поэтому у каждого из объектов есть поле `method`. Поскольку поле является ссылкой на экземпляр делегата, мы можем его вызывать как метод. Создается иллюзия, что мы имеем дело с методом. Но этот «метод» у разных объектов разный, хотя объекты относятся к одному классу. Фактически с помощью анонимных методов мы можем менять код, связанный с полем `method`, таким же образом, как мы меняем значение для числовых полей.

Также в главном методе командой `Bravo show` объявляется переменная `show` с типом `Bravo` (делегат). Значением такой переменной можно присвоить ссылку на метод, принимающий текстовый аргумент и не возвращающий результат. Сначала значение переменной `show` присваивается такой командой:

```
show=delegate(string t){
   Console.WriteLine("Текст: \"{0}\"",t);
};
```

Переменной значением присваивается анонимный метод, который отображает в консольном окне сообщение, содержащее в двойных кавычках текст, переданный аргументом методу. После выполнения такого присваивания командой `show("Bravo")` через переменную `show` вызываем экземпляр делегата и проверяем работу анонимного метода. Затем переменной `show` присваивается в качестве значения новый анонимный метод, как показано ниже:

```
show=delegate(string t){
   for(int k=0;k<t.Length;k++){
      Console.Write("|"+t[k]);
   }
   Console.WriteLine("|");
};
```

На этот раз текстовый аргумент метода отображается посимвольно в консольном окне, с использованием вертикальной черты в качестве разделителя символов. С помощью команды `show("Bravo")` убеждаемся в том, что все происходит так, как описано в анонимном методе.

НА ЗАМЕТКУ

Присваивая переменной `show` в качестве значения различные анонимные методы, мы создаем впечатление, что имеем дело с методом, программный код которого меняется по мере выполнения программы.

Использование анонимных методов

Что-то вы меня больно утесняете, папаша.

из к/ф «Собачье сердце»

Мы рассмотрим несколько примеров, в которых используются анонимные методы. Примеры призваны послужить иллюстрацией к тому, где, когда и как анонимные методы могут оказаться полезными.

Выше упоминалось, что анонимные методы предназначены для присваивания в качестве значений переменным типа делегата. Но это вовсе не означает, что мы не можем передать анонимный метод аргументом другому методу или даже конструктору (правда тут имеются некоторые особенности). Противоречия здесь нет, поскольку соответствующий аргумент описывается как переменная типа делегата, и когда фактическим аргументом передается анонимный метод, то технически переменной типа делегата значением присваивается анонимный метод. То есть все формальности будут выдержаны.

Сначала рассмотрим очень простой пример, в котором имеется делегат `MyDelegate`, соответствующий методам с одним целочисленным аргументом и символьным результатом. В программе описывается класс `MyClass`, у которого есть открытое символьное поле `symbol`, конструктор с двумя аргументами и метод `set()`. Также в классе описано поле `get`, которое является ссылкой на экземпляр делегата `MyDelegate`. Первым аргументом конструктору передается символьное значение, присваиваемое полю `symbol`. Второй аргумент конструктора является

ссылкой на экземпляр делегата `MyDelegate`. Эта ссылка записывается в поле `get`. Программный код представлен в листинге 2.8.

Листинг 2.8. Анонимный метод как аргумент

```
using System;
// Объявление делегата:
delegate char MyDelegate(int n);
// Класс:
class MyClass{
   // Открытое символьное поле:
   public char symbol;
   // Поле, являющееся ссылкой на экземпляр делегата:
   public MyDelegate get;
   // Конструктор, второй аргумент которого ссылка на
   // экземпляр делегата:
   public MyClass(char s,MyDelegate md){
      // Присваивание значения полю:
      symbol=s;
      // Использование ссылки на экземпляр делегата:
      get=md;
   }
   // Метод с аргументом, являющимся ссылкой на экземпляр
   // делегата:
   public void set(MyDelegate md){
      // Использование ссылки на экземпляр делегата:
      get=md;
   }
}
// Класс с главным методом:
class AnonymMethAsArgDemo{
   // Главный метод:
   static void Main(){
```

```
      // Создание объекта. Вторым аргументом является
      // анонимный метод:
      MyClass obj=new MyClass('K', // Символьный аргумент
         delegate(int n){          // Анонимный метод
            return (char)('A'+n);
         }
      );
      // Вызов экземпляра делегата:
      Console.WriteLine("Символ: \'{0}\'",obj.get(3));
      // Вызов метода, аргументом которому передан
      // анонимный метод:
      obj.set(
         delegate(int n){
            return (char)(obj.symbol+n);
         }
      );
      // Вызов экземпляра делегата:
      Console.WriteLine("Символ: \'{0}\'",obj.get(3));
   }
}
```

Ниже показан результат выполнения программы:

Результат выполнения программы (из листинга 2.8)

```
Символ: 'D'
Символ: 'N'
```

Для нас первоочередной интерес представляет код главного метода программы. В нем создается объект `obj` класса `MyClass`. Причем второй аргумент конструктора описан довольно громоздким выражением, которое является описанием анонимного метода:

```
delegate(int n){
   return (char)('A'+n);
}
```

Вся эта конструкция является вторым аргументом конструктора. Это анонимный метод, который при вызове с целочисленным аргументом n результатом возвращает символ, смещенный по отношению к символу 'A' на n позиций. Ссылка на этот анонимный метод записывается через посредника (экземпляр делегата) в поле get. Поэтому при вызове данного экземпляра делегата командой obj.get(3) в качестве результата возвращается символ, отстоящий в кодовой таблице на 3 позиции (в направлении увеличения кода) по отношению к символу 'A'.

НА ЗАМЕТКУ

То есть в данном случае поле symbol объекта при определении анонимного метода не используется. Это, в общем-то, не случайно. При создании объекта obj, когда вызывается конструктор, ссылка obj еще не может использоваться. Поэтому в теле анонимного метода ссылок на объект obj нет.

Далее из объекта obj вызывается метод set(), которому в качестве аргумента передается следующая конструкция:

```
delegate(int n){
   return (char)(obj.symbol+n);
}
```

Это анонимный метод, который присваивается в качестве значения полю get объекта obj. Результатом метод возвращает символ, смещенный на n (аргумент метода) позиций по отношению к символьному значению поля symbol объекта obj. При создании объекта obj поле symbol получило значение 'K'. Поэтому значением выражения obj.get(3) будет символ, смещенный на 3 позиции по отношению к символу 'K'.

ПОДРОБНОСТИ

При вызове метода set() мы использовали анонимный метод, содержащий ссылку obj.symbol. Здесь нет ошибки, поскольку на момент вызова метода set() в переменную obj уже была записана ссылка на объект. В конструкторе, при создании объекта obj, мы не могли использовать ссылку obj.symbol. Причина очевидна: сначала вычисляются аргументы конструктора, а уже затем выполняется его код. Ссылка на объект записывается в переменную obj после того, как объект создан. Поэтому при вычислении аргументов конструктора переменная obj еще не содержит ссылку на объект.

Анонимный метод может возвращаться в качестве результата другого метода. В листинге 2.9 по этому поводу представлен пример.

Листинг 2.9. Анонимный метод как результат

```
using System;
// Объявление делегата:
delegate int MyDelegate();
// Класс с главным методом:
class AnonymAsResultDemo{
   // Статически метод результатом возвращает ссылку на
   // экземпляр делегата:
   static MyDelegate calculate(int n){
      // Локальная переменная:
      int count=0;
      // Результат реализован через анонимный метод:
      return delegate(){
         count+=n;
         return count;
      };
   }
   // Главный метод:
   static void Main(){
      // Переменная типа делегата:
      MyDelegate next=calculate(1);
      for(int i=1;i<=5;i++){
         // Вызов экземпляра делегата:
         Console.Write(next()+" ");
      }
      Console.WriteLine();
      // Новое значение переменной типа делегата:
      next=calculate(3);
            for(int i=1;i<=5;i++){
```

```
            // Вызов экземпляра делегата:
            Console.Write(next()+" ");
        }
        Console.WriteLine();
    }
}
```

При выполнении программы получаем такой результат:

Результат выполнения программы (из листинга 2.9)

```
1 2 3 4 5
3 6 9 12 15
```

Делегат `MyDelegate` описан для создания экземпляров, которые могут ссылаться на методы без аргументов, возвращающие целочисленный результат. Еще мы описываем статический метод `calculate()`, у которого целочисленный аргумент (обозначен как `n`) и который результатом возвращает ссылку на экземпляр делегата `MyDelegate`. В теле метода объявлена локальная целочисленная переменная `count` с начальным нулевым значением. Далее следует `return`-инструкция и возвращаемым значением указано следующее выражение (после этого выражения в коде метода стоит точка с запятой):

```
delegate(){
    count+=n;
    return count;
}
```

Это код анонимного метода без аргументов, который результатом возвращает целое число. Этот анонимный метод возвращается результатом метода `calculate()`.

ПОДРОБНОСТИ

Тип результата для метода `calculate()` указан как `MyDelegate`. Формально это ссылка на экземпляр делегата `MyDelegate`. При вызове метода `calculate()` под результат метода выделяется переменная типа делегата `MyDelegate`. Результатом метод `calculate()`

возвращает анонимный метод, который присваивается переменной, выделенной для записи результата. В итоге создается экземпляр делегата, который ссылается на анонимный метод. Ссылка на этот экземпляр записывается в переменную, выделенную для запоминания результата метода `calculate()`. Поэтому результатом метода `calculate()` является ссылка на экземпляр делегата `MyDelegate`, а этот экземпляр ссылается на упомянутый выше анонимный метод.

В главном методе программы командой `MyDelegate next=calculate(1)` объявляется переменная `next` типа делегата `MyDelegate`, и значением ей присваивается результат вызова метода `calculate()` с аргументом `1`. Затем с помощью оператора цикла несколько раз подряд вычисляется значение выражения `next()`. И каждый раз мы получаем новое значение (числа `1`, `2`, `3` и так далее). Почему так происходит? При вызове метода `calculate()` с аргументом `1` создается локальная переменная `count` с начальным значением `0`. Мы уже знаем, что метод `calculate()` возвращает результатом анонимный метод, который при вызове каждый раз увеличивает значение переменной `count` (в данном случае на `1`) и это новое значение возвращает как результат. В итоге получается так: переменная `next` ссылается на экземпляр делегата, который ссылается на анонимный метод, а этот метод содержит ссылку на переменную `count`, созданную при вызове метода `calculate()`. После завершения вызова метода `calculate()` локальная переменная `count` должна бы удаляться. Но поскольку на эту переменную есть ссылка в анонимном методе, то она «выживает». После первого вызова анонимного метода через переменную `next` значение переменной `count` становится равным `1`. При втором вызове она становится равной `2` и так далее. Поэтому при вычислении значения выражения `next()` мы каждый раз получаем новое число (на единицу больше предыдущего).

После выполнения команды `next=calculate(3)` в переменную `next` записывается ссылка на новый экземпляр делегата, ссылающийся на новый анонимный метод, который ссылается на новую локальную переменную `count`. Она получила начальное значение `0` при вызове метода `calculate()`. И теперь каждый раз при вычислении значения выражения `next()` мы получаем новое число, большее предыдущего на `3`.

Еще одна программа немного напоминает предыдущую. Правда на этот раз последствия использования анонимного метода в качестве результата не такие экзотические. Обратимся к программному коду из листинга 2.10.

Листинг 2.10. Реализация результата с помощью анонимного метода

```
using System;
// Объявление делегата:
delegate double Powers(double x);
// Главный класс:
class MoreAnonymAsResultDemo{
   // Статический метод результатом возвращает ссылку
   // на экземпляр делегата:
   static Powers maker(int n){
      // Локальная переменная типа делегата:
      Powers meth;
      // Значением переменной типа делегата присваивается
      // анонимный метод:
      meth=delegate(double x){
         double s=1;
         // Вычисление произведения:
         for(int i=1;i<=n;i++){
            s*=x;
         }
         return s; // Результат анонимного метода
      };
      return meth; // Результат статического метода
   }
   // Главный метод:
   static void Main(){
      // Первый экземпляр делегата:
      Powers sqr=maker(2);
      // Второй экземпляр делегата:
      Powers cube=maker(3);
      // Вызов экземпляров делегата:
      for(int i=1;i<=5;i++){
```

```
            Console.WriteLine("{0,2}:{1,5}{2,5}{3,5}", i,sqr(i),cube(i),maker(4)(i));
        }
    }
}
```

При выполнении программы получаем следующий результат:

Результат выполнения программы (из листинга 2.10)

```
 1:    1    1    1
 2:    4    8   16
 3:    9   27   81
 4:   16   64  256
 5:   25  125  625
```

Здесь мы объявляем делегат `Powers` для работы с методами, у которых один аргумент типа `double` и такого же типа результат. В главном методе мы описываем статический метод `maker()`. Он имеет целочисленный аргумент (обозначен как `n`), а в качестве результата возвращает ссылку на экземпляр делегата `Powers`. В теле метода объявляется локальная переменная `meth` типа `Powers`. Значением этой переменной присваивается такой анонимный метод:

```
delegate(double x){
    double s=1;
    for(int i=1;i<=n;i++){
        s*=x;
    }
    return s;
}
```

У метода один аргумент (обозначен как `x`) типа `double`, а результатом метод возвращает значение аргумента `x` в степени, определяемой параметром `n` (аргумент метода `maker()`). Для вычисления результата использован оператор цикла, в котором переменная `s` с начальным значением `1` умножается на `x` ровно `n` раз. Ссылка на экземпляр делегата, на который ссылается переменная `meth`, возвращается результатом

метода maker(). Таким образом, при вызове метода maker() с некоторым целочисленным аргументом n мы получаем ссылку на метод (точнее, ссылку на экземпляр делегата, который ссылается на метод). Этот метод при вызове с аргументом x возвращает результатом x в степени n.

В главном методе командами Powers sqr=maker(2) и Powers cube=maker(3) объявляются две переменные sqr и cube типа Powers, и им присваиваются значения. Значением выражения вида sqr(i) является значение аргумента i в степени 2, а значение выражения cube(i) — это значение аргумента i в степени 3. Далее, значение выражения maker(4) — это ссылка на метод, который результатом возвращает значение своего аргумента в четвертой степени. Поэтому значение выражения вида maker(4)(i) есть не что иное, как значение аргумента i в степени 4. Таблица, отображаемая в результате выполнения программы, подтверждает наши выводы.

Лямбда-выражения

А, ничего опасного. Никакой контрреволюции.
из к/ф «Собачье сердце»

У анонимных методов есть конкурент — это *лямбда-выражения*. Лямбда-выражение представляет собой блок программного кода, который определяет метод и предназначен для присваивания в качестве значения переменной типа делегата. В этом смысле и лямбда-выражения, и анонимные методы служат одной цели. Разница лишь в том, как эта цель достигается.

Итак, лямбда-выражение определяет метод. Как же это происходит? Синтаксис описания лямбда-выражения идеологически близок к описанию анонимного метода, но в несколько упрощенном виде. Условно лямбда-выражение можно рассматривать как состоящее из двух частей (левой и правой). Между ними размещается оператор =>. Левая часть определяет аргументы метода. В общем случае аргументы описываются в круглых скобках, для каждого аргумента указывается его тип и название. То есть фактически это круглые скобки с описанием аргументов метода. Отсутствие аргументов отображается пустыми круглыми скобками. В правой части, после оператора => описывается выделенный фигурными скобками блок, в котором размещаются команды,

формирующие тело метода. Общий шаблон описания лямбда-метода такой (жирным шрифтом выделены основные элементы шаблона):

```
(аргументы)=>{
   // Тело метода
}
```

Строго говоря, если взять описание анонимного метода, убрать ключевое слово `delegate` и между блоками из круглых и квадратных скобок поставить оператор =>, то получим лямбда-выражение. Оно используется точно так же, как и анонимный метод. Вместе с тем при использовании лямбда-выражений можно использовать некоторые упрощения, которые часто делают описание соответствующего метода компактным. Ниже перечислены основные правила, которые позволяют «упростить жизнь» при работе с лямбда-выражениями:

- Тип аргументов можно не указывать, если по контексту команды с лямбда-выражением они могут быть определены автоматически (на основе переменной типа делегата, которой присваивается лямбда-выражение).
- Если аргумент один и его тип не указывается, то круглые скобки можно не использовать.
- Если в теле метода всего одна команда, то фигурные скобки можно не использовать.
- Если в теле метода всего одна команда с `return`-инструкцией, то инструкцию `return` можно не использовать.

Например, ниже приведено лямбда-выражение, определяющее метод, который по символьному и целочисленному аргументу вычисляет новый символ:

```
(char s,int n)=>{
   int k=s+n;
   return (char)k;
}
```

Мы могли бы определить такой же метод несколько иначе:

```
(s,n)=>{
   return (char)(s+n);
}
```

Или еще проще:

```
(s,n)=>(char)(s+n)
```

А вот пример лямбда-выражения для метода, который для заданного значения аргумента возвращает квадрат аргумента (произведение аргумента на самого себя):

```
n=>n*n
```

То же самое, с помощью анонимного метода, мы могли бы описать следующим образом:

```
delegate(int n){
   return n*n;
}
```

То есть экономия на программном коде очевидная. Далее мы рассмотрим несколько примеров, которые иллюстрируют, как вместо анонимных методов можно было бы использовать лямбда-выражения. В листинге 2.11 представлена версия программы из листинга 2.7, но вместо анонимных методов использованы лямбда-выражения (здесь и далее для сокращения объема кода несущественные комментарии удалены, а важные места выделены жирным шрифтом).

Листинг 2.11. Знакомство с лямбда-выражениями

```
using System;
delegate int Alpha(int n);
delegate void Bravo(string t);
class MyClass{
   public int number;
   public MyClass(int n){
      number=n;
   }
   public Alpha method;
}
```

```
class LambdaDemo{
   static void Main(){
      MyClass A=new MyClass(100);
      MyClass B=new MyClass(200);
      // Полю объекта значением присваивается
      // лямбда-выражение:
      A.method=n=>A.number+n;
      // Полю объекта значением присваивается
      // лямбда-выражение:
      B.method=(int n)=>{
         return B.number-n;
      };
      int x=80;
      Console.WriteLine("A.method({0})={1}",x,A.method(x));
      A.number=300;
      Console.WriteLine("A.method({0})={1}",x,A.method(x));
      Console.WriteLine("B.method({0})={1}",x,B.method(x));
      Bravo show;
      // Присваивание переменной лямбда-выражения:
      show=t=>Console.WriteLine("Текст: \"{0}\"",t);
      show("Bravo");
      // Присваивание переменной лямбда-выражения:
      show=(string t)=>{
         for(int k=0;k<t.Length;k++){
            Console.Write("|"+t[k]);
         }
         Console.WriteLine("|");
      };
      show("Bravo");
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 2.11)

```
A.method(80)=180
A.method(80)=380
B.method(80)=120
Текст: "Bravo"
|B|r|a|v|o|
```

В листинге 2.12 представлена новая версия программы из листинга 2.8.

Листинг 2.12. Лямбда-выражение как аргумент

```
using System;
delegate char MyDelegate(int n);
class MyClass{
   public char symbol;
   public MyDelegate get;
   public MyClass(char s,MyDelegate md){
      symbol=s;
      get=md;
   }
   public void set(MyDelegate md){
      get=md;
   }
}
class LambdaAsArgDemo{
   static void Main(){
      // Создание объекта. Вторым аргументом является
      // лямбда-выражение:
      MyClass obj=new MyClass('K',n=>(char)('A'+n));
      Console.WriteLine("Символ: \'{0}\'",obj.get(3));
      // Вызов метода, аргументом которому передано
      // лямбда-выражение:
```

```
      obj.set(n=>(char)(obj.symbol+n));
      Console.WriteLine("Символ: \'{0}\'",obj.get(3));
   }
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 2.12)

```
Символ: 'D'
Символ: 'N'
```

Программа в листинге 2.13 — это новая версия программы из листинга 2.9.

Листинг 2.13. Лямбда-выражение как результат

```
using System;
delegate int MyDelegate();
class LambdaAsResultDemo{
   static MyDelegate calculate(int n){
      int count=0;
      // Результат реализован через лямбда-выражение:
      return ()=>{
         count+=n;
         return count;
      };
   }
   static void Main(){
      MyDelegate next=calculate(1);
      for(int i=1;i<=5;i++){
         Console.Write(next()+" ");
      }
      Console.WriteLine();
      next=calculate(3);
            for(int i=1;i<=5;i++){
```

```
            Console.Write(next()+" ");
        }
        Console.WriteLine();
    }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 2.13)

```
1 2 3 4 5
3 6 9 12 15
```

Наконец, в листинге 2.14 показано, как будет выглядеть программа из листинга 2.10, если вместо анонимных методов использовать лямбда-выражения.

Листинг 2.14. Реализация результата с помощью лямбда-выражения

```
using System;
delegate double Powers(double x);
class MoreLambdaAsResultDemo{
    static Powers maker(int n){
        Powers meth;
        // Значением переменной типа делегата присваивается
        // лямбда-выражение:
        meth=x=>{
            double s=1;
            for(int i=1;i<=n;i++){
                s*=x;
            }
            return s;
        };
        return meth;
    }
    static void Main(){
```

```
        Powers sqr=maker(2);
        Powers cube=maker(3);
        for(int i=1;i<=5;i++){
           Console.WriteLine("{0,2}:{1,5}{2,5}{3,5}",i,sqr(i),cube(i),maker(4)(i));
        }
    }
}
```

Ниже приведен результат выполнения программы:

Результат выполнения программы (из листинга 2.14)

```
 1:    1    1    1
 2:    4    8   16
 3:    9   27   81
 4:   16   64  256
 5:   25  125  625
```

Во всех случаях, хотя программный код изменился, результат выполнения программ такой же, как и у их предшественниц.

Несложно заметить, что для небольших (по объему кода) методов использование лямбда-выражений значительно упрощает структуру и читабельность программы.

НА ЗАМЕТКУ

Не все лямбда-выражения в примерах выше описаны оптимальным образом. Причина связана с желанием показать различные способы описания лямбда-выражений.

Но различие между лямбда-выражениями и анонимными методами не сводится к чисто косметическим эффектам. Например, лямбда-выражения могут использоваться для описания индексаторов и свойств, которые имеют только `get`-аксессор. В таком случае вместо блока из фигурных скобок с аксессорами для свойства или индексатора указывается оператор => и значение, считываемое для свойства или индексатора. Небольшой пример по этому поводу представлен в листинге 2.15.

Листинг 2.15. Лямбда-выражения в свойствах и индексаторах

```
using System;
// Класс со свойством и индексатором:
class MyClass{
   // Закрытое текстовое поле:
   private string text;
   // Конструктор с текстовым аргументом:
   public MyClass(string t){
      text=t;
   }
   // Индексатор:
   public char this[int k]=>text[k];
   // Свойство:
   public int length=>text.Length;
}
// Класс с главным методом:
class MoreLambdaDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта:
      MyClass obj=new MyClass("Alpha");
      // Использование индексатора и свойства:
      for(int k=0;k<obj.length;k++){
         Console.Write("|"+obj[k]);
      }
      Console.WriteLine("|");
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 2.15)

```
|A|l|p|h|a|
```

Программа простая: описывается класс `MyClass` с закрытым текстовым полем `text`. Значение полю присваивается при создании объекта (значение передается аргументом конструктору). Индексатор описан так:

```
public char this[int k]=>text[k];
```

Эта запись означает, что при считывании значения проиндексированного объекта результатом возвращается символ из текстового поля с индексом, указанным в квадратных скобках. Похожим образом описано целочисленное свойство `length`:

```
public int length=>text.Length;
```

Значением свойства возвращается размер текста в текстовом поле. В главном методе проверяется функциональность созданного кода класса.

В завершение раздела заметим, что лямбда-выражения считаются более предпочтительным механизмом по сравнению с анонимными методами.

Знакомство с событиями

— Это неслыханно! Что вы здесь делаете?
— Я тут... спим.

из к/ф «Ирония судьбы или с легким паром»

Представим ситуацию, когда есть некоторая программа, а в этой программе имеется несколько объектов, которые «взаимодействуют» друг с другом. Мы хотим добиться того, чтобы при стечении определенных условий для одного из объектов другие объекты на это «среагировали». Объекты «реагируют» через свои методы. Специфика ситуации в том, что решение о необходимости вызвать метод объекту поступает извне, от другого объекта. И такое сообщение должно генерироваться автоматически (как только «звезды расположились нужным образом»). В принципе, для концептуального решения такой задачи нам пригодились бы

экземпляры делегата: с их помощью можно в динамическом режиме устанавливать связи между объектами. Остается открытым вопрос с автоматическим оповещением. И вот в этом случае используется специальный член класса, который называется *событием*.

Событие описывается в классе. Как и для других членов класса, для событий можно указывать спецификатор уровня доступа (обычно это `public`). Затем указывается ключевое слово `event`. После ключевого слова `event` указывается имя делегата и затем уже название события. Название события — это имя, которое мы даем члену класса, подобно тому как мы даем имя полю или методу. Делегат, который указывается в объявлении события, фактически определяет «тип» события. Дело в том, что при возникновении события для его обработки могут вызываться экземпляры делегата. В объявлении события мы указываем имя делегата, экземпляры которого могут вызываться для обработки события.

Таким образом, шаблон объявления события (являющегося открытым членом класса) выглядит следующим образом (жирным шрифтом выделены ключевые элементы шаблона):

```
public event делегат событие;
```

Например, рассмотрим такое объявление:

```
public event MyDelegate MyEvent;
```

В данном случае объявляется событие `MyEvent`, являющееся открытым членом класса (ключевое слово `public`), а для обработки этого события могут вызываться экземпляры делегата `MyDelegate`.

НА ЗАМЕТКУ

Если в объявлении события убрать ключевое слово `event`, то фактически получим объявление поля, являющегося ссылкой на экземпляр делегата.

Все это касается формальной стороны — объявления события. Но остается открытым вопрос о том, что с событием можно делать. Есть две концептуальные задачи, которые решаются при работе с событиями. Во-первых, событие можно вызвать (или сгенерировать). Во-вторых, событие можно связать с экземплярами делегата, которые будут автоматически вызываться при генерировании (возникновении) события. Начнем со второй задачи.

НА ЗАМЕТКУ

Связанные с событием методы, которые вызываются при возникновении события, называются обработчиками события.

Для добавления ссылки на экземпляр делегата, который будет вызываться при генерировании события, этот экземпляр (ссылка на него) добавляется с помощью оператора += (только сокращенная форма!). Используется команда вида `событие+=экземпляр`. На практике вместо ссылки на экземпляр делегата можно использовать ссылку на метод, соответствующий делегату, с которым объявлено событие. То есть можно использовать команду вида `событие+=метод`. После выполнения такой команды, если произойдет событие, то автоматически будет вызван соответствующий метод.

ПОДРОБНОСТИ

При добавлении в событие ссылки на экземпляр делегата или ссылки на метод с помощью оператора += вызывается специальный метод-аксессор. При этом ожидается, что в правой части оператора += указана ссылка на экземпляр делегата. Для запоминания этой ссылки выделяется специальная переменная типа делегата. Если вместо ссылки на экземпляр делегата указана ссылка на метод, то получается, что переменной делегата в качестве значения присваивается указатель на метод. В этом случае автоматически создается экземпляр делегата, который ссылается на метод. Ссылка на этот экземпляр записывается в переменную делегата.

Если мы хотим удалить ссылку на экземпляр делегата из события, то для этой цели используется оператор -= (тоже только сокращенная форма!). Ситуация такая же, как и с оператором +=: мы можем использовать команды вида `событие-=экземпляр` или `событие-=метод`. То есть вместо ссылки на экземпляр мы можем использовать ссылку на метод.

Итак, при возникновении события автоматически вызываются экземпляры делегатов, связанные с этим событием. Но как сгенерировать событие? Ответ простой: его нужно вызвать подобно вызову метода. Указываются имя события и круглые скобки — если нужно, то с аргументами. Количество и тип аргументов определяются делегатом, указанным в объявлении события. Важная особенность события связана с тем, что вызвать (сгенерировать) событие можно только внутри класса.

Сгенерировать событие через ссылку на объект класса не получится, даже если событие в классе объявлено с ключевым словом `public`. Через объект доступны только операции += и −=, с помощью которых в событие добавляются и удаляются ссылки на экземпляры делегата.

НА ЗАМЕТКУ

Спецификатор уровня доступа в объявлении события определяет наследуемость свойства в производном классе и доступность операций += и −= за пределами класса. В любом случае, если в классе есть событие, то вызвать (сгенерировать) событие можно только в программном коде этого класса.

Небольшой пример, в котором используются события, представлен в листинге 2.16.

Листинг 2.16. Знакомство с событиями

```
using System;
// Объявление делегата:
delegate void MyDelegate(string txt);
// Класс с событием:
class MyClass{
   // Объявление события:
   public event MyDelegate MyEvent;
   // Метод для генерирования события:
   public void RaiseMyEvent(string txt){
      // Если для события предусмотрены обработчики:
      if(MyEvent!=null){
         // Генерирование события:
         MyEvent(txt);
      }
   }
}
// Класс:
class Alpha{
```

```
   // Текстовое поле:
   public string name;
   // Конструктор:
   public Alpha(string txt){
      name=txt;
   }
   // Метод отображает сообщение:
   public void show(string msg){
      Console.WriteLine("Объект "+name+":");
      Console.WriteLine(msg);
   }
}
// Класс с главным методом:
class EventDemo{
   // Главный метод:
   static void Main(){
      // Создание объекта с событием:
      MyClass obj=new MyClass();
      // Создание объектов:
      Alpha A=new Alpha("A");
      Alpha B=new Alpha("B");
       // Попытка сгенерировать событие:
      obj.RaiseMyEvent("1-е событие");
      // Добавление обработчика для события:
      obj.MyEvent+=A.show;
      // Генерирование события:
      obj.RaiseMyEvent("2-е событие");
      Console.WriteLine();
      // Добавление обработчика для события:
      obj.MyEvent+=B.show;
      // Генерирование события:
      obj.RaiseMyEvent("3-е событие");
```

```
        Console.WriteLine();
        // Удаление метода из списка обработчиков события:
        obj.MyEvent-=A.show;
        // Генерирование события:
        obj.RaiseMyEvent("4-е событие");
        Console.WriteLine();
        // Удаление методов из списка обработчиков события:
        obj.MyEvent-=A.show;
        obj.MyEvent-=B.show;
        // Попытка сгенерировать событие:
        obj.RaiseMyEvent("5-е событие");
        // Создание экземпляра делегата:
        MyDelegate md=A.show;
        // Добавление метода в список вызовов
        // экземпляра делегата:
        md+=B.show;
        // Добавление экземпляра делегата в список
        // обработчиков события:
        obj.MyEvent+=md;
        // Генерирование события:
        obj.RaiseMyEvent("6-е событие");
    }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 2.16)

```
Объект A:
2-е событие

Объект A:
3-е событие
```

```
Объект B:
3-е событие

Объект B:
4-е событие

Объект A:
6-е событие
Объект B:
6-е событие
```

В программе объявлен делегат `MyDelegate`. Экземпляры делегата могут ссылаться на методы, имеющие один текстовый аргумент и не возвращающие результат.

Класс `MyClass` содержит объявление события `MyEvent`. Оно описано с ключевым словом `event`, типом события указан делегат `MyDelegate`. Это означает, что для обработки события могут использоваться методы, сигнатура которых соответствует характеристикам делегата `MyDelegate`: у метода должен быть текстовый аргумент, и метод не должен возвращать результат. Еще в классе `MyClass` описан открытый метод `RaiseMyEvent()` с текстовым аргументом (обозначен как `txt`). Метод предназначен для генерирования события `MyEvent`. В теле метода с помощью условного оператора проверяется условие `MyEvent!=null`, и если оно истинно, то командой `MyEvent(txt)` генерируется событие `MyEvent`. После генерирования события командой `MyEvent(txt)` автоматически вызываются все методы, зарегистрированные в качестве обработчиков данного события. Каждому из методов аргументом передается текстовое значение `txt`.

ПОДРОБНОСТИ

Зачем нам нужен метод `RaiseMyEvent()`? Понятно, что событие генерируется при выполнении команды `MyEvent(txt)`. Но дело в том, что вызвать (сгенерировать) событие командой `MyEvent(txt)` мы можем только в теле класса. Поэтому мы описываем открытый метод `RaiseMyEvent()`, который можно вызвать через ссылку на объект класса `MyClass`. А при вызове метода `RaiseMyEvent()` уже будет

сгенерировано событие MyEvent (для объекта, из которого вызывается метод RaiseMyEvent()).
Если при генерировании события окажется, что список обработчиков для этого события пустой, то возникнет ошибка. Поэтому перед генерированием события командой MyEvent(txt) в теле метода RaiseMyEvent() сначала проверяется условие MyEvent!=null. Оно истинно, если список обработчиков для события не пустой. Только в этом случае генерируется событие.

В классе Alpha есть открытое текстовое поле name. Конструктор с одним текстовым аргументом позволяет присвоить значение полю при создании объекта. Также в классе есть метод show(). Метод не возвращает результат и имеет один текстовый аргумент (обозначен как msg). Таким образом, метод show() полностью соответствует характеристикам делегата MyDelegate и может быть использован как обработчик события MyEvent. В теле метода show() командой Console.WriteLine("Объект "+name+":") отображается значение поля name объекта, из которого вызывается метод, а затем командой Console.WriteLine(msg) отображается значение аргумента, переданного методу.

В главном методе программы мы создаем объект obj класса MyClass, а также объекты A и B класса Alpha.

Сначала командой obj.RaiseMyEvent("1-е событие") мы пытаемся сгенерировать событие MyEvent для объекта obj. Но на момент выполнения данной команды список обработчиков для события MyEvent объекта obj пуст, поэтому ничего не происходит. Затем командой obj.MyEvent+=A.show в качестве обработчика события регистрируется метод show() объекта A. Поэтому после генерирования события командой obj.RaiseMyEvent("2-е событие") из объекта A вызывается метод show() с аргументом "2-е событие".

На следующем этапе командой obj.MyEvent+=B.show для события обработчиком дополнительно регистрируется еще и метод show() объекта B. Поэтому при генерировании события командой obj.RaiseMyEvent("3-е событие") из объектов A и B с аргументом "3-е событие" последовательно вызывается метод show().

Командой obj.MyEvent-=A.show метод show() объекта A удаляется из списка обработчиков для события MyEvent объекта obj (у события остается один обработчик — метод show() объекта B). При генерировании события командой obj.RaiseMyEvent("4-е событие")

с аргументом "4-е событие" вызывается метод show() объекта B. Далее командами obj.MyEvent-=A.show и obj.MyEvent-=B.show из списка обработчиков события удаляются методы show() объектов A и B. Причем на момент выполнения команды obj.MyEvent-=A.show метода show() объекта A в списке обработчиков нет, так что здесь имеет место попытка удалить отсутствующий в списке метод. Но, как видим, к ошибке это не приводит. А общий итог такой, что список обработчиков события MyEvent объекта obj становится пустым. Как следствие, при выполнении команды obj.RaiseMyEvent("5-е событие") ничего не происходит (сообщения в консольном окне не появляются).

Как отмечалось ранее, в список обработчиков могут добавляться экземпляры делегатов или методы. Методы мы уже добавляли. На финальном этапе прибегнем к помощи экземпляров делегата MyDelegate. Командой MyDelegate md=A.show создается экземпляр md, и в список вызовов делегата добавляется ссылка на метод show() объекта A. Командой md+=B.show туда же добавляется ссылка на метод show() объекта B. А затем командой obj.MyEvent+=md в список обработчиков события MyEvent объекта obj добавляется делегат md. При генерировании события командой obj.RaiseMyEvent("6-е событие") вызывается экземпляр делегата md с аргументом "6-е событие". Вызов этого экземпляра означает вызов метода show() из объектов A и B.

На первый взгляд может показаться, что между событием и экземпляром делегата принципиальной разницы нет. Правда мы не можем «вызвать» (или сгенерировать) событие вне пределов класса, а при операциях с событием можем использовать только операторы += и -=: во всем остальном события и экземпляры делегатов действительно похожи. Но разница между ними все же имеется, причем принципиальная. Событие от экземпляра делегата отличается примерно так же, как свойство отличается от текстового поля (роль текстового поля в данном случае играет ссылка на экземпляр делегата). За событием «спрятана» ссылка на экземпляр делегата и два метода-аксессора, которые отвечают за добавление методов в список обработчиков события и за удаление методов из списка обработчиков события. Если в классе объявлено событие, то все это «богатство» (ссылка на экземпляр делегата и методы-аксессоры) создается автоматически, без нашего участия. Но мы можем реализовать все самостоятельно, в явном виде.

Событие можно объявлять с явным описанием аксессоров. Аксессор, вызываемый при добавлении (с помощью оператора +=) метода в список

обработчиков события, называется add. Аксессор, который вызывается при удалении (с помощью оператора -=) ссылки на метод из списка обработчиков события, называется remove. Ключевое слово value обозначает значение (ссылка на метод или экземпляр делегата), находящееся в правой части выражения с оператором += или -=. Общий шаблон объявления события в таком случае имеет следующий вид (жирным шрифтом выделены ключевые элементы шаблона):

```
public event делегат событие{
   add{
      // Команды add-аксессора
   }
   remove{
      // Команды remove-аксессора
   }
}
```

Фактически событие в этом случае описывается как свойство, только вместо get-аксессора и set-аксессора используются add-аксессор и remove-аксессор. Преимущество такого способа описания события в том, что мы можем изменить используемый по умолчанию алгоритм добавления (удаления) методов в список (из списка) обработчиков события. Как иллюстрацию к сказанному рассмотрим аналог предыдущей программы, но только на этот раз для реализации события мы воспользуемся явным описанием аксессоров, а для записи списка обработчиков события будем использовать закрытое поле, являющееся ссылкой на экземпляр делегата. Интересующий нас программный код представлен в листинге 2.17.

Листинг 2.17. Событие с явным описанием аксессоров

```
using System;
// Объявление делегата:
delegate void MyDelegate(string txt);
// Класс с событием:
class MyClass{
   // Закрытое поле (ссылка на экземпляр делегата)
   // для реализации списка обработчиков события:
```

```
   private MyDelegate myevent;
   // Объявление события (с описанием аксессоров):
   public event MyDelegate MyEvent{
      // Аксессор для добавления метода в список
      // обработчиков события:
      add{
         // Добавление ссылки на метод в список вызовов
         // экземпляра делегата:
         myevent+=value;
      }
      // Аксессор для удаления метода из списка
      // обработчиков события:
      remove{
         // Удаление ссылки на метод из списка вызовов
         // экземпляра делегата:
         myevent-=value;
      }
   }
   // Метод для генерирования события:
   public void RaiseMyEvent(string txt){
      // Если для события предусмотрены обработчики:
      if(myevent!=null){
         // Генерирование события
         // (вызов экземпляра делегата):
         myevent(txt);
      }
   }
}
// Класс:
class Alpha{
   public string name;
   public Alpha(string txt){
```

```
      name=txt;
   }
   public void show(string msg){
      Console.WriteLine("Объект "+name+":");
      Console.WriteLine(msg);
   }
}
// Класс с главным методом:
class EventDemo{
   static void Main(){
      MyClass obj=new MyClass();
      Alpha A=new Alpha("A");
      Alpha B=new Alpha("B");
      obj.RaiseMyEvent("1-е событие");
      obj.MyEvent+=A.show;
      obj.RaiseMyEvent("2-е событие");
      Console.WriteLine();
      obj.MyEvent+=B.show;
      obj.RaiseMyEvent("3-е событие");
      Console.WriteLine();
      obj.MyEvent-=A.show;
      obj.RaiseMyEvent("4-е событие");
      Console.WriteLine();
      obj.MyEvent-=A.show;
      obj.MyEvent-=B.show;
      obj.RaiseMyEvent("5-е событие");
      MyDelegate md=A.show;
      md+=B.show;
      obj.MyEvent+=md;
      obj.RaiseMyEvent("6-е событие");
   }
}
```

Результат выполнения программы точно такой же, как и в предыдущем случае.

Результат выполнения программы (из листинга 2.17)

```
Объект A:
2-е событие

Объект A:
3-е событие
Объект B:
3-е событие

Объект B:
4-е событие

Объект A:
6-е событие
Объект B:
6-е событие
```

В этом примере, по сравнению с программой из листинга 2.16, изменилось только описание метода `MyClass`. Теперь в классе появилось закрытое поле `myevent` типа `MyDelegate`. Это ссылка на экземпляр делегата, через который реализуется список обработчиков для события `MyEvent`. Событие описывается таким блоком кода:

```
public event MyDelegate MyEvent{
   add{
      myevent+=value;
   }
   remove{
      myevent-=value;
   }
}
```

Код `add`-аксессора выполняется при добавлении новой ссылки в список обработчиков для события `MyEvent` с помощью оператора `+=`. В этом случае командой `myevent+=value` соответствующая ссылка на метод или экземпляр делегата (обозначается ключевым словом `value`) добавляется в экземпляр делегата `myevent`. Код `remove`-аксессора выполняется при удалении ссылки на метод или экземпляр делегата из списка обработчиков события `MyEvent`. В этом случае командой `myevent-=value` ссылка, указанная справа от оператора `-=`, удаляется из экземпляра делегата `myevent`.

Несколько изменился код метода `RaiseMyEvent()`, предназначенного для генерирования события. Теперь в теле метода с помощью условного оператора проверяется условие `myevent!=null` (поле `myevent` содержит ссылку на экземпляр делегата), и если условие истинно, то с помощью вызова соответствующего экземпляра делегата происходит «генерирование события».

Все остальное происходит так же, как и в программе из листинга 2.16.

НА ЗАМЕТКУ

Мы продолжим наше знакомство с событиями, когда будем изучать методы создания приложений с графическим интерфейсом.

Резюме

> Истинно вам говорю, 4 мая 1925 года Земля налетит на небесную ось.
>
> *из к/ф «Собачье сердце»*

- Делегат представляет собой тип данных, аналогичный классу. Экземпляр делегата является специальным объектом, который может ссылаться на методы. При описании делегата используется ключевое слово `delegate`, после которого указывается идентификатор типа результата метода, на который может ссылаться экземпляр делегата. Затем указывается имя делегата, и в круглых скобках описываются аргументы. Такие аргументы должны быть у метода, на который ссылается экземпляр делегата.
- Создание экземпляра делегата напоминает создание обычного объекта. Объявляется переменная типа делегата, значением которой

присваивается результат выражения на основе инструкции `new` и имени делегата со ссылкой на метод в круглых скобках. Также можно присвоить значением переменной типа делегата ссылки на метод.

- Ссылка на нестатический метод выполняется так: указывается имя объекта и через точку — имя метода. Ссылка на статический метод выполняется указанием названия класса и (через точку) имени метода.
- Экземпляр делегата может ссылаться на несколько методов. Для добавления ссылки на метод (к уже существующим ссылкам в экземпляре делегата) используют оператор += (или операторы = и +). Для удаления ссылки на метод из экземпляра делегата используют оператор -= (или операторы = и -).
- Экземпляр делегата можно вызывать. Для этого после имени экземпляра делегата (переменной, ссылающейся на экземпляр делегата) указываются круглые скобки и, если нужно, аргументы. Тип и количество аргументов определяются тем, как объявлен делегат, экземпляр которого вызывается. При этом с соответствующими аргументами будут вызваны все методы, на которые ссылается экземпляр делегата. Если вызываемые методы возвращают результат, то результатом выражения с вызовом экземпляра делегата является результат последнего вызванного метода.
- Анонимный метод представляет собой блок кода, определяющий метод и предназначенный для присваивания в качестве значения переменной типа делегата. Анонимный метод описывается так же, как обычный метод, но идентификатор типа результата не указывается, а вместо названия метода указывается ключевое слово `delegate`. Тип результата анонимного метода определяется типом результата, указанного в объявлении соответствующего делегата. Анонимный метод может содержать ссылки на внешние переменные.
- Лямбда-выражения представляют собой альтернативу анонимным методам. Лямбда-выражение определяет метод и предназначено для присваивания в качестве значения переменной делегата. Лямбда-выражение описывается подобно анонимному методу, но только ключевое слово `delegate` не используется, а между круглыми скобками с аргументами и телом описания метода размещается оператор =>. Существуют определенные правила, которые позволяют упростить синтаксис лямбда-выражений. Так, если аргумент один и его тип не указывается, то круглые скобки можно не использовать. При единственной команде в теле метода фигурные скобки можно

не использовать. Если в теле метода есть только одна команда с инструкцией `return`, то инструкцию `return` можно не указывать.

- Событие представляет собой автоматическое уведомление, посылаемое одним объектом прочим объектам. Событие реализуется с помощью специального члена класса. При объявлении события обычно указывается спецификатор доступа `public`, затем ключевое слово `event`, делегат и название события. Делегат, указанный в объявлении события, определяет сигнатуру методов, которые могут использоваться в качестве обработчиков события.
- Для добавления в список обработчиков события ссылок на методы или экземпляры делегатов используется оператор `+=`. Для удаления ссылок на методы и экземпляры делегатов из списка обработчиков события используют оператор `-=`. Событие, как и экземпляр делегата, может быть вызвано (сгенерировано), но это можно сделать только в пределах кода класса. Для генерирования события после имени события указываются круглые скобки с аргументами. Количество и тип аргументов определяются делегатом, указанным при объявлении события. При генерировании события автоматически вызываются все методы и экземпляры делегата, зарегистрированные в качестве обработчиков для данного события.
- Хотя событие напоминает экземпляр делегата, между ними существует разница — примерно такая, как между свойством и полем. С каждым событием связано два метода-аксессора. При добавлении (с помощью оператора `+=`) ссылки на метод в список обработчиков события вызывается `add`-аксессор. При удалении (с помощью метода `-=`) ссылки на метод из списка обработчиков события вызывается `remove`-аксессор. Эти аксессоры можно описать в явном виде при объявлении события.

Задания для самостоятельной работы

Отведай ты из моего кубка.

из к/ф «Иван Васильевич меняет профессию»

1. Напишите программу, в которой объявляется делегат для методов с двумя аргументами (символ и текст) и целочисленным результатом. В главном классе необходимо описать два статических метода. Один

статический метод результатом возвращает количество вхождений символа (первый аргумент) в текстовую строку (второй аргумент). Другой метод результатом возвращает индекс первого вхождения символа (первый аргумент) в текстовую строку (второй аргумент) или значение `-1`, если символ в текстовой строке не встречается. В главном методе создать экземпляр делегата и с помощью этого экземпляра вызвать каждый из статических методов.

2. Напишите программу, в которой объявляется делегат для методов с символьным аргументом, не возвращающих результат. Опишите класс, в котором должно быть символьное поле и метод, позволяющий присвоить значение символьному полю объекта. У метода один символьный аргумент, и метод не возвращает результат. Создайте массив объектов данного класса. Создайте экземпляр делегата. В список вызовов этого делегата необходимо добавить ссылки на метод (присваивающий значение символьному полю) каждого объекта из массива. Проверьте результат вызова такого экземпляра делегата.

3. Напишите программу, в которой объявлен делегат, предназначенный для работы с методами, не имеющими аргумента и возвращающими символьный результат. Опишите класс, у которого есть текстовое поле, а также закрытое поле, являющееся ссылкой на экземпляр делегата. В классе нужно описать открытое свойство, доступное только для чтения. Значением свойства является значение закрытого поля (то есть ссылка на экземпляр делегата). Опишите конструктор класса с двумя аргументами: текстовым (тип `string`) и логическим (тип `bool`). Первый текстовый аргумент определяет значение текстового поля объекта, а второй логический аргумент определяет значение закрытого поля типа делегата. Если второй аргумент истинный, то закрытому полю значением присваивается анонимный метод (или лямбда-выражение), возвращающий результатом первый символ из текстового поля. Если логический аргумент ложный, то закрытому полю значением присваивается анонимный метод (или лямбда выражение), возвращающий результатом последний символ из текстового поля. Создайте объекты класса и проверьте функциональность свойства: его можно вызывать как метод, без аргументов, а результатом является первый или последний символ в текстовом поле (в зависимости от того, с каким логическим аргументом создавался объект).

4. Напишите программу, в которой объявляется делегат для работы с методами, имеющими целочисленный аргумент и целочисленный

результат. Опишите класс с индексатором (доступен только для считывания значения). Индексатор результатом должен возвращать ссылку на экземпляр делегата. Экземпляр делегата ссылается на метод, у которого целочисленный аргумент. Результатом метод возвращает целочисленное значение, получающееся возведением аргумента метода в степень, определяемую индексом объекта. Общий эффект такой: если некоторый объект `obj` класса проиндексировать с неотрицательным индексом `k` и в круглых скобках указать аргумент `n` (команда вида `obj[k](n)`), то результатом такого выражения должно быть значение `n` в степени `k`.

5. Напишите программу, в которой с помощью делегатов и анонимных методов (лямбда-выражений) создается экземпляр делегата, который вызывается без аргументов, а результатом возвращает число из последовательности Фибоначчи — при каждом новом вызове получаем новое число в последовательности. В последовательности Фибоначчи первые два числа равны единице, а каждое следующее число равно сумме двух предыдущих.

6. Напишите программу, в которой с помощью делегатов и анонимных методов (лямбда-выражений) создается экземпляр делегата, который вызывается без аргументов, а результатом возвращает текстовое значение с названием дня недели (`"Понедельник"`, `"Вторник"` и так до `"Воскресенья"`). При каждом новом вызове экземпляра результатом возвращается название следующего дня недели. После `"Воскресенья"` результатом возвращается `"Понедельник"` и так далее.

7. Напишите программу, содержащую статический метод. Первым аргументом статическому методу передается целочисленный массив. Вторым аргументом статическому методу передается ссылка на другой метод. У метода-аргумента должен быть целочисленный аргумент, и он должен возвращать целочисленный результат. Результатом статический метод возвращает целочисленный массив. Элементы этого массива вычисляются как результат вызова метода-аргумента, если ему передавать значения элементов из массива-аргумента. Предложите механизм проверки функциональности данного статического метода.

8. Напишите программу, в которой объявлен делегат, соответствующий методам с целочисленным аргументом и целочисленным результатом. Необходимо описать статический метод с двумя аргументами, каждый из которых является ссылкой на экземпляр делегата (ссылка на метод). Результатом статического метода также является ссылка на экземпляр

делегата. Статический метод работает по такой схеме: при вызове ему передаются ссылки на два метода (методы-аргументы), а результатом статический метод возвращает ссылку на метод (метод-результат). Действие метода-результата на свой аргумент эквивалентно последовательному применению к этому аргументу методов-аргументов. Другими словами, если аргументами статическому методу передаются ссылки `f` и `h` на некоторые методы, то результатом является ссылка на метод, который для аргумента `n` вычисляет значение `f(h(n))`. Предложите механизм проверки функциональности статического метода.

9. Напишите программу, содержащую статический метод с тремя действительными (тип `double`) аргументами. Результатом статического метода является ссылка на метод, вычисляющий квадратичный трехчлен с коэффициентами, определяемыми аргументами статического метода. Другими словами, если статический метод вызывается с числовыми аргументами `a`, `b` и `c`, то результатом является ссылка на метод, который для аргумента `x` типа `double` результатом вычисляет значение выражения `a*x*x+b*x+c`.

10. Напишите программу, в которой есть класс с событием. Событие обрабатывается методами, имеющими текстовый аргумент и не возвращающими результат. У класса должно быть текстовое поле, в которое при создании объекта класса записывается название объекта. В классе должен быть описан метод для генерирования события, который вызывается без аргументов. При генерировании события аргументом передается значение текстового поля объекта, генерирующего событие. Еще один класс, описанный в программе, должен содержать метод с текстовым аргументом, не возвращающий результат. При вызове метод отображает значение своего текстового аргумента. В главном методе программы необходимо создать два объекта первого класса и один объект второго класса. Для событий объектов первого класса обработчиком регистрируется метод объекта второго класса (получается, что метод одного и того же объекта зарегистрирован обработчиком для событий двух объектов). Для каждого из объектов первого класса необходимо сгенерировать событие. При этом метод, зарегистрированный обработчиком, должен выводить название объекта, сгенерировавшего событие.

Глава 3
ПЕРЕЧИСЛЕНИЯ И СТРУКТУРЫ

> Вы напрасно, господа, ходите без калош. Во-первых, вы простудитесь. А во-вторых, вы наследите мне на коврах. А все ковры у меня персидские.
>
> *из к/ф «Собачье сердце»*

В этой главе мы познакомимся с *перечислениями* и *структурами*: узнаем, что это такое и как используется. После изучения материала главы мы будем знать:

- как описывается перечисление;
- как создается и используется переменная типа перечисления;
- как описывается структура;
- чем структура отличается от класса;
- как создается экземпляр структуры;
- какие операции могут выполняться с экземплярами структуры.

Все вопросы и темы, рассматриваемые в главе, подкрепляются примерами.

Знакомство с перечислениями

> Владимир Николаевич, человечество из-за одного камушка с Луны тысячелетия потратило, а тут живой инопланетянин и эцих из неизвестного металла!
>
> *из к/ф «Кин-дза-дза»*

Перечисление — это тип данных, определяемый пользователем. Переменная, объявленная как относящаяся к типу перечисления, может

принимать значение одной из целочисленных констант. Список этих констант фактически детерминирует (определяет) перечисление как тип.

НА ЗАМЕТКУ

Другими словами, имеется набор целочисленных констант и переменная может принимать значение одной из этих констант. Набор констант формирует тип, к которому относится соответствующая переменная. Такой тип называется перечислением.

Таким образом, определяя перечисление как тип данных, мы задаем набор значений, которые может принимать переменная, относящаяся к данному типу. Значения, входящие в набор, целочисленные (по умолчанию типа `int`), но каждое имеет свое уникальное имя (то есть является константой).

Для использования перечисления в программе мы должны сначала описать тип перечисления. После того как тип определен, его можно использовать (в качестве типа переменных, например). Ниже приведен шаблон для объявления перечисления:

```
enum имя {константа,константа,...,константа};
```

Начинается все с ключевого слова `enum`. Оно является индикатором того, что объявляется именно перечисление. После ключевого слова `enum` указывают `имя` перечисления, которое фактически представляет собой название создаваемого типа. Именно `имя` перечисления используется как идентификатор типа данных при объявлении переменных. После имени перечисления в фигурных скобках через запятую перечисляются названия `констант`. Константы, формирующие тип перечисления, предварительно объявлять не нужно. То есть, когда мы в списке перечисляем названия констант, то тем самым мы их объявляем (и инициализируем). По умолчанию первая константа в списке получает значение `0`, вторая константа в списке получает значение `1`, и так далее: значение каждой следующей константы на единицу больше ее предшественницы. Допустим, в программе при объявлении перечисления использована следующая команда:

```
enum Animals {Cat,Dog,Fox,Wolf,Bear};
```

Этой командой определяется перечисление `Animals`. Впоследствии идентификатор `Animals` мы сможем использовать как обозначение для

типа данных. Например, мы сможем объявить переменную, относящуюся к типу перечисления Animals:

```
Animals animal;
```

Объявленная таким образом переменная animal относится к типу Animals, и это означает, что значением переменной может быть одна из констант, формирующих тип Animals (имеются в виду константы Cat, Dog, Fox, Wolf и Bear). Напомним, что константы, формирующие перечисление, являются целочисленными. Их значения — целые числа. В данном случае значение первой в списке {Cat,Dog,Fox,Wolf,Bear} константы Cat равно 0, значение константы Dog равно 1, и так далее — последняя (пятая по счету) в списке константа Bear имеет значение 4. Константы из списка получают эти значения автоматически.

Следует учесть, что константы из списка, определяющего перечисление, несколько отличаются от «обычных» констант, объявленных в индивидуальном порядке. Константы из списка перечисления используются с указанием имени перечисления. Ниже приведена команда, которой присваивается значение переменной animal:

```
animal=Animals.Cat;
```

Переменной animal присваивается значением константа Cat, причем ссылка на эту константу выполнена как Animals.Cat. То есть сначала указывается имя перечисления Animals и через точку — название константы Cat из этого перечисления. Это стандартный способ обращения к константам из перечисления: имени константы предшествует имя перечисления, и разделяются они точкой.

ПОДРОБНОСТИ

Хотя константы из перечисления технически реализуются как целочисленные значения, автоматического преобразования между целочисленными значениями и значениями типа перечисления нет. Обычно приходится выполнять явное приведение типов.

Используемый по умолчанию для реализации значений из перечисления тип int можно заменить на другой целочисленный тип. Для этого идентификатор базового типа указывается через двоеточие после названия перечисления, как показано ниже:

```
enum имя: тип {константа,константа,...,константа};
```

Например, для того чтобы базовым при реализации значений типа перечисления был тип `byte`, а не тип `int`, можем использовать такое объявление для перечисления `Animals`:

```
enum Animals: byte {Cat,Dog,Fox,Wolf,Bear};
```

Также мы можем явно инициализировать константы, формирующие перечисления. В таком случае после имени константы указывается оператор присваивания = и значение константы. Причем не обязательно явно указывать значение для всех констант перечисления. Разрешается указать значения только некоторых констант. В таком случае для констант, значение которых явно не указано, действует прежнее правило: значение очередной константы на единицу больше значения предыдущей константы. Допустим, перечисление объявлено с помощью такой команды:

```
enum Animals: byte {Cat,Dog,Fox=100,Wolf,Bear=200};
```

В этом случае константа `Cat` реализуется с целочисленным значением `0`, константа `Dog` реализуется со значением `1`, константа `Fox` реализуется со значением `100`, значение константы `Wolf` равно `101`, а значение константы `Bear` равно `200`. Небольшая программа, в которой используются перечисления, представлена в листинге 3.1.

Листинг 3.1. Знакомство с перечислениями

```
using System;
// Объявление перечисления:
enum Animals {Cat,Dog,Fox,Wolf,Bear};
// Объявление перечисления с инициализацией констант:
enum Coins {One=1,Two,Five=5,Ten=10,Fifty=50};
// Класс с главным методом:
class EnumDemo{
   // Главный метод:
   static void Main(){
      Console.WriteLine("В мире животных");
      // Переменная типа перечисления:
      Animals animal=Animals.Cat;
      Console.WriteLine("animal: {0,-5} или {1}",animal,(int)animal);
```

```
      // Новое значение переменной:
      animal=Animals.Dog;
      Console.WriteLine("animal: {0,-5} или {1}",animal,(int)animal);
      // Преобразование целого числа в значение
      // типа перечисления:
      animal=(Animals)2;
      Console.WriteLine("animal: {0,-5} или {1}",animal,(int)animal);
      // Сумма переменной и целого числа:
      animal=animal+1;
      Console.WriteLine("animal: {0,-5} или {1}",animal,(int)animal);
      // Применение операции инкремента:
      animal++;
      Console.WriteLine("animal: {0,-5} или {1}",animal,(int)animal);
      Console.WriteLine("В мире финансов");
      // Переменная типа перечисления:
      Coins coin;
      // Объект с константами из перечисления:
      Array names=Enum.GetValues(typeof(Coins));
      // Перебор констант:
      for(int k=0;k<names.Length;k++){
         // Значение переменной:
         coin=(Coins)names.GetValue(k);
         Console.WriteLine("coin: {0,-5} или {1}",coin,(int)coin);
      }
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 3.1)

```
В мире животных
animal: Cat   или 0
```

```
animal: Dog   или 1
animal: Fox   или 2
animal: Wolf  или 3
animal: Bear  или 4
В мире финансов
coin: One   или 1
coin: Two   или 2
coin: Five  или 5
coin: Ten   или 10
coin: Fifty или 50
```

В этой программе мы объявляем два перечисления: перечисление `Animals` объявляется командой `enum Animals {Cat,Dog,Fox,Wolf,Bear}`, а перечисление `Coins` объявляется командой `enum Coins {One=1,Two, Five=5,Ten=10,Fifty=50}`.

В главном методе программы командой `Animals animal=Animals.Cat` объявляется и инициализируется переменная `animal` типа `Animals`. При отображении значения переменной `animal` в консольном окне отображается символьное название константы, которая присвоена в качестве значения переменной. Чтобы узнать целочисленное значение константы, используем инструкцию `(int)animal`, в которой выполняется явное приведение к целочисленному типу.

> **НА ЗАМЕТКУ**
>
> Инструкция `{0,-5}` в текстовой строке в аргументе метода `WriteLine()` означает, что для отображения соответствующего значения выделяется `5` позиций и выравнивание выполняется по левому (из-за знака минус) краю.

Новое значение переменной `animal` присваивается командами `animal=Animals.Dog` (значением присваивается константа из перечисления), `animal=(Animals)2` (выполняется явное приведение целочисленного значения к типу `Animals`), `animal=animal+1` (увеличение текущего значения переменной на единицу), а также `animal++` (применение операции инкремента). При выполнении последних операций мы приняли в расчет, что числовые значения констант отличаются

друг от друга на единицу. Но в общем случае это не всегда так: например, перечисление `Coins`. Для работы с этим перечислением мы командой `Coins coin` объявляем переменную `coin`. Но проблема здесь в том, что константы из перечисления `Coins` имеют «уникальные» целочисленные значения. Лишь для констант `One` и `Two` значения отличаются на единицу, значения всех прочих констант заданы индивидуально. Поэтому перебор ряда натуральных чисел с помощью оператора цикла не очень подходит. Мы поступаем немного хитрее. Сначала с помощью команды `Array names=Enum.GetValues(typeof(Coins))` создаем объект `names` класса `Array`, содержащий набор значений констант из перечисления `Coins`. Для этого из класса `Enum` вызывается статический метод `GetValues()`. Результатом метод возвращает ссылку на объект класса `Array`. Этот объект по свойствам напоминает массив и содержит, как отмечалось, значения, формирующие тип `Coins`. Аргументом методу `GetValues()` передается объект класса `Type`, который содержит информацию о перечислении `Coins`. Этот объект можно получить с помощью инструкции `typeof(Coins)`. Для перебора значений, формирующих тип перечисления `Coins`, мы запускаем оператор цикла, в котором используем индексную переменную `k`. Количество элементов в объекте `names` вычисляется выражением `names.Length`, как для массива. Для считывания очередного значения, содержащегося в объекте `names`, используем метод `GetValue()`. Аргументом методу передается индекс считываемого элемента. Но поскольку значением выражения `names.GetValue(k)` является объектная ссылка класса `object`, то мы используем процедуру явного приведения типа. В итоге получается команда `coin=(Coins)names.GetValue(k)`, которой значение присваивается переменной `coins`. В консольном окне отображается значение этой переменной и его целочисленный «эквивалент».

ПОДРОБНОСТИ

Есть несколько обстоятельств, которые нужно иметь в виду при работе с перечислениями. Во-первых, как мы знаем, константы, формирующие тип перечисления, являются целочисленными. Предполагается, что значением переменной типа перечисления может быть одна из таких констант. Но дело в том, что технических ограничений на значения для переменной типа перечисления нет. Это означает, что переменная может получить целочисленное значение, которое не совпадает со значением ни одной из констант из перечисления. Например, если в рассмотренном выше примере выполнить команду

`animal=(Animals)5`, то ошибки не будет, хотя в списке констант перечисления `Animals` константы со значением `5` нет. Это же замечание относится к операциям вида `animal++` или `animal=animal+1`: если окажется, что новое вычисленное значение не представлено константой в списке перечисления, то оно все рано будет записано в переменную.

Нулевое значение особое: например, поле, тип которого является перечислением, инициализируется с начальным нулевым значением. Причем происходит это, даже если среди констант, формирующих перечисление, нет константы с нулевым значением. Поэтому общая рекомендация состоит в том, чтобы перечисление все же содержало константу с нулевым значением.

Знакомство со структурами

> Нет, давай будем считать, что мы на Земле в какой-то пустыне.
>
> *из к/ф «Кин-дза-дза»*

В языке C# структуры напоминают классы. По сравнению с классами возможности структур скромнее. Зато, в отличие от объектов класса, экземпляры структуры реализуются через механизм прямого доступа, что повышает эффективность обработки данных. Но обо всем по порядку.

Как и класс, структура представляет собой некоторый шаблон, на основе которого создаются объекты. В случае структуры эти объекты мы будем называть *экземплярами структуры*. Здесь концептуально все должно быть более-менее понятно. Описывается структура, в общем-то, так же, как и класс. Разница лишь в том, что вместо ключевого слова `class` используется ключевое слово `struct`. Общий шаблон описания структуры следующий:

```
struct имя{
   // Тело структуры
}
```

После ключевого слова `struct` указывается имя структуры, а в блоке из фигурных скобок описывается тело структуры: поля, методы (в том числе и операторные), индексаторы, свойства и события.

НА ЗАМЕТКУ

В языке C++ также есть структуры. В языке C# структуры кардинально отличаются от структур языка C++. В языке C++ структура может содержать только поля, но не методы. В языке C# структура является аналогом класса.

Вместе с тем, если сравнивать с классами, на структуры накладываются существенные ограничения. Ниже перечислены наиболее важные особенности структур.

- У структур могут быть конструкторы, однако нельзя описать конструктор без аргументов. У каждой структуры есть конструктор без аргументов, используемый по умолчанию. Он существует, даже если в структуре описаны прочие версии конструкторов. Другими словами, у структуры всегда есть конструктор без аргументов и переопределить его нельзя. При этом мы можем описать другие версии конструктора (с аргументами).
- У структур нет деструкторов.
- К структурам неприменимо наследование: структура не может создаваться на основе класса или структуры и не может быть базовой для класса. При этом все структуры встроены в иерархию наследования с классом `Object` на верхнем уровне (то есть переменная класса `Object` может ссылаться на экземпляр структуры).
- При описании членов структуры используют спецификаторы уровня доступа. Если спецификатор доступа не указан, соответствующий член является закрытым и доступен только в пределах кода структуры. Спецификатор уровня доступа `protected` не используется (в нем просто нет смысла, поскольку наследование не поддерживается).
- При объявлении полей в классе полям можно присваивать значения (такое значение будет у поля при создании объекта). Для поля структуры (при объявлении поля) значение можно указать, только если поле статическое (использовано ключевое слово `static`) или является константой (использовано ключевое слово `const`).
- Как и классы, структуры могут реализовать интерфейсы.
- При присваивании экземпляров структуры (одной переменной структуры присваивается другая переменная структуры) выполняется побитовое копирование: значение каждого поля присваиваемого

экземпляра копируется в соответствующее поле экземпляра, которому присваивается значение. Этот механизм копирования обусловлен тем, что структуры относятся к типам с прямым доступом к значению.

Учитывая, что ограничения существенные, возникает естественный вопрос: а зачем вообще нужны структуры? В чем их преимущество? Ответ состоит в том, что структуры — это не ссылочный тип (как классы), а тип с прямым доступом к значению. Это дает преимущество в плане повышения быстродействия программы.

Здесь уместно напомнить, как мы реализуем объекты классов. Для этого нам нужен собственно объект и еще нам нужна объектная переменная, которая на объект ссылается. Доступ к объекту мы получаем через объектную переменную, которая играет роль «посредника». Это ссылочный тип. А вот целочисленная переменная хранит не ссылку на целое число, а само это целое число. Это тип с прямым доступом к значению. Таким же типом является структура. Если мы создаем экземпляр (объект) структуры, то этот экземпляр реализуется как значение некоторой переменной, без каких бы то ни было посредников. Преимущество данного подхода в том, что операции с экземпляром выполняются быстрее, чем в случае, если бы это был аналогичный объект класса, доступ к которому необходимо получать через объектную переменную. Плата за сэкономленное время — ограничение в функциональных возможностях экземпляров структуры по сравнению с объектами классов.

НА ЗАМЕТКУ

Механизм, использованный в языке C# для реализации экземпляров структуры, в языке C++ используется для реализации объектов класса. Также стоит отметить, что целочисленные значения в языке C# на самом деле реализуются как экземпляры структуры.

Экземпляр структуры создается так же, как объявляется переменная базового типа — указывается название структуры и имя переменной:

```
Структура переменная;
```

При создании экземпляра в этом случае поля структуры остаются неинициализированными, и поэтому перед использованием экземпляра структуры полям придется сначала присвоить значения. Если при

создании экземпляра структуры мы хотим использовать конструктор с аргументами, то синтаксис создания экземпляра структуры будет такой же, как и синтаксис команды создания объекта класса, только вместо имени класса указывается имя структуры:

```
Структура переменная=new Структура(аргументы);
```

Как и в случае с созданием объектов класса, используется инструкция `new`. В результате создается экземпляр структуры, который является значением переменной, указанной слева от оператора присваивания. Подобного типа команду (с инструкцией `new`) можно использовать и для создания экземпляра структуры с использованием конструктора без аргументов (конструктор по умолчанию):

```
Структура переменная=new Структура();
```

В этом случае аргументы конструктору не передаются. Эффект от выполнения такого рода команды практически такой же, как и при создании экземпляра структуры командой вида `Структура переменная`. Разница в том, что при вызове конструктора по умолчанию выполняется инициализация полей. Поля инициализируются значениями по умолчанию (для числовых полей это нулевое значение, а для ссылочных типов это пустая ссылка `null`).

ПОДРОБНОСТИ

Если в структуре описывается конструктор, то в теле конструктора полям структуры должны быть присвоены значения.

После того как экземпляр структуры создан, мы можем обращаться к открытым полям и методам этого экземпляра точно так же, как мы это делали с полями и методами объектов. Для этого достаточно указать имя экземпляра структуры и через точку имя поля или имя метода с аргументами в круглых скобках (или пустыми круглыми скобками, если методу не передаются аргументы).

НА ЗАМЕТКУ

Кроме полей и методов, в структуре могут быть описаны свойства, индексаторы и события. Правила их описания и использования такие же, как при работе с объектами и классами. Методы (и индексаторы) для структур могут перегружаться так же, как это делается

для классов. Кроме этого, структура может содержать статические члены. Для обращения к статическим членам вместо имени объекта используют имя структуры.

Программа с описанием структуры, в которой создаются и используются экземпляры структуры, представлена в листинге 3.2.

Листинг 3.2. Знакомство со структурами

```
using System;
// Структура:
struct MyStruct{
   // Целочисленное поле:
   public int code;
   // Текстовое поле:
   public string name;
   // Конструктор:
   public MyStruct(int n,string t){
      code=n;   // Значение целочисленного поля
      name=t;   // Значение текстового поля
   }
   // Метод:
   public void show(){
      // Отображение значений полей:
      Console.WriteLine("Поля \"{0}\" и {1}",name,code);
   }
}
// Класс с главным методом:
class StructDemo{
   // Главный метод:
   static void Main(){
      // Создание экземпляра структуры:
      MyStruct A;
      // Присваивание значений полям:
```

```
      A.code=100;
      A.name="Экземпляр A";
      // Вызов метода:
      A.show();
      // Создание экземпляра структуры:
      MyStruct B=new MyStruct(200,"Экземпляр B");
      // Вызов метода:
      B.show();
      // Присваивание экземпляров:
      A=B;
      // Присваивание значения полю:
      B.code=300;
      // Вызов методов:
      A.show();
      B.show();
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 3.2)

```
Поля "Экземпляр A" и 100
Поля "Экземпляр B" и 200
Поля "Экземпляр B" и 200
Поля "Экземпляр B" и 300
```

Мы описываем структуру `MyStruct` следующим образом (комментарии для удобства удалены):

```
struct MyStruct{
   public int code;
   public string name;
   public MyStruct(int n,string t){
      code=n;
```

```
        name=t;
    }
    public void show(){
        Console.WriteLine("Поля \"{0}\" и {1}",name,code);
    }
}
```

У этой структуры есть два открытых поля: целочисленное (тип `int`) поле `code` и текстовое (тип `string`) поле `name`. Мы также описали конструктор структуры с двумя аргументами (целое число и текст), которые определяют значения полей создаваемого экземпляра. Таким образом, при создании экземпляров структуры мы можем использовать либо конструктор по умолчанию (без аргументов), либо конструктор с двумя аргументами. Еще в структуре описан метод `show()`. Он не имеет аргументов и не возвращает результат. При вызове метод отображает в консольном окне значения полей экземпляра, из которого он был вызван.

В главном методе программы командой `MyStruct A` создается экземпляр `A` структуры `MyStruct`. В данном случае при создании экземпляра его поля не инициализируются. Поэтому нам нужно присвоить значения полям экземпляра.

ПОДРОБНОСТИ

При создании экземпляра структуры вместо команды `MyStruct A` мы могли использовать команду `MyStruct A=new MyStruct()`. В этом случае поле `code` получило бы значение по умолчанию `0`, а поле `name` содержало бы пустую ссылку `null`.

Присваивание значений полям осуществляется командами `A.code=100` и `A.name="Экземпляр A"`. После этого командой `A.show()` из экземпляра `A` структуры вызывается метод `show()`, в результате чего в консольном окне отображаются значения полей экземпляра.

Еще один экземпляр структуры создается командой `MyStruct B=new MyStruct(200,"Экземпляр B")`. В данном случае экземпляр `B` создается вызовом конструктора с двумя аргументами, и поля экземпляра получают значения, переданные конструктору в качестве аргументов. Проверяем значения полей с помощью команды `B.show()`.

В программе есть пример операции присваивания: командой A=B экземпляр B присваивается в качестве значения экземпляру A. Если бы мы имели дело с объектами класса, то копирование выполнялось бы на уровне ссылок на объекты, и в результате обе переменные ссылались бы на один и тот же объект. В данном случае значениями переменных A и B являются не ссылки, а сами экземпляры. Команда присваивания выполняется как побитовое копирование значений полей из экземпляра B в экземпляр A. В результате поле name экземпляра A получает значение "Экземпляр B", а поле code экземпляра A получает значение 200. При этом оба экземпляра остаются физически разными. Поэтому при выполнении команды B.code=300 поле code экземпляра B получает значение 300, а поле code экземпляра A остается со значением 200. В том, что это действительно так, мы убеждаемся с помощью команд A.show() и B.show().

Далее мы рассмотрим некоторые особенности использования структур на практике.

Массив как поле структуры

— Вы можете сказать, что им придет в голову?
— Все что угодно.
— И я того же мнения.

из к/ф «Собачье сердце»

Хотя структура сама по себе относится к типу данных с прямым доступом к значению, среди полей структуры могут быть значения ссылочных типов. Одна из таких ситуаций — когда поле структуры является массивом (ссылкой на массив). Соответствующая программа представлена в листинге 3.3.

Листинг 3.3. Массив как поле структуры

```
using System;
// Структура:
struct MyStruct{
   // Ссылка на символьный массив:
   public char[] symbs;
```

```
   // Метод для отображения содержимого массива:
   public void show(){
      for(int k=0;k<symbs.Length;k++){
         Console.Write("|"+symbs[k]);
      }
      Console.WriteLine("|");
   }
}
// Класс с главным методом:
class ArrayInStructDemo{
   // Главный метод:
   static void Main(){
      // Создание экземпляров структуры:
      MyStruct A,B;
      // Полю присваивается ссылка на массив
      A.symbs=new char[7];
      // Заполнение массива:
      for(int k=0;k<A.symbs.Length;k++){
         A.symbs[k]=(char)('A'+k);
      }
      Console.WriteLine("Экземпляр A:");
      // Содержимое массива:
      A.show();
      // Присваивание экземпляров структуры:
      B=A;
      Console.WriteLine("Экземпляр B:");
      // Содержимое массива:
      B.show();
      // Присваивание значений элементам массива:
      A.symbs[0]='X';
      B.symbs[B.symbs.Length-1]='Y';
      Console.WriteLine("Экземпляр A:");
```

```
      // Содержимое массива:
      A.show();
      Console.WriteLine("Экземпляр B:");
      // Содержимое массива:
      B.show();
   }
}
```

Ниже показан результат выполнения программы:

Результат выполнения программы (из листинга 3.3)

```
Экземпляр A:
|A|B|C|D|E|F|G|
Экземпляр B:
|A|B|C|D|E|F|G|
Экземпляр A:
|X|B|C|D|E|F|Y|
Экземпляр B:
|X|B|C|D|E|F|Y|
```

В программе мы описали структуру `MyStruct`, у которой есть открытое поле `symbs`, являющееся ссылкой на символьный массив. Также в структуре описан метод `show()`, который при вызове отображает в консольном окне содержимое массива, на который ссылается поле `symbs` экземпляра структуры.

В главном методе программы командой `MyStruct A,B` создаются два экземпляра структуры. Затем командой `A.symbs=new char[7]` создается массив, и ссылка на этот массив записывается в поле `symbs` экземпляра `A`. С помощью оператора цикла созданный массив последовательно заполняется символами, начиная с символа `'A'`. Содержимое массива проверяем с помощью команды `A.show()`. Затем выполняется присваивание (команда `B=A`). В результате полю `symbs` экземпляра `B` присваивается значение поля `symbs` экземпляра `A`. Но значение поля `symbs` экземпляра `A` — это ссылка на созданный ранее массив. Получается, что поле `symbs` экземпляра `B` будет ссылаться на точно тот же массив. При выполнении команды `B.show()` действительно получаем уже знакомую нам последовательность

символов. Но чтобы убедиться, что поля в экземплярах A и B ссылаются на один и тот же массив, а не на два одинаковых массива, используем команды A.symbs[0]='X' и B.symbs[B.symbs.Length-1]='Y'. Первой командой меняется значение начального элемента в массиве, на который ссылается поле из экземпляра A, а второй командой меняется значение последнего элемента в массиве, на который ссылается поле из экземпляра B. После выполнения команд A.show() и B.show() убеждаемся, что экземпляры действительно ссылаются на один и тот же массив.

Массив экземпляров структуры

— Это плохо? Плохо? Ответьте, уважаемый доктор.
— Это бесподобно.

из к/ф «Собачье сердце»

Некоторый практический интерес может представлять ситуация, когда создается массив экземпляров структуры. Рассмотрим программу, представленную в листинге 3.4.

Листинг 3.4. Массив экземпляров структуры

```
using System;
// Структура:
struct MyStruct{
   public int code;
}
// Класс с главным методом:
class StructArrayDemo{
   // Главный метод:
   static void Main(){
      // Размер массива:
      int size=7;
      // Создание массива из экземпляров структуры:
      MyStruct[] A=new MyStruct[size];
      // Перебор экземпляров структуры в массиве:
```

```
      for(int k=0;k<A.Length;k++){
         // Полю экземпляра присваивается значение:
         A[k].code=2*k+1;
         // Отображается значение поля экземпляра:
         Console.Write("|"+A[k].code);
      }
      Console.WriteLine("|");
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 3.4)

```
|1|3|5|7|9|11|13|
```

Программа очень простая. В ней описана структура `MyStruct`, в которой всего одно открытое целочисленное поле `code`. В методе `Main()` с помощью команды `MyStruct[] A=new MyStruct[size]` создается массив из экземпляров структуры (переменной `size` предварительно присвоено значение `7`). Элементами созданного массива являются экземпляры структуры `MyStruct`. Поле `code` каждого из экземпляров не инициализировано. Поэтому после создания массива мы присваиваем значения полю `code` каждого из экземпляров. Для этого запускается оператор цикла с индексной переменной `k`. Количество элементов в массиве определяем выражением `A.Length`. При заданном значении индекса `k` обращение к экземпляру структуры выглядит как `A[k]`, а обращение к полю `code` такого экземпляра выполняется в формате `A[k].code`. В экземпляры структуры записываются нечетные числа, которые затем отображаются в консольном окне.

Структуры и метод `ToString()`

Как вы яхту назовете, так она и поплывет.

из м/ф «Приключения капитана Врунгеля»

Хотя структуры не поддерживают наследование, они встроены в иерархию наследования. Поэтому для структур так же, как и для классов,

можно переопределить метод `ToString()`. В листинге 3.5 по этому поводу представлен небольшой пример.

Листинг 3.5. Структуры и метод ToString()

```
using System;
// Структура:
struct MyStruct{
   // Открытое текстовое поле:
   public string name;
   // Открытое целочисленное поле:
   public int code;
   // Конструктор с двумя аргументами:
   public MyStruct(string name,int code){
      // Присваивание значения полям:
      this.name=name;
      this.code=code;
   }
   // Переопределение метода ToString():
   public override string ToString(){
      // Текстовая строка:
      string txt="Экземпляр \""+name+"\"\n";
      txt+="Числовое поле: "+code+"\n";
      // Результат метода:
      return txt;
   }
}
// Класс с главным методом:
class StructToStringDemo{
   // Главный метод:
   static void Main(){
      // Создание экземпляров структуры:
      MyStruct A=new MyStruct("Alpha",100);
```

```
      MyStruct B=new MyStruct("Bravo",200);
      // Операции с экземплярами:
      Console.WriteLine(A);
      string text=B+"Выполнение программы завершено";
      Console.WriteLine(text);
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 3.5)

```
Экземпляр "Alpha"
Числовое поле: 100

Экземпляр "Bravo"
Числовое поле: 200
Выполнение программы завершено
```

У структуры `MyStruct` два открытых поля: текстовое `name` и целочисленное `code`. В структуре описан конструктор с двумя аргументами. Эти аргументы определяют значения полей создаваемого экземпляра структуры.

ПОДРОБНОСТИ

Аргументы конструктора имеют такие же названия, как и поля. Поэтому в теле конструктора использовано ключевое слово `this`, обозначающее экземпляр структуры. Обращение к полям выполняется инструкциями вида `this.name` и `this.code`, а идентификаторы `name` и `code` в теле конструктора означают аргументы.

Также в структуре описан (переопределен) метод `ToString()`. Результатом метод возвращает текстовую строку, которая содержит информацию о значениях полей экземпляра структуры.

НА ЗАМЕТКУ

При описании метода `ToString()` используется ключевое слово `override`.

В главном методе программы командами `MyStruct A=new MyStruct ("Alpha",100)` и `MyStruct B=new MyStruct("Bravo",200)` создаются экземпляры `A` и `B` структуры. При выполнении команды `Console.WriteLine(A)` для экземпляра `A` вызывается метод `ToString()`, и текстовая строка, возвращаемая этим методом, отображается в консольном окне. Затем объявляется текстовая переменная `text`, в качестве значения которой присваивается выражение `B+"Выполнение программы завершено"`. Значение этого выражения вычисляется так: для экземпляра `B` вызывается метод `ToString()`, и полученная в результате строка объединяется со вторым текстовым операндом выражения. Значение текстовой переменной `text` отображается в консольном окне (команда `Console.WriteLine(text)`).

Свойства и индексаторы в структурах

> Все глупости в мире бывают только от умных разговоров.
>
> *из к/ф «Айболит-66»*

В структурах можно использовать индексаторы и свойства. Принцип реализации этих членов структуры такой же, как и в случае с классами. Программа, в которой есть структура с индексаторами и свойствами, представлена в листинге 3.6.

Листинг 3.6. Свойства и индексаторы в структурах

```
using System;
// Структура:
struct MyStruct{
   // Закрытое текстовое поле:
   private string txt;
   // Текстовое свойство:
```

```
   public string text{
      // Аксессор для считывания значения свойства:
      get{
         return txt;
      }
      // Аксессор для присваивания значения свойству:
      set{
         txt=value;
      }
   }
   // Целочисленное свойство (доступно для чтения):
   public int length=>txt.Length;
   // Индексатор с целочисленным индексом:
   public char this[int k]{
      // Аксессор для считывания значения:
      get{
         return txt[k];
      }
      // Аксессор для присваивания значения:
      set{
         // Преобразование текста в символьный массив:
         char[] s=txt.ToCharArray();
         // Замена символа в массиве:
         s[k]=value;
         // Преобразование массива в текст:
         txt=new string(s);
      }
   }
}
// Класс с главным методом:
class StructAndIndPropDemo{
   // Главный метод:
```

```
   static void Main(){
      // Создание экземпляра структуры:
      MyStruct A=new MyStruct();
      // Присваивание значения текстовому свойству:
      A.text="Alpha";
      // Проверка значения текстового свойства:
      Console.WriteLine(A.text);
      // Использование индексатора и
      // целочисленного свойства:
      for(int k=0;k<A.length;k++){
         Console.Write("|"+A[k]);
      }
      Console.WriteLine("|");
      // Использование индексатора и
      // целочисленного свойства:
      A[0]='a';
      A[A.length-1]='A';
      // Проверка значения текстового свойства:
      Console.WriteLine(A.text);
   }
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 3.6)

```
Alpha
|A|l|p|h|a|
alphA
```

В нашем примере структура `MyStruct` содержит закрытое текстовое поле `txt`, а также имеет текстовое свойство `text`. При считывании значения свойства возвращается значение поля `txt`. При присваивании значения свойству значение присваивается полю `txt`.

Еще одно свойство `length` является целочисленным. Оно доступно только для считывания. В качестве значения свойства возвращается размер текста в текстовом поле `txt` (выражение `txt.Length`). При определении свойства был использован упрощенный синтаксис на основе лямбда-выражения.

Индексатор с целочисленным индексом (обозначен как `k`) определен так, что результатом возвращается символ из поля `txt` с соответствующим индексом (выражение `txt[k]`). При присваивании символьного значения проиндексированному экземпляру структуры в текстовом поле `txt` меняется значение символа с данным индексом. Поскольку текстовые значения неизменны, то технически все реализуется следующим образом (код `set`-аксессора для индексатора). Сначала командой `char[] s= txt.ToCharArray()` на основе текста из текстового поля `txt` создается символьный массив `s`. Затем командой `s[k]=value` в этом символьном массиве меняется значение элемента с индексом `k` (индекс в квадратных скобках после имени экземпляра структуры). После этого командой `txt=new string(s)` на основе символьного массива создается текстовый объект, и ссылка на него записывается в поле `txt`.

В главном методе программы командой `MyStruct A=new MyStruct()` создаем экземпляр структуры. С помощью команды `A.text="Alpha"` текстовому свойству этого экземпляра присваивается значение. Проверку значения текстового свойства выполняем командой `Console.WriteLine(A.text)`.

НА ЗАМЕТКУ

Если попытаться вместо команды `MyStruct A=new MyStruct()` использовать команду `MyStruct A`, то программа не скомпилируется. Дело в том, что при создании экземпляра структуры командой `MyStruct A` закрытое поле `txt` не инициализируется, и поэтому попытка обращения к свойству `text` будет заблокирована. Если для создания экземпляра структуры используется команда `MyStruct A=new MyStruct()`, то закрытое поле `txt` получает значение по умолчанию `null`, и мы можем использовать свойство `text` (чтобы присвоить значение).

Для сравнения мы отображаем значение текстового поля посимвольно. Для этого используем оператор цикла, в котором каждый символ считывается индексированием экземпляра структуры (инструкция вида `A[k]`), а размер текста определяется с помощью целочисленного свойства (инструкция `A.length`).

Командами `A[0]='a'` и `A[A.length-1]='A'` меняются значения начального и последнего символов в текстовом поле, после чего результат проверяется с помощью команды `Console.WriteLine(A.text)`.

Экземпляр структуры как аргумент метода

Какая встреча! И какая неприятная!

из к/ф «Айболит-66»

Экземпляр структуры можно передавать методу в качестве агумента. В этом случае важно помнить, что по умолчанию аргументы передаются по значению (то есть в действительности передается копия переменной, фактически указанной аргументом), а структура является типом данных с прямым доступом к значению. В остальном передача экземпляров структуры методу в качестве аргументов достаточно тривиальна. В листинге 3.7 представлена программа, поясняющая ситуацию.

Листинг 3.7. Экземпляр структуры как аргумент метода

```
using System;
// Структура:
struct MyStruct{
   // Ссылка на символьный массив:
   public char[] name;
   // Целочисленное поле:
   public int code;
   // Конструктор:
   public MyStruct(string t,int n){
      // Создание на основе текста символьного массива.
      // Ссылка на массив присваивается значением полю:
      name=t.ToCharArray();
      // Значение целочисленного поля:
      code=n;
   }
}
```

```
// Класс с главным методом:
class StructAsArgDemo{
   // Метод для отображения значений полей экземпляра
   // структуры, переданного аргументом:
   static void show(MyStruct A){
      // Преобразование символьного массива в текст:
      string txt=new string(A.name);
      // Отображение текста:
      Console.WriteLine("Символьный массив [{0}]",txt);
      // Отображение значения целочисленного поля:
      Console.WriteLine("Числовое поле {0}",A.code);
   }
   // Метод для присваивания новых значений полям
   // экземпляров структуры:
   static void maker(MyStruct A,ref MyStruct B,int k,char s){
      Console.WriteLine("Выполняется метод maker()");
      Console.WriteLine("Первый аргумент:");
      // Отображение значений полей экземпляра структуры:
      show(A);
      Console.WriteLine("Второй аргумент:");
      // Отображение значений полей экземпляра структуры:
      show(B);
      // Изменение значения элемента в массиве:
      A.name[k]=s;
      B.name[k]=s;
      // Изменение значения текстового поля:
      A.code++;
      B.code++;
      Console.WriteLine();
      Console.WriteLine("Первый аргумент:");
      // Отображение значений полей экземпляра структуры:
      show(A);
```

```
        Console.WriteLine("Второй аргумент:");
        // Отображение значений полей экземпляра структуры:
        show(B);
        Console.WriteLine("Метод maker() завершен\n");
    }
    // Главный метод:
    static void Main(){
        // Создание экземпляров структуры:
        MyStruct A=new MyStruct("Alpha",100);
        MyStruct B=new MyStruct("Bravo",200);
        // Вызов статического метода:
        maker(A,ref B,4,'R');
        Console.WriteLine("После вызова метода maker()");
        Console.WriteLine("Экземпляр A:");
        // Отображение значений полей экземпляра структуры:
        show(A);
        Console.WriteLine("Экземпляр B:");
        // Отображение значений полей экземпляра структуры:
        show(B);
    }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 3.7)

```
Выполняется метод maker()
Первый аргумент:
Символьный массив [Alpha]
Числовое поле 100
Второй аргумент:
Символьный массив [Bravo]
Числовое поле 200
```

```
Первый аргумент:
Символьный массив [AlphR]
Числовое поле 101
Второй аргумент:
Символьный массив [BravR]
Числовое поле 201
Метод maker() завершен

После вызова метода maker()
Экземпляр A:
Символьный массив [AlphR]
Числовое поле 100
Экземпляр B:
Символьный массив [BravR]
Числовое поле 201
```

У структуры `MyStruct` есть открытое целочисленное поле `code` и открытое поле `name`, представляющее собой ссылку на символьный массив. Конструктор имеет два аргумента: текстовый и целочисленный. Целочисленный аргумент присваивается в качестве значения полю `code`. Текстовый аргумент (обозначен как `t`) командой `t.ToCharArray()` преобразуется в символьный массив (точнее, на основе текста формируется массив), а ссылка на массив записывается в поле `name`.

В классе с методом `Main()` описано два статических метода. Метод `show()` не возвращает результат, а аргументом методу передается экземпляр структуры `MyStruct`. Метод предназначен для отображения значений полей экземпляра структуры, переданного методу в качестве аргумента. В теле метода командой `string txt=new string(A.name)` на основе символьного массива `name` экземпляра `A` (аргумент метода) создается текстовое значение, и ссылка на текстовый объект записывается в переменную `txt`. Значение этой текстовой переменной отображается в консольном окне. Также в консольном окне отображается значение целочисленного поля `code` экземпляра `A`.

Еще один статический метод `maker()` также не возвращает результат. У метода четыре аргумента. Первые два являются экземплярами

структуры MyStruct, причем первый аргумент (обозначен как A) передается по значению, а второй аргумент (обозначен как B и описан с ключевым словом ref) передается по ссылке. Третий аргумент (обозначен как k) является целым числом, а четвертый аргумент (обозначен как s) является символьным значением. Метод выполняется так: сначала отображаются значения полей экземпляров A и B структуры (команды show(A) и show(B)), переданных методу в качестве аргументов, после этого изменяются значения полей этих экземпляров, и новые значения полей отображаются в консольном окне (еще раз используются команды show(A) и show(B)). Изменение полей выполняется следующим образом. Символьное значение s, переданное четвертым аргументом, присваивается элементу с индексом k (третий аргумент метода) массива name экземпляров структуры (команды A.name[k]=s и B.name[k]=s). Еще командами A.code++ и B.code++ на единицу увеличивается значение поля code экземпляров структуры.

В главном методе программы командами MyStruct A=new MyStruct ("Alpha",100) и MyStruct B=new MyStruct("Bravo",200) создаются два экземпляра структуры. После этого выполняется команда maker(A,ref B,4,'R'). Экземпляр A передается методу по значению, то есть передается копия экземпляра A. Экземпляр B передается по ссылке, то есть «в оригинале». Что происходит? Сначала отображаются значения полей для копии экземпляра A и экземпляра B. С оригиналом все просто. С копией ситуация такая. Поле code у копии экземпляра имеет такое же значение, что и поле code оригинала. Поле name копии имеет такое же значение, как и поле name оригинала. Значение этого поля — ссылка на массив. Получается, что и экземпляр A, и его копия, переданная в метод maker() через поле name, ссылаются на один и тот же массив. Поэтому при выполнении метода maker() меняется значение поля code копии экземпляра A, а также меняется значение элемента в массиве, на который ссылаются поля name и экземпляра A, и его копии (переданной в качестве аргумента). Что касается экземпляра B, то, поскольку он передается по ссылке, все изменения происходят именно с этим экземпляром.

После завершения работы метода maker() мы проверяем значения полей экземпляров A и B. Видим, что у экземпляра B изменилось и значение поля code, и значение символа в символьном массиве. У экземпляра A изменилось только значение элемента в символьном массиве (в силу описанных выше причин).

Экземпляр структуры как результат метода

Погубят тебя слишком большие возможности.

из к/ф «Айболит-66»

Экземпляр структуры может возвращаться результатом метода. Небольшая программа, в которой есть метод, возвращающий результатом экземпляр структуры, представлена в листинге 3.8.

Листинг 3.8. Экземпляр структуры как аргумент метода

```
using System;
// Структура:
struct MyStruct{
   // Ссылка на символьный массив:
   public char[] name;
   // Целочисленное поле:
   public int code;
   // Конструктор:
   public MyStruct(char[] s,int n){
      // Значения полей:
      name=s;
      code=n;
   }
   // Метод для отображения значений полей:
   public void show(){
      Console.Write("Массив: ");
      for(int k=0;k<name.Length;k++){
         Console.Write("|"+name[k]);
      }
      Console.WriteLine("|\nЧисловое поле: {0}",code);
   }
}
// Класс с главным методом:
```

```
class StructAsResultDemo{
   // Метод возвращает результатом экземпляр структуры:
   static MyStruct create(string t,int n){
      // Создание символьного массива на основе текста:
      char[] s=t.ToCharArray();
      // Создание экземпляра структуры:
      MyStruct R=new MyStruct(s,n);
      // Результат метода:
      return R;
   }
   // Главный метод:
   static void Main(){
      // Экземпляр структуры:
      MyStruct A;
      // Присваивание значения экземпляру:
      A=create("Alpha",100);
      // Проверка значений полей:
      A.show();
      // Вызов метода из результата вызова метода:
      create("Bravo",200).show();
   }
}
```

Результат выполнения программы будет следующим:

Результат выполнения программы (из листинга 3.8)

```
Массив: |A|l|p|h|a|
Числовое поле: 100
Массив: |B|r|a|v|o|
Числовое поле: 200
```

Структура `MyStruct` имеет поле `name`, являющееся ссылкой на символьный массив, а также в структуре есть целочисленное поле `code`.

Конструктор в структуре описан так, что новый экземпляр может создаваться на основе ссылки на символьный массив и целочисленного значения. Они передаются аргументами конструктору и определяют значения полей экземпляра. Также в структуре описан метод `show()`, который при вызове отображает в консольном окне содержимое символьного массива и значение числового поля.

В классе с главным методом описан еще и статический метод `create()`, который возвращает результатом экземпляр структуры `MyStruct`, а аргументами методу передаются текстовое и целочисленное значения. В теле метода командой `char[] s=t.ToCharArray()` на основе текстового аргумента `t` создается символьный массив `s`, и затем этот символьный массив передается аргументом конструктору в команде `MyStruct R=new MyStruct(s,n)`, которой создается экземпляр `R` структуры `MyStruct`. Командой `return R` экземпляр структуры возвращается результатом метода.

В главном методе командой `MyStruct A` создаем экземпляр структуры, а затем командой `A=create("Alpha",100)` этому экземпляру в качестве значения присваивается результат вызова метода `create()`. Что происходит в этом случае? При вызове метода `create()`, в соответствии с переданными методу аргументами, создается экземпляр структуры. Этот экземпляр присваивается в качестве значения переменной `A`. Происходит побитовое копирование значений полей. После этого у экземпляра `A` поля будут иметь такие же значения, как у экземпляра, вычисленного при вызове метода `create()`. Проверяем это с помощью команды `A.show()`.

ПОДРОБНОСТИ

В структуре `MyStruct` есть поле, представляющее собой ссылку на массив. Когда при вызове метода `create()` создается экземпляр структуры, то создается и массив, на который ссылается поле `name` этого экземпляра. Экземпляр возвращается в качестве результата метода. Обычно результат метода присваивается какой-то переменной. В таком случае происходит побитовое копирование значений полей. Для поля, являющегося ссылкой на массив, копируется ссылка. Получается, что и экземпляр, являющийся результатом метода, и экземпляр, которому присваивается результат вызова метода, ссылаются на один и тот же массив. Но поскольку в теле метода создается локальный экземпляр, то по завершении работы метода

он будет из памяти удален. А массив останется, поскольку на массив есть ссылка в программе.

Команду с вызовом метода `create()` можно интерпретировать как экземпляр делегата со всеми вытекающими отсюда последствиями. Например, мы можем вызвать метод `show()` из подобного выражения, как это сделано в команде `create("Bravo",200).show()`.

Операторные методы в структурах

> Я за разделение труда, доктор. В Большом пусть поют, я буду оперировать.
>
> *из к/ф «Собачье сердце»*

Как отмечалось ранее, в структурах можно использовать операторные методы.

НА ЗАМЕТКУ

Напомним, что операторные методы описываются как открытые статические. Название операторного метода получается объединением ключевого слова `operator` и символа оператора. Количество аргументов операторного метода совпадает с количеством операндов в соответствующей операции (каждый аргумент отождествляется с операндом). Операторный метод должен возвращать результат.

В листинге 3.9 представлен очень скромный пример, в котором есть структура с операторными методами.

Листинг 3.9. Операторные методы в структурах

```
using System;
// Структура:
struct MyNum{
   // Закрытое целочисленное поле:
   private int number;
   // Конструктор:
```

```
   public MyNum(int n){
      // Значение поля:
      number=n;
   }
   // Метод для вычисления суммы экземпляра структуры
   // и числа:
   public static int operator+(MyNum A,int n){
      return !A+n;
   }
   // Метод для вычисления суммы числа и
   // экземпляра структуры:
   public static int operator+(int n,MyNum A){
      return A+n;
   }
   // Метод для вычисления суммы двух экземпляров
   // структуры:
   public static int operator+(MyNum A,MyNum B){
      return !A+!B;
   }
   // Метод для унарного оператора:
   public static int operator!(MyNum A){
      return A.number;
   }
}
// Класс с главным методом:
class StructAndOperatorDemo{
   // Главный метод:
   static void Main(){
      // Создание экземпляров структуры:
      MyNum A=new MyNum(100);
      MyNum B=new MyNum(200);
      // Использование операторных методов для вычисления
```

```
        // значения выражений:
        Console.WriteLine("A: {0}",!A);
        Console.WriteLine("B: {0}",!B);
        Console.WriteLine("A+B: {0}",A+B);
        Console.WriteLine("A+10: {0}",A+10);
        Console.WriteLine("20+B: {0}",20+B);
    }
}
```

После выполнения программы получаем следующий результат:

Результат выполнения программы (из листинга 3.9)

```
A: 100
B: 200
A+B: 300
A+10: 110
20+B: 220
```

Структура `MyNum` содержит закрытое целочисленное поле `number`, конструктор с одним аргументом (определяет значение поля) и несколько версий операторных методов. Точнее, в структуре описаны операторный метод для унарного оператора `!` и три версии операторного метода для бинарного оператора сложения `+`. Действие унарного оператора `!` на экземпляр структуры определено таким образом, что результатом возвращается значение целочисленного поля этого экземпляра. Операторный метод для оператора `+`, когда операндами являются два экземпляра структуры, результатом возвращает сумму целочисленных полей этих экземпляров. Причем в теле операторного метода для получения значения целочисленного поля экземпляра структуры мы использовали оператор `!`. Далее, сумма экземпляра структуры и числа в результате дает сумму целочисленного поля экземпляра и данного числа. Сумма числа и экземпляра структуры вычисляется как сумма экземпляра структуры и числа.

В главном методе программы мы создаем два экземпляра структуры `MyNum` и с их помощью проверяем работу операторных методов.

Структуры и события

Дикари, аж плакать хочется.

из к/ф «Кин-дза-дза»

В структурах можно объявлять и использовать события. Логика действий в этом случае такая же, как и при использовании событий в классах. Небольшой пример, дающий общее представление об использовании событий в структурах, представлен в листинге 3.10.

Листинг 3.10. Структуры и события

```
using System;
// Объявление делегата:
delegate void MyDelegate(string t);
// Первая структура:
struct Alpha{
   // Открытое текстовое поле:
   public string name;
   // Закрытое поле, являющееся ссылкой
   // на экземпляр делегата:
   private MyDelegate myevent;
   // Объявление события:
   public event MyDelegate MyEvent{
      // Аксессор для добавления ссылки на метод
      // в список обработчиков события:
      add{
         myevent+=value;
      }
      // Аксессор для удаления ссылки на метод
      // из списка обработчиков события:
      remove{
         myevent-=value;
      }
```

```
   }
   // Конструктор:
   public Alpha(string t){
      // Значения полей:
      name=t;
      myevent=null;
   }
   // Метод для генерирования событий:
   public void RaiseMyEvent(){
      // Если ссылка не пустая, то вызывается
      // экземпляр делегата:
      if(myevent!=null) myevent("Alpha: "+name);
   }
}
// Вторая структура:
struct Bravo{
   // Открытое текстовое поле:
   public string name;
   // Объявление события:
   public event MyDelegate MyEvent;
   // Метод для генерирования события:
   public void RaiseMyEvent(){
      // Если список обработчиков не пустой, то
      // генерируется событие:
      if(MyEvent!=null) MyEvent("Bravo: "+name);
   }
}
// Класс с главным методом:
class StructAndEventDemo{
   // Статический метод:
   static void show(string t){
      Console.WriteLine("Произошло событие");
```

```
        Console.WriteLine(t);
        Console.WriteLine("------------------");
    }
    // Главный метод:
    static void Main(){
        // Создание экземпляра первой структуры:
        Alpha A=new Alpha("Экземпляр A");
        // Добавление ссылки на метод в список обработчиков
        // события:
        A. MyEvent+=show;
        // Генерирование события:
        A. RaiseMyEvent();
        // Создание экземпляра второй структуры:
        Bravo B=new Bravo();
        // Присваивание значения текстовому полю:
        B.name="Экземпляр B";
        // Добавление ссылки на метод в список обработчиков
        // события:
        B. MyEvent+=show;
        // Генерирование событие:
        B. RaiseMyEvent();
    }
}
```

Ниже показано, как выглядит результат выполнения программы:

Результат выполнения программы (из листинга 3.10)

```
Произошло событие
Alpha: Экземпляр A
------------------
Произошло событие
Bravo: Экземпляр B
------------------
```

В программе есть объявление делегата `MyDelegate`, соответствующего методам с текстовым аргументом, которые не возвращают результат. Экземпляры этого делегата мы планируем использовать для обработки событий. Также мы описали две структуры `Alpha` и `Bravo`. Структуры функционально схожи, но немного по-разному организованы. В структуре `Alpha` есть открытое текстовое поле `name`. Закрытое поле `myevent` типа `MyDelegate` является ссылкой на экземпляр делегата. Мы это поле используем при описании аксессоров события `MyEvent`: при добавлении ссылки на метод в список обработчиков события или удалении ссылки из списка в действительности изменения происходят с полем `myevent`. В структуре также есть конструктор с текстовым аргументом. Текстовый аргумент определяет значение текстового поля `name`, а поле `myevent` в качестве значения получает пустую ссылку. В данном случае это момент не тривиальный, поскольку в конструкторе должны быть присвоены значения всем полям экземпляра структуры.

Для генерирования события в структуре описан метод `RaiseMyEvent()`, в теле которого проверяется условие `myevent!=null` (истинно, если поле `myevent` содержит ссылки на методы), и при истинном условии командой `myevent("Alpha:"+name)` вызывается экземпляр делегата, на который ссылается поле `myevent`.

Вторая структура называется `Bravo`. В ней также есть текстовое поле `name`. Событие `MyEvent` объявлено без явного описания аксессоров. Конструктора у структуры нет. Метод `RaiseMyEvent()` генерирует событие `MyEvent` при условии, что для события зарегистрированы обработчики.

ПОДРОБНОСТИ

То, что в структуре `Bravo` не описан конструктор — не случайно. Каждая версия конструктора, которая описывается в структуре, должна присваивать значения полям экземпляра структуры. За событием «спрятано» поле, которое содержит ссылки на экземпляры делегата, которые вызываются для обработки события. В структуре `Alpha` такое поле было описано явно, и мы ему присваивали значение в конструкторе. В структуре `Bravo` мы для события не описываем аксессоры и не объявляем поле для записи ссылок на экземпляры делегата. Такое поле создается автоматически, и присвоить ему значение в конструкторе было бы проблематично. Поэтому от конструктора мы отказались.

Для обработки событий мы описали статический метод `show()` с текстовым аргументом. Метод при вызове отображает значение своего текстового аргумента вместе с поясняющим текстом.

В главном методе командой `Alpha A=new Alpha("Экземпляр A")` мы создаем экземпляр первой структуры. Командой `A.MyEvent+=show` добавляем ссылку на метод `show()` в список обработчиков события `MyEvent` экземпляра `A`. Командой `A.RaiseMyEvent()` событие генерируется. Аналогичные операции выполняются с экземпляром `B` структуры `Bravo`. Заслуживает внимания лишь команда `Bravo B=new Bravo()`, которой создается экземпляр структуры. Конструктор в структуре `Bravo` мы не описывали, поэтому теоретически вариантов было два: просто объявить экземпляр структуры или использовать `new`-инструкцию. Подходит только второй вариант, поскольку в этом случае вызывается конструктор по умолчанию (без аргументов), который инициализирует значениями по умолчанию поля структуры (в том числе и техническое поле, связанное с событием). При объявлении экземпляра структуры без `new`-инструкции этого не происходит.

Структуры и интерфейсы

> Я не злоупотребляю. Я, дорогой профессор, только в виде опыта.
>
> *из к/ф «Собачье сердце»*

Структуры не поддерживают наследование, но в структурах могут быть реализованы интерфейсы. Общая идея такая же, как и при реализации интерфейсов в классах: методы, свойства, индексаторы и события, объявленные в интерфейсе, должны быть описаны в структуре, реализующей интерфейс. Если структура реализует интерфейс, то в описании структуры после ее имени (через двоеточие) указывается имя реализуемого интерфейса. Если в структуре реализуется несколько интерфейсов, то имена интерфейсов разделяются запятыми. Программа, в которой иллюстрируется механизм реализации интерфейсов в структурах, представлена в листинге 3.11.

Листинг 3.11. Структуры и интерфейсы

```
using System;
// Интерфейс:
```

```
interface MyInterface{
   // Методы:
   void set(int n);
   void show();
}
// Структура наследует интерфейс:
struct MyStruct:MyInterface{
   // Закрытое целочисленное поле:
   private int code;
   // Метод из интерфейса:
   public void set(int n){
      code=n;
   }
   // Метод из интерфейса:
   public void show(){
      Console.WriteLine("Числовое поле "+code);
   }
}
// Класс с главным методом:
class StructAndInterfaceDemo{
   // Главный метод:
   static void Main(){
      // Создание экземпляра структуры:
      MyStruct A=new MyStruct();
      // Вызов метода для присваивания значения полю:
      A.set(100);
      Console.WriteLine("Экземпляр A:");
      // Вызов метода для отображения значения поля:
      A.show();
      // Интерфейсная переменная:
      MyInterface R;
      // Интерфейсной переменной значением присваивается
```

```
        // экземпляр структуры:
        R=A;
        Console.WriteLine("Переменная R:");
        // Вызов метода для отображения значения поля:
        R.show();
        // Вызов метода для присваивания значения полю:
        R.set(200);
        Console.WriteLine("Переменная R:");
        // Вызов метода для отображения значения поля:
        R.show();
        Console.WriteLine("Экземпляр A:");
        // Вызов метода для отображения значения поля:
        A.show();
    }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 3.11)

```
Экземпляр A:
Числовое поле 100
Переменная R:
Числовое поле 100
Переменная R:
Числовое поле 200
Экземпляр A:
Числовое поле 100
```

Интерфейс называется `MyInterface`, и в нем объявлены два метода: метод `set()` с целочисленным аргументом и метод `show()` без аргументов (результат методы не возвращают). Структура `MyStruct` реализует интерфейс `MyInterface`. В структуре объявлено закрытое целочисленное поле `code`. Метод `set()` предназначен для присваивания

значения полю code, а метод show() нужен для отображения значения поля.

В методе Main() командой MyStruct A=new MyStruct() создается экземпляр структуры. Создавать экземпляр нужно именно так (с использованием инструкции new), поскольку поле code закрытое, а перед вызовом любого метода поля должны быть инициализированы. Использованный нами способ создания экземпляра подразумевает вызов конструктора по умолчанию, вследствие чего поле code инициализируется с начальным нулевым значением.

Значение 100 полю code экземпляра A присваиваем с помощью команды A.set(100). Для проверки значения поля экземпляра A используем команду A.show().

Командой MyInterface R объявляется интерфейсная переменная. Поскольку структура MyStruct реализует интерфейс MyInterface, то интерфейсной переменной можно присвоить в качестве значения экземпляр структуры (команда R=A). Через интерфейсную переменную доступ будет только к тем методам, которые объявлены в интерфейсе. Но не это главное. Дело в том, что структура относится к типу данных с прямым доступом к значению. Поэтому при выполнении команды R=A переменная R в качестве значения получает ссылку на копию экземпляра A.

ПОДРОБНОСТИ

При выполнении команды R=A создается копия экземпляра A и ссылка на эту копию записывается в переменную R. Значением переменной R является ссылка, а не экземпляр. Интерфейсная переменная R ссылается на копию экземпляра A структуры MyStruct так же, как объектные переменные ссылаются на объекты классов. Если объявить еще одну интерфейсную переменную P интерфейса MyInterface, а затем выполнить команду P=R, то переменные P и R будут ссылаться на один и тот же экземпляр и этот экземпляр будет копией экземпляра A.

Значение поля code экземпляра, на который ссылается интерфейсная переменная R, проверяем с помощью команды R.show(). После выполнения команды R.set(200) меняется значение поля code экземпляра, на который ссылается интерфейсная переменная R. Значение поля code

экземпляра A остается неизменным. Убеждаемся в этом с помощью команд R.show() и A.show().

Резюме

> Даль моей карьеры видна мне совершенно отчетливо.
>
> *из к/ф «Собачье сердце»*

- Перечисление представляет собой тип данных, формируемый набором целочисленных констант. При объявлении перечисления указывается ключевое слово enum, название перечисления, а в фигурных скобках перечисляются названия констант. По умолчанию константы получают значения, начиная с нуля. Значение каждой следующей константы на единицу больше предыдущей константы в списке. При необходимости значения для констант можно указать в явном виде.
- Переменной, типом которой указано перечисление, можно присваивать в качестве значения одну из констант в перечислении. Константа указывается вместе с именем перечисления. Имя перечисления и название константы разделяются точкой. В случае необходимости можно применять явное преобразование типов.
- Структура напоминает класс, но в отличие от класса, относящегося к ссылочным типам, структура относится к типу с прямым доступом к значению. Описывается структура подобно классу, но вместо ключевого слова class используется ключевое слово struct.
- По сравнению с классами, на структуры накладываются определенные ограничения. Членами структуры могут быть поля, методы, свойства, индексаторы и события. Члены структуры могут быть открытыми и закрытыми (по умолчанию). Ключевое слово protected не используется.
- Методы в структуре могут перегружаться, а также могут использоваться операторные методы.
- В структуре могут описываться конструкторы. Конструктор без аргументов используется по умолчанию, и его нельзя переопределить.
- У структуры нет деструктора.

- Структуры не поддерживают наследование, но могут реализовать интерфейсы.
- Экземпляр структуры может создаваться простым объявлением переменной, типом которой указана структура. В этом случае поля структуры не инициализируются, и перед первым использованием экземпляра структуры полям необходимо присвоить значение.
- Экземпляр структуры можно создавать таким же образом, как и объект класса, с использованием инструкции `new`. В этом случае при создании экземпляра вызывается конструктор, версия которого определяется переданными конструктору аргументами. Если аргументы конструктору не переданы, то вызывается конструктор по умолчанию, в результате чего поля экземпляров структуры инициализируются значениями по умолчанию.
- Значением переменной типа структуры является экземпляр структуры. Если одному экземпляру структуры присваивается в качестве значения другой экземпляр структуры, то выполняется побитовое копирование полей.

Задания для самостоятельной работы

> Предлагаю вам взять несколько журналов в пользу детей Германии. По полтиннику штука.
>
> *из к/ф «Собачье сердце»*

1. Напишите программу, в которой объявляется перечисление для представления дней недели. Предложите методы (или добавьте в главный метод блоки кода), позволяющие по числовому значению определить день недели (с учетом периодической повторяемости дней), а также позволяющие по двум значениям из перечисления определить минимальное количество дней между соответствующими днями недели.

2. Напишите программу, в которой объявляется структура с целочисленным, текстовым и символьным полями. Предложите такие версии конструктора: с тремя аргументами (целое число, текст, символ), с двумя аргументами (целое число и текст) и с одним аргументом (текст). В структуре должен быть метод, при вызове которого отображаются значения полей экземпляра структуры.

3. Напишите программу, содержащую структуру с полем, которое является символьным массивом. Предложите две версии конструктора: с текстовым аргументом (символьный массив формируется на основе текста) и с двумя аргументами (целое число и символ). В последнем случае целочисленный аргумент определяет размер массива, а символьный аргумент определяет значение, которое присваивается всем элементам массива. Опишите в структуре индексатор, который позволял бы считывать значение символа из массива и присваивать значение элементу массива. Предложите метод, который при вызове менял бы порядок символов в массиве на противоположный. Переопределите метод `ToString()` таким образом, чтобы он возвращал текстовую строку, содержащую символы из массива и числовые коды этих символов.

4. Напишите программу, содержащую структуру с целочисленным массивом. Опишите конструктор с одним аргументом, определяющим размер массива. Массив должен заполняться случайными числами. В структуре должны быть методы, возвращающие результатом наибольший элемент в массиве, наименьший элемент в массиве, а также метод, возвращающий среднее значение элементов в массиве (сумма элементов массива, деленная на количество элементов в массиве).

5. Напишите программу, содержащую структуру с тремя закрытыми целочисленными полями. Предложите версии конструктора с одним, двумя и тремя целочисленными аргументами. В структуре должно быть два целочисленных свойства. Одно свойство результатом возвращает наименьшее из значений полей экземпляра структуры. Другое свойство возвращает наибольшее из значений полей экземпляра структуры. Предложите вариант обработки ситуации, когда первому или второму свойству присваивается значение.

6. Напишите программу, в которой есть структура с двумя целочисленными полями. Опишите в структуре операторные методы для выполнения операций сложения, вычитания и умножения двух экземпляров структуры. Во всех случаях результатом должен быть новый экземпляр структуры, значения полей которого вычисляются соответственно как сумма, разность или произведение соответствующих полей экземпляров-операндов. Предложите два операторных метода для унарных операторов, которые возвращали бы результатом наибольшее и наименьшее из значений полей экземпляра структуры.

7. Напишите программу, в которой есть структура с символьным полем. Также в программе должен быть статический метод, аргументом которому передается текстовое значение. Результатом метод возвращает массив из экземпляров структуры. Размер массива определятся длиной текста. Значения символьных полей экземпляров структуры в массиве задаются символами из текста. Предложите метод, которому в качестве аргумента передается массив из экземпляров структуры, а результатом метод возвращает текстовую строку, состоящую из символьных значений полей экземпляров в массиве.

8. Напишите программу, в которой объявлены две структуры. У одной структуры есть текстовое поле, а у другой структуры есть символьное поле. В первой структуре (с текстовым полем) должен быть метод с целочисленным аргументом, который результатом возвращает экземпляр второй структуры (с символьным полем). Значение символьного поля экземпляра-результата определяется как символ из текста (поле экземпляра, из которого вызывается метод) с индексом, определяемым аргументом метода.

9. Напишите программу, в которой есть структура с двумя целочисленными полями. Предложите статический метод, аргументом которому передается целочисленный массив. Результатом метод возвращает экземпляр структуры, первое поле которого равно значению максимального (или минимального) элемента в массиве, а второе поле экземпляра равно индексу этого элемента в массиве.

10. Напишите программу, в которой объявлены две структуры. У первой структуры есть целочисленное поле, а у второй структуры есть два целочисленных поля. Предложите операторный метод, с помощью которого сумма двух экземпляров первой структуры возвращала бы результатом экземпляр второй структуры (поля экземпляра-результата — это поля суммируемых экземпляров). Во второй структуре опишите метод, который результатом возвращает массив из двух экземпляров первой структуры. В этом случае экземпляр с двумя полями, из которого вызывается метод, «разбивается» на два экземпляра, у каждого из которых по одному полю.

Глава 4
УКАЗАТЕЛИ

> Вот, доктор, что происходит, когда исследователь вместо того, чтобы идти параллельно и ощупью с природой, форсирует вопрос и приоткрывает завесу.
>
> *из к/ф «Собачье сердце»*

В этой главе речь пойдет о достаточно интересном механизме языка C# — мы будем обсуждать *указатели* и все, что с ними связано. В частности, мы узнаем:

- что представляют собой указатели и как они используются;
- почему код, в котором используются указатели, является небезопасным;
- какие основные операции можно выполнять с указателями;
- как создаются указатели на структуры;
- как указатели связаны с массивами;
- что связывает указатели с текстовыми строками;
- что такое многоуровневая адресация и как она используется.

Как всегда, будет много примеров, с помощью которых мы проиллюстрируем практические приемы применения указателей.

Знакомство с указателями

> Пусть меня ждут в пирамиде Хеопса у 8-й мумии, 3-й коридор налево.
>
> *из м/ф «Приключения капитана Врунгеля»*

Если мы объявляем переменную базового типа (например, целочисленную), то в памяти под эту переменную выделяется место. Объем

выделяемой памяти зависит от типа переменной. В эту область памяти записывается значение переменной, и в случае необходимости оно оттуда считывается. Когда мы выполняем операции по изменению значения переменной, новое значение заносится в выделенную под переменную область памяти. У этой области памяти есть адрес. Но обычно он нас мало интересует. Мы обращаемся к переменной по имени. Этого вполне достаточно, чтобы присвоить значение переменной, а также изменить или прочитать значение переменной. Другими словами, мы получаем доступ к области памяти не напрямую, а через имя переменной. Вместе с тем в языке C# существует механизм, позволяющий обращаться к памяти напрямую, минуя переменную, под которую эта память выделена. Делается это с помощью *указателей*.

Общая идея состоит в том, чтобы узнать (получить) адрес, по которому записана переменная, и затем обращаться к соответствующей области памяти не через имя переменной, а по адресу области памяти. Адрес переменной записывается как значение в другую переменную, которая называется указателем. Таким образом, указатель — это переменная, значением которой является адрес другой переменной.

НА ЗАМЕТКУ

Технически адрес переменной — это некоторое целое число. Гипотетически мы могли бы запоминать адреса как целые числа. Но это бесперспективный подход, поскольку нам адрес как таковой не очень-то и нужен. Нам нужен адрес как средство доступа к области памяти. Поэтому адреса запоминаются в специальных переменных, которые называются указателями. Значения указателей (адреса) обрабатываются по специальным правилам (называются адресной арифметикой). Запоминая адреса в специальных переменных, мы как бы даем знать, что это именно адреса, а не обычные целые числа, и что обрабатывать данные значения следует как адреса, а не как целые числа.

При объявлении указателя важно отобразить два момента. Во-первых, необходимо показать, что речь идет об объявлении указателя. Во-вторых, нужно явно указать, какого типа значение может быть записано в область памяти, адрес которой будет содержать указатель. Почему это важно? Дело в том, что память разбивается на блоки, размер каждого блока равен 1 байту (это 8 битов). Каждый такой однобайтовый блок имеет адрес. Теперь представим, что объявляется целочисленная

переменная типа int. Для такой переменной выделяется 32 бита, что составляет 4 байта. Таким образом, для целочисленной переменной выделяется 4 однобайтовых блока. Если бы переменная была типа char, то для нее выделялось бы 16 битов, или 2 байта — то есть 2 однобайтовых блока. А для переменной типа byte выделяется всего 1 однобайтовый блок (8 битов). Если под переменную выделяется несколько однобайтовых блоков, то адресом области памяти, в которую записывается значение переменной, является адрес первого однобайтового блока. Получается, что по одному лишь адресу мы можем определить место, начиная с которого в памяти записывается значение переменной. Но нам еще нужно знать размер области памяти, которую занимает переменная. Этот размер можно определить по типу переменной. Поэтому при объявлении указателей важно также определить, с переменными какого типа предполагается работать.

С учетом двух упомянутых выше обстоятельств объявление указателя происходит следующим образом. Указывается идентификатор типа, соответствующий типу переменной, адрес которой может быть записан в указатель, и символ звездочка *. После этой конструкции указывается имя указателя. Например, если мы хотим объявить указатель на целочисленное значение (то есть указатель, значением которого может быть адрес переменной типа int), то можем использовать следующую инструкцию:

```
int* p;
```

В данном случае объявляется указатель p, которому в качестве значения могут присваиваться адреса переменных типа int. Мы в таком случае будем говорить, что p является указателем на целочисленное значение типа int.

Если мы хотим объявить два указателя q и r на значение типа double (указатели, которым в качестве значений можно присваивать адреса переменных типа double), мы могли бы воспользоваться такой командой:

```
double* q,r;
```

Корректным является и такое объявление указателя:

```
char* s;
```

В данном случае объявляется символьный указатель s — значением указателю можно присвоить адрес переменной символьного типа.

Стоит заметить, что объявить указатель можно только для нессылочного типа. Проще говоря, мы не можем объявить указатель на объект класса. Объяснение в данном случае простое и связано с тем, что доступ к объекту мы получаем не напрямую, а через объектную переменную. Значение объектной переменной — не объект, а ссылка на объект (то есть фактически адрес объекта). Вместе с тем, как мы увидим далее, указатели могут применяться при работе с массивами и текстом.

НА ЗАМЕТКУ

При объявлении указателей такие выражения, как `int*` или `double*` (то есть выражение вида `тип*`), с некоторой натяжкой можно интерпретировать как «тип указателя». Во всяком случае, такой подход помогает во многих нетривиальных случаях интуитивно находить правильные ответы и решения.

Также стоит заметить, что в языке C++ объявление `double* q,r` означает, что объявляется указатель `q` на значение типа `double` и переменная (обычная, не указатель) `r` типа `double`. То есть в языке C++ звездочка `*` в объявлении указателя «применяется» только к первой переменной. В языке C# это не так: командой `double* q, r` объявляется два указателя (`q` и `r`) на значение типа `double`.

В правиле, согласно которому при объявлении указателя нужно идентифицировать тип значения, на которое может ссылаться указатель, имеется исключение: мы можем объявить указатель на значение неопределенного типа. В таком случае в качестве идентификатора типа используется ключевое слово `void`. Вот пример объявления такого указателя:

```
void* pnt;
```

Данной командой объявлен указатель `pnt`, и при этом не определен тип значения, на которое может ссылаться указатель (тип переменной, адрес которой может быть присвоен указателю). Такой указатель можно использовать только для запоминания адреса. Никакие иные операции с таким указателем выполнить не удастся. Оно и понятно. Ведь если неизвестно, какого типа переменная записана в области памяти с данным адресом, то и неизвестно, какой объем памяти она занимает. А если объем доступной памяти неизвестен, то и операции с ней не выполняются.

Итак, мы выяснили, как указатель объявляется. Но что же с ним делать? Есть две базовые операции, с которых мы и начнем наше знакомство с использованием указателей. Операции такие:

- Получение адреса переменной.
- Получение доступа к значению, записанному по адресу.

Для получения адреса переменной используют инструкцию `&` (амперсанд). Если ее указать перед именем переменной (получается выражение вида `&переменная`), то значением такого выражения будет адрес области памяти, в которую записано значение переменной. Например, имеется целочисленная переменная `n` типа `int` и указатель `p` на значение типа `int`, объявленные следующим образом:

```
int n;   // Переменная
int* p;  // Указатель
```

Тогда значением выражения `&n` является адрес переменной `n`. Поскольку переменная `n` объявлена с типом `int` и указатель `p` также объявлен с идентификатором `int`, то адрес переменной `n` может быть присвоен указателю `p`. Корректной является следующая команда:

```
p=&n;
```

Обратная операция к получению адреса переменной — получение доступа к значению, записанному по адресу. Для этого перед указателем, содержащим соответствующий адрес, нужно указать звездочку `*` (получается выражение вида `*указатель`). Скажем, выражение `*p` — это значение, записанное по адресу, содержащемуся в указателе `p`. Причем с помощью выражения вида `*указатель` можно не только прочитать значение, записанное по адресу, но и записать по этому адресу новое значение. В частности, мы могли бы воспользоваться такими командами:

```
*p=123;                // Присваивание значения
Console.WriteLine(*p); // Считывание значения
```

Первой из представленных двух команд (выражение `*p=123`) по адресу из указателя `p` записывается целочисленное значение `123`. Если значение указателю `p` присваивается командой `p=&n`, то речь фактически идет о переменной `n`. То есть командой `*p=123` значение

присваивается переменной n. При выполнении команды Console.WriteLine(*p) в консольном окне отображается значение, записанное по адресу из указателя p. Несложно догадаться, что речь идет о значении переменной p.

ПОДРОБНОСТИ

Указатель позволяет получить доступ к области памяти, выделенной под переменную, минуя непосредственное обращение к этой переменной. На каком-то этапе мы узнаем адрес области памяти, в которой хранится значение переменной, и с помощью указателя получаем доступ к этой области памяти. С помощью указателя мы можем прочитать значение, записанное в память, а также мы можем записать туда новое значение. И здесь критически важно знать, какой объем памяти следует использовать при чтении/записи значений. Также нужно знать, как интерпретировать прочитанное значение (скажем, как целочисленное неотрицательное значение, целочисленное значение со знаком или символ). Все это определяется по идентификатору типа, указанному при объявлении указателя. Этот тип должен совпадать с типом переменной, адрес которой заносится в указатель. В случае необходимости можно использовать явное приведение типа. Такие ситуации рассматриваются немного позже.

Перед тем как рассмотреть первый пример, отметим некоторые особенности способа реализации программ, в которых используются указатели.

Как мы увидим далее, получив доступ к области памяти, мы автоматически получаем доступ и к соседним ячейкам. Формально это означает, что мы получаем возможность выполнять операции с памятью напрямую. Такие операции считаются небезопасными, поскольку исполнительная система, под управлением которой выполняются программы, не может проконтролировать безопасность выполнения этих операций. Поэтому соответствующий код считается небезопасным — основной груз ответственности по реализации корректной и неконфликтной работы программы ложится на плечи программиста. В языке C# небезопасный код выделяется в отдельные блоки, которые помечаются ключевым словом unsafe. Например, если в главном методе программы используются указатели, то главный метод можно описать с ключевым словом unsafe (или пометить этим ключевым словом блок, в котором указатели задействованы). Также следует изменить настройки приложения, разрешив

использование небезопасного кода. Для этого на вкладке свойств проекта (можно открыть с помощью команды **Properties** в контекстном меню проекта или в главном меню **Project**) в разделе сборки (раздел **Build**) следует установить флажок опции, разрешающей использование небезопасного кода (опция **Allow unsafe code**).

Теперь перейдем к рассмотрению примера, в котором используются указатели. Интересующая нас программа представлена в листинге 4.1.

Листинг 4.1. Знакомство с указателями

```
using System;
// Класс с главным методом:
class PointersDemo{
   // Главный метод (описан с ключевым словом unsafe):
   unsafe static void Main(){
      // Объявление целочисленной переменной:
      int n;
      // Объявление указателя на значение типа int:
      int* p;
      // Присваивание указателю значения:
      p=&n;
      // Через указатель присваивается значение переменной:
      *p=123;
      // Отображение значения переменной:
      Console.WriteLine("Значение переменной n={0}",n);
      Console.WriteLine("Значение выражения *p={0}",*p);
      Console.WriteLine("Адрес переменной n: {0}",(uint)p);
      Console.WriteLine();
      // Объявление указателя на значение типа byte:
      byte* q;
      // Объявление указателя на значение типа char:
      char* s;
```

```
        // Присваивание указателей и явное приведение типа:
        q=(byte*)p;
        s=(char*)p;
        // Переменной присваивается новое значение:
        n=65601;
        // Проверка значений указателей:
        Console.WriteLine("Адрес в указателе p: {0}",(uint)p);
        Console.WriteLine("Адрес в указателе q: {0}",(uint)q);
        Console.WriteLine("Адрес в указателе s: {0}",(uint)s);
        Console.WriteLine();
        // Отображение значения через указатель:
        Console.WriteLine("Значение типа int: {0}",*p);
        Console.WriteLine("Значение типа byte: {0}",*q);
        Console.WriteLine("Значение типа char: \'{0}\'",*s);
        Console.WriteLine("Значение переменной n={0}",n);
        Console.WriteLine();
        // Присваивание значения через указатель:
        *s='F';
        // Проверка значений:
        Console.WriteLine("Значение типа int: {0}",*p);
        Console.WriteLine("Значение типа byte: {0}",*q);
        Console.WriteLine("Значение типа char: \'{0}\'",*s);
        Console.WriteLine("Значение переменной n={0}",n);
    }
}
```

Для того чтобы скомпилировать эту программу, после того как создан проект и введен программный код, следует открыть окно свойств проекта. Например, в контекстном меню проекта выбрать команду **Properties**, как показано на рис. 4.1.

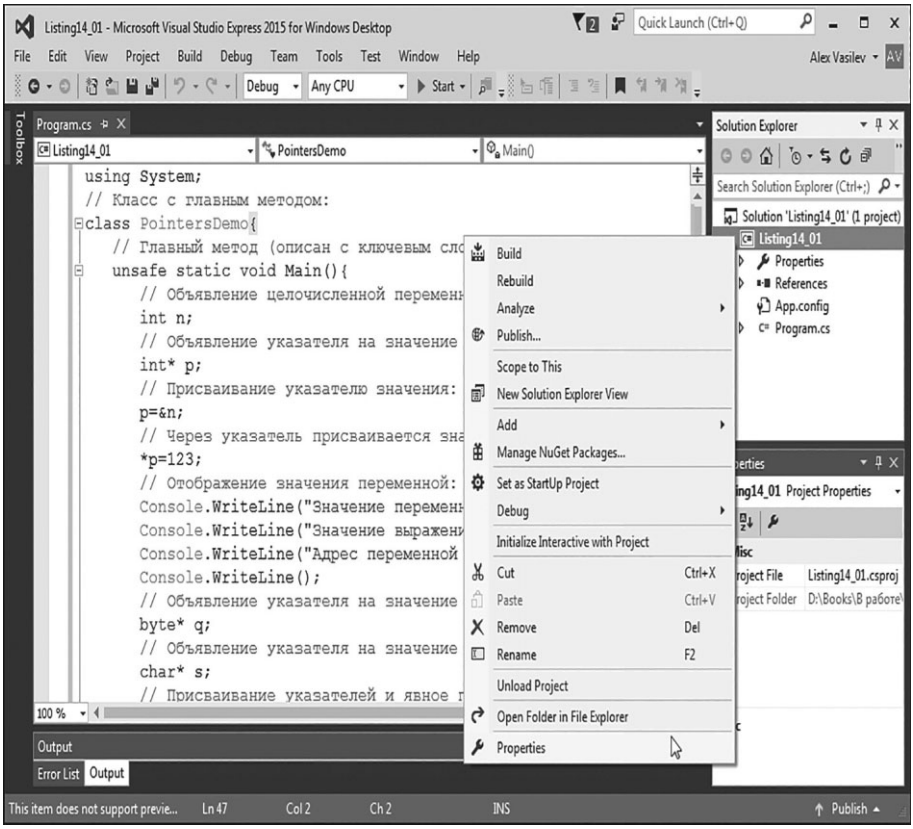


Рис. 4.1. В контекстном меню проекта выбирается команда **Properties**

Должна открыться вкладка свойств проекта, как показано на рис. 4.2.

Там следует выбрать раздел **Build** и установить флажок опции **Allow unsafe code** (см. рис. 4.2). После этого вкладку свойств проекта можно закрыть. Для компилирования и запуска программы на выполнение используем команду **Start Without Debugging** из меню **Debug** (рис. 4.3) или нажимаем комбинацию клавиш <Ctrl>+<F5>.

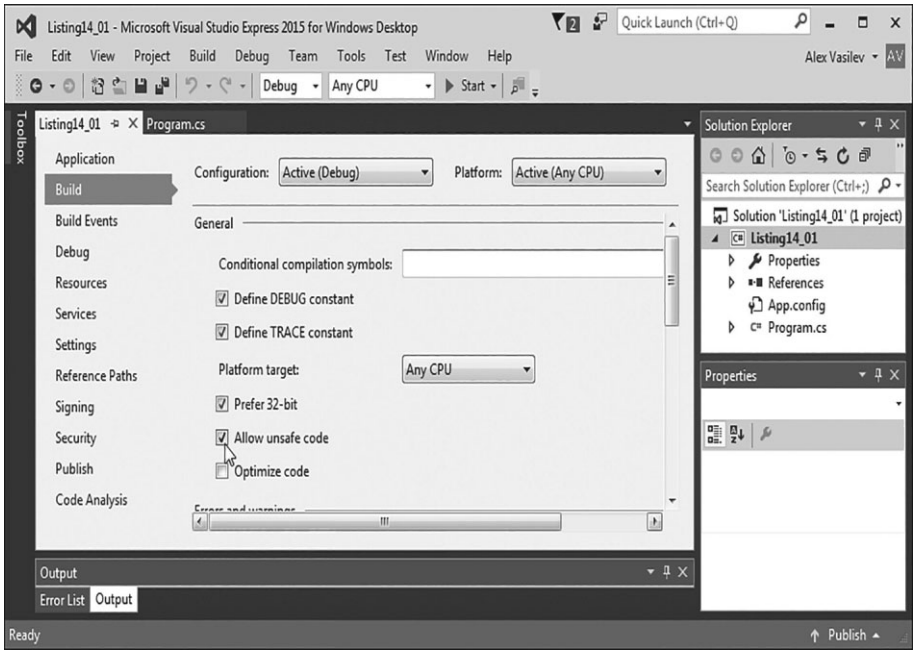


Рис. 4.2. Настройка режима использования небезопасного кода на вкладке свойств проекта

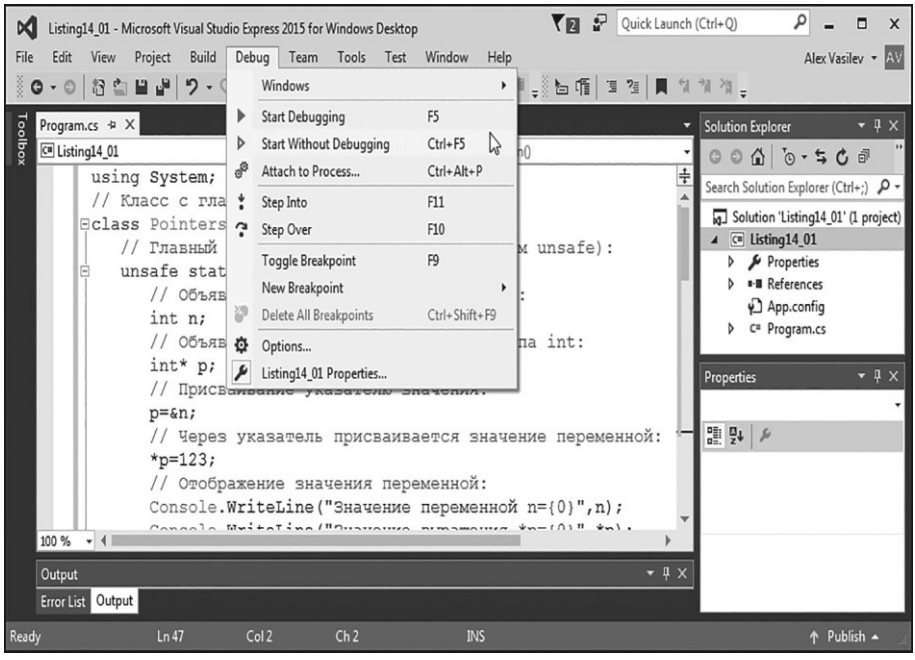


Рис. 4.3. Запуск программы на выполнение с помощью команды **Start Without Debugging** из меню **Debug**

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 4.1)

```
Значение переменной n=123
Значение выражения *p=123
Адрес переменной n: 2028064

Адрес в указателе p: 2028064
Адрес в указателе q: 2028064
Адрес в указателе s: 2028064

Значение типа int: 65601
Значение типа byte: 65
Значение типа char: 'A'
Значение переменной n=65601

Значение типа int: 65606
Значение типа byte: 70
Значение типа char: 'F'
Значение переменной n=65606
```

Главный метод программы описывается с ключевым словом `unsafe`. В теле метода объявляется переменная `n` типа `int`, а также указатель `p` на значение типа `int`. Командой `p=&n` указателю `p` значением присваивается адрес переменной `n`. Причем происходит это еще до того, как переменной `n` присвоено значение. Но проблемы здесь нет, поскольку значение указателя `p` — это адрес переменной `n`. А адрес у переменной появляется в результате объявления этой переменной. То есть, несмотря на то что значение переменной `n` еще не присвоено, адрес у нее уже имеется.

Значение переменной `n` присваивается не напрямую, а через указатель `p`. Для этого мы использовали команду `*p=123`. В результате в область памяти, выделенную под переменную `n`, записывается значение `123`. Именно это значение мы получаем, когда проверяем значение переменной `n` или значение выражения `*p`. Значение выражения `*p` — это число,

записанное по адресу, который является значением указателя p. Чтобы узнать этот адрес, мы используем выражение (uint)p, в котором значение указателя p явно приводится к формату целого неотрицательного числа типа uint. Само по себе значение адреса мало о чем говорит. Более того, он от запуска к запуску меняется, поскольку при разных запусках программы под переменную может выделяться область памяти в разных местах. Важно то, что если уж переменная получила адрес, то он будет неизменным до окончания выполнения программы.

Кроме указателя p, в программе командами byte* q и char* s объявляются еще два указателя. Указатель q может ссылаться на значение типа byte, а указатель s может содержать в качестве значения адрес области памяти, в которую записано значение типа char. После этого командами q=(byte*)p и s=(char*)p указателям присваиваются значения. Несложно сообразить, что оба указателя q и s получают в качестве значения тот же адрес, что записан в указатель p. Убедиться в этом несложно — достаточно сравнить значения выражений (uint)p, (uint)q и (uint)s. От запуска к запуску значения могут быть разными, но между собой они всегда совпадают, поскольку речь идет об одном и том же адресе, записанном в указатели p, q и s. В чем же тогда разница между этими указателями? А разница в том, какое количество однобайтовых ячеек «попадает под контроль» указателя. Адрес, записанный в указатель, — это адрес одной однобайтовой ячейки. Если мы получаем доступ к памяти через указатель p, то, поскольку он предназначен для работы с целыми числами, операции выполняются с четырьмя однобайтовыми ячейками: той, чей адрес записан в указатель, и еще тремя соседними. Причем значение, которое записывается в память, и значение, считываемое из памяти, интерпретируются как целые числа типа int. Если доступ к памяти получаем с помощью указателя s, то операции выполняются с двумя ячейками памяти и значение в этих двух ячейках интерпретируется как символьное (значение типа char). Наконец, если мы получаем доступ к памяти с помощью указателя q, то операции выполняются только с одной ячейкой, а значение в этой ячейке интерпретируется как целочисленное значение типа byte.

Командой n=65601 переменной n присваивается новое значение. Число 65601 можно представить так: $65\,601 = 65\,536 + 64 + 1 = 2^{16} + 2^6 + 2^0$. Поэтому в двоичном коде из 32 битов число 65 601 выглядит как 00000000 00000001 00000000 01000001. Первый, самый младший байт содержит код 01000001, второй байт нулевой (код 00000000), третий байт имеет код 00000001, и четвертый, самый старший байт

тоже нулевой (код 00000000). Значение выражения *p вычисляется по всем четырем байтам и интерпретируется как число типа int. Поэтому мы ожидаемо получаем значение переменной n.

При вычислении значения выражения *q используется только первый байт с кодом 01000001. Этот код интерпретируется как неотрицательное целочисленное значение. Если перевести данный двоичный код в десятичный, то получим число 65.

Наконец, значение выражения *s вычисляется по двум байтам (первому и второму). Получаем бинарный код 00000000 01000001, в котором второй байт нулевой (состоит из одних нулей). Поэтому формально бинарный код соответствует числу 65. Но поскольку s является указателем на символьное значение, то результат интерпретируется как символ. В кодовой таблице символов код 65 имеет буква 'A' (английский алфавит).

При выполнении команды *s='F' в первый и второй байты записывается код символа 'F'. Код этого символа равен 70. Число 70 в двоичном коде (из 16 битов) выглядит как 00000000 01000110. То есть второй байт как был нулевым, так нулевым и остался, а первый изменился: был 01000001, а стал 01000110. Все четыре байта теперь имеют код 00000000 00000001 00000000 01000110. Этот код соответствует числу 65 606. Такое значение получаем при вычислении значения выражения *p, оно же новое значение переменной n.

Когда вычисляется значение выражения *q, то используется только первый байт с кодом 01000110, означающим число 70. При проверке значения выражения *s естественным образом получаем значение 'F'.

Адресная арифметика

> Завтра вечером вылетаете за товаром в квадрат 4. Будьте внимательны, помните о строжайшей секретности этой операции.
>
> *из м/ф «Приключения капитана Врунгеля»*

Выше мы видели, что адрес — это целое число. Мы можем представлять себе одномерную последовательность ячеек (хотя в действительности все намного сложнее — но в данном случае это не важно), каждая размером в 1 байт. У каждой ячейки есть адрес (целое число). Будем исходить

из того, что адрес увеличивается справа налево. Адреса соседних ячеек отличаются на 1. Если имеется некоторая ячейка с адресом, то соседняя ячейка справа имеет адрес на единицу меньше, а соседняя ячейка слева имеет адрес на единицу больше. Для записи значений может использоваться несколько ячеек (будем называть это блоком) — все зависит от типа значения. Например, значение типа `double` занимает 8 ячеек, значение типа `int` занимает 4 ячейки, значение типа `char` занимает 2 ячейки, а значение типа `byte` занимает 1 ячейку. То есть для записи значений разных типов используются блоки памяти разного размера. В указатель при этом записывается адрес первой ячейки в блоке. Поэтому если в памяти подряд размещено два значения типа `byte`, то адреса блоков, в которые записаны эти значения, будут отличаться на 1. Если рядом в памяти два блока с `int`-значениями, то адреса этих блоков отличаются на 4. Адреса соседних `double`-блоков отличаются на 8. Таким образом, дискретность изменения адресов соседних блоков памяти определяется типом значений, которые записаны в эти блоки.

НА ЗАМЕТКУ

Здесь приводятся объемы памяти, которые должны выделяться для значений разных типов по стандарту языка C#. Узнать фактический объем памяти (в байтах), выделяемый компилятором под значение определенного типа, можно с помощью инструкции `sizeof`. Например, результатом выражения `sizeof(int)` является целое число, равное количеству байтов, выделяемых для значения типа `int` (должно быть равно 4). Значение выражения `sizeof(char)` — объем памяти (в байтах), выделяемый для значения типа `char` (по стандарту должно быть 2).

Считывание значения из области памяти и запись в область памяти значения с использованием указателей — далеко не единственные поддерживаемые в C# операции. Есть группа важных и полезных операций, которые могут выполняться с указателями.

НА ЗАМЕТКУ

Далее под ячейкой будет подразумеваться однобайтовый блок, а под блоком в общем случае мы будем подразумевать группу из нескольких ячеек. Фраза «указатель ссылается на ячейку» понимается в том смысле, что указатель содержит адрес соответствующей ячейки в качестве значения.

- Указатели можно сравнивать с помощью операторов ==, !=, <, >, <= и >=. При этом сравниваются числовые значения адресов.
- Можно вычислять разность указателей (одного типа). Результатом будет целое число, равное количеству блоков между ячейками, на которые ссылаются указатели. Другими словами, целое число определяет, на сколько блоков памяти смещена одна ячейка по отношению к другой. Размер блока памяти (единица, в которой измеряется смещение) определяется типом базового значения, указанного при объявлении указателей. Допустим, `p` и `q` являются указателями на значения типа `int`, а разность указателей `q-p` равна `3`. Это означает, что если взять ячейку с адресом из указателя `p` и выполнить смещение на `3` `int`-блока (в направлении увеличения адреса), то получим ячейку, адрес которой записан в указатель `q`. Поскольку один `int`-блок — это `4` однобайтовых блока, то разность `q-p`, равная `3`, означает, что ячейка, на которую ссылается указатель `q`, смещена по отношению к ячейке, на которую ссылается указатель `p`, на `12` однобайтовых блоков. Соответственно, разность адресов в указателях `q` и `p` равна `12`. Если бы указатели `p` и `q` были предназначены для работы с `char`-значениями, то речь бы шла о смещении на `3` блока, каждый из которых состоит из `2` ячеек (значение типа `char` записывается в `2` однобайтовых блока). Поэтому адрес ячейки, на которую ссылается указатель `q`, был бы больше адреса из указателя `p` на `6`. Отрицательная разность указателей означает, что смещение выполняется уменьшением адреса.
- К указателю можно прибавлять целое число, а также из указателя можно вычитать целое число. Результатом будет адрес ячейки, смещенной по отношению к адресу из исходного указателя на количество блоков, определяемых прибавляемым или вычитаемым числом. Размер блока определяется типом, указанным при объявлении указателя. При прибавлении положительного числа адрес увеличивается, а при вычитании положительного числа адрес уменьшается. Так, если `p` является указателем для `int`-значений, то результатом выражения `p+3` является адрес ячейки, смещенной (с увеличением адреса) по отношению к ячейке с адресом из указателя `p` на `3` `int`-блока (или `12` однобайтовых блоков). Результат выражения `p-3` вычисляется аналогичным образом, но адрес при этом уменьшается.
- Указатели можно индексировать: после имени указателя в квадратных скобках указывается целочисленный индекс, который может быть в том числе и отрицательным. Результатом выражения

на основе проиндексированного указателя является значение, записанное в блоке, смещенном по отношению к ячейке, на которую ссылается указатель. Количество блоков для смещения определяется индексом. Если индекс положительный — адрес при смещении увеличивается, если индекс отрицательный — адрес при смещении уменьшается. Через выражение на основе проиндексированного указателя можно прочитать значение в блоке памяти и записать значение в этот блок памяти (присвоив выражению значение). Например, если `p` является указателем на `int`-значение, то выражение `p[3]` представляет собой `int`-значение в блоке, смещенном на `3` `int`-блока по отношению к ячейке с адресом из указателя `p`.

Перечисленные выше правила имеют простые и очевидные последствия. Например, если указатели (одного типа) ссылаются на соседние блоки, то разность этих указателей равна `1` или `-1`, в зависимости от того, как относительно друг друга расположены блоки. При этом разность адресов, записанных в указатели, отличается на значение, равное объему (в байтах) блока памяти, выделяемого для записи значения данного типа. Далее, если `p` — некоторый указатель, то `p[0]` — значение, записанное в блоке, на который ссылается `p` (то есть то же, что и `*p`). Значениями выражений `p+1` и `p-1` являются адреса блоков, соседних с блоком, на который ссылается указатель `p`. Если `k` — целое число, то выражение `p[k]` эквивалентно выражению `*(p+k)`, и так далее (таких интересных соотношений довольно много).

Для большей наглядности рассмотрим небольшой пример, в котором используются правила адресной арифметики (а если более конкретно, то индексируются указатели). Обратимся к программе в листинге 4.2.

Листинг 4.2. Индексирование указателей

```
using System;
class MiniArrayDemo{
   // Главный метод:
   unsafe static void Main(){
      // Объявление числовой переменной:
      double miniarray;
      // Размер "byte-массива":
      int m=sizeof(double)/sizeof(byte);
```

```
// Указатель неопределенного типа:
void* pnt;
// Значение указателя:
pnt=&miniarray;
// Указатель на byte-значение:
byte* p;
// Значение указателя:
p=(byte*)pnt;
// Перебор блоков памяти с помощью указателя:
for(int k=0;k<m;k++){
   // В блок памяти записывается значение:
   p[k]=(byte)(k+1);
   // Отображение значения из блока памяти:
   Console.Write("|"+p[k]);
}
Console.WriteLine("|");
// Новое значение указателя:
p=(byte*)pnt+m-1;
// Перебор блоков памяти с помощью указателя:
for(int k=0;k<m;k++){
   // Использован отрицательный индекс:
   Console.Write("|"+p[-k]);
}
Console.WriteLine("|");
// Размер "char-массива":
int n=sizeof(double)/sizeof(char);
// Указатель на char-значение:
char* q;
// Значение указателя:
q=(char*)pnt;
// Перебор блоков памяти с помощью указателя:
for(int k=0;k<n;k++){
```

```
            // В блок памяти записывается значение:
            q[k]=(char)('A'+k);
            // Отображение значения из блока памяти:
            Console.Write("|"+q[k]);
         }
         Console.WriteLine("|");
         // Новое значение указателя:
         q=(char*)pnt+n-1;
         // Перебор блоков памяти с помощью указателя:
         for(int k=0;k<n;k++){
            // Использован отрицательный индекс:
            Console.Write("|"+q[-k]);
         }
         Console.WriteLine("|");
      }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 4.2)

```
|1|2|3|4|5|6|7|8|
|8|7|6|5|4|3|2|1|
|A|B|C|D|
|D|C|B|A|
```

В программе реализована очень простая идея. Объявляется переменная типа `double`. Под такую переменную выделяется 8 байтов. Сначала каждый такой байт интерпретируется как переменная типа `byte` (занимает 1 байт памяти). Получается 8 однобайтовых блоков, в каждый из которых записывается значение. Индексируя указатели, мы можем получать доступ к этим блокам как к элементам массива.

Эту же область памяти размером в 8 байтов, выделенную под `double`-переменную, можно интерпретировать как 4 блока по 2 байта. В каждый из этих 4 блоков можно записать `char`-значение (под переменную типа

char выделяется 2 байта). Получая доступ к char-значениям путем индексирования указателя, мы создаем иллюзию массива из 4 символьных элементов. Таким образом, одну и ту же область памяти можем использовать как byte-массив из 8 элементов или как char-массив из 4 элементов.

Если более детально, то в программе объявляется переменная miniarray типа double. Значение целочисленной переменной m вычисляется выражением sizeof(double)/sizeof(byte). Это отношение объема памяти (в байтах), выделяемой для значений типа double (значение 8), и объема памяти, выделяемой для значений типа byte (значение 1). Также командой void* pnt мы объявляем указатель неопределенного типа. Операции адресной арифметики к такому указателю неприменимы. Фактически все, что можно с таким указателем сделать, записать в него адрес. Именно это мы делаем с помощью команды pnt=&miniarray, которой указателю pnt в качестве значения присваивается адрес переменной miniarray.

ПОДРОБНОСТИ

Область памяти, выделенной под переменную miniarray, состоит из 8 блоков, размер каждого блока равен 1 байту. Адрес переменной miniarray — это адрес первого из 8 блоков. Именно адрес первого блока записывается в указатель pnt.

Командой byte* p объявляется указатель p на значение типа byte. Значение указателю присваиваем командой p=(byte*)pnt. В этом случае, во-первых, адрес из указателя pnt копируется в указатель p. Во-вторых, пришлось использовать явное приведение типа. Без этого не обойтись, поскольку pnt является указателем на значение неопределенного типа, а указатель p объявлен для работы с byte-значениями. В итоге указатель p содержит адрес первого из 8 блоков, выделенных для переменной miniarray.

НА ЗАМЕТКУ

Хотя указатели pnt и p содержат один и тот же адрес, между ними есть принципиальная разница. Поскольку тип данных, на которые может ссылаться указатель p, определен, то к указателю p могут применяться операции адресной арифметики.

Для записи значений в однобайтовые блоки запускаем оператор цикла, в котором индексная переменная k пробегает значения от 0 до m-1 включительно (то есть перебираются все однобайтовые блоки). За каждый цикл при заданном индексе k сначала выполняется команда p[k]=(byte)(k+1), которой в однобайтовый блок записывается значение, а затем это значение отображается в консольном окне с помощью команды Console.Write("|"+p[k]). В обеих командах использована инструкция вида p[k], в которой индексируется указатель p. Выражение p[k] представляет собой значение, записанное в блок памяти, смещенный на k позиций по отношению к блоку памяти, на который ссылается указатель p. Поскольку указатель p объявлен для работы со значениями типа byte, а под значение этого типа выделяется 1 байт, то при смещениях используются однобайтовые блоки.

ПОДРОБНОСТИ

В команде p[k]=(byte)(k+1) использовано явное приведение к типу byte. Необходимость явного приведения типа обусловлена тем, что сумма k+1 представляет собой значение типа int, а выражение p[k] подразумевает работу со значениями типа byte.

При выполнении команды p=(byte*)pnt+m-1 указателю p присваивается новое значение. Это адрес последнего блока в области памяти, выделенной для переменной miniarray типа double. В соответствии с использованной командой значение указателя pnt приводится к типу byte*, а к полученному результату прибавляется число m-1 (точнее, сначала прибавляется значение переменной m, а затем вычитается число 1). Приведение к типу byte* необходимо для того, чтобы с указателем (адресом) можно было выполнять арифметические операции (в том числе и прибавлять к указателю числа). Значением выражения (byte*)pnt+m-1 является адрес блока памяти, смещенного на m-1 позиций по отношению к блоку, на который ссылается указатель pnt. Поскольку в приведенной команде мы указатель pnt приводили к типу byte*, а под значение типа byte выделяется 1 байт памяти, то смещение на одну позицию означает смещение на один однобайтовый блок. Таким образом, результат выражения (byte*)pnt+m-1 — это адрес однобайтового блока, смещенного по отношению к первому блоку (его адрес записан в указатель pnt) на m-1 позиций. Это последний блок в области памяти, выделенной под переменную miniarray типа double.

ПОДРОБНОСТИ

Точнее, значение выражения `(byte*)pnt+m-1` вычисляется следующим образом: от блока, на который ссылается указатель `pnt`, выполняется `m` смещений с увеличением адреса, а потом `1` смещение с уменьшением адреса, и адрес полученного блока возвращается в качестве результата. Можно было воспользоваться командой `(byte*)pnt+(m-1)`. В таком случае сразу выполнялось бы `m-1` смещений с увеличением адреса.

После присваивания нового значения указателю `p` еще раз выполняется оператор цикла, в котором диапазон изменения индексной переменной `k` такой же, как и в предыдущем случае. За каждый цикл при заданном значении индекса `k` отображается значение выражения `p[-k]`. Особенность данного выражения в том, что индекс отрицательный. Поскольку речь идет об индексировании указателя, то отрицательный индекс означает, что смещение выполняется уменьшением адреса. Другими словами, значение выражения `p[-k]` — это значение в блоке памяти, который смещен на `k` позиций (так, что адрес уменьшается) по отношению к блоку памяти, на который ссылается указатель `p`. Поскольку `p` ссылается на последний блок области памяти, выделенной для переменной `miniarray`, то проиндексированный отрицательным индексом указатель дает значение соответствующего предыдущего блока. То есть выражение `p[0]` возвращает значение последнего блока, выражение `p[-1]` возвращает значение предпоследнего блока, выражение `p[-2]` позволяет узнать значение блока перед предпоследним, и так далее. В итоге получается, что в консоли отображаются значения, записанные в однобайтовые блоки, но отображаются они в обратном порядке (от последнего блока к начальному).

Значение целочисленной переменной `n` вычисляется на основе выражения `sizeof(double)/sizeof(char)` как отношение объема памяти, выделяемой для `double`-значений (8 байтов), к объему памяти, выделяемой для `char`-значений (2 байта). В итоге переменная `n` получает значение `4`. В данном случае мы интерпретируем память, выделенную под переменную `miniarray`, как 4 последовательно размещенных блока, каждый по 2 байта. Для получения доступа к этим двухбайтовым блокам объявляется указатель `q` на значение типа `char`. Значение указателю `q` присваивается командой `q=(char*)pnt`. Это адрес первого блока в области памяти, выделенной под переменную `miniarray`. Но, в отличие от случая с указателем `p`, здесь указатель `q` предназначен

для работы с символьными значениями. Поэтому базовые блоки, на которые выполняется смещение, двухбайтовые (то есть состоят из двух однобайтовых блоков), а значения в этих двухбайтовых блоках интерпретируются как символьные. Сначала в операторе цикла мы заносим с помощью команды `q[k]=(char)('A'+k)` в эти блоки символы, начиная с `'A'` (и заканчивая `'D'`), и отображаем их в консольном окне. А затем «перебрасываем» указатель `q` на последний двухбайтовый блок (команда `q=(char*)pnt+n-1`) и с помощью еще одного оператора цикла, индексируя указатель `q` отрицательными индексами (выражение `q[-k]`), отображаем символьные значения двухбайтовых блоков в обратном порядке.

ПОДРОБНОСТИ

Результатом выражения `(char*)pnt+n-1` является адрес последнего двухбайтового блока в области памяти, выделенной под переменную `miniarray`. Указатель `pnt` приводится к типу `char*`, поэтому все смещения при выполнении операций адресной арифметики выполняются на уровне блоков размером в 2 байта (такого размера блок выделяется под значение типа `char`). Значение выражения `(char*)pnt+n-1` — это адрес блока, смещенного `n-1` раз (с увеличением адреса) по отношению к блоку, на который ссылается указатель `pnt`. Смещение на одну позицию означает смещение на два однобайтовых блока.

Еще один пример, в котором иллюстрируются операции с указателями, представлен в листинге 4.3. В каком-то смысле он напоминает предыдущую программу (из листинга 4.2), но реализовано все немного иначе.

Листинг 4.3. Операции с указателями

```
using System;
// Класс с главным методом:
class OperationsWithPointersDemo{
   // Главный метод:
   unsafe static void Main(){
      // Объявление переменной типа double:
      double val;
      // Целочисленная индексная переменная:
      int k=1;
```

```
// Указатели на значения типа double:
double* start,end;
// Значения указателей:
start=&val;
end=start+1;
// Отображение адресов:
Console.WriteLine("Адрес start\t{0}",(uint)start);
Console.WriteLine("Адрес end\t{0}",(uint)end);
// Разность адресов:
Console.WriteLine("Разность адресов {0,6}",(uint)end-(uint)start);
// Разность указателей:
Console.WriteLine("Разность double-указателей\t{0}",end-start);
Console.WriteLine("Разность int-указателей \t{0}",(int*)end-(int*)start);
Console.WriteLine("Разность char-указателей\t{0}",(char*)end-(char*)start);
Console.WriteLine("Разность byte-указателей\t{0}",(byte*)end-(byte*)start);
// Указатель на значение типа byte:
byte* p=(byte*)start;
// Указатель на значение типа char:
char* q=(char*)start;
// Указатель на значение типа int:
int* r=(int*)start;
Console.WriteLine("Тип byte:");
Console.WriteLine("Адрес\tЗначение");
// Заполнение блоков памяти значениями и отображение
// значений из блоков памяти:
while(p<end){
   // Значение записывается в блок памяти:
   *p=(byte)k;
   // Отображение адреса и значения из блока памяти:
   Console.WriteLine("{0}\t{1}",(uint)p,*p);
   // Увеличение значения указателя:
   p++;
```

```
            // Новое значение переменной:
            k+=2;
        }
        Console.WriteLine("Тип char:");
        Console.WriteLine("Адрес\t3начение");
        // Заполнение блоков памяти значениями и отображение
        // значений из блоков памяти:
        for(k=0;q+k<end;k++){
            // Значение записывается в блок памяти:
            *(q+k)=(char)('A'+2*k);
            // Отображение адреса и значения из блока памяти:
            Console.WriteLine("{0}\t{1}",(uint)(q+k),*(q+k));
        }
        Console.WriteLine("Тип int:");
        Console.WriteLine("Адрес\t3начение");
        // Заполнение блоков памяти значениями и отображение
        // значений из блоков памяти:
        for(k=0;&r[k]<end;k++){
            // Значение записывается в блок памяти:
            r[k]=5*(k+1);
            // Отображение адреса и значения из блока памяти:
            Console.WriteLine("{0}\t{1}",(uint)&r[k],r[k]);
        }
    }
}
```

Результат выполнения программы может быть таким, как показано ниже:

Результат выполнения программы (из листинга 4.3)

```
Адрес start     1830908
Адрес end       1830916
Разность адресов      8
```

```
Разность double-указателей      1
Разность int-указателей         2
Разность char-указателей        4
Разность byte-указателей        8
Тип byte:
Адрес   Значение
1830908 1
1830909 3
1830910 5
1830911 7
1830912 9
1830913 11
1830914 13
1830915 15
Тип char:
Адрес   Значение
1830908 A
1830910 C
1830912 E
1830914 G
Тип int:
Адрес   Значение
1830908 5
1830912 10
```

В этой программе мы объявляем переменную `val` типа `double`, а также указатели `start` и `end`, предназначенные для работы со значениями типа `double`. Командой `start=&val` указатель `start` получает в качестве значения адрес переменной `val`. Значение указателю `end` присваивается командой `end=start+1`, в результате чего в указатель `end` записывается адрес блока, расположенного сразу за областью памяти, выделенной под переменную `val`.

ПОДРОБНОСТИ

Область памяти, выделенная под переменную `val`, состоит из 8 однобайтовых блоков. В указатель `start` записывается адрес первого из этих 8 блоков. При прибавлении числа `1` к указателю `start` (в команде `end=start+1`) выполняется смещение на один блок размера, равного объему памяти, выделяемой под `double`-значение (8 байтов). В результате в указатель `end` записывается адрес блока, расположенного за 8 однобайтовыми блоками, выделенными под переменную `val`.

Значения адресов, которые вычисляются выражениями `(uint)start` и `(uint)end`, отображаются в консольном окне. В принципе, сами адреса от запуска к запуску меняются. Но разница между значениями адресов (вычисляется как разность `(uint)end-(uint)start`) всегда одна и та же и равна `8`. Причина в том, что адрес приписывается каждому однобайтовому блоку. Адреса соседних однобайтовых блоков отличаются на 1. Под значение типа `double` выделяется 8 однобайтовых блоков, отсюда и результат. Но вот если мы вычислим разность указателей `end-start`, то получим значение `1`. Указатели `end` и `start` объявлены для работы с `double`-значениями. Разность этих указателей — это количество значений типа `double`, которые можно записать между соответствующими адресами. В данном случае между адресами (с учетом начального блока, на который ссылается указатель `start`) находится 8 однобайтовых значений. В эту область можно записать одно значение типа `double`.

При вычислении выражения `(int*)end-(int*)start` мы получаем значение `2`. Здесь, как и при вычислении выражения `end-start`, вычисляется разность указателей. Но эти указатели предварительно приведены к типу `int*`. То есть мы имеем дело с указателями, предназначенными для работы с целочисленными значениями типа `int`. Поэтому значением выражения `(int*)end-(int*)start` является целое число, равное количеству ячеек для значений типа `int`, которые помещаются в области памяти между адресами из указателей `end` и `start`. Как мы уже знаем, там 8 однобайтовых ячеек, а для записи значения типа `int` нужно 4 байта. Получается, что в данную область памяти можно записать два значения типа `int`.

Аналогично при вычислении выражения `(char*)end-(char*)start` в результате получаем число `4`. Это количество `char`-блоков,

которые помещаются в области памяти, выделенной под переменную val. Ну и несложно догадаться, почему результатом выражения (byte*)end-(byte*)start является число 8.

Командами byte* p=(byte*)start, char* q=(char*)start и int* r=(int*)start объявляются указатели p, q и r на значения типа byte, char и int соответственно. В каждый из указателей копируется адрес, записанный в указатель start. Поскольку указатели разного типа, то используется явное приведение типов. Далее действуем так: с помощью каждого из указателей заполняем значениями область памяти, выделенную под переменную val, и отображаем эти значения в консольном окне. Технология решения задачи каждый раз разная.

Для заполнения области памяти byte-значениями запускаем оператор цикла while, в котором проверяется условие p<end (адрес в указателе p меньше адреса в указателе end). За каждый цикл командой *p=(byte)k в однобайтовую ячейку, на которую ссылается указатель p, записывается значение целочисленной переменной k (начальное значение 1), а затем отображается адрес ячейки (инструкция (uint)p) и значение в этой ячейке (инструкция *p). Командой p++ увеличивается значение указателя (в результате указатель будет указывать на соседнюю ячейку), а командой k+=2 текущее значение переменной k увеличивается на 2.

В следующем операторе цикла for переменная k принимает начальное значение 0. За каждый цикл значение переменной k увеличивается на 1. Оператор выполняется, пока истинно условие q+k<end. Значение выражения q+k — это указатель на блок памяти (предназначенный для записи char-значения), смещенный на k позиций по отношению к блоку, на который ссылается указатель q. Условие состоит в том, что адрес этого блока должен быть меньше адреса из указателя end.

В теле оператора цикла командой *(q+k)=(char)('A'+2*k) в блок памяти, на который ссылается указатель q+k, записывается символьное значение, которое получается прибавлением к коду символа 'A' значения 2*k с последующим преобразованием к типу char (получаем буквы через одну, начиная с буквы 'A'). Также отображается адрес блока (инструкция (uint)(q+k)) и значение в этом блоке (инструкция *(q+k)).

В еще одном операторе цикла for проверяется условие &r[k]<end. Значение выражения r[k] — это значение в блоке, смещенном на k позиций по отношению к блоку, на который ссылается указатель r. В данном случае смещение выполняется на блоки, в которые можно записать

int-значения. Выражение &r[k] — это адрес блока, в который записано значение r[k]. Этот адрес должен быть меньше адреса из указателя end.

В теле оператора цикла командой r[k]=5*(k+1) в соответствующий блок записывается значение (числа 5 и 10, а больше не помещается). Адрес блока (инструкция (uint)&r[k]) и значение из блока (инструкция r[k]) отображаются в консольном окне.

НА ЗАМЕТКУ

При отображении адресов, если используется указатель на значение типа byte, дискретность изменения адреса равна 1. При отображении адресов с помощью указателей на значения типа char дискретность изменения адреса равна 2. При использовании указателя на значение типа int дискретность изменения адреса равна 4. Фактически дискретность изменения адреса определяется количеством однобайтовых блоков, используемых для записи значения соответствующего типа.

Указатели на экземпляр структуры

> Тогда помните: для вас законы не писаны. Действуйте только по моим инструкциям.
>
> *из м/ф «Приключения капитана Врунгеля»*

Если структура не содержит членов ссылочного типа, то для экземпляра такой структуры можно создать указатель. При объявлении указателя на экземпляр структуры тип указателя описывают как имя структуры, после которого стоит звездочка *. Например, если имеется структура MyStruct (без членов ссылочного типа) и мы хотим объявить указатель pnt на экземпляр такой структуры, то объявление указателя может выглядеть следующим образом:

```
MyStruct* pnt;
```

Указателю на экземпляр структуры в качестве значения присваивается адрес экземпляра структуры. Адрес экземпляра можно получить, разместив перед именем экземпляра инструкцию &. Чтобы по указателю получить доступ к экземпляру структуры, можем перед указателем поставить звездочку *. Но поскольку в экземпляре структуры нас обычно

интересуют поля и методы, удобнее воспользоваться оператором -> (стрелка). Он используется в формате `указатель->поле` или `указатель->метод(аргументы)`. Небольшой пример, в котором использованы указатели на структуры, представлен в листинге 4.4.

Листинг 4.4. Указатели на экземпляр структуры

```
using System;
// Описание структуры:
struct MyStruct{
   // Поле:
   public int code;
   // Метод:
   public void show(){
      // Отображение значения поля:
      Console.WriteLine("Поле code: "+code);
   }
}
// Класс с главным методом:
class StructsAndPointersDemo{
   // Главный метод:
   unsafe static void Main(){
      // Создаются экземпляры структуры:
      MyStruct A,B;
      // Объявляется указатель на экземпляр структуры:
      MyStruct* p;
      // Указателю присваивается значение:
      p=&A;
      // Обращение к полю через указатель:
      p->code=123;
      // Вызов метода через указатель:
      p->show();
      // Вызов метода через экземпляр структуры:
      A.show();
```

```
      Console.WriteLine("--------------");
      // Новое значение указателя:
      p=&B;
      // Обращение к полю через указатель:
      p->code=321;
      // Вызов метода через указатель:
      p->show();
      // Вызов метода через экземпляр структуры:
      B.show();
      Console.WriteLine("--------------");
      // Еще один способ обратиться к полю:
      (*p).code=456;
      // Еще один способ вызвать метод:
      (*p).show();
      // Вызов метода через экземпляр структуры:
      B.show();
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 4.4)

```
Поле code: 123
Поле code: 123
--------------
Поле code: 321
Поле code: 321
--------------
Поле code: 456
Поле code: 456
```

В программе описана структура `MyStruct`, в которой есть открытое целочисленное поле `code` и открытый метод `show()`. Метод при вызове

отображает в консольном окне значение поля code экземпляра структуры, из которого вызывается.

В главном методе программы командой MyStruct A,B создаются экземпляры A и B структуры MyStruct. Командой MyStruct* p объявляется указатель p на экземпляр структуры MyStruct. Значение указателю присваивается командой p=&A. После ее выполнения указатель p ссылается на экземпляр A. Поэтому инструкция p->code является ссылкой на поле code экземпляра A. Например, командой p->code=123 полю code экземпляра A присваивается значение 123. Командой p->show() из экземпляра A вызывается метод show(). Для проверки этот же метод вызывается командой A.show(). Результаты обоих вызовов совпадают, как и должно быть.

После выполнения команды p=&B указатель p ссылается на экземпляр B. Поэтому при выполнении команды p->code=321 значение присваивается полю code экземпляра B. Соответственно, результат выполнения команды p->show() такой же, как и команды B.show(), поскольку в обоих случаях вызывается метод show() экземпляра B.

Стоит также заметить, что доступ к экземпляру можно получить с помощью инструкции *p. Например, при выполнении инструкции (*p).code=456 (круглые скобки нужны для изменения порядка применения операторов) присваивается значение полю code экземпляра B. Метод show() из этого экземпляра можно вызвать с помощью инструкции (*p).show().

Инструкция fixed

> Форму будете создавать под моим личным контролем.
>
> *из к/ф «Чародеи»*

Представим себе такую ситуацию. В программе создан объект класса, и у этого объекта есть поле (например, целочисленное). Мы не можем создать указатель для объекта, но можем создать указатель для поля, поскольку фактически это некоторая переменная. Но при этом мы сталкиваемся с потенциальной проблемой. Дело в том, что если в какой-то момент окажется, что в программе ссылок на объект больше нет (а это вполне реальная ситуация), то объект из памяти может быть удален. И тогда указатель будет

ссылаться на область памяти, которая больше не используется для хранения значения поля. Такие ситуации считаются (и являются) потенциально опасными. Они блокируются: просто так указатель на поле объекта не создать, приходится использовать «специальный подход».

НА ЗАМЕТКУ

То, что на поле объекта ссылается указатель, само по себе не защищает объект от удаления. В расчет принимаются только объектные переменные, которые ссылаются на объект. Если таких ссылок нет, объект помещается в очередь на удаление.

Для предотвращения удаления (системой сборки мусора) переменных (например, поля объекта), на которые ссылаются указатели, используется инструкция `fixed`. Шаблон использования этой инструкции представлен ниже (жирным шрифтом выделены ключевые элементы шаблона):

```
fixed(тип* указатель=&переменная){
   // Команды
}
```

После ключевого слова `fixed` в круглых скобках объявляется указатель, которому присваивается значение (адрес переменной или поля объекта). Команды, выполняемые с привлечением данного указателя, размещаются в отдельном блоке, выделенном фигурными скобками. Указатель, объявленный в `fixed`-блоке, доступен только в пределах этого блока. Объект, на поле которого ссылается указатель, не будет удален из памяти до тех пор, пока выполняются команды в `fixed`-блоке. Программа, в которой иллюстрируется использование инструкции `fixed`, представлена в листинге 4.5.

Листинг 4.5. Использование инструкции fixed

```
using System;
// Класс с полем:
class MyClass{
   // Поле:
   public int number;
   // Конструктор:
   public MyClass(int n){
      number=n;
```

```
   }
   // Деструктор:
   ~MyClass(){
      Console.WriteLine("Удален объект с полем "+number);
   }
}
// Главный класс программы:
class FixedDemo{
   // Главный метод:
   unsafe static void Main(){
      // Использован fixed-блок:
      fixed(int* p=&new MyClass(123).number){
         // Отображение значения поля:
         Console.WriteLine("Значение поля number: "+*p);
         // Полю присваивается новое значение:
         *p=321;
         // Отображение сообщения:
         Console.WriteLine("Завершение fixed-блока");
      }
   }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 4.5)

```
Значение поля number: 123
Завершение fixed-блока
Удален объект с полем 321
```

В программе описывается класс `MyClass`, у которого есть открытое целочисленное поле `number`, конструктор с одним аргументом (определяет значение поля), а также деструктор, который при удалении объекта из памяти выводит в консоль соответствующее сообщение со значением поля `number` удаляемого объекта.

Главный метод состоит из fixed-блока, в котором объявлен указатель p на целочисленное значение, и этому указателю в качестве значения присвоено выражение &new MyClass(123).number. Проанализируем его. Инструкция new MyClass(123) создает объект класса MyClass, и поле number этого объекта равно 123. Значением инструкции является ссылка на созданный объект. Эта ссылка никуда не записывается, и поэтому объект является анонимным. Сразу после создания он попадает в очередь на удаление. Точнее, должен был бы попасть. Инструкцией new MyClass(123).number мы обращаемся к полю number этого объекта, а символ & перед всем выражением означает, что нас интересует адрес этого поля. Таким образом, значением выражения &new MyClass(123).number является адрес поля number созданного анонимного объекта, и этот адрес записывается в указатель p. В итоге объект, хотя он и анонимный, гарантированно не удаляется из памяти, пока выполняется fixed-блок.

Значение поля объекта проверяем с использованием выражения *p. Командой *p=321 полю объекта присваивается новое значение. После этого завершается выполнение fixed-блока (об этом выводится сообщение) и завершается выполнение главного метода. Но перед этим созданный объект удаляется из памяти. Для данного объекта вызывается деструктор, и в консольном окне появляется сообщение со значением поля удаляемого объекта. Убеждаемся, что речь идет о том объекте, на поле number которого ссылался указатель p.

Указатели и массивы

Живьем брать демонов!

из к/ф «Иван Васильевич меняет профессию»

Указатели в языке C# тесно связаны с массивами. Точнее, при работе с массивами весьма эффектно могут использоваться указатели. Чтобы понять, почему такое возможно, необходимо учесть несколько обстоятельств. А именно:

- Имя массива является указателем на первый элемент массива.
- В памяти элементы массива расположены подряд, один за другим.
- Указатель на массив определяется в блоке fixed.

Зная имя массива, мы можем получить доступ к области памяти, в которую записан первый (начальный) элемент. Поскольку элементы массива расположены подряд, то, индексируя указатель (или выполняя иные операции адресной арифметики), мы можем получить доступ к любому элементу массива. Правда автоматического контроля на предмет выхода за пределы массива уже не будет. Инструкция `fixed` в данном случае нужна для того, чтобы гарантировать неизменность места размещения массива в памяти (это позволяет избежать перемещения массива при оптимизации использования ресурсов). Небольшой пример, в котором совместно используются указатель и массив, представлен в листинге 4.6.

Листинг 4.6. Указатели и массивы

```
using System;
// Класс с главным методом:
class ArrayAndPointerDemo{
   // Главный метод:
   unsafe static void Main(){
      // Размер массива:
      int size=10;
      // Объявление указателя:
      int* q;
      // Создание массива:
      int[] nums=new int[size];
      Console.WriteLine("Заполнение массива:");
      // Используем fixed-блок:
      fixed(int* p=nums){
         // Указателю присваивается значение:
         q=p+size-1;
         // Перебор элементов массива:
         for(int k=0;k<nums.Length;k++){
            // Элементу массива присваивается значение:
            p[k]=k+1;
            // Отображение значения элемента массива:
            Console.Write("|"+nums[k]);
```

```
            }
            Console.WriteLine("|");
            // Перебор элементов массива с помощью указателя:
            while(p<=q){
               // Возведение значения элемента в квадрат:
               (*q)*=(*q);
               // Изменение значения указателя:
               q--;
            }
         }
         // Отображение содержимого массива:
         Console.WriteLine("Массив после изменения:");
         for(int k=0;k<nums.Length;k++){
            // Отображение значения элемента:
            Console.Write("|"+nums[k]);
         }
         Console.WriteLine("|");
      }
}
```

Как выглядит результат выполнения программы, показано ниже:

Результат выполнения программы (из листинга 4.6)

```
Заполнение массива:
|1|2|3|4|5|6|7|8|9|10|
Массив после изменения:
|1|4|9|16|25|36|49|64|81|100|
```

В этой программе мы создаем целочисленный массив `nums`. Размер массива определяется переменной `size` (значение `10`). Также мы объявляем указатель `q` на целочисленное значение. Затем используется `fixed`-блок, в котором объявляется указатель `p` на целое число, а значением указателю присваивается имя массива `nums`. Поскольку имя массива

является указателем на начальный элемент массива, то указатель p ссылается на этот же начальный элемент. Командой q=p+size-1 присваивается значение указателю q. Учитывая, что переменная size определяет размер массива, а указатель p содержит ссылку на начальный элемент массива, несложно сообразить, что указатель q будет ссылаться на последний элемент в массиве.

Для заполнения массива использован оператор цикла for, в котором индексная переменная k последовательно принимает значения индексов элементов массива nums. При этом значения элементам массива присваиваются командой p[k]=k+1. Поскольку указатель p, как отмечалось выше, ссылается на начальный элемент массива, то из правила индексирования указателей следует, что выражение вида p[k] аналогично выражению nums[k]. Убеждаемся в этом, отображая в консольном окне значения элементов массива.

Еще раз элементы массива перебираются с помощью оператора цикла while. Оператор цикла выполняется, пока истинно условие p<=q (адрес в указателе p не превышает адрес в указателе q).

> **НА ЗАМЕТКУ**
>
> Напомним, что в начальный момент указатель p ссылается на начальный элемент массива, а указатель q ссылается на последний элемент массива.

За каждый цикл командой (*q)*=(*q) текущее значение в элементе, на который ссылается указатель q, умножается на себя же (то есть значение элемента возводится в квадрат), после чего командой q-- указатель q перебрасывается на предыдущий элемент в массиве. Проверка показывает, что после выполнения оператора цикла все элементы массива действительно возведены в квадрат.

Указатели и текст

Зина, подскажи мне что-нибудь по-славянски!
из к/ф «Иван Васильевич меняет профессию»

Как мы знаем, текстовые значения в языке C# реализуются в виде объектов класса String. Стандартная схема такая: имеется объектная переменная класса String, которая ссылается на объект этого же класса,

а объект, в свою очередь, содержит текстовое значение. И, как мы помним, ранее утверждалось, что текстовые объекты неизменны: нельзя поменять текст в текстовом объекте, а можно только создать новый объект и ссылку на этот новый объект записать в объектную переменную. Создается иллюзия, что текст изменился.

Реальность немного сложнее. Технически текст реализуется в виде символьного массива. Эта «традиция» уходит корнями в языки C и C++. В этих языках в конце такого символьного массива размещается нуль-символ `'\0'`, который служит индикатором окончания текста. Данный символ (не следует его путать с символьным изображением цифры ноль — это не одно и то же) имеет нулевой код и добавляется в конец символьного массива автоматически. Но это, еще раз подчеркнем, в языках C и C++. В языке C# упомянутый выше символьный массив «спрятан» в текстовом объекте, на который ссылается переменная класса `String`. И хотя в целях совместимости обычно в конце символьного массива нуль-символ `'\0'` добавляется, никакой особой роли в C# он не играет. Размер текста запоминается отдельно, а нуль-символ обрабатывается, как и все прочие символы.

ПОДРОБНОСТИ

Резюмируем. Текстовая строка в языке C# реализуется как объект. В этом объекте «спрятан» символьный массив. Массив содержит символы из текста. Обычно в конце символьного массива добавляется нуль-символ, чтобы обеспечить корректную обработку текста в случае совместного использования разных технологий. Но этот нуль-символ не является частью текстовой строки. Также нуль-символ не является индикатором окончания текста. Размер текстовой строки запоминается в виде целого числа, которое тоже «спрятано» в текстовом объекте.

Используя указатели, мы можем «проникнуть» в текстовый объект и получить прямой доступ к элементам символьного массива, который фактически содержит текст. С помощью указателей мы можем не только прочитать отдельные символы в массиве, но также и изменить их непосредственно в текстовом объекте. Пример, в котором иллюстрируется данный подход, представлен в листинге 4.7.

Листинг 4.7. Указатели и текст

```
using System;
// Класс с главным методом:
class StringAndPointerDemo{
```

```
   // Главный метод:
   unsafe static void Main(){
      // Текстовая строка:
      String txt="Программируем на C#";
      // Отображение текста:
      Console.WriteLine(txt);
      // Указатель на начальный элемент строки:
      fixed(char* p=txt){
         // Перебор символов строки:
         for(int k=0;p[k]!='\0';k++){
            // Отображение символа:
            Console.Write("|"+p[k]);
            // Изменение значения символа в объекте:
            p[k]=(char)('A'+k);
         }
         Console.WriteLine("|");
      }
      // Отображение текста:
      Console.WriteLine(txt);
   }
}
```

Результат выполнения программы будет следующим:

Результат выполнения программы (из листинга 4.7)

```
Программируем на C#
|П|р|о|г|р|а|м|м|и|р|у|е|м| |н|а| |C|#|
ABCDEFGHIJKLMNOPQRS
```

Мы объявляем текстовую переменную `txt` со значением `"Программируем на C#"`. В `fixed`-блоке объявляется символьный указатель `p`, которому в качестве значения присваивается переменная `txt`. В результате в указатель `p` записывается адрес начального элемента символьного

массива, содержащего текст. Получается так: текстовая переменная `txt` содержит ссылку на объект, который содержит символьный массив с текстом (символы из текста являются элементами массива). Указатель `p` ссылается на начальный элемент в этом символьном массиве. Индексируя указатель в операторе цикла, последовательно получаем доступ к каждому элементу в символьном массиве. Оператор цикла выполняется, пока истинно условие `p[k]!='\0'`, то есть пока не будет прочитан нуль-символ. За каждую итерацию цикла сначала отображается текущее символьное значение `p[k]` элемента массива, а затем командой `p[k]=(char)('A'+k)` этому элементу присваивается новое значение. В результате в текстовом объекте, на который ссылается переменная `txt`, символы текста меняются на цепочку символов из алфавита, начиная с буквы `'A'`.

> **НА ЗАМЕТКУ**
>
> Выше мы исходили из того, что символьный массив, через который реализуется текст, завершается нуль-символом `'\0'`. Но это не самый надежный критерий проверки окончания текста. Более того, теоретически сам текст может содержать в себе нуль-символ. Поэтому в общем случае разумнее использовать свойство `Length`.

Многоуровневая адресация

> Передайте Зинаиде Михайловне, что Розалия Францевна говорила Анне Ивановне: «Капитолина Никифоровна дубленки предлагает».
>
> *из к/ф «Иван Васильевич меняет профессию»*

В языке C# мы можем создавать указатели на указатели. Другими словами, мы можем объявить указатель, значением которого является адрес другого указателя, в который записывается адрес обычной переменой. Допускается создание и более глубоких цепочек указателей, но на практике это используется не так часто.

Чтобы понять, как объявляется указатель на указатель, следует учесть, что при объявлении указателя с использованием идентификатора типа, на значение которого может ссылаться указатель, указывается звездочка `*`. Например, инструкция `int*` используется при объявлении указателя на переменную типа `int`. Тогда при объявлении указателя на указатель

на значение типа `int` используется идентификатор `int**`. Как иллюстрацию к использованию указателей на указатели рассмотрим программу в листинге 4.8.

Листинг 4.8. Многоуровневая адресация

```
using System;
// Главный класс:
class PointerToPointerDemo{
   // Главный метод:
   unsafe static void Main(){
      // Целочисленные переменные:
      int A,B;
      // Указатель на целочисленную переменную:
      int* p;
      // Указатель на указатель на целочисленную
      // переменную:
      int** q;
      // Значение указателя на указатель:
      q=&p;
      // Значение указателя:
      p=&A;
      // Переменной A присваивается значение
      // (через указатель на указатель):
      **q=123;
      // Проверка значения переменной A:
      Console.WriteLine(A);
      // Новое значение указателя:
      *q=&B;
      // Переменной B присваивается значение
      // (через указатель):
      *p=321;
      // Проверка значения переменной B:
      Console.WriteLine(B);
```

```
    }
}
```

Ниже показано, каким будет результат выполнения программы:

Результат выполнения программы (из листинга 4.8)

```
123
321
```

В программе мы объявляем переменные `A` и `B` типа `int`, указатель `p` на значение типа `int`, а командой `int** q` объявляется указатель `q`, который может ссылаться на указатель, который, в свою очередь, может ссылаться на переменную типа `int`.

Командой `q=&p` в указатель `q` записывается адрес переменной `p`, которая сама является указателем. Указателю `p` значение присваивается командой `p=&A`. Что мы получили? Имеются переменные (указатель ведь — это тоже переменная) `A`, `p` и `q`. Адрес переменной `A` записан в переменную `p`, а адрес переменной `p` записан в переменную `q`. Выражение `*q` — это значение, записанное по адресу, который хранится в `q`. В указателе `q` хранится адрес указателя `p`. Поэтому выражение `*q` является эквивалентом `p`. Далее, если `*q` — это то же, что и `p`, то `**q` — это то же, что и `*p`. А `*p` — это значение переменной `A`, поскольку указатель `p` ссылается на переменную `A`. Поэтому при выполнении команды `**q=123` переменной `A` присваивается значение `123`.

А вот когда выполняется команда `*q=&B`, то в указатель `p` записывается адрес переменной `B`. После этого выполнение команды `*p=321` приводит к тому, что переменной `B` присваивается значение `321`.

Массив указателей

> Минуточку! За чей счет этот банкет? Кто оплачивать будет?
>
> *из к/ф «Иван Васильевич меняет профессию»*

Указатели можно использовать практически так же, как и другие переменные. Нужно только помнить, что при использовании указателей

мы имеем дело с адресами. С другой стороны, использование указателей иногда позволяет привнести в программу некоторую пикантность. В листинге 4.9 представлена программа, в которой задействованы массивы указателей.

Листинг 4.9. Массив указателей

```
using System;
class ArrayOfPointersDemo{
   unsafe static void Main(){
      // Переменные:
      int x=100,y=200,z=300;
      // Массив указателей на целочисленные значения:
      int*[] nums=new int*[3];
      // Значение первого элемента массива:
      nums[0]=&x;
      // Значение второго элемента массива:
      nums[1]=&y;
      // Значение третьего элемента массива:
      nums[2]=&z;
      // Отображение значений переменных:
      Console.WriteLine("Числа: {0}, {1} и {2}",nums[0][0],*nums[1],nums[2][0]);
      // Массив указателей на символьные значения:
      char*[] symbs=new char*[3];
      // Значение первого элемента массива:
      symbs[0]=(char*)&x;
      // Значение второго элемента массива:
      symbs[1]=(char*)&y;
      // Значение третьего элемента массива:
      symbs[2]=(char*)&z;
      // Символьная переменная:
      char s='A';
      // Заполнение значениями области памяти,
      // выделенной под переменные:
```

```
      for(int i=0;i<symbs.Length;i++){
         for(int j=0;j<sizeof(int)/sizeof(char);j++){
            // В блок памяти записывается значение:
            symbs[i][j]=s;
            // Новое значение символьной переменной:
            s++;
            // Отображение значения из блока памяти:
            Console.Write(symbs[i][j]+" ");
         }
         Console.WriteLine();
      }
      // Проверка результата:
      Console.WriteLine("Числа: {0}, {1} и {2}",x,y,z);
      Console.WriteLine("Проверка: {0}, {1} и {2}",nums[0][0],*nums[1],nums[2][0]);
      Console.WriteLine("Еще раз: {0}, {1} и {2}",*(int*)symbs[0],
*(int*)symbs[1],*(int*)symbs[2]);
      Console.WriteLine("Для сравнения: {0}, {1} и {2}",*symbs[0],*symbs[1],
*symbs[2]);
   }
}
```

В результате выполнения программы в консольном окне отображаются следующие сообщения:

Результат выполнения программы (из листинга 4.9)

```
Числа: 100, 200 и 300
A B
C D
E F
Числа: 4325441, 4456515 и 4587589
Проверка: 4325441, 4456515 и 4587589
Еще раз: 4325441, 4456515 и 4587589
Для сравнения: A, C и E
```

Проанализируем программный код и результат его выполнения. Мы объявляем три целочисленные переменные x, y и z (со значениями 100, 200 и 300 соответственно). Массив из трех указателей на целочисленные значения создается командой int*[] nums=new int*[3]. Элементы массива nums являются указателями, и в качестве значений им можно присваивать адреса целочисленных переменных, что мы и делаем с помощью команд nums[0]=&x, nums[1]=&y и nums[2]=&z. Таким образом, nums[0] — это адрес переменной x, nums[1] — это адрес переменной y, а nums[2] — это адрес переменной z. С учетом правил адресной арифметики получаем, что nums[0][0] — это значение переменной x, *nums[1] — это значение переменной y, а nums[2][0] — это значение переменной z.

Командой char*[] symbs=new char*[3] создается массив из трех указателей на символьные значения. Значения элементам массива присваиваем командами symbs[0]=(char*)&x, symbs[1]=(char*)&y и symbs[2]=(char*)&z. Поскольку переменные x, y и z целочисленные, а элементы массива symbs являются указателями на символьные значения, то нам пришлось выполнить явное приведение типа. Что получается? Например, значением элемента symbs[0] является адрес переменной x. Но переменная записана в 4 байтах, а указатель symbs[0] получает доступ к блокам размером в 2 байта. Поэтому symbs[0][0] — это значение (интерпретируемое как символ) в первых двух однобайтовых блоках области памяти, выделенной под переменную x. А выражение symbs[0][1] — это символьное значение в следующих двух однобайтовых блоках области памяти, выделенной под переменную x. Это же касается других элементов массива symbs и переменных y и z. В итоге получается, что на основе трех блоков памяти (каждый по 4 байта), выделенных под переменные x, y и z, мы организовали видимость символьного массива из трех строк и двух столбцов. Этот импровизированный двумерный символьный массив заполняется с помощью вложенных операторов цикла, а соответствующие символьные значения отображаются в консольном окне.

ПОДРОБНОСТИ

Значение выражения symbs.Length — это количество элементов в массиве symbs (то есть 3). Выражение sizeof(int)/sizeof(char) определяет, сколько char-блоков помещается внутри int-блока (результат равен 2).

Область памяти, выделенная под импровизированный двумерный символьный массив, используется также при работе с массивом nums,

и в нее же записаны значения переменных x, y и z. Например, значение переменной x можно узнать по имени этой переменной с помощью инструкции nums[0][0] или с помощью выражения *(int*)symbs[0]. В последнем случае следует учесть, что symbs[0] есть указатель на первые два байта из области памяти, выделенной под переменную x. Выражение (int*)symbs[0] с явным приведением типа есть указатель на все четыре байта области, выделенной под переменную x. А выражение *(int*)symbs[0] — это значение в области памяти, выделенной под переменную x. Что касается значения переменной x, то в первые два байта этой переменной записывался код символа 'A' (это число 65), а в два других байта записан код символа 'B' (это число 66). Если бинарный код в четырех байтах интерпретировать как значение типа int, то с учетом того, что смещение на два однобайтовых блока означает умножение на 2^{16} = 65 536, то получаем 65 + 66 × 65 536 = 4 325 441. Ну а значение выражения *symbs[0] — это символ 'A', записанный в первые два байта области памяти, выделенной под переменную x.

НА ЗАМЕТКУ

Читателю предлагается самостоятельно объяснить результат вычисления прочих подобных выражений, использованных в программе.

Резюме

Какой дурак на Плюке правду думает? Абсурд!

из к/ф «Кин-дза-дза»

- Указатель представляет собой переменную, значением которой является адрес другой переменной. При объявлении указателя указывается тип переменной, на которую может ссылаться указатель, и звездочка *. Указатели объявляют для значений базовых типов.
- Получить адрес переменной можно, указав перед именем переменной символ &. Если перед именем указателя поставить звездочку *, то получим доступ к значению, записанному по адресу, хранящемуся в указателе.
- Правила адресной арифметики определяют, какие операции и как могут выполняться с указателями. В частности, указатели можно сравнивать, указатели можно вычитать, индексировать, а также можно прибавлять к указателю целое число и вычитать из указателя целое число.

- Если структура не содержит членов ссылочного типа, то для экземпляра структуры можно объявить указатель.
- Блок `fixed` используется в случае, если необходимо обеспечить безопасное использование области памяти, на которую ссылается указатель.
- Имя массива является указателем на начальный элемент массива. Элементы массива в памяти расположены подряд, один за другим.
- С помощью указателей можно получить доступ к символам текста в текстовом объекте. Методы использования указателей при работе с текстом фактически означают работу с символьным массивом.
- Можно объявить указатель, который ссылается на указатель. Это называется многоуровневой адресацией.
- Блок программного кода, в котором используются указатели, помечается ключевым словом `unsafe`. Соответствующее приложение также необходимо компилировать в режиме использования небезопасного кода.

Задания для самостоятельной работы

— Капу, капу жми!
— Нажал уже...

из к/ф «Кин-дза-дза»

1. Напишите программу, в которой объявляется переменная типа `int`. В первый и последний байты области памяти, выделенной под эту переменную, запишите число `1`, а в два внутренних байта запишите символ `'A'`.

2. Напишите программу, в которой объявляется переменная типа `double`. В область памяти, выделенную под эту переменную, запишите такие значения: в первый байт запишите значение `1`, в следующие два байта запишите символ `'A'`, в следующие четыре байта запишите значение `2` и в оставшийся восьмой байт запишите значение `3`.

3. Напишите программу, в которой объявляются три переменные типа `int`. Первые две переменные получают случайные значения. Область памяти, выделенная под третью переменную, заполняется следующим образом: первые два байта копируются из первой переменной, а следующие два байта копируются из второй переменной. Предложите способ проверки корректности вычислений.

4. Напишите программу, в которой объявляется переменная типа `int`. Переменной в качестве значения присваивается случайное число. Затем (с использованием указателей) выполняется циклический сдвиг байтов: первый байт становится вторым, второй байт становится третьим, третий байт становится четвертым, а четвертый байт становится первым. Предложите способ проверки правильности вычислений.

5. Напишите программу, содержащую структуру с символьным и целочисленным полями, а также метод, который при вызове отображает значения этих полей. Создайте указатель на экземпляр структуры и с помощью этого указателя присвойте значения полям экземпляра структуры и вызовите метод из экземпляра.

6. Напишите программу, в которой есть класс с целочисленным полем. На основе класса создайте объект. С помощью указателей запишите в первые два байта области памяти, выделенной под поле объекта, символ `'A'`, а в два следующие байта — символ `'B'`. Проверьте значение поля и объясните результат.

7. Напишите программу, в которой создается массив из двух элементов типа `int`. Заполните этот массив по байтам последовательностью натуральных чисел. Проверьте значения байтов, значения элементов целочисленного массива и объясните результат.

8. Напишите программу, в которой объявляется переменная типа `double` и одномерный массив, элементы которого являются указателями на значение типа `byte`. В этот массив записываются адреса однобайтовых блоков из области памяти, выделенной под `double`-переменную, но через один: адрес первого блока, третьего, пятого и так далее. Используя этот массив указателей путем их индексирования, следует заполнить все байты в области памяти, выделенной под `double`-переменную, последовательностью натуральных чисел и проверить результат.

9. Напишите программу, в которой создается текстовый объект, а потом с помощью указателей непосредственно в этом объекте выполняется инверсия текста: порядок символов заменяется на противоположный.

10. Напишите программу, в которой объявляется переменная типа `int`, а также указатель на указатель на значение типа `char`. С помощью этого указателя нужно записать в первые два байта в области памяти, выделенной под переменную типа `int`, символ `'A'`, а в следующие два байта — символ `'B'`. Проверьте значение целочисленной переменной и объясните результат.

Глава 5
ОБРАБОТКА ИСКЛЮЧЕНИЙ

> Товарищ эцилопп! Они у нас вещи украли! Поймайте их!
>
> *из к/ф «Кин-дза-дза»*

Эта глава посвящена обработке *исключительных ситуаций*. Мы узнаем, как создавать программные коды, устойчивые к возникновению непредвиденных ситуаций. Среди наших ближайших задач есть такие:

- знакомство с основными принципами обработки исключений;
- применение конструкции `try-catch` для перехвата исключений;
- знакомство с основными классами исключений;
- использование нескольких `catch`-блоков для перехвата исключений разных типов;
- использование вложенных `try`-блоков и блока `finally`;
- искусственное генерирование исключений;
- создание классов пользовательских исключений;
- применение инструкций `checked` и `unchecked`.

Как обычно, все рассматриваемые темы иллюстрируются специально подобранными примерами.

Принципы обработки исключений

> Здравствуйте! Мы — наши туристы, отстали от группы. Подбросьте нас до города, а там мы как-нибудь уже сами! Переводи.
>
> *из к/ф «Кин-дза-дза»*

Программы создаются для того, чтобы они работали эффективно, быстро и надежно. К сожалению, очень сложно обезопасить программный

код от возможных ошибок. Здесь имеются в виду не ошибки, связанные с некорректным синтаксисом или неправильно реализованным алгоритмом. Имеются в виду ошибки, которые возникают в процессе выполнения программы и появление которых потенциально невозможно или сложно предвидеть. Пример подобной ситуации представить легко. Допустим, при выполнении программы пользователь должен ввести целое число. Значение, которое вводит пользователь (в консольном окне или поле ввода), программа считывает и пытается интерпретировать как целое число. Если же пользователь ввел не целое число, а что-либо другое, возникает ошибка. Причем эта ошибка не связана с некорректным программным кодом. Она обусловлена действиями пользователя. Предусмотреть в программе, что именно введет пользователь, не получится. Поэтому каждый раз, когда такая программа запускается на выполнение, потенциально может возникнуть ошибка. Проблема в том, что если ошибка возникает, то программа завершает выполнение (и появляется сообщение об ошибке). Это очень неудобно. Подобных ситуаций пытаются избегать. И возможности для этого имеются.

НА ЗАМЕТКУ

Напомним, что очень краткое и поверхностное знакомство с обработкой исключений у нас состоялось ранее, в первой части книги, в главе 3.

Чтобы понять, как программа может реагировать на возникновение ошибки, имеет смысл хотя бы кратко узнать, что происходит, если случается ошибка. А происходит следующее. Если при выполнении некоторой команды возникает ошибка, то выполнение этой команды прекращается. Но не все так просто. Еще автоматически создается объект, который содержит информацию о возникшей ошибке. Саму ошибку называют *исключением* или исключительной ситуацией, а объект называется объектом исключения или объектом ошибки. Этот объект передается в программу для обработки. Иногда говорят, что объект исключения «вбрасывается» в программу. Далее возможны два варианта. Вариант первый: в программе предусмотрена обработка исключений. Если так, то исключение обрабатывается, и программа продолжает работу. Вариант второй: в программе не предусмотрена обработка исключительных ситуаций. В таком случае используется механизм обработки по умолчанию: выполнение программного кода прекращается, и выводится сообщение об ошибке.

НА ЗАМЕТКУ

Как отмечалось выше, при возникновении ошибки создается объект. Объекты, как мы знаем, создаются на основе классов. В языке C# имеется иерархия классов, каждый из которых соответствует ошибке определенного типа. При возникновении ошибки объект исключения создается на основе класса, который соответствует типу возникшей ошибки.

Итак, мы можем научить программу реагировать на возникновение ошибки. Главное преимущество в наличии системы перехвата и обработки исключений в том, что программа не завершает работу в аварийном режиме. Хотя, откровенно говоря, есть и другие позитивные моменты. Даже больше — иногда они настолько положительны, что приходится генерировать исключения специально. Такой способ программирования мы также обсудим.

Использование конструкции `try-catch`

— Они будут на карачках ползать, а мы на них плевать!
— А зачем?
— Как зачем? Удовольствие получать.

из к/ф «Кин-дза-дза»

Мы уже сталкивались с конструкцией `try-catch`. В самом простом варианте эта конструкция используется следующим образом. Программный код, при выполнении которого может возникнуть ошибка, заключается в отдельный блок (выделяется фигурными скобками), помеченный ключевым словом `try`. Данный блок будем называть `try`-блоком, а содержащийся в нем код — контролируемым кодом. После `try`-блока следует `catch`-блок. В этом `catch`-блоке размещается программный код, предназначенный для обработки исключений. Общий шаблон использования конструкции `try-catch` такой (жирным шрифтом выделены ключевые элементы шаблона):

```
try{
   // Контролируемый код
}
```

```
catch{
   // Код для обработки исключений
}
```

Если при выполнении контролируемого программного кода в `try`-блоке ошибок не было, то `catch`-блок игнорируется. По завершении выполнения кода из `try`-блока начинают выполняться команды после конструкции `try-catch`. Если же при выполнении команд в `try`-блоке возникла ошибка, то выполнение команд в `try`-блоке прекращается и начинают выполняться команды в `catch`-блоке. По завершении выполнения `catch`-блока начинают выполняться команды после конструкции `try-catch`. Напомним, что это наиболее простая схема перехвата и обработки исключений. Перехватываются исключения всех типов (все виды ошибок, которые, в принципе, можно перехватить и обработать), а объект исключения, который создается при возникновении ошибки, никак не используется.

Если мы хотим использовать при обработке ошибки объект исключения, то необходимо немного видоизменить способ описания `catch`-блока. Правда изменения минимальны: после ключевого слова `catch` в круглых скобках указывается имя класса, определяющего тип перехватываемых ошибок, а также формальное обозначение для объекта исключения. То есть используется следующий шаблон описания `catch`-блока:

```
catch(класс объект){
   // Код для обработки исключений
}
```

Если в качестве типа для объекта исключения указать класс `Exception`, то в `catch`-блоке будут перехватываться исключения всех типов, и при этом у нас будет возможность использовать объект исключения.

ПОДРОБНОСТИ

Если в `catch`-блоке указать класс перехватываемых ошибок, то этот блок будет обрабатывать ошибки данного класса и всех его подклассов. Класс `Exception` является базовым для прочих классов исключений. Поэтому если указать тип исключения как `Exception`, то в соответствующем `catch`-блоке будут перехватываться исключения всех типов.

Небольшой пример, в котором при обработке исключительной ситуации используется объект ошибки, представлен в листинге 5.1.

Листинг 5.1. Использование конструкции try-catch

```
using System;
// Класс с главным методом:
class UsingTryCatchDemo{
   // Главный метод:
   static void Main(){
      // Отображение сообщения:
      Console.WriteLine("Начинается выполнение программы");
      // Объявление переменных:
      int A=10,B=0;
      // Отображение значений переменных:
      Console.WriteLine("Переменная A="+A);
      Console.WriteLine("Переменная B="+B);
      // Контролируемый код:
      try{
         // Отображение сообщения:
         Console.WriteLine("Вычисляется выражение A/B");
         // Вычисление выражения (попытка деления на ноль):
         Console.WriteLine("Результат: "+A/B);
         // Отображение выражения (команда не выполняется):
         Console.WriteLine("Вычисления закончены");
      }
      // Блок обработки исключения:
      catch(Exception e){
         // Отображение сообщения:
         Console.WriteLine("Возникла ошибка!");
         // Тип ошибки:
         Console.WriteLine("Тип ошибки: "+e.GetType().Name);
         // Описание ошибки:
```

```
            Console.WriteLine("Описание: "+e.Message);
            // Источник ошибки (программа):
            Console.WriteLine("Программа: "+e.Source);
            // Метод, в котором произошла ошибка:
            Console.WriteLine("Метод: "+e.TargetSite.Name);
            // Место (строка) с ошибкой:
            Console.WriteLine("Место ошибки: "+e.StackTrace);
        }
        // Отображение сообщения:
        Console.WriteLine("Программа завершила выполнение");
    }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 5.1)

```
Начинается выполнение программы
Переменная A=10
Переменная B=0
Вычисляется выражение A/B
Возникла ошибка!
Тип ошибки: DivideByZeroException
Описание: Попытка деления на нуль.
Программа: Listing05_01
Метод: Main
Место ошибки:    в UsingTryCatchDemo.Main() в D:\Book\Listing05_01\
Program.cs:строка 18
Программа завершила выполнение
```

Программа достаточно простая. Вся интрига закручивается в главном методе. В начале выполнения программы в консольном окне появляется сообщение соответствующего содержания. Также объявляются две целочисленные переменные: переменная `A` со значением `10` и переменная `B`

со значением 0. Значения переменных отображаются в консольном окне. После этого начинает выполняться try-блок с контролируемым кодом.

Сначала командой Console.WriteLine("Вычисляется выражение A/B") выводится сообщение о вычислении частного двух целочисленных значений. При выполнении команды Console.WriteLine("Результат: "+A/B) предпринимается попытка вычислить значение выражения A/B. Но при нулевом значении переменной B возникает ошибка, связанная с делением на ноль. Поэтому дальше команды в try-блоке не выполняются (то есть до выполнения команды Console.WriteLine("Вычисления закончены") в try-блоке дело не доходит). Вместо этого создается объект исключения, и этот объект передается для обработки в catch-блок.

ПОДРОБНОСТИ

Выполнение команды Console.WriteLine("Результат: "+A/B) начинается с вычисления значения аргумента, переданного методу WriteLine(). При вычислении значения аргумента вычисляется выражение A/B. Как отмечалось выше, в этом месте возникает ошибка. Поэтому выполнение команды Console.WriteLine("Результат: "+A/B) прекращается, и никакого сообщения в консольном окне не будет.

В catch-блоке объект (точнее, ссылка на объект) исключения обозначен как e. Именно через переменную e в теле catch-блока мы обращаемся к объекту исключения. Там командой Console.WriteLine("Возникла ошибка!") отображается сообщение о возникновении ошибки, а далее с использованием объекта исключения e в консольном окне отображаются некоторые дополнительные сведения о том, что произошло.

У объекта исключения e есть метод GetType(). Метод результатом возвращает объект класса Type, содержащий информацию о типе объекта e. Этот объект класса Type имеет текстовое свойство Name, значение которого — название класса, к которому относится объект исключения e. Поэтому значение выражения e.GetType().Name — это название класса ошибки, которая произошла. Как следует из результата выполнения программы, произошла ошибка класса DivideByZeroException (из пространства имен System).

Текстовое свойство Message объекта e содержит описание ошибки. В данном случае значением свойства является текст "Попытка

деления на нуль.". Далее, значением свойства Source является текст с названием приложения или объекта, вызвавшего исключение. Значением свойства TargetSite является объект класса MethodBase. Объект содержит информацию о методе, вызвавшем исключение. Название метода можно получить с помощью свойства Name. Таким образом, значением выражения e.TargetSite.Name является текст "Main" с названием главного метода, в котором произошла ошибка. Значением свойства StackTrace является текст с указанием места (вплоть до номера строки кода), в котором возникла ошибка.

На этом выполнение catch-блока завершается. Работа программы завершается выполнением команды Console.WriteLine("Программа завершила выполнение").

ПОДРОБНОСТИ

При делении на ноль произошла ошибка класса DivideByZeroException, который является производным от класса ArithmeticException. Класс ArithmeticException является производным от класса SystemException, который, в свою очередь, является производным от класса Exception. В описании catch-блока объект ошибки e указан как относящийся к классу Exception. Свойства и методы объекта e, использованные в программе, — это свойства и методы объекта класса Exception. На самом деле объект ошибки относится к другому классу (имеется в виду класс DivideByZeroException). Но поскольку класс DivideByZeroException является производным от класса Exception, то, во-первых, ошибки класса DivideByZeroException перехватываются в catch-блоке, и, во-вторых, свойства и методы, которые есть у объекта класса Exception, также есть и у объекта класса DivideByZeroException.

Стоит также заметить, что если мы в программном коде изменим значение переменной B с 0 на 2 (вместо инструкции B=0 следует использовать инструкцию B=2), то результат выполнения программы будет таким, как показано ниже:

Результат выполнения программы (из листинга 5.1)

```
Начинается выполнение программы

Переменная A=10

Переменная B=2
```

```
Вычисляется выражение A/B
Результат: 5
Вычисления закончены
Программа завершила выполнение
```

В этом случае при выполнении `try`-блока ошибок не возникает. Поэтому выполняются все команды в `try`-блоке, а команды в `catch`-блоке игнорируются.

Основные классы исключений

— Чего-нибудь не так?
— Все так. Только Земля далеко.

из к/ф «Кин-дза-дза»

Как отмечалось выше, существует целая иерархия классов, описывающих различные исключительные ситуации. Большинство практически важных классов исключений содержатся в пространстве имен `System`. На вершине иерархии наследования классов находится класс `Exception`. Производными от класса `Exception` являются классы `SystemException` и `ApplicationException`. Класс `SystemException` является базовым для классов исключений, соответствующих системным ошибкам. Класс `ApplicationException` обычно используют для создания пользовательских исключений (этот вопрос рассматривается немного позже). В табл. 5.1 приведены некоторые классы исключений, которые нередко используются на практике.

Табл. 5.1. Некоторые классы исключений

Класс	Описание
`ArgumentException`	Исключение, связанное с некорректным аргументом
`ArithmeticException`	Исключение, связанное с выполнением арифметических операций. У этого класса имеется три производных класса: `DivideByZeroException` (ошибка деления на ноль), `NotFiniteNumberException` (получено бесконечное значение) и `OverflowException` (ошибка, связанная с переполнением)
`ArrayTypeMismatchException`	Исключение, связанное с попыткой записать в массив значение недопустимого типа

`FormatException`	Исключение, обусловленное недопустимым форматом
`IndexOutOfRangeException`	Исключение, связанное с выходом индекса за допустимые пределы
`InvalidCastException`	Исключение, обусловленное недопустимым преобразованием типов
`InvalidProgramException`	Исключение, связанное с неправильной компиляцией программы
`MemberAccessException`	Исключение, обусловленное неудачной попыткой доступа к члену класса. У класса этого исключения есть три производных класса: `FieldAccessException` (ошибка доступа к полю), `MethodAccessException` (ошибка доступа к методу) и `MissingMemberException` (отсутствует член класса)
`NotImplementedException`	Исключение, связанное с попыткой выполнить нереализованный метод или операцию
`NotSupportedException`	Исключение, связанное с попыткой выполнения неподдерживаемого метода или операции
`NullReferenceException`	Исключение, связанное с использованием пустой ссылки
`OutOfMemoryException`	Исключение, связанное с недостаточным объемом памяти
`RankException`	Исключение, связанное с передачей массива неправильной размерности
`StackOverflowException`	Исключение, связанное с переполнением стека
`TimeoutException`	Исключение, связанное с окончанием времени, выделенного на выполнение процесса или операции

Конечно, это лишь небольшая часть классов исключений. В действительности иерархия классов наследования более чем обширна, она содержит самые разнообразные классы, так сказать, «на все случаи жизни». Конкретные классы мы будем использовать и обсуждать по мере необходимости. А далее рассмотрим пример, в котором при выполнении программы генерируются (и обрабатываются) исключения разных типов. Обратимся к программному коду в листинге 5.2.

Листинг 5.2. Исключения разных типов

```
using System;
// Класс с главным методом:
class UsingExceptionsDemo{
   // Главный метод:
   static void Main(){
```

```
// Объект для генерирования случайных чисел:
Random rnd=new Random();
// Переменная массива:
int[] nums;
// Целочисленные переменные:
int x,n;
// Оператор цикла:
for(int k=1;k<10;k++){
   // Отображение номера цикла:
   Console.Write("[{0}] ",k);
   // Начальное значение переменной (номер команды):
   n=1;
   // Контролируемый блок кода:
   try{
      // Команда №1. Попытка создать массив:
      nums=new int[2*rnd.Next(3)-1];
      // Увеличение значения счетчика команд:
      n++;
      // Команда №2. Попытка вычислить частное:
      x=1/rnd.Next(3);
      // Увеличение значения счетчика команд:
      n++;
      // Команда №3. Попытка присвоить значение
      // элементу массива:
      nums[rnd.Next(2)-1]=x;
      // Увеличение значения счетчика команд:
      n++;
      // Команда №4. Ошибка преобразования:
      nums[0]=Int32.Parse("ноль");
   }
   // Блок обработки исключений:
   catch(Exception e){
```

```
            // Отображение номера команды с ошибкой:
            Console.Write("Команда №{0}: ",n);
            // Отображение типа (класса) исключения:
            Console.Write(e.GetType().Name);
            // Отображение описания для исключения:
            Console.WriteLine(" - "+e.Message);
        }
    }
  }
}
```

Возможный результат выполнения программы (с поправкой на то, что используется генератор случайных чисел) представлен ниже:

Результат выполнения программы (из листинга 5.2)

```
[1] Команда №3: IndexOutOfRangeException – Индекс находился вне границ массива.
[2] Команда №2: DivideByZeroException – Попытка деления на нуль.
[3] Команда №3: IndexOutOfRangeException – Индекс находился вне границ массива.
[4] Команда №1: OverflowException – Переполнение в результате выполнения
арифметической операции.
[5] Команда №4: FormatException – Входная строка имела неверный формат.
[6] Команда №2: DivideByZeroException – Попытка деления на нуль.
[7] Команда №4: FormatException – Входная строка имела неверный формат.
[8] Команда №1: OverflowException – Переполнение в результате выполнения
арифметической операции.
[9] Команда №2: DivideByZeroException – Попытка деления на нуль.
```

Программа, в общем-то, простая. В главном методе командой `Random rnd=new Random()` создается объект `rnd` класса `Random` для генерирования случайных чисел. Также мы объявляем переменную `nums`, которая является переменной целочисленного одномерного массива (может ссылаться на одномерный целочисленный массив). Также мы объявляем целочисленные переменные `x` и `n`, которые нам нужны для проведения вычислений.

Затем запускается оператор цикла, в котором индексная переменная k последовательно принимает значения от 1 до 9 включительно. В начале цикла командой Console.Write("[{0}] ",k) отображается номер цикла, а переменная n получает начальное (для данного цикла) единичное значение (команда n=1). После этого следует try-блок с контролируемым кодом. Там есть несколько команд, которые потенциально могут привести к возникновению ошибки (а одна команда просто вызывает ошибку). Перед каждой такой командой (за исключением первой) выполняется команда n++. Это позволяет нам вести учет потенциально опасных команд, а в случае возникновения ошибки мы будем знать, какая из четырех «опасных» команд стала фатальной.

Итак, сначала командой nums=new int[2*rnd.Next(3)-1] выполняется попытка создать массив. Она не всегда успешная. Размер массива определяется выражением 2*rnd.Next(3)-1. Значение выражения rnd.Next(3) — это случайное целое число в диапазоне от 0 до 2 включительно (то есть 0, 1 или 2). Поэтому значением выражения 2*rnd.Next(3)-1 с одинаковой вероятностью является число -1, 1 или 3. Если это число -1, то мы пытаемся создать массив отрицательного размера. В этом случае генерируется исключение класса OverflowException. Но если на этом этапе все прошло нормально, то затем выполняется команда x=1/rnd.Next(3). Здесь мы пытаемся присвоить значение целочисленной переменной x. Мы уже знаем, что значение выражения rnd.Next(3) есть число 0, 1 или 2. Если это число 1 или 2, то выполняется команда деления нацело и переменная x получает значение (1 или 0, но это не принципиально). Но если значение выражения окажется равным 0, то при вычислении значения переменной x возникает исключение класса DivideByZeroException, связанное с попыткой деления на ноль. Но если и тут повезло и ошибки не было, то она может возникнуть при выполнении команды nums[rnd.Next(2)-1]=x. Дело в том, что выражение rnd.Next(2)-1 принимает с равной вероятностью значения -1 и 0. Индекс, выходящий за пределы допустимого диапазона (в том числе и отрицательный индекс), — это исключение класса IndexOutOfRangeException. Если же на этом этапе тоже все в порядке, то на команде nums[0]=Int32.Parse("ноль") везение точно заканчивается. Здесь предпринята попытка преобразовать в целое число текст «ноль», что приводит к исключению класса FormatException.

Таким образом, при выполнении команд цикла точно генерируется исключение одного из четырех упомянутых выше типов. Такое исключение перехватывается и обрабатывается в catch-блоке. В нем отображается

номер команды, при выполнении которой возникла ошибка (значение переменной `n`), класс исключения (выражение `e.GetType().Name`), а также описание возникшей ошибки (выражение `e.Message`).

Использование нескольких `catch`-блоков

— Нас плюкане транклюкировали, пока мы на гастролях были.
— За что?
— За то, что мы их не успели.
— А вы их за что?
— Чтоб над головой не маячили!

из к/ф «Кин-дза-дза»

В предыдущем примере в процессе выполнения программы генерировались исключения разных классов, но перехватывались и обрабатывались они в одном блоке. Другими словами, для обработки ошибок разных типов мы использовали один и тот же блок программного кода. Правда мы использовали объект исключения, что позволяло нам определить фактический класс сгенерированного исключения, но принципиально это мало что меняет: какая бы ошибка ни возникла, для обработки выполнялся один и тот же код. Но процесс перехвата и обработки исключений можно организовать более гибким способом. Мы можем для каждого типа ошибки предусмотреть персональный блок обработки. Делается это просто. После `try`-блока указывается не один, а несколько `catch`-блоков. В каждом `catch`-блоке указывается (в круглых скобках) тип ошибки, который обрабатывается этим блоком. При возникновении ошибки в `try`-блоке выполнение команд контролируемого кода прекращается и начинается просмотр `catch`-блоков на предмет того, совпадает ли класс сгенерированного исключения с классом ошибки, указанным в `catch`-блоке. Если совпадение найдено, соответствующий `catch`-блок используется для обработки исключения. Если все `catch`-блоки просмотрены и совпадение не найдено, ошибка остается необработанной.

НА ЗАМЕТКУ

Совпадение класса сгенерированного исключения с классом исключения в `catch`-блоке понимается в том смысле, что классы должны

совпадать или класс сгенерированного исключения должен быть производным от класса, указанного в `catch`-блоке. Другими словами, в `catch`-блоке обрабатываются исключения, класс которых указан в описании `catch`-блока, а также исключения всех производных классов. Поэтому, например, блок, в котором типом ошибки указан класс `Exception`, указывается последним в последовательности блоков, поскольку такой блок обрабатывает все ошибки.

В описании `catch`-блока тип исключения указывается в круглых скобках после ключевого слова `catch`. При необходимости там же можно обозначить объект исключения и использовать его для получения информации о возникшем событии. В листинге 5.3 представлена программа (которая является вариацией предыдущего примера из листинга 5.2). В ней используется несколько `catch`-блоков для обработки исключений. Поскольку многие команды в программе должны быть уже знакомы читателю, то для сокращения объема программного кода комментарии удалены.

Листинг 5.3. Использование нескольких catch-блоков

```
using System;
class MoreExceptionsDemo{
   static void Main(){
      Random rnd=new Random();
      int[] nums;
      int x;
      for(int k=1;k<10;k++){
         Console.Write("[{0}] ",k);
         try{
            nums=new int[2*rnd.Next(3)-1];
            x=1/rnd.Next(3);
            nums[rnd.Next(2)-1]=x;
            nums[0]=Int32.Parse("ноль");
         }
         catch(OverflowException){
            Console.WriteLine("Ошибка №1: Неверный размер массива");
         }
```

```
            catch(DivideByZeroException){
               Console.WriteLine("Ошибка №2: Деление на ноль");
            }
            catch(IndexOutOfRangeException){
               Console.WriteLine("Ошибка №3: Неверный индекс элемента");
            }
            catch(FormatException){
               Console.WriteLine("Ошибка №4: Неверный формат числа");
            }
         }
      }
}
```

С поправкой на использование генератора случайных чисел результат выполнения программы может быть следующим:

Результат выполнения программы (из листинга 5.3)

```
[1] Ошибка №1: Неверный размер массива
[2] Ошибка №2: Деление на ноль
[3] Ошибка №4: Неверный формат числа
[4] Ошибка №3: Неверный индекс элемента
[5] Ошибка №4: Неверный формат числа
[6] Ошибка №2: Деление на ноль
[7] Ошибка №2: Деление на ноль
[8] Ошибка №1: Неверный размер массива
[9] Ошибка №3: Неверный индекс элемента
```

В данном случае мы отказались от переменной `n`, с помощью которой ранее запоминалась команда, вызывавшая ошибку. Принцип выполнения программного кода такой же, как и в предыдущем случае (см. Листинг 5.2). Отличие лишь в том, что при возникновении ошибки (причины возникновения ошибок не изменились) обработка ошибки выполняется одним из четырех `catch`-блоков. Класс ошибки, обрабатываемой `catch`-блоком, указан в круглых скобках после ключевого слова

catch. Объект ошибки при этом мы не объявляли (хотя могли бы), поскольку в процессе обработки ошибок он не используется. В зависимости от того, какого типа ошибка произошла, для обработки ошибки выполняется код соответствующего catch-блока.

НА ЗАМЕТКУ

Совсем не обязательно в индивидуальном порядке обрабатывать все ошибки. Другими словами, мы можем описать catch-блоки только для некоторых типов ошибок и предусмотреть catch-блок для ошибок всех прочих типов. Делается это просто. Например, мы хотим, чтобы по отдельным алгоритмам обрабатывались исключения классов OverflowException и FormatException, а для всех других исключений использовалась одна и та же схема обработки. В таком случае мы используем три catch-блока: сначала указываются блоки для ошибок типа OverflowException и FormatException, а затем указывается catch-блок для обработки исключений класса Exception. При возникновении ошибки блоки проверяются в том порядке, как они указаны. Если исключение не будет передано для обработки ни в первый, ни во второй блок, то оно будет обработано третьим блоком, поскольку в нем обрабатываются исключения всех классов, являющихся производными классами от класса Exception.

Вложенные конструкции try-catch и блок finally

> Вот потому, что вы говорите то, что не думаете, и думаете то, что не думаете, вот в клетках и сидите.
>
> *из к/ф «Кин-дза-дза»*

Один try-блок (внешний) может содержать в себе try-catch конструкции (внутренние). Интерес представляет ситуация, когда ошибка происходит во внутреннем try-блоке. У этого внутреннего try-блока есть catch-блоки, предназначенные для обработки исключений. Возможны две ситуации:

- Возникшая во внутреннем try-блоке ошибка перехватывается и обрабатывается в catch-блоке, связанном с данным try-блоком.
- Ошибка, возникшая во внутреннем try-блоке, не перехватывается в catch-блоке, связанном с данным try-блоком.

В первом случае все происходит штатно: ошибка обрабатывается, и после выполнения соответствующего `catch`-блока начинают выполняться команды после внутренней `try-catch` конструкции. А вот во втором случае, когда ошибка не обрабатывается во внутренней `try-catch` конструкции, в игру вступает внешняя `try-catch` конструкция. А именно, исключение, сгенерированное во внутреннем `try`-блоке, передается для обработки во внешнюю `try-catch` конструкцию. Если исключение обрабатывается в каком-то из `catch`-блоков этой конструкции, то дальше выполняются команды после внешней `try-catch` конструкции. Если ошибка не обрабатывается и во внешнем `try`-блоке, то исключение передается дальше, в следующую внешнюю `try-catch` конструкцию — конечно, если такая имеется. В конце концов, ошибка или будет обработана, или закончатся внешние `try-catch` конструкции. В последнем случае выполнение программы завершается и появляется сообщение об ошибке.

Конструкцию `try-catch`, состоящую из `try`-блока и `catch`-блоков, можно дополнить еще и `finally`-блоком. Этот блок указывается после `catch`-блоков. То есть можно использовать конструкцию `try-catch-finally` такого вида (жирным шрифтом выделены ключевые элементы шаблона):

```
try{
   // Контролируемый код
}
catch(класс объект){
   // Команды для обработки исключения
}
finally{
   // Обязательные для выполнения команды
}
```

В этой конструкции `catch`-блоков может быть несколько. Команды из блока `finally` выполняются в любом случае — и если ошибка в `try`-блоке возникла, и если ошибка в `try`-блоке не возникла. Возможны три принципиальные ситуации:

- При выполнении кода в `try`-блоке ошибка не возникла. Тогда код в `catch`-блоках игнорируется, а выполняется код в блоке `finally`, и после этого начинается выполнение команд после конструкции `try-catch-finally`.

- При выполнении контролируемого кода (в try-блоке) возникает ошибка, которая обрабатывается в одном из catch-блоков. В таком случае после обработки выполняется код в блоке finally, а затем выполняются команды после конструкции try-catch-finally.
- Выполняется try-блок, и при этом возникает ошибка. Среди catch-блоков нет такого, который был бы предназначен для обработки ошибки данного типа. В таком случае объект исключения передается для обработки во внешнюю конструкцию try-catch. Но перед этим выполняются команды в блоке finally.

Собственно, ради вот этого третьего сценария обычно и используют блок finally. Небольшой пример, в котором иллюстрируется использование вложенных try-catch конструкций и блока finally, представлен в листинге 5.4.

Листинг 5.4. Вложенные конструкции try-catch и блок finally

```
using System;
// Класс с главным методом:
class NestedTryDemo{
   // Статический метод:
   static void generator(string num){
      // Отображение сообщения:
      Console.WriteLine("Метод с аргументом \"{0}\" начинает работу",num);
      // Текстовая переменная:
      string msg="Массив не создан!";
      // Контролируемый код:
      try{
         // Преобразование текста в целое число:
         int m=Int32.Parse(num);
         // Создание массива:
         int[] array=new int[m];
         // Новое значение текстовой переменной:
         msg="Создан массив:";
         // Оператор цикла:
         for(int k=0;k<=4;k++){
```

```
            // Элементу массива присваивается значение:
            array[k]=12/(4-k);
            // К тексту добавляется значение элемента:
            msg+=" "+array[k];
         }
      }
      // Обработка ошибки выхода за пределы массива:
      catch(IndexOutOfRangeException){
         // Отображение сообщения:
         Console.WriteLine("Ошибка в методе: выход за пределы массива");
      }
      // Код для обязательного выполнения:
      finally{
         // Отображение значения текстовой переменной:
         Console.WriteLine(msg);
      }
      // Отображение сообщения:
      Console.WriteLine("Метод завершил работу");
   }
   // Главный метод:
   static void Main(){
      // Текстовый массив:
      string[] args={"один","-2","5","2"};
      // Символьный массив из одного элемента:
      char[] symbs=new char[1];
      // Оператор цикла:
      for(int k=0;k<args.Length;k++){
         // Контролируемый код (внешний блок):
         try{
            // Отображение сообщения:
            Console.WriteLine("Выполняется цикл №{0}:",k+1);
            // Контролируемый код (внутренний блок):
```

```
            try{
               // Вызов статического метода:
               generator(args[k]);
               // Команда с ошибкой (неверный индекс):
               symbs[1]='A';
            }
            // Обработка ошибки деления на ноль
            // (внутренний блок):
            catch(DivideByZeroException e){
                // Отображение сообщения:
               Console.WriteLine("Первый внутренний catch-блок: {0}",
e.GetType().Name);
            }
            // Обработка ошибки выхода за пределы массива
            // (внутренний блок):
            catch(IndexOutOfRangeException e){
               // Отображение сообщения:
               Console.WriteLine("Второй внутренний catch-блок: {0}",
e.GetType().Name);
            }
            // Обязательный для выполнения код:
            finally{
               // Элементу массива присваивается значение:
               symbs[0]=(char)('A'+k);
               // Отображение сообщения:
               Console.WriteLine("Символ: \'{0}\'",symbs[0]);
            }
         }
         // Обработка всех ошибок (внешний блок):
         catch(Exception e){
            // Отображение сообщения:
```

```
            Console.WriteLine("Внешний catch-блок: {0}",e.GetType().Name);
        }
        // Отображение сообщения:
        Console.WriteLine("Цикл №{0} завершен...\n",k+1);
    }
  }
}
```

Результат выполнения программы может быть таким:

Результат выполнения программы (из листинга 5.4)

```
Выполняется цикл №1:
Метод с аргументом "один" начинает работу
Массив не создан!
Символ: 'A'
Внешний catch-блок: FormatException
Цикл №1 завершен...

Выполняется цикл №2:
Метод с аргументом "-2" начинает работу
Массив не создан!
Символ: 'B'
Внешний catch-блок: OverflowException
Цикл №2 завершен...

Выполняется цикл №3:
Метод с аргументом "5" начинает работу
Создан массив: 34612
Первый внутренний catch-блок: DivideByZeroException
Символ: 'C'
Цикл №3 завершен...
```

```
Выполняется цикл №4:
Метод с аргументом "2" начинает работу
Ошибка в методе: выход за пределы массива
Создан массив: 34
Метод завершил работу
Второй внутренний catch-блок: IndexOutOfRangeException
Символ: 'D'
Цикл №4 завершен...
```

В программе, в классе с главным методом описан статический метод `generator()`. Метод не возвращает результат и имеет текстовый аргумент (обозначен как `num`). Предполагается, что это текстовое представление целого числа. Поэтому командой `int m=Int32.Parse(num)` в теле метода выполняется попытка преобразовать аргумент в целое число. Эта команда размещена внутри `try`-блока, однако ошибка класса `FormatException`, которая может возникнуть при попытке преобразования текста в число, в `catch`-блоке не обрабатывается, поскольку этот блок предназначен для обработки ошибок класса `IndexOutOfRangeException`. Поэтому если ошибка на этом этапе возникла, то метод завершает работу и ошибка передается во внешнюю `try-catch` конструкцию (если такая есть) для обработки. Но перед этим выполняется блок `finally`, в котором отображается значение текстовой переменной `msg`. Начальное значение переменной — текст `"Массив не создан!"`. Но если преобразование текстового аргумента в число проходит успешно, а также без ошибок выполняется команда `int[] array=new int[m]`, которой создается целочисленный массив, то командой `msg="Создан массив:"` переменной `msg` присваивается новое значение, а затем в операторе цикла командой `msg+=" "+array[k]` к текущему текстовому значению переменной дописывается значение элемента массива `array` с индексом `k`. Предварительно значение элементу присваивается командой `array[k]=12/(4-k)`. Важно и то, что в операторе цикла индексная переменная `k` принимает значения от `0` до `4` включительно. Поэтому здесь возможны две неприятности. Во-первых, если размер массива меньше `4`, то возникает ошибка, связанная с выходом за пределы массива. Во-вторых, если дело доходит до вычисления элемента с индексом `4`, то получаем ошибку деления на ноль. Как отмечалось выше, ошибка выхода за пределы массива обрабатывается в `catch`-блоке, в котором отображается сообщение

соответствующего содержания. После обработки ошибки выполняется блок `finally` и команда, которая выводит сообщение о том, что метод завершил работу. Если же возникает ошибка деления на ноль, то она в методе не обрабатывается, а выбрасывается из метода для дальнейшей обработки внешними блоками обработки (если они есть), но перед этим выполняется блок `finally`.

НА ЗАМЕТКУ

Возможны следующие сценарии.

- При вызове метода `generator()` с текстовым аргументом, который не является текстовым представлением целого числа, появляется сообщение о начале работы метода, а затем выполняется блок `finally`, выводящий сообщение о том, что массив не создан. Метод выбрасывает (генерирует и не обрабатывает) исключение класса `FormatException`.
- Это же происходит в случае, если методу в качестве аргумента передано текстовое представление для отрицательного числа, но метод выбрасывает исключение класса `OverflowException`.
- Если аргументом методу передано текстовое представление для числа не меньше `4`, то появляется сообщение о начале работы метода, заполняется четыре элемента массива, и их значения отображаются в консоли при выполнении блока `finally`, и выбрасывается исключение класса `DivideByZeroException`.
- Если аргументом методу передано текстовое представление для числа меньше `4`, то появляется сообщение о начале работы метода, создается массив соответствующего размера, массив заполняется и генерируется ошибка класса `IndexOutOfRangeException`. Но эта ошибка обрабатывается в `catch`-блоке (появляется сообщение о выходе за пределы массива). При выполнении `finally`-блока отображается содержимое массива, а затем появляется сообщение о завершении работы метода.

В главном методе программы командой `string[] args={"один", "-2","5","2"}` создается текстовый массив. Элементы массива используются для передачи аргументами методу `generator()`. Также командой `char[] symbs=new char[1]` создается вспомогательный символьный массив из одного элемента. После этого запускается оператор цикла, в котором перебираются индексы элементов массива `args`.

Тело оператора цикла состоит из внешней `try-catch` конструкции и команды, отображающей сообщение о завершении выполнения

соответствующего цикла. Во внешнем `catch`-блоке обрабатываются исключения класса `Exception` (то есть фактически все ошибки). При обработке исключения отображается название класса ошибки.

Внешний `try`-блок состоит из команды, которой отображается сообщение о начале выполнения цикла, а также внутренней конструкции `try-catch-finally`. Внутренний `try`-блок состоит из команд `generator(args[k])` и `symbs[1]='A'`. Вторая из команд содержит ошибку: поскольку массив `symbs` состоит всего из одного элемента, то элемента с индексом `1` у массива `symbs` нет. Выполнение данной команды приводит к ошибке класса `IndexOutOfRangeException`. Но команда `symbs[1]='A'` выполняется, только если при выполнении команды `generator(args[k])` не было выброшено исключение.

НА ЗАМЕТКУ

При выполнении внутреннего `try`-блока в любом случае возникает ошибка. Это или ошибки класса `FormatException`, `OverflowException` или `DivideByZeroException`, выбрасываемые при вызове метода `generator()`, или ошибка класса `IndexOutOfRangeException`, возникающая при выполнении команды `symbs[1]='A'`.

Для внутреннего `try`-блока предусмотрены `catch`-блоки, обрабатывающие ошибки классов `DivideByZeroException` и `IndexOutOfRangeException`. Есть также блок `finally`, при выполнении которого командой `symbs[0]=(char)('A'+k)` присваивается значение единственному элементу массива `symbs`, а затем значение этого элемента отображается в консольном окне.

Чтобы понять результат выполнения программы, необходимо учесть следующие обстоятельства.

- При вызове метода `generator()` с аргументом `"один"` происходит ошибка класса `FormatException`, связанная с преобразованием текста в число. Ошибка не обрабатывается во внутренней конструкции `try-catch`, а передается для обработки во внешнюю конструкцию `try-catch`. Перед этим выполняется `finally`-блок из метода и `finally`-блок из внутренней конструкции `try-catch-finally`.
- При вызове метода `generator()` с аргументом `"-2"` происходит ошибка класса `OverflowException`, которая не обрабатывается ни в методе, ни во внутренней конструкции `try-catch-finally`.

Обработка выполняется во внешнем catch-блоке, но предварительно выполняются finally-блоки в методе и во внутренней конструкции try-catch-finally.

- При вызове методе generator() с аргументом "5" генерируется ошибка класса DivideByZeroException, которая не обрабатывается в методе, но обрабатывается во внутренней конструкции try-catch-finally. Перед обработкой выполняется блок finally из метода. Затем после выполнения кода из соответствующего внутреннего catch-блока выполняется finally-блок внутренней конструкции try-catch-finally.
- Если метод generator() вызывается с аргументом "2", то генерируется ошибка класса IndexOutOfRangeException, которая обрабатывается в методе. После выполнения catch-блока в методе выполняется finally-блок. Метод завершает работу сообщением соответствующего содержания. После этого во внутреннем try-блоке при выполнении команды symbs[1]='A' генерируется еще одно исключение класса IndexOutOfRangeException. Оно обрабатывается в одном из внутренних catch-блоков, а после него выполняется блок finally внутренней конструкции try-catch-finally.

Генерирование исключений

— Тормози!
— Как я могу затормозить, когда ты всю тормозную жидкость выпил?

из к/ф «Кин-дза-дза»

Исключения можно генерировать преднамеренно. Речь идет не о том, чтобы написать команду, выполнение которой приводит к ошибке. Речь идет об использовании специальной инструкции, выполнение которой имеет такие же последствия, как и возникновение ошибки.

Сгенерировать исключение достаточно просто. Для этого используют инструкцию throw, после которой указывается объект исключения. Объект исключения можно создать в явном виде (то есть так, как создаются объекты обычных классов). Также в catch-блоке можно воспользоваться объектом, созданным автоматически при возникновении реальной ошибки (повторное генерирование исключения). Правда

исключение, перехваченное в `catch`-блоке, может быть сгенерировано повторно и более простым способом. Для этого используют инструкцию `throw`, не указывая явно объект исключения.

НА ЗАМЕТКУ

Само по себе создание объекта на основе класса исключения не означает генерирования исключения. Необходимо использовать инструкцию `throw`, после которой указать объект исключения.

В листинге 5.5 представлена программа, в которой искусственно генерируются исключения.

Листинг 5.5. Генерирование исключений

```
using System;
// Класс с главным методом:
class ThrowDemo{
   // Главный метод:
   static void Main(){
      // Внешний try-блок:
      try{
         // Внутренний try-блок:
         try{
            Console.WriteLine("Сейчас будет ошибка...");
            // Генерирование исключения:
            throw new ArithmeticException("Какая-то ошибка");
         }
         // Перехват исключения во внутреннем блоке:
         catch(ArithmeticException e){
            Console.WriteLine(e.Message);
            // Повторное генерирование исключения:
            throw;
         }
      }
```

```
      // Перехват исключения во внешнем блоке:
      catch(ArithmeticException e){
         Console.WriteLine("Еще раз: \"{0}\"",e.Message);
      }
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 5.5)

```
Сейчас будет ошибка...
Какая-то ошибка
Еще раз: "Какая-то ошибка"
```

В этой программе используются вложенные `try`-блоки. Во внутреннем `try`-блоке после выполнения команды `Console.WriteLine("Сейчас будет ошибка...")` генерируется исключение класса `ArithmeticException`. Для этого мы использовали инструкцию `throw`, после которой указана команда `new ArithmeticException("Какая-то ошибка")` создания анонимного объекта класса `ArithmeticException`. Текст, переданный в качестве аргумента конструктору класса, становится значением свойства `Message` объекта исключения.

НА ЗАМЕТКУ

В принципе, мы могли бы создать обычный (не анонимный) объект класса `ArithmeticException`, а затем указать этот объект в `throw`-инструкции. Еще раз подчеркнем, что само по себе создание объекта исключения не приводит к генерированию ошибки.

Перехват сгенерированного исключения выполняется во внутреннем `catch`-блоке. Там командой `Console.WriteLine(e.Message)` отображается описание ошибки, а затем командой `throw` исключение генерируется повторно. Важный момент состоит в том, что объект исключения точно тот же, что был создан ранее при искусственном генерировании исключения. Проверить это легко. Повторно сгенерированное исключение перехватывается и обрабатывается во внешнем `catch`-блоке. В нем отображается значение свойства `Message` объекта

исключения, и оно такое же, как аргумент, переданный конструктору класса `ArithmeticException` при создании объекта исключения.

Механизм искусственного генерирования исключений используется в самых разных ситуациях (например, для множественной обработки исключений) и позволяет значительно повысить гибкость программного кода.

Пользовательские классы исключений

Рации у нас нет, мы ее выкинули.

из к/ф «Кин-дза-дза»

Мы можем создать собственный класс исключений. Схема очень простая и базируется на механизме наследования классов. Пользовательский класс исключения обычно создается наследованием класса `ApplicationException` (или `Exception`). Объекты такого класса могут иметь дополнительные свойства и методы, ориентированные на решение определенной прикладной задачи. Генерируются исключения пользовательских классов искусственно, а перехватываются и обрабатываются таким же способом, как и другие исключения. Программа, в которой описывается класс пользовательского исключения, а затем генерируется и обрабатывается исключение пользовательского типа, представлена в листинге 5.6.

Листинг 5.6. Пользовательские классы исключений

```
using System;
// Класс пользовательского исключения:
class MyException:ApplicationException{
   // Закрытое целочисленное поле:
   private int code;
   // Конструктор:
   public MyException(int n,string txt):base(txt){
      code=n;
   }
   // Переопределение метода ToString():
```

```
   public override string ToString(){
      // Текстовая строка:
      string res=Message+"\nКод ошибки: "+code;
      // Результат метода:
      return res;
   }
}
// Главный класс:
class MyExceptionDemo{
   // Главный метод:
   static void Main(){
      // Контролируемый код:
      try{
         // Генерирование пользовательского исключения:
         throw new MyException(123,"Большая ошибка");
      }
      // Перехват и обработка исключения:
      catch(Exception e){
         Console.WriteLine(e);
      }
   }
}
```

Ниже показано, как выглядит результат выполнения программы:

Результат выполнения программы (из листинга 5.6)

```
Большая ошибка
Код ошибки: 123
```

Мы описали класс `MyException`, который наследует класс `ApplicationException`. В классе описано закрытое целочисленное поле `code`, а также мы описали конструктор класса с двумя аргументами. Первый целочисленный аргумент определяет значение поля `code`.

Второй текстовый аргумент передается конструктору базового класса. Поэтому текстовый аргумент конструктора класса `MyException` определяет свойство `Message` объекта исключения. Кроме этого, в классе `MyException` переопределяется метод `ToString()`. Результатом метод возвращает текстовую строку, которая содержит описание ошибки (значение свойства `Message`) и значение поля `code`.

В главном методе программы в контролируемом блоке командой `throw new MyException(123,"Большая ошибка")` генерируется исключение пользовательского типа. Для перехвата исключения использован `catch`-блок, в котором указано имя класса `Exception`. Поскольку класс `MyException` является производным от класса `ApplicationException`, а он, в свою очередь, является производным от класса `Exception`, то данный `catch`-блок перехватывает сгенерированное исключение. Обработка исключения состоит в том, что выполняется команда `Console.WriteLine(e)`. Для объекта исключения `e` автоматически вызывается метод `ToString()`, и в итоге в консольном окне появляется сообщение с описанием ошибки и значением целочисленного поля объекта ошибки. Эти значения были указаны при создании объекта исключения в команде с ключевым словом `throw`.

Инструкции `checked` и `unchecked`

> Ребята! Со спичкой мы пошутили. Мы радость хотели доставить вам!
>
> *из к/ф «Кин-дза-дза»*

В некоторых случаях удобно использовать инструкции `checked` и `unchecked`. С помощью инструкции `checked` можно выделить блок или выражение, которые в таком случае контролируются на предмет переполнения (выхода значения за допустимые границы). Если происходит переполнение, то генерируется исключение класса `OverflowException`. Шаблон использования инструкции `checked` для выделения блока команд такой:

```
checked{
   // Команды
}
```

Если мы хотим контролировать отдельное выражение, то шаблон использования инструкции `checked` будет следующим:

```
checked(выражение)
```

Напротив, инструкция `unchecked` используется, если мы не хотим, чтобы при переполнении генерировалось исключение класса `OverflowException`. Шаблоны использования инструкции `unchecked` такие же, как и для инструкции `checked`. Если с помощью инструкции `unchecked` контролируем блок кода, то используем такой шаблон:

```
unchecked{
   // Команды
}
```

Если с помощью инструкции `unchecked` контролируется отдельное выражение, то используется оно следующим образом:

```
unchecked(выражение)
```

Стоит сразу заметить, что по умолчанию используется режим, при котором исключения в случае переполнения не генерируется. С другой стороны, использование инструкций `checked` и `unchecked` оправдывается тем, что в общем случае могут быть задействованы разные глобальные настройки среды выполнения и применение инструкций позволит в явном виде идентифицировать способ реагирования на возникновение переполнения.

Программа, в которой есть пример использования инструкций `checked` и `unchecked`, представлена в листинге 5.7.

Листинг 5.7. Инструкции checked и unchecked

```
using System;
// Класс с главным методом:
class CheckedDemo{
   // Главный метод:
   static void Main(){
      // Переменные:
      byte a=100,b=200,c;
```

```
      // Контролируемый код:
      try{
         // Проверка на предмет переполнения:
         c=checked((byte)(a+b));
      }
      // Обработка исключения:
      catch(OverflowException){
         Console.WriteLine("Есть ошибка: "+(a+b));
      }
      // Нет проверки на предмет переполнения:
      c=unchecked((byte)(a+b));
      Console.WriteLine("Нет  ошибки: "+c);
   }
}
```

Результат выполнения программы будет следующим:

Результат выполнения программы (из листинга 5.7)

```
Есть ошибка: 300
Нет  ошибки: 44
```

Программа очень простая. В главном методе объявляются три переменные типа `byte`: переменная `a` со значением `100`, переменная `b` со значением `200` и переменная `c`, значение которой при объявлении не присваивается.

Значение переменной `c` сначала присваивается командой `c=checked((byte)(a+b))` внутри `try`-блока. Сумма значений переменных `a` и `b` равна `300`, и это число выходит за верхнюю допустимую границу `255`. Поскольку соответствующее выражение помечено инструкцией `checked`, то генерируется исключение класса `OverflowException`. Оно перехватывается и обрабатывается в `catch`-блоке. Там выполняется команда `Console.WriteLine("Есть ошибка: "+(a+b))`, и реальная сумма значений переменных `a` и `b` появляется в диалоговом окне.

Еще раз значение переменной `c` присваивается командой `c=unchecked((byte)(a+b))`. Но теперь мы использовали инструкцию

unchecked. Поэтому исключение, связанное с выходом за верхнюю границу значения, присваиваемого переменной c , не генерируется.

НА ЗАМЕТКУ

Если бы мы не использовали инструкцию unchecked, то исключение тоже не генерировалось бы. Напомним, этот режим используется по умолчанию.

Фактическое значение переменной c проверяем с помощью команды Console.WriteLine("Нет ошибки: "+c).

ПОДРОБНОСТИ

Присваиваемое переменной c число 300 можно представить в виде $300 = 256 + 44 = 256 + 32 + 8 + 4 = 2^8 + 2^5 + 2^3 + 2^2$. Но для записи значений типа byte отводится всего 8 битов. В присваиваемом значении старшие биты отбрасываются (то есть пропадает слагаемое 2^8). Переменная c получает значение $2^5 + 2^3 + 2^2 = 32 + 8 + 4 = 44$.

Использование исключений

> У нас в уезде писарь был. Год рождения в паспорте одной циферкой записывал — чернила, шельмец, вишь, экономил.
>
> *из к/ф «Формула любви»*

В этом разделе мы рассмотрим несколько примеров, в которых используется обработка исключительных ситуаций. Начнем с программы, в которой решается линейное уравнение вида $Ax = B$. В общем случае решением этого уравнения является значение $x = B/A$. Но эта формула справедлива, только если параметр $A \neq 0$. А если $A = 0$, то возможны два варианта. При $B \neq 0$ уравнение решений не имеет, а при $B = 0$ решением является любое число. Соответствующий программный код представлен в листинге 5.8.

Листинг 5.8. Решение линейного уравнения

```
using System;
// Класс с главным методом:
```

```
class EquationDemo{
   // Метод для решения уравнения:
   static double solve(double A,double B){
      // Если значение параметра A отлично от нуля:
      if(A!=0) return B/A;
      // Текстовая переменная:
      string msg;
      // Если параметр B имеет нулевое значение:
      if(B==0) msg="Решение - любое число";
      // Если значение параметра B отлично от нуля:
      else msg="Решений нет";
      // Генерируется исключение:
      throw new ArithmeticException(msg);
   }
   // Главный метод:
   static void Main(){
      // Числовые переменные:
      double A,B,x;
      Console.WriteLine("Решение уравнения Ax=B");
      // Контролируемый код:
      try{
         Console.Write("Параметр A=");
         // Считывается значение параметра A:
         A=Double.Parse(Console.ReadLine());
         Console.Write("Параметр B=");
         // Считывается значение параметра B:
         B=Double.Parse(Console.ReadLine());
         // Вычисление решения:
         x=solve(A,B);
         // Отображение решения:
         Console.WriteLine("Решение x={0}",x);
      }
```

```
      // Обработка исключения:
      catch(Exception e){
         // Отображение описания исключения:
         Console.WriteLine(e.Message);
      }
   }
}
```

Результат выполнения программы может быть таким (здесь и далее жирным шрифтом выделены введенные пользователем значения):

Результат выполнения программы (из листинга 5.8)

```
Решение уравнения Ax=B
Параметр A=5
Параметр B=12
Решение x=2,4
```

Таким:

Результат выполнения программы (из листинга 5.8)

```
Решение уравнения Ax=B
Параметр A=0
Параметр B=12
Решений нет
```

Таким:

Результат выполнения программы (из листинга 5.8)

```
Решение уравнения Ax=B
Параметр A=0
Параметр B=0
Решение – любое число
```

Или таким:

Результат выполнения программы (из листинга 5.8)

```
Решение уравнения Ax=B

Параметр A=5

Параметр B=b

Входная строка имела неверный формат.
```

Для решения уравнения мы описали специальный статический метод `solve()`. У метода два аргумента типа `double`. Это параметры `A` и `B` уравнения. Результатом метод возвращает частное `A/B`. Но такое значение вычисляется только в случае, если первый аргумент `A` отличен от нуля. Если первый аргумент нулевой, то проверяется (на предмет равенства нулю) значение второго аргумента `B`. В зависимости от результатов проверки, текстовой переменной `msg` присваивается значение, и эта переменная передается в качестве аргумента конструктору класса `ArithmeticException` при генерировании исключения. Таким образом, метод `solve()` результатом возвращает решение уравнения или метод генерирует исключение класса `ArithmeticException`, если решений нет или если решением может быть любое число. Эта информация содержится в описании исключения (свойство `Message` объекта исключения).

В главном методе программы сначала считываются значения параметров `A` и `B`, а затем для вычисления решения уравнения вызывается метод `solve()`. Все эти команды выполняются внутри `try`-блока, а соответствующий `catch`-блок перехватывает все исключения, в том числе и связанные с некорректно введенными значениями.

НА ЗАМЕТКУ

Если деление на ноль выполняется на множестве действительных чисел, то исключение не генерируется (в отличие от случая, когда деление на ноль выполняется на множестве целых чисел).

В следующем примере есть метод, у которого два аргумента: текст и символ. Результатом метод возвращает индекс первого вхождения символа в текст. Если заданного символа в тексте нет, то метод генерирует исключение. Рассмотрим программный код в листинге 5.9.

Листинг 5.9. Поиск символа в тексте

```
using System;
// Класс с главным методом:
class FindSymbolDemo{
   // Метод для поиска символа в тексте:
   static int find(string txt,char s){
      // Перебор символов в тексте:
      for(int k=0;k<txt.Length;k++){
         // Если есть совпадение:
         if(txt[k]==s) return k;
      }
      // Генерируется исключение:
      throw new Exception("Символа \'"+s+"\' в тексте \""+txt+"\" нет");
   }
   // Главный метод:
   static void Main(){
      // Текстовая переменная:
      string txt;
      // Символьная переменная:
      char s;
      // Целочисленная переменная:
      int index;
      // Контролируемый код:
      try{
         Console.Write("Введите текст: ");
         // Считывание текста:
         txt=Console.ReadLine();
         Console.Write("Введите символ: ");
         // Считывание символа:
         s=(char)Console.Read();
         // Определение индекса символа в тексте:
```

```
            index=find(txt,s);
            // Отображение результата:
            Console.WriteLine("Индекс символа: "+index);
        }
        // Обработка исключений:
        catch(Exception e){
            // Описание исключения:
            Console.WriteLine(e.Message);
        }
    }
}
```

Возможный результат выполнения программы представлен ниже (жирным шрифтом выделены введенные пользователем значения):

Результат выполнения программы (из листинга 5.9)

```
Введите текст: Изучаем язык C#
Введите символ: ы
Индекс символа: 10
```

Если символа в тексте нет, то результат будет следующим:

Результат выполнения программы (из листинга 5.9)

```
Введите текст: Изучаем язык C#
Введите символ: д
Символа 'д' в тексте "Изучаем язык C#" нет
```

Схему мы использовали простую, напоминающую подход, реализованный в предыдущем примере (см. Листинг 5.8). В статическом методе `find()` перебираются символы в тексте (первый аргумент), и если символ из текста совпадает с тем символом, который передан вторым аргументом методу, то индекс символа возвращается результатом метода. Если оператор цикла завершился, а результат не возвращен, то генерируется исключение класса `Exception`. Описание исключения содержит информацию о том, что данного символа в тексте нет.

В главном методе считывается текст и символ. Затем вызывается метод `find()`. Команды размещены в `try`-блоке, а исключения обрабатываются в `catch`-блоке.

ПОДРОБНОСТИ

Для считывания символа использован статический метод `Read()` из класса `Console`. Метод возвращает код считанного символа. Поэтому мы воспользовались явным приведением типа для преобразования кода в символ.

В следующем примере обработка исключений используется для прекращения работы оператора цикла. Рассмотрим программу в листинге 5.10.

Листинг 5.10. Завершение работы оператора цикла

```
using System;
// Класс с индексатором:
class MyClass{
   // Закрытое поле - ссылка на массив:
   private int[] nums;
   // Конструктор:
   public MyClass(int size){
      nums=new int[size];
   }
   // Индексатор:
   public int this[int k]{
      // Аксессор для считывания значения:
      get{
         return nums[k];
      }
      // Аксессор для присваивания значения:
      set{
         nums[k]=value;
      }
   }
```

```
}
// Главный класс:
class WhileAndExceptionDemo{
   // Метод для отображения значений элементов
   // массива из объекта:
   static void show(MyClass obj){
      // Целочисленная переменная:
      int k=0;
      // Контролируемый блок:
      try{
         // Оператор цикла:
         while(true){
            // Отображение значения элемента массива:
            Console.Write("|"+obj[k]);
            // Увеличение значения индексной переменной:
            k++;
         }
      }
      // Обработка исключений:
      catch(IndexOutOfRangeException){
         Console.WriteLine("|");
      }
   }
   // Главный метод:
   static void Main(){
      // Создание объекта класса:
      MyClass obj=new MyClass(10);
      // Целочисленная переменная:
      int k=0;
      // Контролируемый код:
      try{
         // Оператор цикла:
```

```
            do{
               // Присваивание значения элементу массива
               // в объекте:
               obj[k]=2*k+1;
               // Увеличение значения индексной переменной:
               k++;
            }while(true);
         }
         // Обработка исключений:
         catch(IndexOutOfRangeException){
            // Отображение содержимого массива из объекта:
            show(obj);
         }
      }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 5.10)

```
|1|3|5|7|9|11|13|15|17|19|
```

В этом примере мы описали класс `MyClass`, в котором есть закрытое поле `nums`, являющееся ссылкой на целочисленный массив. Массив создается при вызове конструктора. Аргументом конструктору передается число, определяющее размер массива. Для доступа к элементам массива используется индексатор. Проблема в том, что массив закрытый, но в классе не определены свойство или метод, которые позволили бы узнать размер массива.

В главном классе описан статический метод `show()`, который не возвращает результат. Аргументом методу передается объект класса `MyClass`. Метод предназначен для отображения содержимого массива из объекта, переданного методу в качестве аргумента. Для этого в `try`-блоке запускается формально бесконечный оператор цикла `while`. За каждую итерацию цикла отображается значение проиндексированного объекта (значение элемента массива из объекта), и индекс

увеличивается на единицу. Как только будет предпринята попытка считать значение несуществующего элемента, будет сгенерировано исключение класса `IndexOutOfRangeException`. Как следствие, выполнение оператора цикла прекращается, а исключение обрабатывается в `catch`-блоке.

В главном методе командой `MyClass obj=new MyClass(10)` создается объект `obj` класса `MyClass`, в котором «спрятан» массив из `10` элементов. Для заполнения массива в `try`-блоке запускается бесконечный цикл с оператором `do-while`. Командой `obj[k]=2*k+1` очередному элементу массива присваивается значение (нечетное число), а затем командой `k++` увеличивается значение индекса. Оператор цикла завершает выполнение при возникновении исключения класса `IndexOutOfRangeException` (попытка присвоить значение несуществующему элементу). При обработке этого исключения в `catch`-блоке с помощью команды `show(obj)` отображается содержимое массива из объекта `obj` (последовательность из `10` нечетных чисел).

В следующем примере описывается пользовательский класс исключения, который позволяет создавать объекты на основе уже существующих объектов исключений. Также в программе используется повторное генерирование исключений (имеется в виду генерирование исключения в `catch`-блоке). Соответствующая программа представлена в листинге 5.11.

Листинг 5.11. Пользовательский класс на основе объекта

```
using System;
// Класс исключения:
class MyException:Exception{
   // Закрытые текстовые поля:
   private string time;
   private string source;
   // Конструктор создания объекта исключения:
   public MyException(Exception obj):base("Ошибка класса MyException"){
      // Класс исходной ошибки:
      source=obj.GetType().Name;
      // Дата и время генерирования ошибки:
```

```
      time=DateTime.Now.ToString();
   }
   // Переопределение метода ToString():
   public override string ToString(){
      string txt=Message+"\n";
      txt+="Исходный класс ошибки: "+source+"\n";
      txt+="Дата и время генерирования: "+time;
      return txt;
   }
}
// Главный класс:
class MoreMyExceptionDemo{
   // Главный метод:
   static void Main(){
      // Внешний try-блок:
      try{
         // Внутренний try-блок:
         try{
            // Массив из одного элемента:
            int[] a={1};
            // Обращение к несуществующему элементу
            // массива (генерирование ошибки):
            a[1]=2;
         }
         // Внутренний catch-блок:
         catch(Exception e){
            // Создание объекта исключения:
            MyException me=new MyException(e);
            // Генерирование исключения:
            throw me;
         }
      }
```

```
        // Внешний catch-блок:
        catch(Exception e){
            Console.WriteLine(e);
        }
    }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 5.11)

```
Ошибка класса MyException
Исходный класс ошибки: IndexOutOfRangeException
Дата и время генерирования: 19.09.201720:14:25
```

Интерес представляет класс `MyException`. Класс имеет два закрытых текстовых поля `source` и `time`. Также в классе описан конструктор, который позволяет создавать объекты класса `MyException` на основе объектов класса `Exception` (а значит, и на основе объектов всех производных классов от класса `Exception`). Конструктору производного класса передается текст `"Ошибка класса MyException"`, который определяет значение свойства `Message` создаваемого объекта. Полю `source` присваивается значение `obj.GetType().Name`. Это название класса для объекта исключения. На основе этого класса создается новый объект исключения. Полю `time` в качестве значения присваивается выражение `DateTime.Now.ToString()`. Здесь мы использовали структуру `DateTime` из пространства имен `System`. С помощью статического свойства `Now` получаем ссылку на экземпляр структуры `DateTime`, содержащий информацию о текущей дате и времени (определяется по системным часам компьютера). С помощью метода `ToString()`, который вызывается из этого экземпляра, получаем текстовую строку с датой и временем. Именно эта текстовая строка становится значением поля `time`.

Метод `ToString()` переопределен так, что результатом возвращается текст, содержащий значение свойства `Message` объекта исключения, а также значения полей `source` и `time` (с дополнительными текстовыми пояснениями).

В главном классе мы используем вложенные конструкции try-catch. Во внутреннем try-блоке командой int[] a={1} создается массив из одного элемента. После этого размещена команда a[1]=2, при выполнении которой происходит ошибка (обращение к несуществующему элементу массива). Эта ошибка перехватывается и обрабатывается во внутреннем catch-блоке. Обработка состоит в том, что командой MyException me=new MyException(e) создается объект me класса MyException. Объект создается на основе объекта исключения e, переданного для обработки. После этого командой throw me искусственно генерируется исключение пользовательского класса. Это исключение перехватывается во внешнем catch-блоке. Там объект исключения передается аргументом методу WriteLine(), в результате чего для объекта вызывается метод ToString(), и полученное текстовое значение отображается в консольном окне. Хочется верить, что содержимое сообщения комментариев не требует.

Следующая программа содержит класс MyClass, у которого есть целочисленное (тип int) свойство number. Но значение этого свойства сохраняется в закрытом поле типа byte. С помощью инструкции checked и обработки исключений свойство реализовано таким образом, что при попытке присвоить свойству значение, выходящее за допустимые границы для типа byte, свойству присваивается значение 255. Интересующий нас код представлен в листинге 5.12.

Листинг 5.12. Обработка исключений в свойстве

```
using System;
// Класс со свойством:
class MyClass{
   // Закрытое поле типа byte:
   private byte num;
   // Открытое свойство типа int:
   public int number{
      // Аксессор для считывания значения свойства:
      get{
         // Значение свойства:
         return num;
```

```
      }
      // Аксессор для присваивания значения свойству:
      set{
         // Контролируемый код:
         try{
            // Генерирование исключения при переполнении:
            checked{
               // Значение поля:
               num=(byte)value;
            }
         }
         // Обработка исключений:
         catch(OverflowException){
            // Значение поля:
            num=255;
         }
      }
   }
}
// Главный класс:
class CheckedDemo{
   // Главный метод:
   static void Main(){
      // Объект класса:
      MyClass obj=new MyClass();
      // Присваивание значения свойству:
      obj.number=100;
      // Проверка значения свойства:
      Console.WriteLine("Значение свойства: "+obj.number);
      // Присваивание значения свойству:
      obj.number=300;
      // Проверка значения свойства:
```

```
        Console.WriteLine("Значение свойства: "+obj.number);
    }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 5.12)

```
Значение свойства: 100
Значение свойства: 255
```

В классе `MyClass` для свойства `number` `get`-аксессор описан так, что результатом возвращается значение поля `num`. А вот в `set`-аксессоре использована конструкция `try-catch`. В `try`-блоке размещен блок `checked`, а в нем — команда `num=(byte)value`. Если при присваивании значения полю `num` будет иметь место переполнение, то сгенерируется исключение класса `OverflowException`. Для обработки этого исключения предназначен `catch`-блок, в котором командой `num=255` полю `num` присваивается максимально возможное значение `255`.

В главном методе программы командой `MyClass obj=new MyClass()` создается объект `obj` класса `MyClass`. Проверка показывает, что при выполнении команды `obj.number=100` полю `num` присваивается значение `100`, а при выполнении команды `obj.number=300` это поле, как и ожидалось, получает значение `255`.

Резюме

— Это твое заднее слово?
— Заднее не бывает!

из к/ф «Кин-дза-дза»

- При возникновении ошибки в процессе выполнения программы она завершает свою работу. Существует возможность обработки подобных ситуаций. В этом случае используется конструкция `try-catch`.
- Если при выполнении кода в блоке `try` ошибок не было, то `catch`-блок игнорируется. Если при выполнении кода в блоке `try` возникла ошибка, то просматриваются блоки `catch`, связанные с данным

блоком try. Каждый catch-блок предназначен для обработки ошибок определенного типа. Если подходящий catch-блок найден, то выполняется код этого блока. Если нужный catch-блок не найден, то ошибка передается для обработки во внешнюю конструкцию try-catch (если такая есть). Если ошибка не обработана, программа завершает свое выполнение.

- В описании catch-блока указывается класс ошибки, обрабатываемой в блоке. Блок обрабатывает ошибки данного класса, а также всех его подклассов. При необходимости в catch-блоке можно объявить и использовать объект ошибки. Объект ошибки создается автоматически при возникновении ошибки.
- Классы ошибок (исключений) образуют иерархию наследования. На вершине этой иерархии находится класс Exception. У него есть производные классы SystemException и ApplicationException (последний обычно используется при создании классов пользовательских исключений).
- Конструкции try-catch могут быть вложенными. В таком случае, если исключение не обрабатывается во внутренней конструкции, оно передается во внешнюю конструкцию try-catch. Также можно использовать блок finally, который выполняется в обязательном порядке и при возникновении ошибки, и если ошибка не возникла.
- Исключения могут генерироваться вручную. Для этого используют инструкцию throw, после которой указывается имя объекта исключения. Объект исключения можно создать самостоятельно или воспользоваться объектом, созданным автоматически при возникновении ошибки. Для повторного генерирования исключения в catch-блоке используют инструкцию throw, после которой объект исключения не указывается.
- Допускается создавать собственные классы исключений. Такие классы создаются путем наследования классов Exception или ApplicationException.
- Инструкции checked и unchecked позволяют переходить в режим и выходить из режима контроля переполнения при выполнении арифметических операций.

Задания для самостоятельной работы

Я не бездействовал. Я сразу на каппу нажал. Скрипач свидетель.

из к/ф «Кин-дза-дза»

1. Напишите программу, в которой пользователь вводит через консоль размер целочисленного массива. Массив создается, заполняется натуральными числами, а затем содержимое массива отображается в консольном окне. Предусмотреть обработку исключений, связанных с тем, что пользователь ввел некорректное значение для размера массива или ввел нечисловое значение.

2. Напишите программу, в которой пользователь последовательно вводит два целых числа. Программа вычисляет остаток от деления большего числа на меньшее число. Программный код следует написать с учетом обработки возможных ошибок.

3. Напишите программу, в которой уравнение вида $Ax = B$ решается на множестве целых чисел. Решением является такое целое число x, которое, будучи умноженным на целое число A, дает целое число B. Решение существует только в том случае, если целое число B без остатка делится на целое число A или если оба параметра A и B равны нулю. Предусмотреть обработку исключительных ситуаций.

4. Напишите программу, в которой решается квадратное уравнение $Ax^2 + Bx + C = 0$. В общем случае уравнение имеет два решения $x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$. Предложите схему обработки исключительных ситуаций. Для вычисления квадратного корня можно использовать статический метод `Sqrt()` из класса `Math`.

5. Напишите программу, содержащую статический метод, первым аргументом которого является целочисленный массив, а вторым аргументом — целочисленное значение. Методом при вызове выполняется проверка относительно наличия в массиве (первый аргумент) элемента со значением, определяемым вторым аргументом. Если такой элемент есть, то результатом возвращается индекс этого элемента (первого из них, если таких элементов несколько). Если элемента с указанным значением в массиве нет, то метод генерирует исключение. Предложите способ проверки функциональности метода, включая обработку исключений.

6. Напишите программу, содержащую статический метод, не возвращающий результат. Аргументом методу передается целое число. Если число четное, то метод генерирует исключение класса `ArithmeticException`, а если число нечетное, то генерируется исключение класса `OverflowException`. В главном методе выполняется оператор цикла, в котором за каждый цикл пользователь вводит целое число, оно передается аргументом методу. Организовать обработку событий таким образом, чтобы в результате появлялось сообщение о том, четное число или нечетное. Оператор цикла должен завершать работу, если пользователь вводит не число.

7. Напишите программу, содержащую класс с индексатором. Через индексатор можно получить доступ к элементам закрытого целочисленного массива. Используя обработку исключительных ситуаций, опишите статический метод, аргументом которому передается объект класса, а результатом возвращается размер закрытого массива в этом объекте.

8. Напишите программу, в которой описывается класс для ошибки пользовательского типа. У класса должно быть поле, являющееся ссылкой на символьный массив. В качестве аргумента конструктору класса передается целое число, определяющее размер массива. Массив заполняется последовательностью букв (например, начиная с буквы `'A'`). В главном методе программы для реализации символьного массива создается объект исключения пользовательского класса. Для отображения содержимого массива искусственно генерируется исключение, а обработка исключения сводится к отображению содержимого массива из объекта исключения.

9. Напишите программу, в которой описаны два класса пользовательских исключений (один создается наследованием класса `Exception`, а другой создается наследованием класса `ApplicationException`). Опишите класс с закрытым символьным массивом и индексатором, который позволяет присваивать значения элементам массива и считывать значения элементов массива. Определите индексатор таким образом, чтобы при слишком большом индексе (выходящем за верхнюю границу) генерировалось исключение первого пользовательского класса исключений, а при отрицательном индексе генерировалось исключение второго пользовательского класса. В главном методе предложите механизм проверки функциональности индексатора для индексируемых объектов.

10. Напишите программу, содержащую класс с индексатором. При индексировании объекта в качестве значения соответствующего выражения

возвращается целое число и, соответственно, присваивается целое число типа `int`. Присваиваемые значения записываются в массив с элементами типа `byte`. Используя обработку исключительных ситуаций и инструкцию `checked`, опишите индексатор так, чтобы при попытке присвоить проиндексированному объекту значение, выходящее за допустимый диапазон для значений типа `byte`, соответствующий элемент массива получал значение `255`.

Глава 6
МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ

На двух лошадях скакать — седалища не хватит!

из к/ф «Формула любви»

Пришло время обсудить такой мощный и эффектный механизм, как *многопоточное программирование*. Нам предстоит узнать много нового и интересного. Среди тем, рассматриваемых далее, будут такие:

- принципы реализации многопоточных приложений;
- знакомство с классом `Thread`;
- создание дочерних потоков и способы управления ими;
- получение доступа к объекту главного потока;
- создание фоновых потоков;
- знакомство с методами синхронизации.

В процессе обсуждения этих тем мы будем также рассматривать примеры, которые поясняют и иллюстрируют способы практической реализации базовых концепций многопоточного программирования.

Класс `Thread` и создание потоков

Ох, тяжко мне! Молви еще раз, ты не демон?

из к/ф «Иван Васильевич меняет профессию»

Прежде чем приступить к рассмотрению вопросов, связанных с многопоточным программированием, имеет смысл уточнить, что же это такое. Вообще, *потоками* называются разные части программы, которые выполняются одновременно. Именно в одновременном выполнении

нескольких блоков программного кода кроется «изюминка» многопоточного программирования. В языке C# предусмотрены простые и эффективные средства для реализации многопоточного способа программирования. С этими средствами мы намерены познакомиться.

Стратегическая задача состоит из двух подзадач. Во-первых, необходимо определить блоки программного кода, которые будут выполняться одновременно (код, выполняемый в потоке). Во-вторых, необходимо запустить эти коды на выполнение в специальном режиме (чтобы они выполнялись одновременно). Для запуска программного кода на выполнение в режиме потока нам понадобится специальный объект — объект потока. Создается такой объект на основе класса `Thread`.

ПОДРОБНОСТИ

Класс `Thread` определен в пространстве имен `System.Threading`. Класс `Thread` определен с инструкцией `sealed`. Это означает, что данный класс не может быть базовым для создания производных классов.

Создание объекта потока не означает, что поток начинает выполняться. Для запуска потока на выполнение из объекта этого потока необходимо вызвать метод `Start()`.

ПОДРОБНОСТИ

Если метод `Start()` вызывается из объекта потока, который уже запущен на выполнение, то генерируется исключение класса `ThreadStateException`.

То есть здесь все относительно просто: создаем объект класса `Thread` и вызываем из этого объекта метод `Start()`. Но открытым остается вопрос о том, как связать объект потока с программным кодом, который должен выполняться в потоке. Такое «связывание» происходит на этапе создания объекта потока. Происходит это так: при создании объекта потока на основе класса `Thread` в качестве аргумента конструктору передается ссылка на экземпляр делегата `ThreadStart`. Предполагается, что экземпляр делегата содержит ссылку на метод или методы, которые будут вызваны при выполнении потока. Делегат `ThreadStart` определен для методов без аргументов, не возвращающих результат.

ПОДРОБНОСТИ

При создании объекта класса `Thread` аргументом конструктору можно передать ссылку на метод без аргументов, не возвращающий результат. Хотя формально аргументом конструктора должна быть ссылка типа `ThreadStart`, это работает. Принцип такой. Под аргумент конструктора выделяется переменная типа `ThreadStart`. Значением этой переменной присваивается ссылка на метод. В результате создается экземпляр делегата, который ссылается на метод, а ссылка на экземпляр присваивается переменной, выделенной под аргумент конструктора.

Если мы хотим, чтобы в потоке выполнялось несколько методов (последовательно, один за другим), то имеет смысл создать экземпляр делегата `ThreadStart`, записать в него ссылки на соответствующие методы, а ссылку на экземпляр передать аргументом конструктору класса `Thread` (при создании объекта потока).

Простой пример, в котором в программе создается дочерний поток, представлен в листинге 6.1.

Листинг 6.1. Знакомство с потоками

```
using System;
using System.Threading;
// Класс с главным методом:
class ThreadDemo{
   // Статический метод (для вызова в дочернем потоке):
   static void run(){
      Console.WriteLine("Запускаем дочерний поток");
      // Оператор цикла:
      for(int k=0;k<=5;k++){
         // Отображение сообщения:
         Console.WriteLine("Дочерний поток: {0}",(char)('A'+k));
         // Задержка выполнения потока в 1 секунду:
         Thread.Sleep(1000);
      }
      Console.WriteLine("Дочерний поток завершен");
   }
```

```
   // Главный метод:
   static void Main(){
      Console.WriteLine("Запускаем главный поток");
      // Создание объекта дочернего потока:
      Thread mt=new Thread(run);
      // Запуск дочернего потока:
      mt.Start();
      // Оператор цикла:
      for(int k=1;k<=5;k++){
         // Отображение сообщения:
         Console.WriteLine("Главный поток:  "+k);
         // Задержка выполнения потока в 2 секунды:
         Thread.Sleep(2000);
      }
      Console.WriteLine("Главный поток завершен");
   }
}
```

Результат выполнения программы может быть таким:

Результат выполнения программы (из листинга 6.1)

```
Запускаем главный поток
Главный поток:  1
Запускаем дочерний поток
Дочерний поток: A
Дочерний поток: B
Главный поток:  2
Дочерний поток: C
Дочерний поток: D
Главный поток:  3
Дочерний поток: E
Дочерний поток: F
Главный поток:  4
Дочерний поток завершен
```

```
Главный поток:  5
Главный поток завершен
```

Программа достаточно простая. В главном классе описывается статический метод `run()`, который не возвращает результат и у которого нет аргументов (то есть этот метод соответствует библиотечному делегату `ThreadStart`). Мы собираемся выполнять код метода `run()` в режиме дочернего потока. В теле метода командой `Console.WriteLine("Запускаем дочерний поток")` отображается сообщение о начале выполнения дочернего потока, после чего запускается оператор цикла, в котором индексная переменная `k` пробегает значения от `0` до `5` включительно, увеличивая каждый раз свое значение на единицу. За каждую итерацию цикла командой `Console.WriteLine("Дочерний поток: {0}",(char)('A'+k))` в консольном окне отображается сообщение, которое содержит очередную букву латинского алфавита, начиная с буквы `'A'`. После этого командой `Thread.Sleep(1000)` выполняется задержка в выполнении потока длительностью в 1 секунду. Здесь мы вызываем статический метод `Sleep()` из класса `Thread`. Действие метода состоит в том, что выполняется временная остановка в выполнении потока. Время приостановки потока определяется аргументом метода `Sleep()` и измеряется в миллисекундах (1 секунда содержит 1000 миллисекунд).

НА ЗАМЕТКУ

Таким образом, при выполнении метода `run()` в консольном окне отображаются сообщения, но отображаются они с интервалом в 1 секунду.

После завершения оператора цикла командой `Console.WriteLine("Дочерний поток завершен")` отображается сообщение о завершении выполнения потока.

НА ЗАМЕТКУ

Для использования класса `Thread` в программе, кроме пространства имен `System`, подключается еще и пространство имен `System.Threading`.

Выполнение программы начинается с выполнения главного метода. В главном методе программы командой `Console.WriteLine("Запускаем главный поток")` отображается сообщение о начале

выполнения программы (главного потока). Затем с помощью команды `Thread mt=new Thread(run)` создается объект `mt` класса `Thread`. Это объект для дочернего потока. Аргументом конструктору передается имя метода `run()`. Следовательно, при выполнении потока, связанного с объектом `mt`, будет выполняться код метода `run()`.

ПОДРОБНОСТИ

Конструктор класса `Thread` рассчитан на передачу в качестве аргумента ссылки на экземпляр делегата `ThreadStart`. Под эту ссылку выделяется соответствующая переменная (переменная типа делегата `ThreadStart`). По факту аргументом передается ссылка на метод. В результате переменной типа делегата присваивается ссылка на метод. Ситуация обрабатывается следующим образом: создается экземпляр делегата, ссылающийся на данный метод, а ссылка на экземпляр делегата записывается в переменную делегата. Поэтому вместо создания экземпляра делегата и передачи ссылки на этот экземпляр конструктору класса `Thread` мы можем сразу передать аргументом конструктору ссылку на метод.

Но сам по себе факт создания объекта потока не означает, что поток начинает выполняться. Поток следует запустить на выполнение. Для этого из объекта потока вызывается метод `Start()` (команда `mt.Start()`). В этот момент запускается дочерний поток: начинает выполняться метод `run()`. Но поскольку он выполняется в дочернем потоке, то следующая команда в главном потоке начинает выполняться, не дожидаясь окончания выполнения метода `run()`. В главном потоке выполняется оператор цикла. На каждой итерации цикла командой `Console.WriteLine("Главный поток: "+k)` отображается сообщение со значением индексной переменной `k`. Затем командой `Thread.Sleep(2000)` выполняется задержка в 2 секунды в выполнении главного потока. По завершении оператора цикла в главном потоке командой `Console.WriteLine("Главный поток завершен")` в консольном окне отображается сообщение.

НА ЗАМЕТКУ

Получается, что главный и дочерний потоки одновременно отображают сообщения в консольном окне. Но для дочернего потока интервал между сообщениями составляет 1 секунду, а главный поток выводит сообщения с интервалом в 2 секунды. Понятно, что дочерний поток завершает работу раньше главного потока.

Использование потоков

> Кому чего в городе купить? Принимаю заказы до одной тонны.
>
> *из к/ф «Девчата»*

Далее мы рассмотрим различные способы создания потоков в программе. Еще одна программа иллюстрирует подход, при котором в программе создается несколько дочерних потоков, причем на этот раз программный код потоков реализуется на основе нестатических методов. Рассмотрим код в листинге 6.2.

Листинг 6.2. Несколько дочерних потоков

```
using System;
using System.Threading;
// Класс с методом для выполнения в потоке:
class MyClass{
   // Количество сообщений:
   private int count;
   // Время задержки между сообщениями:
   private int time;
   // Текст для отображения:
   private string text;
   // Метод для выполнения в потоке:
   public void show(){
      Console.WriteLine(text+": Начало...");
      for(int i=1;i<=count;i++){
         // Отображение сообщения:
         Console.WriteLine(text+" -> "+i);
         // Задержка в выполнении потока:
         Thread.Sleep(time);
      }
      Console.WriteLine(text+": Завершение...");
   }
```

```
   // Конструктор:
   public MyClass(int c,int t,string txt){
      count=c;
      time=t;
      text=txt;
   }
}
// Главный класс:
class ManyThreadsDemo{
   // Главный метод:
   static void Main(){
      Console.WriteLine("Главный поток: Начало...");
      // Создание объектов:
      MyClass objA=new MyClass(5,2000,"Поток A");
      MyClass objB=new MyClass(6,1200,"Поток B");
      MyClass objC=new MyClass(8,1000,"Поток C");
      // Создание объектов потоков:
      Thread A=new Thread(objA.show);
      Thread B=new Thread(objB.show);
      Thread C=new Thread(objC.show);
      // Запуск дочерних потоков:
      A. Start();
      B. Start();
      C. Start();
      Console.WriteLine("Главный поток: Завершение...");
   }
}
```

Ниже показан возможный результат выполнения программы:

Результат выполнения программы (из листинга 6.2)

```
Главный поток: Начало...
Поток A: Начало...
```

```
Поток B: Начало...
Поток B -> 1
Поток C: Начало...
Поток A -> 1
Поток C -> 1
Главный поток: Завершение...
Поток C -> 2
Поток B -> 2
Поток A -> 2
Поток C -> 3
Поток B -> 3
Поток C -> 4
Поток B -> 4
Поток A -> 3
Поток C -> 5
Поток B -> 5
Поток C -> 6
Поток B -> 6
Поток A -> 4
Поток C -> 7
Поток C -> 8
Поток B: Завершение...
Поток A -> 5
Поток C: Завершение...
Поток A: Завершение...
```

В этой программе в главном потоке создаются три дочерних потока. Дочерние потоки создаются на основе нестатического метода `show()`, описанного в классе `MyClass`. У класса `MyClass` есть:

- закрытое целочисленное поле `count`, определяющее количество сообщений (не считая начального и конечного), отображаемых в дочернем потоке;

- закрытое целочисленное поле `time`, определяющее время приостановки выполнения потока;
- текстовое поле `text`, значение которого определяет текст сообщений, отображаемых в потоке.

Значения полей передаются аргументами конструктору при создании объекта и используются в методе `show()`. В теле конструктора командой `Console.WriteLine(text+": Начало...")` выводится сообщение о начале выполнения метода (потока). Затем запускается оператор цикла (количество итераций цикла определяется значением поля `count`), и на каждой итерации командой `Console.WriteLine(text+"->"+i)` отображается сообщение. Для задержки в выполнении потока использована команда `Thread.Sleep(time)`. После завершения оператора цикла командой `Console.WriteLine(text+": Завершение...")` отображается сообщение о завершении выполнения метода (и, следовательно, потока).

В главном методе отображается сообщение о начале выполнения главного потока (команда `Console.WriteLine("Главный поток: Начало...")`). Далее создаются три объекта `objA`, `objB` и `objC` класса `MyClass`. Это «обычные» объекты. На их основе создаются объекты дочерних потоков `A`, `B` и `C`. Каждый раз аргументом конструктору класса `Thread` передается ссылка на метод `show()` объекта класса `MyClass` (инструкции `objA.show`, `objB.show` и `objC.show`). Для запуска каждого из потоков на выполнение из объекта потока вызывается метод `Start()` (команды `A. Start()`, `B. Start()` и `C. Start()`). В результате одновременно с выполнением главного потока в режиме дочерних потоков начинается выполнение кода метода `show()` из объектов `A` (первый дочерний поток), `B` (второй дочерний поток) и `C` (третий дочерний поток). В это время командой `Console.WriteLine("Главный поток: Завершение...")` в главном потоке выводится сообщение о завершении потока.

НА ЗАМЕТКУ

Параметры, влияющие на выполнение потоков, подобраны так, что главный поток завершается раньше дочерних потоков. По умолчанию дочерние потоки являются приоритетными, поэтому продолжают работу даже при завершении главного потока. Позже мы познакомимся с фоновыми потоками, которые автоматически завершаются при завершении главного потока.

Иногда возникает необходимость в одном потоке (например, главном) выполнить определенную команду только после завершения другого потока. В таких случаях используют свойство `IsAlive`, которое возвращает значение `true`, если поток (для объекта, из которого запрашивается свойство) еще выполняется. Если поток завершен, значение свойства равно `false`.

Вызов метода `Join()` означает ожидание завершения потока, из объекта которого вызван метод.

НА ЗАМЕТКУ

Если вызвать метод `Join()` из объекта потока, который еще не стартовал, генерируется исключение класса `ThreadStateException`.

Программа, в которой используются свойство `IsAlive` и метод `Join()`, представлена в листинге 6.3.

Листинг 6.3. Координация завершения потоков

```
using System;
using System.Threading;
// Главный класс:
class JoinDemo{
    // Метод для дочернего потока:
   static void run(){
      Console.WriteLine("Дочерний поток запущен...");
      // Оператор цикла:
      for(int k=1;k<=5;k++){
         // Отображение сообщения:
         Console.WriteLine("Дочерний поток: "+k);
         // Задержка в выполнении потока:
         Thread.Sleep(2500);
      }
      Console.WriteLine("Дочерний поток завершен...");
   }
   // Главный метод:
```

```
   static void Main(){
      // Начальное сообщение главного потока:
      Console.WriteLine("Главный поток запущен...");
      // Объект дочернего потока:
      Thread t=new Thread(run);
      // Запуск дочернего потока:
      t.Start();
      // Оператор цикла:
      for(char s='A';s<='F';s++){
         // Отображение сообщения:
         Console.WriteLine("Главный поток: "+s);
         // Задержка в выполнении потока:
         Thread.Sleep(1000);
      }
      // Ожидание завершения дочернего потока:
      if(t.IsAlive) t.Join();
      // Последнее сообщение главного потока:
      Console.WriteLine("Главный поток завершен...");
   }
}
```

Возможный результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 6.3)

```
Главный поток запущен...
Дочерний поток запущен...
Дочерний поток: 1
Главный поток: A
Главный поток: B
Главный поток: C
Дочерний поток: 2
Главный поток: D
```

```
Главный поток: E
Дочерний поток: 3
Главный поток: F
Дочерний поток: 4
Дочерний поток: 5
Дочерний поток завершен...
Главный поток завершен...
```

В главном методе программы запускается дочерний поток. В дочернем потоке выполняется статический метод `run()`. И дочерний, и главный потоки выводят в консольное окно сообщения. Параметры выполнения потоков подобраны таким образом, что главный поток выполняется значительно быстрее. Но в главном потоке размещен условный оператор, в котором проверяется значение свойства `IsAlive` объекта дочернего потока `t`. Если условие истинно (то есть если дочерний поток еще выполняется), то выполняется команда `t.Join()`. Выполнение команды приводит к тому, что главный поток ожидает выполнения дочернего потока, и только после этого в главном потоке выполняется команда `Console.WriteLine("Главный поток завершен...")`.

Следующая программа, представленная в листинге 6.4, иллюстрирует способ создания дочерних потоков на основе экземпляра делегата `ThreadStart` (имеется в виду явное создание экземпляра делегата и его передача в качестве аргумента конструктору класса `Thread`), а также на основе лямбда-выражения (когда лямбда-выражение передается аргументом конструктору класса `Thread`).

Листинг 6.4. Поток на основе экземпляра делегата

```
using System;
using System.Threading;
// Класс с методом для запуска в потоке:
class MyClass{
   // Закрытое текстовое поле:
   private string text;
   // Закрытое целочисленное поле:
   private int time;
```

```
   // Конструктор:
   public MyClass(string txt,int t){
      text=txt;
      time=t;
   }
   // Метод без аргументов:
   public void show(){
      for(int k=1;k<=5;k++){
         // Отображение сообщения:
         Console.WriteLine(text+"\t"+k);
         // Задержка в выполнении потока:
         Thread.Sleep(time);
      }
   }
}
// Главный класс:
class ThreadStartDemo{
   // Статический метод с двумя аргументами:
   static void run(string txt,int time){
      for(int k=1;k<=5;k++){
         // Отображение сообщения:
         Console.WriteLine(txt+"\t"+k);
         // Задержка в выполнении потока:
         Thread.Sleep(time);
      }
   }
   // Главный метод:
   static void Main(){
      // Начальное сообщение в главном потоке:
      Console.WriteLine("Главный поток запущен...");
      // Создание объектов класса MyClass:
      MyClass A=new MyClass("Объект A",1000);
```

```
        MyClass B=new MyClass("Объект B",1500);
        // Создание экземпляра делегата:
        ThreadStart ts=A.show;
        // Добавление метода в список вызовов
        // экземпляра делегата:
        ts+=B.show;
        // Создание объекта потока на основе экземпляра
        // делегата:
        Thread first=new Thread(ts);
        // Создание объекта потока на основе
        // лямбда-выражения:
        Thread second=new Thread(()=>{
           run("Метод Alpha",1200);
           run("Метод Bravo",800);
        });
        // Запуск первого потока:
        first.Start();
        // Запуск второго потока:
        second.Start();
        // Ожидание окончания выполнения первого потока:
        if(first.IsAlive) first.Join();
        // Ожидание окончания выполнения второго потока:
        if(second.IsAlive) second.Join();
        // Последнее сообщение в главном потоке:
        Console.WriteLine("Главный поток завершен...");
    }
}
```

Ниже показано, как может выглядеть результат выполнения программы:

Результат выполнения программы (из листинга 6.4)

```
Главный поток запущен...
Объект A        1
```

```
Метод Alpha     1
Объект A        2
Метод Alpha     2
Объект A        3
Метод Alpha     3
Объект A        4
Метод Alpha     4
Объект A        5
Метод Alpha     5
Объект B        1
Метод Bravo     1
Объект B        2
Метод Bravo     2
Метод Bravo     3
Объект B        3
Метод Bravo     4
Метод Bravo     5
Объект B        4
Объект B        5
Главный поток завершен...
```

У нас есть класс `MyClass` с методом `show()`, который выводит в консольное окно сообщения. Текст сообщения и задержка между сообщениями определяются полями объекта, из которого вызывается метод. При этом у метода нет аргументов и метод не возвращает результат.

В главном классе описан статический метод `run()`. Метод также выводит сообщения в консольное окно. Он не возвращает результат, но у него два аргумента (текстовый и целочисленный). Аргументы определяют текст сообщений и задержку между этими сообщениями.

В главном методе создаются объекты `A` и `B` класса `MyClass`. Командой `ThreadStart ts=A.show` мы создаем экземпляр `ts` делегата `ThreadStart`, который ссылается на метод `show()` объекта `A`. После этого командой `ts+=B.show` в список вызовов экземпляра делегата добавляется еще и ссылка на метод `show()` объекта `B`.

На основе экземпляра ts создается объект first для первого дочернего потока (команда Thread first=new Thread(ts)). При запуске этого потока командой first.Start() сначала вызывается метод show() из объекта A, а по завершении выполнения метода вызывается метод show() из объекта B.

> **НА ЗАМЕТКУ**
>
> Таким образом, в первом дочернем потоке сначала выполняется метод show() из объекта A, а затем выполняется метод show() из объекта B.

Еще один дочерний поток создается на основе лямбда-выражения. При создании объекта second классу Thread аргументом передается такое лямбда-выражение:

```
()=>{
   run("Метод Alpha",1200);
   run("Метод Bravo",800);
}
```

Это лямбда-выражение определяет метод без аргументов, при вызове которого последовательно вызывается метод run() с разными аргументами. Поэтому при запуске второго дочернего потока на выполнение командой second.Start() сначала вызывается метод run() с аргументами "Метод Alpha" и 1200, а затем этот же метод run() вызывается с аргументами "Метод Bravo" и 800.

> **НА ЗАМЕТКУ**
>
> Второй дочерний поток выполняется следующим образом: сначала выполняется команда run("Метод Alpha",1200), а после того как команда выполнена, выполняется команда run("Метод Bravo",800).

Перед отображением последнего сообщения в главном потоке ожидается выполнение первого и второго дочерних потоков.

Фоновые потоки

> Если мы допустим беспорядок в документации, потомки нам этого не простят.
>
> *из к/ф «Гостья из будущего»*

Как отмечалось ранее, если главный поток завершает работу, то дочерние потоки автоматически не завершаются. Такие дочерние потоки называются *приоритетными*. Однако есть категория потоков, которые называются *фоновыми*. Их особенность в том, что при завершении главного потока фоновые потоки завершаются автоматически.

У каждого объекта потока есть свойство `IsBackground`. Свойство может принимать значение `true` или `false`. Если свойство `IsBackground` объекта потока равно `true`, то поток является фоновым. Если значение свойства `IsBackground` объекта потока равно `false`, то поток является приоритетным. По умолчанию значение свойства `IsBackground` для объектов потоков равно `false`, поэтому все потоки, которые мы создавали ранее, были приоритетными. Если мы хотим создать фоновый поток, то необходимо свойству `IsBackground` объекта этого потока присвоить значение `true`. Пример создания фонового потока представлен в программе в листинге 6.5.

Листинг 6.5. Фоновый поток

```
using System;
using System.Threading;
// Класс с главным методом:
class BackgroundDemo{
   // Статический метод для выполнения в фоновом потоке:
   static void run(){
      // Отображение сообщения:
      Console.WriteLine("Дочерний поток запущен...");
      // Целочисленная переменная:
      int k=1;
```

```
        // Бесконечный цикл:
        while(true){
            // Отображение сообщения:
            Console.WriteLine("Дочерний поток: "+k);
            // Увеличение значения переменной:
            k++;
            // Задержка в выполнении потока:
            Thread.Sleep(2100);
        }
    }
    // Главный метод:
    static void Main(){
        // Отображение начального сообщения:
        Console.WriteLine("Главный поток запущен...");
        // Создание объекта потока:
        Thread t=new Thread(run);
        // Фоновый поток:
        t.IsBackground=true;
        // Запуск фонового потока:
        t.Start();
        // Оператор цикла:
        for(char s='A';s<='F';s++){
            // Отображение сообщения:
            Console.WriteLine("Главный поток:  "+s);
            // Задержка в выполнении главного потока:
            Thread.Sleep(1000);
        }
        // Отображение последнего сообщения:
        Console.WriteLine("Главный поток завершен...");
    }
}
```

Возможный результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 6.5)

```
Главный поток запущен...
Главный поток:  A
Дочерний поток запущен...
Дочерний поток: 1
Главный поток:  B
Главный поток:  C
Дочерний поток: 2
Главный поток:  D
Главный поток:  E
Дочерний поток: 3
Главный поток:  F
Главный поток завершен...
```

Особенность этого примера в том, что метод `run()`, который выполняется в дочернем потоке, содержит бесконечный цикл (в операторе `while` условием указано `true`). Теоретически такой метод (или поток на его основе) должен был бы работать бесконечно. Но поток, который создается на основе метода `run()`, является фоновым. Он становится фоновым после выполнения команды `t.IsBackground=true`, где `t` является объектом потока. Поэтому после запуска (команда `t.Start()`) дочерний фоновый поток выполняется до тех пор, пока выполняется породивший его главный поток. При завершении главного потока дочерний фоновый поток завершается автоматически.

Обычно фоновые потоки используются для решения вспомогательных задач.

Операции с потоками

> Боже, какой типаж! Браво, браво! Слушайте, я не узнаю вас в гриме! Кто вы такой?
>
> *из к/ф «Иван Васильевич меняет профессию»*

Существуют свойства и методы, которые позволяют выполнять различные операции с потоками. Например, у класса `Thread` есть статическое

свойство CurrentThread. Значением этого свойства является ссылка на объект потока, в котором запрашивается свойство.

НА ЗАМЕТКУ

Например, мы имели дело с главным потоком, но никогда ранее не упоминали объект главного потока. Меж тем он существует, и к нему можно получить доступ. Для этого в главном потоке следует воспользоваться свойством CurrentThread класса Thread.

У объекта потока есть свойство Name, определяющее название потока. Свойство Priority объекта потока определяет приоритет потока. Возможные значения свойства — константы Highest, AboveNormal, Normal (значение по умолчанию), BelowNormal и Lowest из перечисления ThreadPriority.

НА ЗАМЕТКУ

Приоритет потока влияет на доступ потока к системным ресурсам, особенно если несколько потоков конкурируют за один и тот же ресурс. Обычно определяющим является не само значение приоритета, а то, у какого из потоков приоритет выше.

С помощью метода Abort() можно завершить выполнение потока, из объекта которого вызывается метод. Небольшой пример, в котором используются перечисленные выше свойства, представлен в листинге 6.6.

Листинг 6.6. Операции с потоками

```
using System;
using System.Threading;
// Главный класс:
class CurrentThreadDemo{
   // Статическое поле:
   static int state=0;
   // Статический метод:
   static void run(bool type){
      // Ссылка на объект потока,
      // в котором вызывается метод:
```

```
    Thread t=Thread.CurrentThread;
    // Отображение сообщения с названием потока:
    Console.WriteLine("Поток "+t.Name+" запущен...");
    // Бесконечный цикл:
    while(true){
       // Изменение значения статического поля:
       if(type) state++;
       else state--;
       // Поток приостанавливает выполнение:
       Thread.Sleep(1000);
    }
}
// Главный метод:
static void Main(){
    // Получение ссылки на главный поток:
    Thread t=Thread.CurrentThread;
    // Название потока:
    t.Name="Base";
    // Отображение сообщения с названием потока:
    Console.WriteLine("Главный поток {0} запущен...",t.Name);
    // Отображение значения статического поля:
    Console.WriteLine("Начальное значение: {0}",state);
    // Создание объекта для первого дочернего потока:
    Thread up=new Thread(()=>run(true));
    // Создание объекта для второго дочернего потока:
    Thread down=new Thread(()=>run(false));
    // Название для первого потока:
    up.Name="Alpha";
    // Приоритет для первого потока:
    up.Priority=ThreadPriority.Highest;
    // Название для второго потока:
    down.Name="Bravo";
```

```
        // Приоритет для второго потока:
        down.Priority=ThreadPriority.Lowest;
        // Запуск первого потока:
        up.Start();
        // Запуск второго потока:
        down.Start();
        // Приостановлено выполнение главного потока:
        Thread.Sleep(5000);
        // Завершение выполнения первого потока:
        up.Abort();
        // Завершение выполнения второго потока:
        down.Abort();
        // Итоговое значение статического поля:
        Console.WriteLine("Итоговое значение: {0}",state);
        // Сообщение о завершении главного потока:
        Console.WriteLine("Главный поток {0} завершен...",t.Name);
    }
}
```

Результат выполнения программы может быть таким:

Результат выполнения программы (из листинга 6.6)

```
Главный поток Base запущен...
Начальное значение: 0
Поток Alpha запущен...
Поток Bravo запущен...
Итоговое значение: -1
Главный поток Base завершен...
```

В главном классе объявляется статическое целочисленное поле `state` с начальным нулевым значением. Еще у нас есть статический метод `run()` с аргументом логического типа (обозначен как `type`). В теле метода командой `Thread t=Thread.CurrentThread` объявляется

объектная переменная t класса Thread, а в качестве значения переменной присваивается ссылка на объект потока, в котором выполняется метод. Ссылку на объект потока мы получаем с помощью статического свойства CurrentThread класса Thread. После этого появляется сообщение с названием потока, в котором выполняется метод. Название потока получаем с помощью инструкции t.Name. При этом мы приняли во внимание, что переменная t ссылается на объект потока. После отображения сообщения запускается оператор цикла while, в котором условием указано значение true. Поэтому формально получаем бесконечный цикл (оператор цикла, который не останавливается). Если метод run() вызван с аргументом true, то за каждую итерацию цикла значение поля state увеличивается на единицу. Если метод run() вызван с аргументом false, то за каждую итерацию значение поля state уменьшается на единицу. После каждого изменения значения поля state выполняется пауза в 1 секунду (команда Thread.Sleep(1000)).

В главном методе программы командой Thread t=Thread.CurrentThread в переменную t записывается ссылка на главный поток программы.

НА ЗАМЕТКУ

В методе run() была аналогичная команда. Но, во-первых, переменная t в методе run() и переменная t в главном методе — это совершенно разные переменные. Во-вторых, значением инструкции Thread.CurrentThread является ссылка на объект потока, в котором эта инструкция выполняется. Поэтому в методе run() в переменную t записывается ссылка на объект потока, в котором выполняется метод run(), а в главном методе в переменную t записывается ссылка на объект главного потока.

Командой t.Name="Base" главному потоку присваивается название "Base". Отображается сообщение о запуске главного потока и начальное значение поля state. Дочерние потоки (объекты) создаются командами Thread up=new Thread(()=>run(true)) и Thread down=new Thread(()=>run(false)). В этих командах методы, выполняемые в потоках, определяются с помощью лямбда-выражений. В итоге в методе up будет выполняться метод run() с аргументом true (поток увеличивает значение поля state), а в потоке down будет выполняться метод run() с аргументом false (поток уменьшает значение поля state). Перед запуском для потоков задаются названия

(команды `up.Name="Alpha"` и `down.Name="Bravo"`) и приоритет (команды `up.Priority=ThreadPriority.Highest` и `down.Priority=ThreadPriority.Lowest`). Для запуска потоков на выполнение используем команды `up.Start()` и `down.Start()`.

> **НА ЗАМЕТКУ**
>
> Откровенно говоря, в данном случае приоритет потоков мало на что влияет. Команды, определяющие приоритет потоков, приведены исключительно как иллюстрация к способу использования соответствующих свойств объекта потока, не более. Мы останавливаем выполнение дочерних потоков из главного потока.

Командой `Thread.Sleep(5000)` выполняется приостановка выполнения главного потока на 5 секунд, после чего командами `up.Abort()` и `down.Abort()` завершается выполнение дочерних потоков.

> **НА ЗАМЕТКУ**
>
> Напомним, что метод `run()` содержит бесконечный цикл, а созданные потоки не являются фоновыми, поэтому они должны были бы выполняться бесконечно долго.

После завершения работы дочерних потоков проверяется значение поля `state`. От запуска к запуску это значение может меняться, но обычно оно не сильно отличается от начального значения `0` (поскольку каждый из дочерних потоков успевает выполнить примерно одинаковое количество циклов по изменению значения поля). Завершается выполнение главного потока отображением сообщения соответствующего содержания.

Синхронизация потоков

> Ни минуты покоя. Даже в воскресенье не могут вернуться не все сразу.
>
> *из к/ф «Гостья из будущего»*

Обычно разные потоки используют общие ресурсы. Скажем, в предыдущем примере два потока одновременно пытались изменить значение одного и того же статического поля. Чтобы понять, какие потенциальные

проблемы могут возникнуть в подобных ситуациях, рассмотрим пример, не имеющий прямого отношения к программированию.

Предположим, что имеется банковский счет, на котором есть определенная сумма денег. Доступ к этому счету имеет несколько человек, которые могут снимать деньги со счета и пополнять счет. Пускай для определенности на счету находится сумма в 1000 денежных единиц (не важно каких). Два клиента пытаются одновременно выполнить такие операции:

- первый клиент пытается внести на счет сумму в 100 денежных единиц;
- второй клиент пытается снять со счета сумму в 100 денежных единиц.

Итог операции должен быть нулевой: сколько денег снято, столько же и поступает на счет. Но теперь рассмотрим технологию процесса. Деньги снимаются и зачисляются через удаленные терминалы. Операция сводится к приему и выдаче наличных и изменению записи о состоянии счета. При внесении наличных на счет сначала выполняется запрос о получении текущего состояния счета, затем об увеличении этого значения на 100 (сумма, перечисляемая на счет) и, наконец, о выполнении записи о новом состоянии счета. При снятии наличных также считывается текущая сумма на счете, эта сумма уменьшается на 100 (то, что снимает со счета клиент), и новое значение записывается как состояние счета. Например, если сначала выполняется операция по зачислению средств, то будет прочитано значение 1000 (начальное состояние счета), вычислена сумма 1100 и это значение записано в качестве новой суммы средств на счете. Далее со счета снимаются наличные: считывается сумма 1100 на счете, вычисляется новое значение 1000, и оно записывается как текущее состояние счета. Если поменять порядок выполнения операций (сначала снять деньги, а потом внести такую же сумму на счет), конечный результат не изменится. Но давайте представим, что оба процесса (снятие наличных и перечисление наличных на счет) выполняются практически одновременно. Скажем, при зачислении денег на счет считывается сумма 1000 и до того, как будет записано новое состояние счета, выполняется снятие денег со счета. Второй процесс также считает текущую сумму 1000. Первый процесс вычислит новое значение 1100, а второй процесс вычислит новое значение 900. Дальше все зависит от того, какой процесс быстрее сделает запись о новом состоянии счета. Если первый процесс окажется быстрее, то сначала будет записано значение 1100, а затем второй процесс исправит это значение на 900.

Если задачу быстрее решит второй процесс, то итоговая сумма окажется равной 1100.

В чем тут проблема? Проблема в том, что один процесс вмешался в работу другого процесса. Какой выход из ситуации? Можно заблокировать ресурс (в данном случае это счет): пока с ресурсом работает один процесс, другим процессам доступ к ресурсу запрещен. Как только первый процесс завершил работу с ресурсом, с ним могут работать другие процессы (но в том же режиме — пока процесс работает с ресурсом, другие процессы к ресурсу доступа не имеют).

Похожая ситуация может возникнуть в программировании при одновременном обращении нескольких потоков к одному общему ресурсу (например, объекту). Если мы хотим избежать проблем из-за одновременного доступа нескольких потоков к общему объекту, этот объект можно заблокировать на время выполнения потока. Для этого используют инструкцию `lock`. В круглых скобках после инструкции указывается блокируемый объект, а в фигурных скобках — блок команд, при выполнении которых объект блокируется. Общий шаблон использования инструкции `lock` в этом случае следующий:

```
lock(объект){
   // Команды
}
```

Пример, в котором используется блокировка объекта во время доступа к нему потоков, представлен в листинге 6.7.

Листинг 6.7. Синхронизация потоков

```
using System;
using System.Threading;
// Класс для создания объекта:
class MyClass{
   // Открытое целочисленное поле:
   public int state=0;
}
// Класс с главным методом:
class LockDemo{
```

```
   // Метод для выполнения в потоке:
   static void run(bool type,MyClass obj){
      // Целочисленные переменные:
      int val,k=1;
      // Объект для генерирования случайных чисел:
      Random rnd=new Random();
      // Оператор цикла:
      while(k<=5){
         // Блокировка объекта:
         lock(obj){
            // Считывание значения поля:
            val=obj.state;
            // Отображение прочитанного значения:
            Console.WriteLine("{0,4}: прочитано значение {1,2}",
Thread.CurrentThread.Name,val);
            // Случайная задержка в выполнении потока:
            Thread.Sleep(rnd.Next(1000,3001));
            // Новое значение переменной:
            if(type) val++;
            else val--;
            // Полю объекта присваивается новое значение:
            obj.state=val;
            // Отображение нового значения поля:
            Console.WriteLine("{0,4}: присвоено значение {1,2}",
Thread.CurrentThread.Name,obj.state);
            // Новое значение индексной переменной:
            k++;
         }
      }
   }
   // Главный метод:
   static void Main(){
```

```
      // Создание объекта:
      MyClass obj=new MyClass();
      // Объект первого потока:
      Thread up=new Thread(()=>run(true,obj));
      // Название первого потока:
      up.Name="UP";
      // Создание объекта второго потока:
      Thread down=new Thread(()=>run(false,obj));
      // Название второго потока:
      down.Name="DOWN";
      // Запуск первого потока на выполнение:
      up.Start();
      // Запуск второго потока на выполнение:
      down.Start();
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 6.7)

```
  UP: прочитано значение  0
  UP: присвоено значение  1
DOWN: прочитано значение  1
DOWN: присвоено значение  0
  UP: прочитано значение  0
  UP: присвоено значение  1
DOWN: прочитано значение  1
DOWN: присвоено значение  0
  UP: прочитано значение  0
  UP: присвоено значение  1
DOWN: прочитано значение  1
```

```
DOWN: присвоено значение  0
  UP: прочитано значение  0
  UP: присвоено значение  1
DOWN: прочитано значение  1
DOWN: присвоено значение  0
  UP: прочитано значение  0
  UP: присвоено значение  1
DOWN: прочитано значение  1
DOWN: присвоено значение  0
```

В программе описан класс `MyClass`, у которого есть целочисленное открытое поле `state` (с начальным нулевым значением).

> **НА ЗАМЕТКУ**
>
> Нестатическое поле `state` инициализировано начальным нулевым значением непосредственно в инструкции объявления поля в классе. Это означает, что у всех объектов класса `MyClass` при создании поле `state` будет получать начальное нулевое значение.

Класс `MyClass` нам нужен для того, чтобы на его основе создать объект, за который будут конкурировать дочерние потоки. Дочерние потоки создаются на основе статического метода `run()`, описанного в главном классе (точнее, потоки создаются на основе анонимных методов, в которых вызывается метод `run()`). У метода два аргумента: аргумент `type` логического типа и аргумент `obj`, который является ссылкой на объект класса `MyClass`. При выполнении метода запускается оператор цикла `while`, в котором командой `val=obj.state` в переменную `val` записывается текущее значение поля `state` объекта `obj` (объект, переданный методу в качестве аргумента). Это значение отображается в консольном окне. Затем выполняется задержка в выполнении потока. Время задержки является случайной величиной в диапазоне от 1 до 3 секунд. Далее, в зависимости от значения первого логического аргумента метода, значение переменной `val` увеличивается или уменьшается на единицу, это новое значение присваивается полю `state` объекта `obj` (команда `obj.state=val`), после чего новое значение поля отображается в консольном окне.

ПОДРОБНОСТИ

Для генерирования случайных чисел командой `Random rnd=new Random()` создается объект `rnd` класса `Random`. Результатом выражения `rnd.Next(1000,3001)` является целое случайное число в диапазоне возможных значений от `1000` до `3000`.

Инструкции `{0,4}` и `{1,2}` в строке форматирования в методе `WriteLine()` означают, что для отображения первого (после строки форматирования) аргумента выделяется не менее `4` позиций, а для отображения второго аргумента выделяется не менее `2` позиций.

Значением выражения `Thread.CurrentThread.Name` является имя потока, в котором вычисляется это выражение. Здесь уместно напомнить, что статическое свойство `CurrentThread` класса `Thread` результатом возвращает ссылку на объект потока, в котором вычисляется значение свойства.

Но самое важное — это то, что все описанные выше действия выполняются внутри блока на основе `lock`-инструкции: блок начинается с выражения `lock(obj)`, означающего, что на время выполнения команд в следующем после этого выражения блоке объект `obj` блокируется для доступа из других потоков.

В главном методе программы создается объект `obj` класса `MyClass`. Также создаются объекты `up` и `down` класса `Thread`. В обоих случаях объекты потоков создаются на основе лямбда-выражений. В потоке `up` выполняется метод `run()` с аргументами `true` и `obj`. В потоке `down` выполняется метод `run()` с аргументами `false` и `obj`. Поэтому при выполнении потока `up` выполняются операции по увеличению значения поля `state` объекта `obj`. Напротив, поток `down` выполняет операции по уменьшению значения поля `state` объекта `obj`.

Командами `up.Name="UP"` и `down.Name="DOWN"` задаются названия для потоков, а командами `up.Start()` и `down.Start()` потоки запускаются на выполнение.

Что можно сказать о результате выполнения программы? Сообщения от потоков выводятся парами, а конечное значение поля `state` объекта `obj` совпадает с начальным нулевым значением этого поля. Это последствия блокирования объекта `obj` при работе с ним потоков.

Для сравнения интересно посмотреть, каким может быть результат выполнения программы, если не использовать блокировку объекта `obj`.

Для этого в программном коде (см. Листинг 6.7) следует закомментировать инструкцию `lock(obj)` (с открывающей фигурной скобкой) и закрывающую фигурную скобку блока в теле метода `run()`. Соответствующие места в программном коде выделены жирным шрифтом. Возможный результат выполнения программы в этом случае может выглядеть так, как показано ниже:

Результат выполнения программы (из листинга 6.7)

```
  UP: прочитано значение  0
DOWN: прочитано значение  0
  UP: присвоено значение  1
  UP: прочитано значение  1
DOWN: присвоено значение -1
DOWN: прочитано значение -1
DOWN: присвоено значение -2
  UP: присвоено значение  2
  UP: прочитано значение  2
DOWN: прочитано значение  2
  UP: присвоено значение  3
  UP: прочитано значение  3
DOWN: присвоено значение  1
DOWN: прочитано значение  1
  UP: присвоено значение  4
  UP: прочитано значение  4
DOWN: присвоено значение  0
DOWN: прочитано значение  0
  UP: присвоено значение  5
DOWN: присвоено значение -1
```

Общий вывод состоит в том, что условие равенства начального и конечного значений поля `state` объекта `obj` не выполнено. И это является следствием нескоординированной работы потоков с общим ресурсом, которым в данном случае является объект `obj`.

Использование потоков

— Как это вы успеваете, Филипп Филиппович?
— Успевает всюду тот, кто никуда не торопится.

из к/ф «Собачье сердце»

В этом разделе мы рассмотрим несколько примеров, в которых используются потоки. В примере, представленном в листинге 6.8, два дочерних потока заполняют символьный массив. Один из потоков заполняет массив с конца до начала, присваивая элементам массива кириллические буквы. Второй поток заполняет массив с начала и до конца, присваивая элементам массива латинские буквы. Потоки работают, пока не «встретятся» где-то посредине массива. Рассмотрим представленный далее программный код.

Листинг 6.8. Заполнение массива потоками

```
using System;
using System.Threading;
// Главный класс:
class FillingArrayDemo{
   // Главный метод:
   static void Main(){
      // Размер массива:
      int size=20;
      // Создание массива:
      char[] symbs=new char[size];
      // Заполнение массива "звездочками":
      for(int k=0;k<symbs.Length;k++){
         symbs[k]='*';
         Console.Write("|"+symbs[k]);
      }
      Console.WriteLine("|");
      // Индекс первого и последнего элемента в массиве:
      int first=0,second=symbs.Length-1;
      // Объектные переменные для потоков:
```

```
Thread A,B;
// Создание объекта для первого потока:
A=new Thread(()=>{
   // Начальный символ для заполнения массива:
   char start='Я';
   // Бесконечный цикл:
   while(true){
      // Если второй индекс больше первого индекса:
      if(second>first){
         // Значение элемента:
         symbs[second]=start;
         // Новый символ для присваивания:
         start--;
         // Новое значение индекса:
         second--;
         // Задержка в выполнении потока:
         Thread.Sleep(100);
      }
      // Если второй индекс не больше первого:
      else{
         // Завершение в выполнении потока:
         Thread.CurrentThread.Abort();
      }
   }
});
// Создание объекта для второго потока:
B=new Thread(()=>{
   // Начальный символ для заполнения массива:
   char start='А';
   // Бесконечный цикл:
   while(true){
      // Ессли первый индекс меньше второго:
```

```
            if(first<second){
               // Значение элемента:
               symbs[first]=start;
               // Новый символ для присваивания:
               start++;
               // Новое значение индекса:
               first++;
               // Задержка в выполнении потока:
               Thread.Sleep(100);
            }
            // Если первый индекс не меньше второго:
            else{
               // Завершение выполнения потока:
               Thread.CurrentThread.Abort();
            }
         }
      });
      // Запуск первого потока на выполнение:
      A. Start();
      // Запуск второго потока на выполнение:
      B. Start();
      // Ожидание выполнения первого потока:
      if(A. IsAlive) A. Join();
      // Ожидание выполнения второго потока:
      if(B. IsAlive) B. Join();
      // Отображение содержимого массива:
      for(int k=0;k<symbs.Length;k++){
         Console.Write("|"+symbs[k]);
      }
      Console.WriteLine("|");
   }
}
```

Результат выполнения программы может быть таким, как показано ниже:

Результат выполнения программы (из листинга 6.8)

```
|*|*|*|*|*|*|*|*|*|*|*|*|*|*|*|*|*|*|*|*|*|*|*|*|*|*|
|A|B|C|D|E|F|G|H|I|*|Ц|Ч|Ш|Щ|Ъ|Ы|Ь|Э|Ю|Я|
```

После создания символьного массива `symbs` он заполняется символом `'*'` (звездочка). В целочисленную переменную `first` записывается значение первого элемента массива, а в переменную `second` записывается значение последнего элемента массива. Первый поток создается на основе лямбда-выражения, определяющего метод, который выполняется при запуске первого потока на выполнение. В этом методе символьная переменная `start` получает начальное значение `'Я'`, после чего запускается формально бесконечный цикл. В теле оператора цикла, при истинности условия `second>first`, выполняются следующие команды:

- Текущее значение переменной `start` присваивается элементу массива `symb` с индексом `second` (команда `symbs[second]=start`).
- Вследствие выполнения команды `start--` переменная `start` значением получает предыдущую (по отношению к текущему значению) букву из алфавита.
- Командой `second--` значение переменной `second` уменьшается на единицу.
- Чтобы поток не выполнил работу слишком быстро, командой `Thread.Sleep(100)` выполняем небольшую задержку в выполнении потока.

Если же при проверке условия `second>first` оно окажется ложным, то в `else`-блоке условного оператора командой `Thread.CurrentThread.Abort()` завершается выполнение потока. Здесь мы с помощью статического свойства `CurrentThread` класса `Thread` получили ссылку на объект потока, в котором выполняется метод, и из этого объекта вызвали метод `Abort()`, в результате чего поток прекращает работу.

Второй поток выполняется похожим образом, но элементы заполняются с начала до конца, и для заполнения используются латинские буквы.

Оба потока выполняются до тех пор, пока значение переменной `first` меньше значения переменной `second`. Значение переменной `first` увеличивается в процессе заполнения массива, а значение переменной `second` уменьшается. Параметры программы подобраны так, что наиболее вероятный сценарий следующий: в какой-то момент значения переменных `first` и `second` станут одинаковыми, и потоки завершат свое выполнение. При этом элемент, индекс которого равен совпадающим значениям переменных `first` и `second`, останется со старым значением — ни один из потоков не присвоит новое значение этому элементу.

В следующей программе создается двумерный числовой массив, и этот массив построчно заполняется. Заполнение каждой строки выполняется отдельным потоком. Объектные переменные, через которые реализуются ссылки на потоки, организованы в массив. Это общая идея. Далее рассмотрим способ ее реализации. Интересующий нас программный код представлен в листинге 6.9.

Листинг 6.9. Массив объектных переменных для потоков

```
using System;
using System.Threading;
// Главный класс:
class ThreadArrayDemo{
   // Главный метод:
   static void Main(){
      // Двумерный массив:
      int[,] nums=new int[6,9];
      // Массив из объектных переменных для потоков:
      Thread[] t=new Thread[nums.GetLength(0)];
      // Перебор элементов массива:
      for(int i=0;i<t.Length;i++){
         // Локальная переменная для цикла:
         int p=i;
         // Создание объекта потока:
         t[i]=new Thread(()=>{
            // Перебор элементов в строке
```

```
            // целочисленного массива:
            for(int j=0;j<nums.GetLength(1);j++){
               // Элементу присваивается значение:
               nums[p,j]=(p+1)*(j+1);
               // Приостановка выполнения потока:
               Thread.Sleep(100);
            }
         });
         // Запуск потока на выполнение:
         t[i].Start();
      }
      // Ожидание завершения дочерних потоков:
      for(int i=0;i<t.Length;i++){
         if(t[i].IsAlive) t[i].Join();
      }
      // Отображение содержимого двумерного целочисленного
      // массива:
      for(int i=0;i<nums.GetLength(0);i++){
         for(int j=0;j<nums.GetLength(1);j++){
            Console.Write("{0,-4}",nums[i,j]);
         }
         Console.WriteLine();
      }
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 6.9)

```
1   2   3   4   5   6   7   8   9
2   4   6   8   10  12  14  16  18
3   6   9   12  15  18  21  24  27
```

```
4   8   12  16  20  24  28  32  36
5   10  15  20  25  30  35  40  45
6   12  18  24  30  36  42  48  54
```

Мы создаем двумерный целочисленный массив `nums`. Еще мы создаем массив `t` из объектных переменных класса `Thread`. Размер массива `t` равен количеству строк в массиве `nums` (вычисляется инструкцией `nums.GetLength(0)`). После этого перебираются элементы массива `t`, и на каждой итерации цикла при фиксированном индексе `i` выполняются такие команды:

- Объявляется локальная целочисленная переменная `p`, и в качестве значения ей присваивается текущее значение индекса `i`. Важно, что переменная объявляется внутри цикла. Поэтому переменная создается (под нее выделяется место в памяти) за каждый цикл заново. По идее, область доступности и время существования этой переменной определяются циклом. Но как мы увидим далее, на эту переменную будет ссылаться метод, реализуемый через лямбда-выражение (используется при создании потока). Поэтому переменная `p` будет существовать, пока работает поток. Но самое важное то, что у каждого потока будет «своя» переменная `p`.
- Создается объект потока. При создании объекта класса `Thread` (объект потока, ссылка на который записывается в элемент `t[i]`) аргументом конструктору передается лямбда-выражение, определяющее метод, выполняемый в потоке. Этим методом запускается оператор цикла, в котором переменная `j` пробегает значения второго индекса для элементов массива `nums`, при условии, что первый индекс равен `p`. Командой `nums[p,j]=(p+1)*(j+1)` соответствующему элементу присваивается значение (то есть значение элемента равно произведению порядкового номера строки на порядковый номер столбца). После присваивания значения элементу целочисленного массива командой `Thread.Sleep(100)` делается небольшая пауза в выполнении потока.
- После создания объекта потока командой `t[i].Start()` этот поток запускается на выполнение.

Таким образом, после выполнения оператора цикла будут запущены дочерние потоки. Количество дочерних потоков равно количеству строк в двумерном массиве. Ссылки на объекты потоков записаны в массив `t`. Каждый поток заполняет одну строку двумерного массива.

ПОДРОБНОСТИ

Несложно заметить, что переменная `p` фактически дублирует переменную `i` в операторе цикла. Почему нельзя было вместо `p` использовать `i`? Переменная `p` для каждого цикла своя. Переменная `i` для всех циклов общая. Если бы в лямбда-выражении вместо переменной `p` использовалась переменная `i`, то методы, выполняемые в разных потоках, ссылались бы на одну и ту же переменную `i`, которая изменяет свое значение в процессе выполнения оператора цикла. Выполнение кода становится непредсказуемым — точнее, предсказуемо заканчивается ошибкой.

Далее по плану содержимое массива должно отображаться в консольном окне. Но для этого главный поток должен дождаться окончания выполнения дочерних потоков. Поэтому запускается оператор цикла, в котором перебираются все элементы из массива `t`. На каждой итерации цикла для данного индекса `i` проверяется условие `t[i].IsAlive`. Если оно истинно (поток продолжает работу), то ожидается завершение потока (команда `t[i].Join()`). После того как все дочерние потоки завершили работу (это означает, что двумерный массив заполнен), с помощью вложенных операторов цикла отображается содержимое двумерного массива `nums`.

НА ЗАМЕТКУ

Инструкция `{0,-4}` в строке форматирования в методе `WriteLine()` означает, что под соответствующий аргумент при отображении в консольном окне отводится `4` позиции, а выравнивание выполняется по левому краю (поскольку второе число отрицательное).

В программе из листинга 6.10 также заполняется двумерный целочисленный массив. Но на этот раз для заполнения массива используется специальный метод. При запуске метода ему передается ссылка на массив (который следует заполнить), индекс строки (начиная с которой заполняется массив) и ссылка на объект для генерирования случайных чисел. Если указанная для заполнения строка массива не является последней, то метод запускает дочерний поток, в котором вызывается этот же метод, но заполнять он должен следующую строку. Таким образом, в результате вызова метода построчно заполняется двумерный массив: начиная с той строки, которая указана при вызове метода, и до последней строки массива включительно. Каждая строка заполняется отдельным потоком.

Теперь рассмотрим программный код, представленный ниже.

Листинг 6.10. Создание потока в потоке

```
using System;
using System.Threading;
// Главный класс:
class ThreadInThreadDemo{
   // Статический метод для заполнения двумерного массива
   // случайными числами:
   static void fill(int[,] a,int k,Random rnd){
      // Объектная переменная для потока:
      Thread t=null;
      // Если не последняя строка:
      if(k<a.GetLength(0)-1){
         // Создание объекта потока:
         t=new Thread(()=>{
            // Вызов в потоке метода:
            fill(a,k+1,rnd);
         });
         // Запуск потока на выполнение:
         t.Start();
      }
      // Заполнение строки в массиве:
      for(int m=0;m<a.GetLength(1);m++){
         // Значение элемента - случайное число:
         a[k,m]=k*10+rnd.Next(10);
      }
      // Если объект потока существует и поток еще
      // выполняется, то необходимо дождаться
      // завершения потока:
      if(t!=null&&t.IsAlive) t.Join();
   }
```

```
   // Главный метод:
   static void Main(){
      // Создается двумерный массив:
      int[,] nums=new int[6,9];
      // Объект для генерирования случайных чисел:
      Random rnd=new Random();
      // Заполнение массива:
      fill(nums,0,rnd);
      // Отображение содержимого массива:
      for(int i=0;i<nums.GetLength(0);i++){
         for(int j=0;j<nums.GetLength(1);j++){
            Console.Write("{0,-4}",nums[i,j]);
         }
         Console.WriteLine();
      }
   }
}
```

Результат выполнения программы может быть таким:

Результат выполнения программы (из листинга 6.10)

```
7   0   6   5   4   5   9   7   1
13  15  18  18  16  19  11  10  13
21  20  20  21  28  24  25  23  22
30  34  30  39  36  36  35  38  31
43  49  48  45  49  49  42  40  46
55  59  51  52  59  58  55  54  51
```

В главном классе программы, кроме метода `Main()`, также описывается статический метод `fill()` с тремя аргументами: ссылка на двумерный целочисленный массив `a`, целочисленный индекс строки `k` (начиная с которой должен заполняться массив) и ссылка `rnd` на объект класса `Random` (с помощью которого будут генерироваться случайные числа). В теле метода объявляется объектная переменная `t` класса `Thread`.

Ее начальное значение равно `null` (пустая ссылка). В условном операторе проверяется условие `k<a.GetLength(0)-1`, истинное в случае, если обозначенная для заполнения строка не является последней в двумерном массиве. Если так, то создается объект потока и ссылка на этот объект записывается в переменную `t`. Поток определен таким образом, что при его выполнении вызывается метод `fill()` (команда `fill(a,k+1,rnd)`), но на этот раз его второй аргумент (индекс строки для заполнения) на единицу больше. После этого поток запускается на выполнение (команда `t.Start()`) и выполняется заполнение строки случайными числами. Для этого использован оператор цикла, в котором индексная переменная `m` определяет второй индекс элемента массива (первый индекс элемента массива — это индекс `k` отмеченной для заполнения строки). Значение элементу присваивается командой `a[k,m]=k*10+rnd.Next(10)`. Поэтому первая строка (индекс `k` равен `0`) заполняется случайными числами от `0` до `9`, вторая строка (индекс `k` равен `1`) заполняется случайными числами от `10` до `19`, третья строка (индекс `k` равен `2`) заполняется случайными числами от `20` до `29`, и так далее.

После заполнения строки числами в условном операторе проверяется условие `t!=null&&t.IsAlive`. Если оно истинно, то вследствие команды `t.Join()` метод ожидает завершения выполнения потока, после чего работа метода завершается.

ПОДРОБНОСТИ

В условном операторе проверяется условие `t!=null&&t.IsAlive`. Состоит оно в том, что переменная `t` содержит непустую ссылку (попросту это означает, что дочерний поток был создан и запущен), и при этом поток, на объект которого ссылается переменная `t`, еще выполняется. Важно то, что использован оператор логического и `&&`, работающий по упрощенной схеме: если первое условие `t!=null` ложно, то второе условие `t.IsAlive` проверяться не будет. Проверка условия `t.IsAlive` при ложном условии `t!=null` приводит к ошибке.

В главном методе программы создается двумерный массив `nums` и объект `rnd` класса `Random` для генерирования случайных чисел. Массив заполняется при выполнении команды `fill(nums,0,rnd)`, после чего мы проверяем содержимое массива с помощью вложенных операторов цикла.

НА ЗАМЕТКУ

Метод `fill()` ожидает завершения потока, если такой запускался при вызове метода. Поэтому завершение выполнения метода `fill()` означает, что запущенный методом поток также завершен. В дочернем потоке вызывается метод `fill()`, который может запустить еще один поток, и так далее. Но в силу того же обстоятельства (метод `fill()` ожидает завершения дочернего потока, запущенного методом) получается, что каждый дочерний поток ожидает завершения запущенного из него дочернего потока.

В следующем примере дочерний поток используется для вычисления суммы бесконечного ряда $\sum_{n=1}^{\infty} n^2/n!$ («точное» значение этой бесконечной суммы равно $2e \approx 5{,}43656$, где через $e \approx 2{,}71828$ обозначено основание натурального логарифма). Программа выполняет вычисления следующим образом. В главном методе объявляется `double`-переменная с начальным нулевым значением. Создается и запускается дочерний поток с бесконечным циклом. Там последовательно вычисляются слагаемые, и каждое вычисленное слагаемое прибавляется к текущему значению переменной, выделенной для запоминания значения суммы. При этом главный поток на время приостанавливает работу. После паузы из главного потока завершается выполнение дочернего потока, а вычисленное значение для суммы отображается в консольном окне. Программный код, в котором реализован описанный подход, представлен в листинге 6.11.

Листинг 6.11. Вычисление суммы

```
using System;
using System.Threading;
// Главный класс:
class SumCalcDemo{
   // Главный метод:
   static void Main(){
      // Переменная для записи значения суммы:
      double s=0;
      Console.WriteLine("Вычисление суммы");
      // Контрольное значение для суммы:
      Console.WriteLine("Контрольное значение: {0}",2*Math.E);
```

```
      // Создание объекта потока:
      Thread calc=new Thread(()=>{
         // Индексная переменная:
         int n=1;
         // Добавка к сумме:
         double q=1;
         // Бесконечный цикл:
         do{
            // Прибавление слагаемого к сумме:
            s+=q;
            // Новое значение для индексной переменной:
            n++;
            // Вычисление добавки к сумме
            // для следующего цикла:
            q=n*n;
            for(int k=1;k<=n;k++){
               q/=k;
            }
            // Приостановка в выполнении потока:
            Thread.Sleep(100);
         }while(true);
      });
      // Запуск потока на выполнение:
      calc.Start();
      // Приостановка выполнения главного потока:
      Thread.Sleep(1000);
      // Завершение выполнения дочернего потока:
      calc.Abort();
      // Отображение результата вычислений:
      Console.WriteLine("Вычисленное значение: {0}",s);
   }
}
```

Результат выполнения программы может быть таким, как показано ниже:

Результат выполнения программы (из листинга 6.11)

```
Вычисление суммы
Контрольное значение: 5,43656365691809
Вычисленное значение: 5,43656029541446
```

Думается, программный код особых комментариев не требует. Отметим лишь несколько обстоятельств. Во-первых, для сравнения приводится «точный» результат для суммы. Для этого мы использовали статическую константу `E` (основание натурального логарифма) из класса `Math` (пространство имен `System`). Во-вторых, слагаемые для суммы вычислялись «в лоб», без всяких упрощений и оптимизаций, строго в соответствии с формулой для слагаемого суммы. В-третьих, в дочернем потоке между прибавлением слагаемых к сумме выполнялась небольшая пауза. Делалось это с простой целью: чтобы дочерний поток не выполнялся слишком быстро и можно было проследить, как время вычисления суммы влияет на результат.

Резюме

Друзья! Пацаки! Вот ваш носок, спасибо.

из к/ф «Кин-дза-дза»

- Многопоточное программирование подразумевает, что некоторые части программы или блоки программного кода выполняются одновременно. Для создания потока (запуска потока на выполнение) необходимо определить программный код, который будет выполняться в потоке, и запустить программный код на выполнение в режиме дочернего потока.
- Создание потока подразумевает создание объекта потока на основе класса `Thread` из пространства имен `System.Threading`. Аргументом конструктору класса `Thread` передается ссылка на экземпляр делегата `TreadStart`. Делегат подразумевает работу с методами, которые не имеют аргументов и не возвращают результат. Для запуска потока на выполнение из объекта потока вызывается метод `Start()`.

- Существует ряд свойств и методов, полезных при работе с потоками. Статический метод `Sleep()` класса `Thread` позволяет выполнить временную приостановку в выполнении потока, из которого вызывается метод. Время (в миллисекундах) задержки в выполнении потока указывается в качестве аргумента при вызове метода. Метод `Join()` используется, если необходимо дождаться выполнения потока, из объекта которого вызван метод. С помощью метода `Abort()` можно завершить выполнение потока. Свойство `IsAlive` позволяет определить, выполняется ли поток, для объекта которого запрашивается свойство. Свойство `Name` определяет название потока, а свойство `Priority` определяет приоритет проекта. Статическое свойство `CurrentThread` результатом возвращает ссылку на объект потока, из которого запрашивается свойство.
- По умолчанию дочерние потоки являются приоритетными, поэтому при завершении главного потока работа дочерних потоков продолжается. Можно создать фоновый поток, который автоматически завершается при завершении главного потока. Для этого свойству `IsBackground` объекта потока необходимо присвоить значение `true` (по умолчанию это свойство имеет значение `false`).
- При выполнении потоков, если они обращаются к одному общему ресурсу, выполняется синхронизация потоков. Один из способов решения этой задачи состоит в блокировке объекта, к которому обращаются потоки. Если использовать инструкцию `lock`, после которой в круглых скобках указать объект, то этот объект будет заблокирован для других потоков, пока выполняется выделенный блок программного кода.

Задания для самостоятельной работы

Всем лежать! Полчаса!

из к/ф «Кин-дза-дза»

1. Напишите программу, в которой для заданного целочисленного значения вычисляется факториал, двойной факториал и число из последовательности Фибоначчи. Для вычисления каждого из значений запускается дочерний поток. Факториал $n!$ для числа n вычисляется как произведение чисел от 1 до этого числа включительно, то есть $n! = 1 \times 2 \times 3 \times ... \times n$. Двойной факториал $n!!$ вычисляется как произведение натуральных

чисел «через одно», то есть $n!! = n \times (n - 2) \times (n - 4) \times ...$ (последний множитель равен 2 или 1 в зависимости от того, четное число n или нечетное). В последовательности Фибоначчи первые два числа равны 1, а каждое следующее число равно сумме двух предыдущих (получаются числа 1, 1, 2, 3, 5, 8, 13, 21 и так далее).

2. Напишите программу, в которой в главном потоке целочисленная переменная через определенные промежутки получает случайное значение. Два дочерних потока периодически (через определенные промежутки времени) проверяют значение переменной. Первый поток проверяет, является ли значение переменной нечетным, а второй поток проверяет, делится ли значение переменной на 3. Если проверка успешная, то соответствующий поток выводит в консольное окно сообщение.

3. Напишите программу, в которой символьный массив заполняется с помощью двух дочерних потоков: первый поток заполняет элементы с четными индексами латинскими буквами, а второй поток заполняет элементы с нечетными индексами кириллическими буквами.

4. Напишите программу, в которой два потока заполняют целочисленный массив. Один поток заполняет массив с начала до конца числами, являющимися степенями двойки (2^0, 2^1, 2^2 и так далее). Другой поток заполняет массив с конца до начала числами, являющимися степенями тройки (3^0, 3^1, 3^2 и так далее).

5. Напишите программу, содержащую класс с двумя полями: одно является ссылкой на целочисленный массив, а второе поле является ссылкой на символьный массив. Создайте объект класса, а также запустите на выполнение два дочерних потока. Один дочерний поток должен заполнить символьный массив объекта, а второй дочерний поток должен заполнить целочисленный массив объекта. Способ заполнения предложите самостоятельно (например, случайные символы и числа).

6. Напишите программу, в которой создается одномерный целочисленный массив и запускаются два дочерних потока. Один дочерний поток выполняет поиск элемента с наибольшим значением, а второй поток выполняет поиск элемента с наименьшим значением. Найденные значения отображаются в консольном окне.

7. Напишите программу, в которой создается двумерный целочисленный массив. Массив заполняется построчно: первая строка заполняется числами, являющимися степенями двойки (2^0, 2^1, 2^2 и так далее), вторая

строка заполняется степенями тройки (3^0, 3^1, 3^2 и так далее), и так далее. Для заполнения каждой строки запускается дочерний поток. Объектные переменные для потоков следует организовать в виде массива.

8. Напишите программу, в которой создается двумерный символьный массив. Массив заполняется случайными символами. Заполнение выполняется по столбикам. Для заполнения каждого столбика запускается отдельный дочерний поток.

9. Напишите программу, в которой с помощью дочернего потока вычисляется сумма $\sum_{n=1}^{\infty} 2^n(n + 1)/n!$ («точное» значение для суммы равно $3e^2 - 1 \approx 21{,}167168$).

10. Напишите программу, в которой с помощью дочернего потока вычисляется произведение $\prod_{n=2}^{\infty}(n^3 - 1)/(n^3 + 1)$. Сравните результат с «точным» значением $2/3$.

Глава 7
ОБОБЩЕННЫЕ ТИПЫ

А пацаки и чатлане — это национальность?

из к/ф «Кин-дза-дза»

В этой главе речь пойдет об *обобщенных типах*. В процессе работы с материалом главы нам предстоит:

- узнать, как тип данных может выступать в качестве параметра;
- научиться описывать и использовать обобщенные методы, в которых тип данных является параметром;
- познакомиться с обобщенными классами и структурами, в которых тип данных определяется в качестве параметра;
- научиться описывать и использовать обобщенные интерфейсы;
- познакомиться с обобщенными делегатами;
- узнать, как накладываются ограничения на параметры, определяющие тип данных.

Будут рассмотрены и другие вопросы, прямо или косвенно касающиеся использования обобщенных типов.

Передача типа данных в виде параметра

— А чем они друг от друга отличаются?
— Ты что, дальтоник, Скрипач? Зеленый цвет от оранжевого отличить не можешь?

из к/ф «Кин-дза-дза»

Идея использования обобщенных типов базируется на том, что при работе с методами, классами, структурами, интерфейсами или делегатами тип фактически используемых данных передается с помощью специального параметра.

НА ЗАМЕТКУ

Те «конструкции», которые описываются в данной главе, в англоязычной литературе описываются термином generics, что можно было бы перевести как обобщения (часто именно так и переводят). Иногда применяют термин «шаблоны» или «универсальные шаблоны». Но эта терминология, скорее, из языка C++. Мы будем использовать термин «обобщенные типы».

Чтобы лучше понять, зачем все это нужно и как реализуется на практике, представим следующую ситуацию. Допустим, мы хотим описать статический метод, который получает два аргумента одного и того же типа, и при выполнении метода эти аргументы обмениваются значениями. Что примечательно в таком методе? Примечательно в нем то, что алгоритм, который необходимо реализовать в методе, практически не зависит от типа аргументов, передаваемых методу. Самое важное и фактически единственное условие состоит в том, что оба аргумента должны относиться к одному и тому же типу. А какой именно это тип — вопрос второстепенный. Конечно, мы могли бы описать несколько версий метода для аргументов разного типа. Но, во-первых, это очень неудобно и нерационально (поскольку придется фактически переписывать несколько раз один и тот же код — с минимальными поправками на конкретный тип аргументов). Во-вторых, такой подход не всегда приемлем, поскольку типов данных (с учетом различных библиотечных и пользовательских классов) очень много и перебрать их все нет физической возможности. Так какой же выход из ситуации? А выход простой и элегантный. Мы описываем алгоритм выполнения метода, но при этом не конкретизируем тип аргументов, а лишь указываем, что они относятся к одному типу. При вызове метода данный алгоритм выполняется с учетом того, к какому типу относятся аргументы, переданные методу. В этом состоит основная идея, связанная с использованием обобщенных методов.

Примерно такой же подход используется при работе с обобщенными классами. Скажем, мы хотим описать класс, у которого имеется одно поле некоторого типа. Еще у класса есть конструктор с одним аргументом (того же типа, что и тип поля). Аргумент конструктора определяет значение поля. В классе можно описать метод (без аргументов), отображающий значение поля, а также метод с одним аргументом, позволяющий присвоить полю значение. В общем и целом — стандартная ситуация. Особенность ее лишь в том, что способ описания такого класса практически не зависит от того, к какому типу относится поле класса.

Важно лишь, чтобы тип поля, тип аргумента конструктора и тип аргумента метода совпадали. Ну, еще тип должен быть таким, чтобы значение этого типа можно было отобразить в консольном окне (то есть чтобы такая операция имела смысл). Вот, собственно, и все. А теперь давайте представим, что нам необходимо несколько классов, отличающихся лишь типом поля. В рамках традиционного подхода для каждого типа данных пришлось бы описывать отдельный класс. Причем если в случае с методом можно было бы воспользоваться перегрузкой (и разные версии метода имели бы одно и то же имя), то при описании нескольких классов каждому классу пришлось бы давать собственное имя. При этом содержимое каждого такого класса было бы во многом одинаково.

НА ЗАМЕТКУ

Теоретически можно было бы описать один класс с полем типа (класса) `Object`. Но и в таком подходе есть свои серьезные недостатки, так что это тоже не является решением проблемы.

Вместо создания нескольких однотипных классов мы можем описать один обобщенный класс. От обычного класса обобщенный отличается тем, что в нем не указан конкретный тип поля. Просто указывается, что поле есть и оно какого-то типа. И аргумент конструктора и метода — того же типа. А какой именно это тип, станет понятно, когда будет создаваться объект. Преимущество данного подхода как минимум в том, что нет необходимости многократно дублировать сходные блоки программного кода.

НА ЗАМЕТКУ

Обычно интерес представляет объект класса (в данном случае имеется в виду обычный, не обобщенный класс). Объект, как мы знаем, создается на основе класса. В этом смысле класс представляет собой некую абстракцию, которая служит «каркасом» для создания объекта. Обобщенный класс — это еще более высокий уровень абстракции. С определенной долей упрощения обобщенный класс можно рассматривать как «каркас», на основе которого формируется «обычный» класс, и уже на основе этого «обычного» класса создается объект. Процесс формирования «обычного» класса на основе обобщенного происходит при создании объекта. Как это делается на практике, мы рассмотрим далее.

Конечно, данная схема очень упрощенная и не совсем точная, но она позволяет понять место и роль обобщенных классов в программном коде.

Похожим образом идея «обобщенности» распространяется на структуры, интерфейсы и делегаты. Мы последовательно рассмотрим способ реализации данной концепции и начнем с методов и классов.

Обобщенные методы

Нет, ты тоже пацак! Ты — пацак, и он — пацак!
А я — чатланин! И они — чатлане!

из к/ф «Кин-дза-дза»

Итак, знакомство с обобщенными типами мы начнем с *обобщенных методов*.

> **НА ЗАМЕТКУ**
>
> Сначала рассмотрим статические методы, а затем уделим внимание обобщенным методам в обычных (не обобщенных) классах.

Принципиальное отличие обобщенного метода от обычного состоит в том, что при описании обобщенного метода вместо ключевого слова, определяющего тип данных, используется *обобщенный параметр типа*: некоторый идентификатор, обозначающий тип данных. Этот идентификатор используется в описании метода для обозначения типа данных. При вызове метода вместо обобщенного параметра типа «подставляется» фактический идентификатор типа, который обычно определяется по типу аргумента (или аргументов), переданного методу.

При описании обобщенного метода идентификатор, используемый в качестве обобщенного параметра типа, указывается в угловых скобках после имени метода. Именно наличие угловых скобок (с идентификатором обобщенного параметра типа) после имени метода является признаком того, что метод — обобщенный. Шаблон описания статического обобщенного метода выглядит так:

```
static тип_результата имя_метода<параметр_типа>(аргументы){
   // Код метода
}
```

Ситуацию можно описать и иначе: в обобщенном методе после имени метода в угловых скобках можно указать идентификатор, который

в коде метода используется для обозначения типа данных. Причем под «кодом метода» подразумевается и описание аргументов, и идентификатор типа результата. Например, ниже приведено описание статического обобщенного метода с одним обобщенным параметром типа:

```
public static void show<T>(T arg){
   Console.WriteLine(arg);
}
```

Это очень простой метод. Он не возвращает результат, у него один аргумент (обобщенного типа), и все, что делает этот метод, — выводит в консоль значение своего аргумента. При вызове метода ему фактически можно передавать аргумент любого типа. Тип аргумента (и конкретное значение параметра `T`) определяется по команде, которой вызывается метод. Скажем, если для вызова метода использована команда `show(123)`, то вместо параметра `T` будет подставлено значение `Int32`. Также при вызове обобщенного метода можно (а иногда это просто необходимо) явно указать, какое значение следует использовать в качестве обобщенного типа. Такое значение (идентификатор типа) указывается в команде вызова обобщенного метода в угловых скобках после имени метода (но перед круглыми скобками с аргументами метода). В этом смысле корректной является команда `show<int>(123)`, в которой мы явно указали, что в качестве обобщенного типа следует использовать тип `int`.

ПОДРОБНОСТИ

Идентификатор `int`, который мы используем для обозначения типа целочисленных переменных, на самом деле является псевдонимом для типа среды .Net Framework, который называется `Int32` (это структура из пространства имен `System`). Поэтому при проверке типа для `int`-значений получаем название типа `Int32`. Аналогичная ситуация имеет место и для других базовых типов.

Определить тип аргумента, фактически переданного методу, можно с помощью метода `GetType()`. Метод `GetType()` вызывается из объекта, тип которого нужно определить. Результатом является объект класса `Type`. Объект класса `Type` содержит информацию о типе исходного объекта (того объекта, из которого вызывался метод `GetType()`). У объекта класса `Type` есть свойство `Name`. Значение свойства `Name` — название типа (класс объекта, из которого вызывался метод `GetType()`).

Для определения значения обобщенного типа (то есть когда мы хотим выяснить, какое значение использовано для обобщенного

параметра) можно воспользоваться инструкцией `typeof`, после которой (в круглых скобках) указывается идентификатор типа. Результатом является объект класса `Type`. Название типа можно определить с помощью свойства `Name`.

Небольшой пример, в котором объявляется и используется статический обобщенный метод, представлен в листинге 7.1.

Листинг 7.1. Статические обобщенные методы

```
using System;
// Класс с целочисленным полем:
class MyClass{
   // Целочисленное поле:
   public int code;
   // Конструктор:
   public MyClass(int n){
      code=n;
   }
   // Переопределение метода ToString():
   public override string ToString(){
      return "поле = "+code;
   }
   // Операторный метод для преобразования объекта к
   // целочисленному типу:
   public static implicit operator int(MyClass obj){
      return obj.code;
   }
   // Операторный метод для преобразования целого числа
   // в объект:
   public static implicit operator MyClass(int n){
      return new MyClass(n);
   }
}
```

```
// Класс с главным методом:
class GenStatMethDemo{
   // Обобщенный метод (с аргументом обобщенного типа):
   static void show<T>(T arg){
      // Обобщенный параметр:
      Console.WriteLine("Обобщенный параметр: {0}",typeof(T).Name);
      // Значение аргумента:
      Console.WriteLine("Значение аргумента: {0}",arg);
      Console.WriteLine("-----------------------------");
   }
   // Главный метод:
   static void Main(){
      // Вызов обобщенных методов:
      show(100);
      show<int>(200);
      show<double>(300);
      show<MyClass>(400);
      show('A');
      show<char>('B');
      show<int>('C');
      show("Alpha");
      MyClass obj=new MyClass(500); // Создание объекта
      show(obj);
      show<int>(obj);
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 7.1)

```
Обобщенный параметр: Int32
Значение аргумента: 100
```

```
------------------------------
Обобщенный параметр: Int32
Значение аргумента: 200
------------------------------
Обобщенный параметр: Double
Значение аргумента: 300
------------------------------
Обобщенный параметр: MyClass
Значение аргумента: поле = 400
------------------------------
Обобщенный параметр: Char
Значение аргумента: A
------------------------------
Обобщенный параметр: Char
Значение аргумента: B
------------------------------
Обобщенный параметр: Int32
Значение аргумента: 67
------------------------------
Обобщенный параметр: String
Значение аргумента: Alpha
------------------------------
Обобщенный параметр: MyClass
Значение аргумента: поле = 500
------------------------------
Обобщенный параметр: Int32
Значение аргумента: 500
------------------------------
```

В этой программе мы описываем самый обычный класс `MyClass`, объект которого мы планируем передавать статическому обобщенному методу. В классе описано целочисленное поле `code`, его значение определяется аргументом конструктора. Метод `ToString()` переопределен

для класса так, что результатом возвращается текстовая строка со значением поля code. Еще в классе определены два операторных метода. Один определяет способ неявного преобразования объекта в целое число (в этом случае результатом является значение поля code объекта), а другой метод определят способ преобразования целого числа в объект (результатом операции является ссылка на вновь созданный объект, а значение поля code объекта определяется исходным целочисленным значением).

В классе с главным методом описан обобщенный статический метод show(). У метода один обобщенный параметр (обозначен как T). При вызове метода в консольном окне отображаются значение обобщенного параметра и значение аргумента. Значение обобщенного параметра определяется инструкцией typeof(T).Name.

В главном методе программы метод show() вызывается с несколькими аргументами, в том числе и с аргументом, являющимся объектом класса MyClass (при отображении значения аргумента, когда аргументом является объект класса MyClass, вызывается метод ToString()). Например, при выполнении команды show(100) обобщенный тип определяется по типу переданного методу аргумента (целое число). В команде show<int>(200) значение для обобщенного типа указано явно. Здесь значение для обобщенного типа совпадает со значением аргумента метода. В команде show<double>(300) это не так: аргумент является целым числом, а для обобщенного типа указано значение double. Но поскольку существует автоматическое приведение типов и оно применимо для преобразования типа int в тип double, то команда является корректной. Похожая ситуация имеет место и в отношении команды show<MyClass>(400), в которой аргументом является целое число, а для обобщенного параметра указано значение MyClass. Но поскольку в классе MyClass определен метод для неявного преобразования объектов этого класса в целочисленные значения, то команда также является корректной.

Команды show('A') и show<char>('B') особых комментариев не требуют. В команде show<int>('C') для обобщенного типа указано значение int, хотя аргумент относится к типу char. Как и в предыдущих аналогичных случаях, в игру вступает автоматическое приведение типа (значение типа char преобразуется в значение типа int).

При выполнении команды show("Alpha") в качестве обобщенного типа используется класс String. Наконец, в команде show(obj) аргументом методу передается объект obj класса MyClass, поэтому

в качестве обобщенного типа используется класс `MyClass`. А вот в команде `show<int>(obj)` аргумент метода хотя и является объектом класса `MyClass`, но значение для обобщенного типа указано явно, поэтому будет задействовано автоматическое преобразование объекта класса `MyClass` к целочисленному значению. Благо, мы описали в классе `MyClass` соответствующий операторный метод.

Обобщенный метод может содержать несколько обобщенных параметров. В таком случае идентификаторы для обобщенных параметров указываются через запятую в угловых скобках в описании метода. При вызове такого метода фактические значения для аргументов указываются в угловых скобках после имени метода или определяются автоматически на основе типов аргументов, переданных методу. В листинге 7.2 представлена программа, в которой есть обобщенные методы с несколькими обобщенными параметрами.

Листинг 7.2. Несколько обобщенных параметров

```
using System;
// Главный класс:
class MoreGenMethDemo{
   // Метод с двумя обобщенными параметрами:
   static void show<T,U>(T a,U b){
      // Отображение значений аргументов:
      Console.WriteLine("Первый аргумент {0}",a);
      Console.WriteLine("Второй аргумент {0}",b);
   }
   // Главный метод:
   static void Main(){
      // Вызов обобщенного метода:
      show(100,'A');
      show('B',"Bravo");
      show('C','D');
      show<int,char>('C','D');
   }
}
```

Результат выполнения программы такой, как показано ниже:

Результат выполнения программы (из листинга 7.2)

```
Первый аргумент 100
Второй аргумент A
Первый аргумент B
Второй аргумент Bravo
Первый аргумент C
Второй аргумент D
Первый аргумент 67
Второй аргумент D
```

Принципы описания и использования методов с несколькими обобщенными параметрами точно такие, как и в случае с методами, имеющими один параметр. Просто нужно сделать поправку на то, что параметров теперь несколько.

Схема использования обобщенных методов не сводится лишь к передаче методам параметров разных типов. Скажем, аргументом метода может быть массив из элементов обобщенного типа или метод может подобный массив возвращать как результат. Небольшая зарисовка к сказанному представлена в программе в листинге 7.3.

Листинг 7.3. Различные обобщенные методы

```
using System;
// Класс с главным методом:
class MuchMoreGenMethDemo{
   // Обобщенный метод с аргументом - массивом из
   // элементов обобщенного типа:
   static void show<T>(T[] array){
      // Отображение значений элементов массива:
      for(int k=0;k<array.Length;k++){
         Console.Write("|"+array[k]);
      }
      Console.WriteLine("|");
   }
```

```
// Обобщенный метод возвращает результатом массив из
// элементов обобщенного типа:
static T[] create<T>(int n){
   // Создание массива из элементов обобщенного типа:
   T[] array=new T[n];
   // Результат метода - ссылка на массив:
   return array;
}
// Обобщенный метод для заполнения массива значениями:
static void fill<T>(T[] a,T b){
   // Перебираются элементы массива:
   for(int k=0;k<a.Length;k++){
      // Элементу массива присваивается значение:
      a[k]=b;
   }
}
// Обобщенный метод результатом возвращает значение
// случайного элемента из массива:
static T getRand<T>(T[] a,Random rnd){
   // Результат метода:
   return a[rnd.Next(a.Length)];
}
// Главный метод:
static void Main(){
   // Создание объекта для генерирования чисел:
   Random rnd=new Random();
   // Создание массивов:
   string[] A={"one","two","three"};
   int[] B={2,3,5,8,13,21};
   char[] C={'c','h','a','r','l','i'};
   // Вызов обобщенных методов:
   show(A);
```

```
      show(B);
      Console.WriteLine("Случайные числа:");
      for(int k=1;k<=10;k++){
         Console.Write(getRand(B,rnd)+" ");
      }
      Console.WriteLine();
      show(C);
      Console.WriteLine("Случайные символы:");
      for(int k=1;k<=10;k++){
         Console.Write(getRand(C,rnd)+" ");
      }
      Console.WriteLine();
      // Создание символьного массива вызовом метода:
      char[] symbs=create<char>(6);
      // Заполнение массива символами:
      fill(symbs,'W');
      // Отображение содержимого массива:
      show(symbs);
      // Создание целочисленного массива вызовом метода:
      int[] nums=create<int>(7);
      // Заполнение массива числами:
      fill(nums,3);
      // Отображение содержимого массива:
      show(nums);
   }
}
```

Результат выполнения программы может (с учетом того, что используется генератор случайных чисел) быть таким:

Результат выполнения программы (из листинга 7.3)

```
|one|two|three|
|2|3|5|8|13|21|
```

```
Случайные числа:
2 21 5 5 21 5 8 8 21 8
|c|h|a|r|l|i|
Случайные символы:
a i c r i a r c a h
|W|W|W|W|W|W|
|3|3|3|3|3|3|
```

В классе с главным методом описывается несколько статических обобщенных методов. Есть метод `show()`, аргументом которого является массив обобщенного типа (обобщенный тип обозначен как `T`, а тип аргумента обозначен как `T[]` — то есть ссылка на массив, элементы которого относятся к типу `T`). Методом отображаются значения элементов массива.

Обобщенный метод `create()` получает в качестве аргумента целое число, а результатом возвращает массив (не заполненный) из элементов обобщенного типа. В теле метода командой `T[] array=new T[n]` создается массив из `n` (целочисленный аргумент метода) элементов типа `T` (обобщенный параметр). Ссылка на этот массив возвращается результатом метода (команда `return array`).

Обобщенный метод `fill()` предназначен для заполнения массива значениями (все элементы массива получают одинаковые значения). Ссылка на массив и присваиваемое элементам массива значение передаются аргументами методу.

Наконец, в программе есть метод `getRand()`, который результатом возвращает значение одного из элементов массива, переданного первым аргументом методу (первый аргумент обозначен как `a`). Элемент выбирается случайным образом. Для этого в качестве второго аргумента методу передается ссылка на объект класса `Random` (второй аргумент обозначен как `rnd`), с помощью которого генерируется случайное число (индекс элемента). Индекс генерируется инструкцией `rnd.Next(a.Length)` (случайное число в диапазоне от `0` до `a.Length-1` включительно). Результатом метода возвращается значение элемента с соответствующим индексом (команда `return a[rnd.Next(a.Length)]`).

В методе `Main()` создается объект `rnd` для генерирования случайных чисел, а также три массива `A` (текстовый), `B` (целочисленный) и `C` (символьный). Для отображения содержимого массивов используются

команды `show(A)`, `show(B)` и `show(C)`. Результатом выражения `getRand(B,rnd)` является значение одного из элементов (выбирается случайным образом) массива `B`. Аналогично результат выражения `getRand(C,rnd)` — это значение случайно выбранного элемента из массива `C`.

При выполнении команды `char[] symbs=create<char>(6)` создается символьный массив из `6` (аргумент метода `create()`) элементов, и ссылка на массив записывается в переменную `symbs`. В данном случае мы явно указали значение `char` для обобщенного параметра, поскольку по аргументу метода `create()` автоматически определить значение обобщенного параметра не получится. В силу той же причины значение `int` для обобщенного параметра явно указывается в команде `int[] nums=create<int>(7)`, которой создается целочисленный массив из `7` элементов. Ссылка на массив записывается в переменную `nums`. Для заполнения массивов используем команды `fill(symbs,'W')` и `fill(nums,3)`. Отображается содержимое массивов с помощью метода `show()`.

Примечателен и тот факт, что обобщенные методы можно перегружать: в программе может быть несколько версий обобщенного метода. Как это делается на практике, показано в листинге 7.4.

Листинг 7.4. Перегрузка обобщенных методов

```
using System;
// Главный класс:
class GenMethOverloadDemo{
   // Первая версия метода:
   static void show<T>(T a){
      Console.WriteLine("Версия №1");
      Console.WriteLine("Аргумент: {0}",a);
   }
   // Вторая версия метода:
   static void show(int a){
      Console.WriteLine("Версия №2");
      Console.WriteLine("Число: {0}",a);
   }
```

```
   // Третья версия метода:
   static void show<T,U>(T a,U b){
      Console.WriteLine("Версия №3");
      Console.WriteLine("Первый: {0}",a);
      Console.WriteLine("Второй: {0}",b);
   }
   // Четвертая версия метода:
   static void show<T>(T a,int b){
      Console.WriteLine("Версия №4");
      Console.WriteLine("Первый: {0}",a);
      Console.WriteLine("Число:  {0}",b);
   }
   // Главный метод:
   static void Main(){
      // Вызов обобщенного метода:
      show('A');
      show(123);
      show<int>('A');
      show<int>(123);
      show('B',"Bravo");
      show('C',321);
      show<char,int>('C',321);
   }
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 7.4)

```
Версия №1
Аргумент: A
Версия №2
Число: 123
Версия №1
```

```
Аргумент: 65
Версия №1
Аргумент: 123
Версия №3
Первый: B
Второй: Bravo
Версия №4
Первый: C
Число:  321
Версия №3
Первый: C
Второй: 321
```

В этой программе мы описали четыре версии метода `show()`:

- Обобщенный метод с одним аргументом обобщенного типа.
- Обычный (не обобщенный) метод с целочисленным аргументом.
- Обобщенный метод с одним аргументом обобщенного типа и одним аргументом целочисленного типа.
- Обобщенный метод с двумя аргументами обобщенного типа.

Все версии метода `show()` при вызове отображают в консольном окне значения аргумента или аргументов, но в любом случае можно определить версию метода, которая была вызвана.

В главном методе метод `show()` вызывается с разными аргументами. При выполнении команды `show('A')` однозначно вызывается обобщенный метод с одним аргументом. А вот при выполнении команды `show(123)` появляется некоторая интрига относительно версии метода, которую следует вызвать (обобщенный метод с одним аргументом или обычный метод с целочисленным аргументом). Приоритет остается за обычной версией метода. Но если мы при вызове метода в угловых скобках укажем значение для обобщенного параметра (как в команде `show<int>(123)`), то будет вызван обобщенный метод. Аналогичная ситуация имеет место и в случае передачи двух аргументов методу `show()`. При выполнении команды `show('B',"Bravo")` вариантов, кроме как вызывать метод с двумя аргументами обобщенного типа, нет.

А вот при выполнении команды `show('C',321)` вызывается версия метода, в которой второй аргумент описан как обычный целочисленный аргумент. Если же мы указываем в угловых скобках значения для обобщенных параметров (команда `show<char,int>('C',321)`), то вызывается версия метода с двумя аргументами обобщенного типа.

НА ЗАМЕТКУ

Ситуация может быть достаточно запутанной. Например, мы желаем описать две версии обобщенного метода с двумя аргументами. У первой версии метода первый аргумент обобщенного типа, а второй — целочисленный. Во второй версии метода первый аргумент является целочисленным, а второй — обобщенного типа. В таком случае вызов метода с двумя целочисленными аргументами будет невозможен, поскольку невозможно однозначно определить, какая версия метода вызывается.

Обобщенный метод может быть и нестатическим: такой метод описывается в обычном (не обобщенном) классе. В листинге 7.5 приведен пример подобной ситуации.

Листинг 7.5. Нестатические обобщенные методы

```
using System;
// Класс с обобщенным методом:
class MyClass{
   // Закрытое текстовое поле:
   public string text;
   // Конструктор:
   public MyClass(string t){
      text=t;
   }
   // Обобщенный метод:
   public void show<X>(X arg){
      Console.WriteLine(text+": "+arg);
   }
}
```

```
// Главный класс:
class MyClassWithGenMethodDemo{
   // Главный метод:
   static void Main(){
      // Создание объектов:
      MyClass A=new MyClass("Alpha");
      MyClass B=new MyClass("Bravo");
      // Вызов обобщенного метода:
      A.show(100);
      B.show(200);
      A.show('A');
      B.show<char>('B');
      B.show<int>('B');
   }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 7.5)

```
Alpha: 100
Bravo: 200
Alpha: A
Bravo: B
Bravo: 66
```

В программе описывается класс `MyClass` с текстовым полем. В классе есть обобщенный метод `show()`. Метод получает аргумент обобщенного типа и отображает в консольном окне значение этого аргумента. Вместе со значением аргумента отображается значение текстового поля объекта. В главном методе создается два объекта класса `MyClass`, и из этих объектов вызывается метод `show()`. Результаты вызовов вполне предсказуемы.

Обобщенные классы

Уэф, ты когда-нибудь видел, чтобы такой маленький пацак был таким меркантильным кю?

из к/ф «Кин-дза-дза»

Обобщенный класс — это класс, в котором некоторые (или все) типы представлены обобщенными параметрами. Конкретные значения для типов данных определяются на этапе создания объекта класса. Зачем такие классы нужны и какая от них польза? Вообще, обобщенный класс представляет собой конструкцию еще более абстрактную, чем обычный класс. Например, мы хотим создавать объекты с двумя полями. Если мы описываем обычный класс, то в таком классе мы должны указать типы полей. Но если мы описываем обобщенный класс, то типы полей можно указать с помощью обобщенных параметров, а затем, уже при создании объекта, определяются значения типов. Получается, что обобщенный класс представляет собой некий шаблон для обычного класса. То есть он «инкапсулирует» в себе различные классы однотипной структуры — в данном случае это классы, у которых два поля, не важно, какого типа.

Обобщенные классы удобно использовать в тех случаях, когда объекты, с которыми предстоит иметь дело, должны иметь однотипную структуру, но при этом тип данных может быть разным. Допустим, нам необходимо хранить данные в виде массива и для этой цели предполагается использовать специальные объекты-контейнеры, содержащие соответствующий массив и имеющие методы для работы с данными из массива. Если так, то тип элементов массива отходит на второй план. Его можно реализовать через обобщенный параметр, а в самом обобщенном классе описать универсальные алгоритмы (реализовав их в виде методов), применимые для данных любого типа. А конкретный тип данных, которые будут храниться в контейнере, определяется на этапе создания объекта.

НА ЗАМЕТКУ

Возвращаясь к примеру с классом с двумя полями, разницу между обычным и обобщенным классом можно определить так. Обычный класс позволяет создавать объекты с двумя полями определенного типа (скажем, `int` и `char`). С помощью обобщенного класса можно создавать объекты с двумя полями любого типа.

Как же описывается обобщенный класс? В принципе, так же, как и обычный (не обобщенный). Только в угловых скобках после названия класса указывается обобщенный параметр (или параметры), обозначающий тип данных. Этот параметр (или параметры) используется в теле класса, отождествляя собой некоторый тип (какой именно, становится понятно при создании объекта на основе класса). А объект класса тоже создается как обычный объект, но кроме названия обобщенного класса в угловых скобках после имени класса указывается значение (или значения) для обобщенных параметров. Шаблон описания обобщенного класса следующий:

```
class имя_класса<параметр(ы)_типа>{
   // Код класса
}
```

При создании объекта на основе обобщенного класса используем следующий шаблон:

```
класс<тип(ы)> объект=new класс<тип(ы)>();
```

Фактически, если под названием класса подразумевать имя класса с угловыми скобками и указанными в них значениями типов данных, процесс создания объекта представляется традиционным.

Теперь рассмотрим программу в листинге 7.6, в которой есть примеры описания и использования обобщенных классов.

Листинг 7.6. Знакомство с обобщенными классами

```
using System;
// Обобщенный класс с одним параметром:
class Alpha<X>{
   // Поле обобщенного типа:
   public X code;
   // Конструктор:
   public Alpha(X a){
      code=a;
   }
   // Метод:
   public void show(){
```

```
      Console.WriteLine("Поле типа {0}: {1}",typeof(X).Name,code);
   }
}
// Обобщенный класс с двумя параметрами:
class Bravo<X,Y>{
   // Поля обобщенного типа:
   public X first;
   public Y second;
   // Конструктор:
   public Bravo(X a,Y b){
      first=a;
      second=b;
   }
   // Метод:
   public void show(){
      Console.WriteLine("Первое поле типа {0}: {1}",typeof(X).Name,first);
      Console.WriteLine("Второе поле типа {0}: {1}",typeof(Y).Name,second);
   }
}
// Главный класс:
class GenClassDemo{
   // Главный метод:
   static void Main(){
      // Создание объектов обобщенного класса:
      Alpha<int> A=new Alpha<int>(123);
      Alpha<string> B=new Alpha<string>("object B");
      Alpha<char> C=new Alpha<char>('C');
      // Вызов методов из объектов обобщенного класса:
      A.show();
      B.show();
      C.show();
      Console.WriteLine();
```

```
        // Создание объектов обобщенного класса:
        Bravo<int,char> objA=new Bravo<int,char>(123,'A');
        Bravo<string,bool> objB=new Bravo<string,bool>("objB",true);
        Bravo<char,char> objC=new Bravo<char,char>('B','C');
        // Вызов методов из объектов обобщенного класса:
        objA.show();
        objB.show();
        objC.show();
    }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 7.6)

```
Поле типа Int32: 123
Поле типа String: object B
Поле типа Char: C

Первое поле типа Int32: 123
Второе поле типа Char: A
Первое поле типа String: objB
Второе поле типа Boolean: True
Первое поле типа Char: B
Второе поле типа Char: C
```

Программа простая. В ней мы описываем два обобщенных класса: класс `Alpha` описан с одним обобщенным параметром (обозначен как `X`), а класс `Bravo` описан с двумя обобщенными параметрами (обозначены как `X` и `Y`). В описании классов обобщенные параметры указываются в угловых скобках после имени класса.

В классе `Alpha` есть поле `code` обобщенного типа `X`, а также метод `show()`, при вызове которого в консольном окне отображается информация о значении обобщенного параметра `X` и значении поля `code`. Также в классе `Alpha` есть конструктор с одним аргументом (типа `X`).

Переданный конструктору аргумент присваивается в качестве значения полю `code`.

Класс `Bravo` подобен классу `Alpha`, но с поправкой на то, что в классе `Bravo` два обобщенных параметра (`X` и `Y`), два поля (поле `first` типа `X` и поле `second` типа `Y`), у конструктора два аргумента, а метод `show()` при вызове отображает значение параметров `X` и `Y`, а также значение полей `first` и `second`.

> **НА ЗАМЕТКУ**
>
> Методы в обобщенном классе описываются как обычные: то есть после имени метода обобщенные параметры в угловых скобках указывать не нужно. В случае необходимости они просто используются в методе. Это же относится к вызову методов. Если объект создается на основе обобщенного класса, то значения для обобщенных параметров определяются на этапе создания объекта. При вызове метода из объекта нет необходимости как-то определять эти параметры.

В главном методе создаются объекты классов `Alpha` и `Bravo`, и из этих объектов вызывается метод `show()`. Хочется верить, что результат выполнения соответствующих команд особых комментариев не требует.

> **НА ЗАМЕТКУ**
>
> При создании объекта на основе обобщенного класса значения обобщенных параметров указываются в угловых скобках после имени класса и при объявлении объектной переменной, и в инструкции создания объекта класса.

Обобщенные структуры

Здравствуйте! Мы пацаки с планеты Земля.

из к/ф «Кин-дза-дза»

Структуры также могут быть обобщенными. Принцип описания и использования обобщенных структур, в общем-то, такой же, как и в случае с обобщенными классами. Просто нужно помнить, что мы имеем дело со структурами, со всеми вытекающими отсюда последствиями. Пример использования обобщенных структур представлен в листинге 7.7.

Листинг 7.7. Знакомство с обобщенными структурами

```
using System;
// Обобщенная структура:
struct MyStruct<X,Y>{
   // Поля обобщенных типов:
   public X first;
   public Y second;
   // Конструктор:
   public MyStruct(X a,Y b){
      first=a;
      second=b;
   }
   // Метод для отображения значений полей:
   public void show(){
      Console.WriteLine("Первое поле: {0}",first);
      Console.WriteLine("Второе поле: {0}",second);
   }
}
// Главный класс:
class GenStructDemo{
   // Главный метод:
   static void Main(){
      // Создание экземпляра структуры:
      MyStruct<string,int> A;
      // Присваивание значений полям экземпляра структуры:
      A.first="MyStruct";
      A.second=100;
      // Вызов метода из экземпляра структуры:
      A.show();
      // Создание экземпляра структуры:
      MyStruct<char,int> B=new MyStruct<char,int>('B',200);
      // Вызов метода из экземпляра структуры:
```

```
      B.show();
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 7.7)

```
Первое поле: MyStruct
Второе поле: 100
Первое поле: B
Второе поле: 200
```

Программа представляет собой упрощенный вариант предыдущего примера, но на этот раз мы вместо обобщенного класса использовали обобщенную структуру. Структура называется `MyStruct`, и у нее два обобщенных параметра (обозначены как `X` и `Y`). В структуре описаны два поля обобщенного типа, конструктор с двумя аргументами и метод `show()` для отображения значений полей. В главном методе программы создаются экземпляры структуры, полям экземпляров в случае необходимости присваиваются значения. Для поверки значений полей экземпляров структуры вызывается метод `show()`.

Обобщенные интерфейсы

> А, вот видишь: у вас такой же оголтелый расизм, как и здесь, на Плюке! Только власть захватили не чатлане, а пацаки.
>
> *из к/ф «Кин-дза-дза»*

Интерфейс также может быть обобщенным. Класс, который создается путем реализации обобщенного интерфейса, может быть либо обобщенным, либо обычным. Пример в листинге 7.8 иллюстрирует эту ситуацию.

Листинг 7.8. Знакомство с обобщенными интерфейсами

```
using System;
// Обобщенный интерфейс:
```

```
interface MyInterface<X,Y>{
   void set(X a,Y b);
   void show();
}
// Обобщенный класс реализует обобщенный интерфейс:
class Alpha<X,Y>:MyInterface<X,Y>{
   private X first;
   private Y second;
   public void set(X a,Y b){
      first=a;
      second=b;
   }
   public void show(){
      Console.WriteLine("Поля {0} и {1}",first,second);
   }
}
// Обобщенный класс реализует обобщенный интерфейс:
class Bravo<Y>:MyInterface<int,Y>{
   private int first;
   private Y second;
   public void set(int a,Y b){
      first=a;
      second=b;
   }
   public void show(){
      Console.WriteLine("Поля {0} и {1}",first,second);
   }
}
// Обычный класс реализует обобщенный интерфейс:
class Charlie:MyInterface<int,char>{
   private int first;
   private char second;
```

```
   public void set(int a,char b){
      first=a;
      second=b;
   }
   public void show(){
      Console.WriteLine("Поля {0} и {1}",first,second);
   }
}
// Главный класс:
class GenInterfaceDemo{
   // Главный метод:
   static void Main(){
      Console.WriteLine("Первый объект");
      // Создание объекта на основе обобщенного класса:
      Alpha<string,double> A=new Alpha<string,double>();
      // Использование интерфейсной переменной:
      MyInterface<string,double> Ra=A;
      // Вызов методов:
      A.set("Alpha",100.0);
      Ra.show();
      Ra.set("alpha",150.0);
      A.show();
      Console.WriteLine("Второй объект");
      // Создание объекта на основе обобщенного класса:
      Bravo<string> B=new Bravo<string>();
      // Использование интерфейсной переменной:
      MyInterface<int,string> Rb=B;
      // Вызов методов:
      B.set(200,"Bravo");
      Rb.show();
      Rb.set(250,"bravo");
      B.show();
```

```
        Console.WriteLine("Третий объект");
        // Создание объекта на основе обычного класса:
        Charlie C=new Charlie();
        // Использование интерфейсной переменной:
        MyInterface<int,char> Rc=C;
        // Вызов методов:
        C.set(300,'C');
        Rc.show();
        Rc.set(350,'D');
        C.show();
    }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 7.8)

```
Первый объект
Поля Alpha и 100
Поля alpha и 150
Второй объект
Поля 200 и Bravo
Поля 250 и bravo
Третий объект
Поля 300 и C
Поля 350 и D
```

Мы описали обобщенный интерфейс `MyInterface` с двумя обобщенными параметрами `X` и `Y`. В интерфейсе объявлены два метода. Метод `set()` не возвращает результат, и у него два аргумента обобщенных типов. Метод `show()` не возвращает результат, и у него нет аргументов. Любой класс, реализующий интерфейс `MyInterface`, должен содержать описание этих методов.

Также мы описываем три класса, реализующих интерфейс `MyInterface`. Класс `Alpha` является обобщенным с двумя обобщенными параметрами

X и Y. Класс реализует интерфейс с теми же параметрами (соответствующая инструкция в описании класса имеет вид MyInterface<X,Y>). В классе есть два закрытых поля first и second обобщенных типов, метод set(), предназначенный для присваивания значений полям, а также метод show(), отображающий значение полей в консольном окне.

Класс Bravo также является обобщенным, и у него только один обобщенный параметр (обозначен как Y). В классе реализуется интерфейс MyInterface, но для первого обобщенного параметра для этого интерфейса указано значение int. Соответственно, инструкция, определяющая реализуемый интерфейс, имеет вид MyInterface<int,Y>. Класс Bravo описан подобно классу Alpha, но только одно поле относится к типу int, а второе поле относится к обобщенному типу. Соответствующим образом описан и метод set().

Класс Charlie является обычным (не обобщенным). Он реализует версию обобщенного интерфейса MyInterface, но с явно указанными значениями int и char для обобщенных параметров.

В главном методе программы создаются объекты классов Alpha, Bravo и Charlie. Кроме этого, объявляются интерфейсные переменные, которым в качестве значения присваиваются ссылки на соответствующие объекты. При объявлении интерфейсных переменных указываются имя интерфейса и в угловых скобках значения обобщенных параметров. Значения обобщенных параметров, указанные при объявлении интерфейсных переменных, должны соответствовать значениям обобщенных параметров, использованных при создании объекта.

НА ЗАМЕТКУ

Напомним, что интерфейсная переменная может ссылаться на объект любого класса, который реализует данный интерфейс. Через интерфейсную переменную можно получить доступ только к тем методам объекта, которые объявлены в интерфейсе.

Обобщенные классы и наследование

— А фамилию, позвольте узнать?
— Фамилию? Я согласен наследственную принять.

из к/ф «Собачье сердце»

Рассмотренный в предыдущем разделе механизм реализации обобщенных интерфейсов применим и к наследованию классов. Идея очень простая: один класс может наследовать другой класс, даже если последний является обобщенным. При этом производный класс может быть как обобщенным, так и обычным. В программе в листинге 7.9 приведен пример, похожий на пример из предыдущего раздела, но вместо реализации интерфейса задействовано наследование обобщенного абстрактного класса.

Листинг 7.9. Наследование обобщенных классов

```
using System;
// Обобщенный абстрактный класс:
abstract class MyClass<X,Y>{
   // Поля обобщенного типа:
   protected X first;
   protected Y second;
   // Абстрактные методы:
   abstract public void set(X a,Y b);
   abstract public void show();
}
// Обобщенный класс наследует обобщенный абстрактный класс:
class Alpha<X,Y>:MyClass<X,Y>{
   public override void set(X a,Y b){
      first=a;
      second=b;
   }
   public override void show(){
```

```
      Console.WriteLine("Поля {0} и {1}",first,second);
   }
}
// Обобщенный класс наследует обобщенный абстрактный класс:
class Bravo<Y>:MyClass<int,Y>{
   public override void set(int a,Y b){
      first=a;
      second=b;
   }
   public override void show(){
      Console.WriteLine("Поля {0} и {1}",first,second);
   }
}
// Обычный класс наследует обобщенный абстрактный класс:
class Charlie:MyClass<int,char>{
   public override void set(int a,char b){
      first=a;
      second=b;
   }
   public override void show(){
      Console.WriteLine("Поля {0} и {1}",first,second);
   }
}
// Главный класс:
class GenInterfaceDemo{
   // Главный метод:
   static void Main(){
      Console.WriteLine("Первый объект");
      // Создание объекта на основе обобщенного класса:
      Alpha<string,double> A=new Alpha<string,double>();
      // Использование объектной переменной базового
```

```
        // абстрактного класса:
        MyClass<string,double> Ra=A;
        // Вызов методов:
        A.set("Alpha",100.0);
        Ra.show();
        Ra.set("alpha",150.0);
        A.show();
        Console.WriteLine("Второй объект");
        // Создание объекта на основе обобщенного класса:
        Bravo<string> B=new Bravo<string>();
        // Использование объектной переменной базового
        // абстрактного класса:
        MyClass<int,string> Rb=B;
        // Вызов методов:
        B.set(200,"Bravo");
        Rb.show();
        Rb.set(250,"bravo");
        B.show();
        Console.WriteLine("Третий объект");
        // Создание объекта на основе обычного класса:
        Charlie C=new Charlie();
        // Использование объектной переменной базового
        // абстрактного класса:
        MyClass<int,char> Rc=C;
        // Вызов методов:
        C.set(300,'C');
        Rc.show();
        Rc.set(350,'D');
        C.show();
    }
}
```

Результат выполнения программы такой же, как и в предыдущем случае (см. Листинг 7.8):

Результат выполнения программы (из листинга 7.9)

```
Первый объект
Поля Alpha и 100
Поля alpha и 150
Второй объект
Поля 200 и Bravo
Поля 250 и bravo
Третий объект
Поля 300 и C
Поля 350 и D
```

В этой программе мы описали абстрактный обобщенный класс `MyClass` с двумя обобщенными параметрами (обозначены как `X` и `Y`). В классе объявлены два `protected`-поля `first` и `second` обобщенных типов, а также абстрактные методы `set()` и `show()`. Классы `Alpha`, `Bravo` и `Charlie` наследуют (но по-разному) абстрактный базовый класс `MyClass`. В производных класса переопределяются методы `set()` и `show()`. В главном методе программы создаются объекты классов `Alpha`, `Bravo` и `Charlie`, а также проверяется работа методов `set()` и `show()`.

Обобщенные делегаты

— Ну и зараза же ты, родной...
— Он хуже. Он просто кю!

из к/ф «Кин-дза-дза»

Обобщенным может быть делегат. При объявлении обобщенного делегата обобщенные параметры указываются в угловых скобках после имени делегата. При создании экземпляра делегата значения обобщенных параметров указываются в угловых скобках после названия делегата. Как все это выглядит на практике, показано в листинге 7.10.

Листинг 7.10. Обобщенный делегат

```
using System;
// Обобщенный делегат:
delegate X GenMeth<X,Y>(Y arg);
// Класс:
class MyClass{
   // Целочисленное поле:
   public int code;
   // Метод с символьным аргументом возвращает символьный
   // результат:
   public char getChar(char start){
      return (char)(start+code);
   }
}
// Главный класс:
class GenDelegateDemo{
   // Статический метод с целочисленным аргументом
   // возвращает текстовый результат:
   static string getStr(int num){
      return "Число "+num;
   }
   // Статический метод с текстовым аргументом возвращает
   // целочисленный результат:
   static int getInt(string txt){
      return Int32.Parse(txt);
   }
   // Главный метод:
   static void Main(){
      // Объект класса:
      MyClass obj=new MyClass();
      obj.code=5;
```

```
        // Экземпляры обобщенного делегата:
        GenMeth<string,int> A=getStr;
        GenMeth<int,string> B=new GenMeth<int,string>(getInt);
        GenMeth<char,char> C=obj.getChar;
        // Вызов экземпляров делегата:
        Console.WriteLine(A(123));
        Console.WriteLine(B("100")+200);
        Console.WriteLine(C('A'));
    }
}
```

При выполнении программы получаем такой результат:

Результат выполнения программы (из листинга 7.10)

```
Число 123
300
F
```

В программе мы объявляем обобщенный делегат `GenMeth` с двумя обобщенными параметрами `X` и `Y`. В соответствии с описанием делегата, экземпляр делегата может ссылаться на метод, который имеет один аргумент типа `Y` и возвращает результат типа `X`. Фактически это означает, что при создании экземпляра делегата в качестве метода, на который экземпляр будет ссылаться, можно выбрать любой метод с одним аргументом, возвращающий результат.

В программе описан класс `MyClass` с целочисленным полем `code`. В классе есть метод `getChar()`. У метода символьный аргумент, и метод возвращает символьный результат. Результат метода вычисляется так: к коду символа, переданного аргументом методу, прибавляется значение поля `code`, и полученное значение является кодом символа-результата.

В главном классе описано несколько статических методов. Методу `getStr()` в качестве аргумента передается целочисленное значение, а результатом метод возвращает текстовую строку, получающуюся объединением текста `"Число "` и значения аргумента метода.

Статический метод `getInt()`, как предполагается, аргументом получает текстовое представление для целого числа, а результатом возвращает собственно число из текста (фактически метод дублирует работу статического метода `Parse()` из структуры `Int32`).

В главном методе программы создается объект `obj` класса `MyClass`, полю `code` этого объекта присваивается значение `5`, после чего ссылка на метод `getChar()` объекта используется в команде `GenMeth<char,char> C=obj.getChar`, которой создается экземпляр для обобщенного делегата `GenMeth`. После имени делегата указаны значения для обобщенных параметров. Команды `GenMeth<string,int> A=getStr` и `GenMeth<int,string> B=new GenMeth<int,string>(getInt)` также дают представление о том, как создаются экземпляры обобщенного делегата. Созданные экземпляры делегата вызываются в главном методе.

Ограничения на параметры типа

> Уйди отсюда! Как советовать, так все — чатлане, а как работать, так...
>
> *из к/ф «Кин-дза-дза»*

В некоторых случаях при описании обобщенных методов, классов, структур, интерфейсов или делегатов необходимо наложить ограничения на значения для обобщенных параметров. Например, ограничение может быть таким: значением обобщенного параметра должно быть только имя класса. Или наоборот, имя класса не может быть значением обобщенного параметра. Или значением обобщенного параметра должно быть имя класса, являющегося производным от какого-то определенного класса, и так далее. Все это можно реализовать, причем достаточно легко.

Если обобщенный параметр не может принимать произвольное значение, а должен соответствовать определенным критериям, то в описании соответствующего обобщенного класса, метода, структуры, интерфейса или делегата указывается специальная инструкция, которая начинается с ключевого слова `where`. После этого ключевого слова следует название параметра (для которого определяется ограничение) и через двоеточие некоторое выражение. Это выражение, собственно, и определяет

ограничение, накладываемое на обобщенный параметр. Наиболее характерные случаи представлены в табл. 7.1 (далее через T обозначен обобщенный параметр).

Табл. 7.1. Ограничения на обобщенные параметры

Выражение	Значение
`where T: class`	Значением обобщенного параметра `T` может быть имя класса
`where T: struct`	Значением обобщенного типа `T` может быть только нессылочный тип (базовый тип или структура)
`where T: new()`	Значением обобщенного параметра `T` может быть тип, для которого предусмотрен конструктор без аргументов. Если на обобщенный параметр накладывается несколько ограничений, то данное ограничение должно быть последним
`where T: базовый_класс`	Значением обобщенного параметра `T` может быть имя класса, который является производным для указанного после двоеточия базового класса (обозначен как `базовый_класс`)
`where T: интерфейс`	Значением обобщенного параметра `T` может быть имя класса, реализующего данный `интерфейс`

Как все это выглядит на практике, иллюстрируют следующие примеры. В листинге 7.11 представлена программа, в которой описан обобщенный класс с одним параметром. На этот параметр накладывается ограничение: его значением может быть только класс, производный от одного из обычных классов, также описанных в программе.

Листинг 7.11. Ограничения на обобщенные параметры

```
using System;
// Базовый класс:
class Alpha{
   // Поле:
   protected int code;
   // Конструктор:
   public Alpha(int n){
      code=n;
   }
   // Виртуальный метод:
   public virtual void show(){
      Console.WriteLine("Класс Alpha: "+code);
```

```
   }
}
// Производный класс:
class Bravo:Alpha{
   // Конструктор:
   public Bravo(int n):base(n){}
   // Переопределение метода:
   public override void show(){
      Console.WriteLine("Класс Bravo: "+code);
   }
}
// Производный класс:
class Charlie:Bravo{
   // Конструктор:
   public Charlie(int n):base(n){}
   // Переопределение метода:
   public override void show(){
      Console.WriteLine("Класс Charlie: "+code);
   }
}
// Обобщенный класс с ограничением на обобщенный параметр:
class MyClass<T> where T: Alpha{
   // Поле - ссылка на объект:
   public Alpha obj;
   // Конструктор обобщенного класса:
   public MyClass(T t){
      // Ссылка на объект:
      obj=t;
      // Вызов метода:
      obj.show();
   }
}
```

```
// Главный класс:
class GenRestDemo{
   // Главный метод:
   static void Main(){
      // Создание объектов обычных классов:
      Alpha A=new Alpha(100);
      Bravo B=new Bravo(200);
      Charlie C=new Charlie(300);
      // Создание объектов на основе обобщенного класса:
      MyClass<Alpha> objA=new MyClass<Alpha>(A);
      MyClass<Bravo> objB=new MyClass<Bravo>(B);
      MyClass<Charlie> objC=new MyClass<Charlie>(C);
      MyClass<Alpha> objD=new MyClass<Alpha>(C);
   }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 7.11)

```
Класс Alpha: 100
Класс Bravo: 200
Класс Charlie: 300
Класс Charlie: 300
```

Во-первых, мы описали три обычных класса `Alpha`, `Bravo` и `Charlie`. Класс `Alpha` является базовым для класса `Bravo`, а класс `Bravo` является базовым для класса `Charlie`. В классе `Alpha` описано защищенное целочисленное поле `code`, конструктор с одним аргументом и виртуальный метод `show()`, который при вызове отображает название класса и значение поля `code`. В производных классах `Bravo` и `Charlie` метод `show()` переопределяется так, чтобы отображалось имя соответствующего класса (нам это нужно для того, чтобы можно было определить, из объекта какого класса вызывается метод).

Класс `MyClass` является обобщенным. У него один обобщенный параметр, обозначенный как `T`. Ограничение `where T: Alpha`, накладываемое

на этот параметр, означает, что значением параметра T может быть имя класса, являющегося производным от класса Alpha (класс Alpha также попадает в данную категорию). В классе MyClass есть поле obj. Это ссылка на объект класса Alpha. Конструктору обобщенного класса передается один аргумент — ссылка на объект обобщенного типа T. Данная ссылка в качестве значения присваивается полю obj. Такая операция возможна, поскольку, как мы помним, значением обобщенного параметра T может быть не любой класс, а только производный класс от класса Alpha. Далее, объектная переменная базового класса может ссылаться на объект производного класса. Поэтому поле obj, являющееся объектной переменной класса Alpha, может ссылаться на объект класса, который «скрывается» за параметром T.

В конструкторе обобщенного класса MyClass из объекта, на который ссылается поле obj, вызывается метод show() (команда obj.show()). В итоге при создании объекта на основе обобщенного класса появляется сообщение, в котором указано имя обычного класса (Alpha, Bravo или Charlie). Это класс объекта, на который записана ссылка в поле obj. Также в сообщении отображается значение поля code этого объекта.

В главном методе создается объект A класса Alpha со значением 100 для поля code, объект B класса Bravo со значением 200 для поля code и объект C класса Charlie со значением 300 для поля code. Далее на основе обобщенного класса MyClass создается четыре объекта objA, objB, objC и objD. Каждый раз при создании такого объекта появляется сообщение в консольном окне. Причем при создании последнего объекта objD значением обобщенного параметра указан класс Alpha, хотя фактически в качестве аргумента конструктору класса MyClass передается объект C класса Charlie. Но, как видим, эта ситуация обрабатывается корректно.

Еще один пример использования ограничения для обобщенного параметра типа представлен в листинге 7.12. На этот раз ограничение состоит в том, что значением обобщенного параметра может быть только класс, у которого есть конструктор без аргументов.

НА ЗАМЕТКУ

Когда возникает необходимость накладывать ограничение, связанное с наличием конструктора без аргументов? Например, в обобщенном классе создается объект обобщенного типа. Нужно вызвать конструктор класса, который обозначен как обобщенный параметр.

Если мы уверены, что у этого класса есть конструктор без аргументов, то можем вызвать такой конструктор, даже не зная, с каким классом в действительности придется иметь дело. А уверенными мы можем быть, если наложим ограничение, которое состоит в том, что у класса, являющегося значением обобщенного параметра, есть конструктор без аргументов.

Рассмотрим представленный далее программный код.

Листинг 7.12. Ограничение new() на параметр типа

```
using System;
// Первый класс:
class Alpha{
   public int code;
   public override string ToString(){
      return "Alpha: "+code;
   }
}
// Второй класс:
class Bravo{
   public string text;
   public override string ToString(){
      return "Bravo: "+text;
   }
}
// Обобщенный класс с ограничением на обобщенный параметр:
class MyClass<T> where T: new(){
   // Ссылка на объект:
   public T obj;
   // Конструктор:
   public MyClass(){
      // Создание объекта обобщенного типа:
      obj=new T();
   }
```

```
   // Метод:
   public void show(){
      Console.WriteLine(obj);
   }
}
// Главный класс:
class MoreGenRestDemo{
   // Главный метод:
   static void Main(){
      // На основе обобщенного класса создается объект:
      MyClass<Alpha> objA=new MyClass<Alpha>();
      objA.obj.code=123;
      objA.show();
      // На основе обобщенного класса создается объект:
      MyClass<Bravo> objB=new MyClass<Bravo>();
      objB.obj.text="text";
      objB.show();
   }
}
```

Результат выполнения программы показан ниже:

Результат выполнения программы (из листинга 7.12)

```
Alpha: 123
Bravo: text
```

В программе описаны обычные классы `Alpha` (с целочисленным полем `code`) и `Bravo` (с текстовым полем `text`). В каждом классе переопределен метод `ToString()`. Метод результатом возвращает текстовую строку с названием класса и значением поля. Ни в одном из классов не описан конструктор, и это означает, что каждый класс имеет конструктор без аргументов.

Обобщенный класс `MyClass` имеет один обобщенный параметр `T`, на который наложено ограничение `where T: new()`. Оно означает, что класс,

который будет использован как значение обобщенного параметра (при создании объекта на основе обобщенного класса), должен иметь конструктор без аргументов. Классы `Alpha` и `Bravo` удовлетворяют этому критерию.

В обобщенном классе `MyClass` имеется поле `obj` обобщенного типа `T`. Значением полю присваивается ссылка на объект, который создается при вызове конструктора. Соответствующая команда имеет вид `obj=new T()`. То есть объект создается инструкцией `new T()`, в которой мы обобщенный параметр `T` используем как имя класса. Собственно, ради возможности использовать данную инструкцию на параметр `T` накладывается ограничение.

У обобщенного класса `MyClass` есть метод `show()`. При его вызове в консольном окне «печатается» объект, на который ссылается поле `obj` (в этом случае для объекта `obj` вызывается метод `ToString()`).

В методе `Main()` на основе обобщенного класса `MyClass` создаются два объекта: при создании объекта `objA` значением обобщенного параметра является класс `Alpha`, а при создании объекта `objB` значением обобщенного параметра является класс `Bravo`. У каждого из объектов `objA` и `objB` есть поле `obj`. Для объекта `objA` поле `obj` ссылается на объект класса `Alpha`, а для объекта `objB` поле `obj` ссылается на объект класса `Bravo`. Командами `objA.obj.code=123` и `objB.obj.text="text"` полям объектов, на которые есть ссылка в поле `obj`, присваиваются значения. После этого командами `objA.show()` и `objB.show()` проверяется результат.

На один и тот же параметр может накладываться несколько ограничений сразу. Подобная ситуация представлена в листинге 7.13.

Листинг 7.13. Несколько ограничений на обобщенный параметр

```
using System;
// Базовый класс:
class Alpha{
   // Целочисленное поле:
   public int code;
   // Переопределение метода ToString():
   public override string ToString(){
      return "Alpha: "+code;
   }
```

```
}
// Производный класс:
class Bravo:Alpha{
   // Текстовое поле:
   public string text;
   // Переопределение метода ToString():
   public override string ToString(){
      return "Bravo: "+code+" и "+text;
   }
}
// Обобщенный класс:
class MyClass<X,Y>
   where X: class,new() // Ограничения на параметр X
   where Y: X,new(){    // Ограничения на параметр Y
   // Первое поле обобщенного типа:
   public X first;
   // Второе поле обобщенного типа:
   public Y second;
   // Конструктор:
   public MyClass(){
      // Создаются объекты обобщенных типов:
      first=new X();
      second=new Y();
   }
   // Переопределение метода ToString():
   public override string ToString(){
      return "MyClass->|"+first+"|"+second+"|";
   }
}
// Главный класс:
class MultiGenRestDemo{
   // Главный метод:
```

```
   static void Main(){
      // Создание объекта обобщенного класса:
      MyClass<Alpha,Bravo> obj=new MyClass<Alpha,Bravo>();
      // Присваивание значений полям:
      obj.first.code=100;
      obj.second.code=200;
      obj.second.text="text";
      // Отображение характеристик объекта:
      Console.WriteLine(obj);
   }
}
```

Ниже показано, как выглядит результат выполнения программы:

Результат выполнения программы (из листинга 7.13)

```
MyClass->|Alpha: 100|Bravo: 200 и text|
```

В программе описан класс `Alpha`, который является базовым для класса `Bravo`. У класса `Alpha` есть целочисленное поле `code`, а в классе `Bravo` к нему добавляется текстовое поле `text`. В каждом из классов переопределен метод `ToString()`: методом возвращается текстовая строка с названием класса и значением полей.

В программе также описан обобщенный класс `MyClass` с двумя обобщенными параметрами (обозначены как `X` и `Y`). На параметр `X` наложено ограничение `where X: class,new()`, означающее, что значением этого параметра может быть класс, у которого есть конструктор без аргументов. На параметр `Y` накладывается ограничение `where Y: X,new()`. Ограничение означает, что значением параметра `Y` может быть класс с конструктором без аргументов. А еще этот класс должен быть производным от класса, являющегося значением параметра `X`. В обобщенном классе есть поле `first` типа `X` и поле `second` типа `Y`. В конструкторе класса `MyClass` создаются объекты классов `X` и `Y`, и ссылки на эти объекты записываются в поля `first` и `second`. А метод `ToString()` для класса `MyClass` определен так, что отображаются название класса и текстовые представления (а значит, значения полей) объектов, на которые ссылаются поля `first` и `second`.

В главном методе командой `MyClass<Alpha,Bravo> obj=new MyClass<Alpha,Bravo>()` создается объект `obj` обобщенного класса `MyClass`. В качестве значения для обобщенного параметра `X` указан класс `Alpha`, а значением параметра `Y` указан класс `Bravo` (который является производным от класса `Alpha`). После присваивания значений полям объект `obj` «выводится» в консоль (в этом случае вызывается метод `ToString()`, описанный в классе `MyClass`).

Похожим образом накладываются ограничения на обобщенные параметры в интерфейсах. В листинге 7.14 представлена программа, в которой описывается обобщенный интерфейс (на обобщенные параметры которого накладываются ограничения), а затем путем реализации этого интерфейса создается обобщенный класс.

Листинг 7.14. Обобщенный интерфейс с ограничением на обобщенный параметр

```
using System;
// Обобщенный интерфейс с ограничением
// на обобщенный параметр:
interface MyInterface<T> where T: struct{
   void show();
}
// Обобщенный класс реализует обобщенный интерфейс:
class MyClass<T>:MyInterface<T> where T: struct{
   // Поле обобщенного типа:
   public T val;
   // Конструктор:
   public MyClass(T v){
      val=v;
   }
   // Описание метода из интерфейса:
   public void show(){
      Console.WriteLine("Тип {0}: значение {1}",typeof(T).Name,val);
   }
}
```

```
// Главный класс:
class GenInterfaceRestDemo{
   static void Main(){
      // Создание объектов на основе обобщенного класса:
      MyClass<int> A=new MyClass<int>(100);
      MyClass<char> B=new MyClass<char>('B');
      // Вызов метода из объектов:
      A.show();
      B.show();
   }
}
```

При выполнении программы получаем следующий результат:

Результат выполнения программы (из листинга 7.14)

```
Тип Int32: значение 100
Тип Char: значение B
```

Обобщенный интерфейс `MyInterface` описан с обобщенным параметром `T`, на который накладывается ограничение `where T: struct`. Ограничение означает, что обобщенный параметр `T` может «скрывать» за собой нессылочный тип. В интерфейсе объявлен метод `show()` без аргументов, не возвращающий результат.

Обобщенный класс `MyClass` реализует интерфейс `MyInterface`. Ограничение на обобщенный параметр в классе указано такое же, как и в интерфейсе. В классе описано поле `val` обобщенного типа. Значение поля отображается при вызове метода `show()`. Также отображается значение обобщенного параметра.

НА ЗАМЕТКУ

Значение полю `val` присваивается в конструкторе. Напрямую здесь ограничение, накладываемое на обобщенный параметр, не используется, хотя это и имеет значение: при присваивании значения полю `val` в него копируется значение аргумента (то есть копируется значение, а не выполняется ссылка на уже существующий объект).

В главном методе создаются объекты A и B класса MyClass. Значение полю val объектов присваивается при создании объектов. Для проверки значений полей из объектов вызывается метод show().

Еще один пример из листинга 7.15 иллюстрирует, как ограничения на обобщенные параметры реализуются в случае использования обобщенных делегатов и методов.

Листинг 7.15. Обобщенные делегаты и методы с ограничениями на параметры

```
using System;
// Обобщенный делегат с ограничением
// на обобщенный параметр:
delegate T MD<T>(T[] arg) where T: struct;
// Главный класс:
class GenDelegateRestDemo{
   // Статический обобщенный метод с ограничением на
   // обобщенный параметр:
   static T First<T>(T[] t)  where T: struct{
      return t[0];
   }
   // Статический обобщенный метод (без ограничений на
   // обобщенный параметр):
   static T Last<T>(T[] t){
      return t[t.Length-1];
   }
   // Обычный (не обобщенный) статический метод:
   static int MaxVal(int[] a){
      int val=a[0];
      for(int k=1;k<a.Length;k++){
         if(val<a[k]) val=a[k];
      }
      return val;
   }
```

```
   // Главный метод:
   static void Main(){
      // Создание экземпляров делегата:
      MD<int> A=First<int>;
      MD<char> B=Last;
      MD<int> C=MaxVal;
      // Вызов экземпляров делегата:
      int x=A(new int[]{10,7,3,1});
      char y=B(new char[]{'A','C','F'});
      int z=C(new int[]{1,7,2,5,11,9,3,6});
      // Проверка результата:
      Console.WriteLine("Значения: {0}, {1} и {2}",x,y,z);
   }
}
```

Ниже показано, как выглядит результат выполнения программы:

Результат выполнения программы (из листинга 7.15)

```
Значения: 10, F и 11
```

В данном случае мы описываем обобщенный делегат `MD` с одним обобщенным параметром (обозначен как `T`). На обобщенный параметр накладывается ограничение `where T: struct`, то есть параметром может быть нессылочный тип. Делегат соответствует методам, у которых аргумент является массивом, а результат — значением того же типа, что и элементы массива. Также в программе есть три статических метода: два обобщенных и один обычный (не обобщенный). Обобщенный метод `First()` результатом возвращает значение первого элемента массива, переданного аргументом методу. На обобщенный параметр метода накладывается такое же ограничение, как и на параметр обобщенного делегата. Обобщенный метод `Last()` результатом возвращает значение последнего элемента массива, переданного аргументом методу (на обобщенный параметр метода ограничения не накладываются). Наконец, аргументом методу `MaxVal()` передается целочисленный массив, а результатом возвращается значение наибольшего элемента в массиве.

В главном методе на основе статических методов создаются экземпляры для обобщенного делегата MD, причем при присваивании экземпляру ссылки на обобщенный метод последний мы указываем как с явной спецификацией значения обобщенного типа (команда MD <int> A=First<int>), так и без нее (команда MD<char> B=Last). При создании экземпляра делегата на основе обычного (не обобщенного) метода значение для обобщенного параметра указывается только после названия делегата (команда MD<int> C=MaxVal). Конечно, характеристики такого метода должны соответствовать характеристикам делегата. Также в главном методе есть примеры вызова созданных экземпляров делегата MD.

Резюме

— Побойся неба! ПЖ жив, и я — счастлив.
— А я еще больше счастлив!

из к/ф «Кин-дза-дза»

- При описании методов и классов (а также структур, интерфейсов и делегатов) тип может передаваться через параметр — в описании метода или класса тип обозначен параметром, а конкретное значение для типа данных определяется на этапе создания объекта или при вызове метода. Идентификатор (обобщенный параметр), обозначающий тип, указывается в угловых скобках после имени класса или названия метода. Если обобщенных параметров несколько, они разделяются запятыми.
- При вызове обобщенного метода фактическое значение обобщенного параметра (или параметров) определяется на основе аргументов, переданных методу. Также можно указать значения для обобщенных параметров явно, указав их в угловых скобках после имени метода (в команде вызова метода). Значения обобщенных параметров указываются явно и в том случае, если значения обобщенных параметров не могут быть определены автоматически по команде вызова метода.
- При создании объекта на основе обобщенного класса после имени класса в угловых скобках указываются значения для обобщенных параметров. Такой же принцип используется при работе с обобщенными структурами, интерфейсами и делегатами.

- Обобщенные методы можно перегружать. Обобщенные классы могут наследоваться. При этом на основе обобщенного класса могут создаваться как обобщенные классы, так и обычные.
- На обобщенные параметры могут накладываться ограничения (например, необходимость иметь конструктор без аргументов, быть производным классом от какого-либо класса или реализовать определенный интерфейс). Ограничение задается выражением, которое состоит из ключевого слова `where`, после которого указывается название обобщенного параметра, двоеточие и инструкция, определяющая накладываемое на параметр ограничение.

Задания для самостоятельной работы

> Такое предложение. Находим Скрипача, летим к местному правительству...
>
> *из к/ф «Кин-дза-дза»*

1. Напишите программу, в которой есть обобщенный статический метод с двумя аргументами обобщенного типа. При вызове метода аргументы должны обмениваться значениями.

2. Напишите программу, в которой есть статический обобщенный метод. Метод не возвращает результат, а аргументом методу передается массив с элементами обобщенного типа. При вызове метода выполняется циклическая перестановка элементов массива: первый элемент становится вторым, второй элемент становится третьим, и так далее, а последний элемент становится первым.

3. Напишите программу, содержащую обобщенный класс со свойством обобщенного типа. Значением свойства возвращается значение закрытого поля обобщенного типа. При присваивании значения свойству значение присваивается закрытому полю.

4. Напишите программу, содержащую обобщенный класс с индексатором обобщенного типа. Также в классе должно быть поле, являющееся ссылкой на массив из элементов обобщенного типа. Считывание и присваивание значений с использованием индексатора подразумевают считывание значений элементов массива и присваивание значений элементам массива.

5. Напишите программу, в которой создается цепочка из объектов, созданных на основе обобщенного класса: имеется последовательность объектов, и в этой последовательности первый объект ссылается на второй, второй объект ссылается на третий, и так далее. Обобщенный класс, помимо прочего, должен содержать поле обобщенного типа и поле, являющееся ссылкой на следующий объект в цепочке. Предложите методы для работы с такой цепочкой объектов.

6. Напишите программу, в которой есть статический обобщенный метод. Аргументом методу передается массив из элементов обобщенного типа. Результатом метод возвращает значение наибольшего/наименьшего элемента в массиве. Чтобы можно было сравнивать значения обобщенного типа, на этот обобщенный тип следует наложить ограничение: тип должен быть таким, что реализует интерфейс `IComparable`. Если так, то для сравнения значений может использоваться метод `CompareTo()`. Метод вызывается из одного объекта (первый объект) обобщенного типа, а аргументом ему передается другой (второй) объект обобщенного типа (сравниваемые объекты). Результатом метода является целое число. Отрицательное число возвращается, если первый объект «меньше» второго. Положительное число возвращается, если первый объект «больше» второго. Если объекты равны друг другу, то результатом метода `CompareTo()` является нулевое значение.

7. Напишите программу со статическим обобщенным методом, которому в качестве аргумента передается массив из элементов обобщенного типа. При вызове метода выполняется сортировка элементов массива в порядке возрастания/убывания. Для возможности сравнения значений элементов массива друг с другом (больше/меньше) следует наложить на обобщенный тип ограничение, состоящее в необходимости реализации интерфейса `IComparable`. Для сравнения значений обобщенных типов использовать метод `CompareTo()`.

8. Напишите программу, содержащую обобщенный статический метод. В качестве аргумента методу передается значение обобщенного типа. Методом в консольное окно выводится одно из трех разных сообщений, в зависимости от типа переданного аргумента. Отслеживаются такие случаи: аргументом является значение типа `int`, аргументом является значение типа `char`, и аргументом передается значение типа, отличного от `int` и `char`.

9. Напишите программу, в которой есть обобщенный класс. У обобщенного класса есть поле, являющееся ссылкой на массив из элементов

обобщенного типа. Опишите для класса операторный метод для оператора +, позволяющий вычислять сумму двух объектов класса. Результатом является объект этого же класса. Массив в объекте-результате получается объединением массивов суммируемых объектов.

10. Напишите программу, в которой есть обобщенный класс с двумя обобщенными параметрами (первый и второй). У класса есть два поля, являющихся ссылками на массивы из элементов обобщенного типа (первого и второго). Массивы имеют одинаковые размеры. Опишите индексатор, в котором индексом указывается значение второго обобщенного типа, а результатом индексатора является значение первого обобщенного типа. Предполагается, что индексом указывается одно из значений из второго массива, а считывается или присваивается значение для соответствующего элемента из первого массива. Предложите механизм добавления новых элементов в массивы и удаления уже существующих элементов из массивов. Операции должны выполняться так, чтобы количество элементов в обоих массивах было одинаковым.

Глава 8
ПРИЛОЖЕНИЯ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ

Профессор, это будет небывалый в мире эксперимент.

из к/ф «Собачье сердце»

В этой главе мы обсудим вопросы, связанные с созданием приложений с графическим интерфейсом. В частности, мы узнаем:

- каковы основные принципы создания приложений с графическим интерфейсом;
- как создается окно формы;
- как реализуются графические компоненты;
- что нужно делать для обработки событий, связанных с графическими компонентами;
- о некоторых возможностях в плане отображения графики.

Понятно, что это далеко не полный список приемов и методик, с которыми нам предстоит познакомиться. На нашем пути будет много нового и интересного, поэтому имеет смысл сразу приступить к делу.

Принципы создания графического интерфейса

— Господин Калиостро, а как насчет портрета?
— Погодите вы, голубчик, с портретом! Дайте ему со скульптурой разобраться.

из к/ф «Формула любви»

Тема, связанная с созданием приложений с графическим интерфейсом, более чем многогранна и исключительно объемна. Мы, в силу объективных причин, все вопросы осветить не сможем, но постараемся

остановиться на наиболее важных и принципиальных. Начнем с базовых принципов, которые лежат в основе технологии создания приложений с графическим интерфейсом.

В общих чертах наша задача состоит в том, чтобы написать программный код, при выполнении которого отображается окно с элементами управления. Это могут быть кнопки, списки, опции, переключатели — список элементов довольно обширный, но все же ограниченный. Среди элементов есть статические — такие, например, как текстовая метка, основное назначение которой состоит в том, чтобы отображать текст. Но большинство элементов функциональные. С ними можно выполнять некоторые манипуляции (например, щелкать по кнопке или выбирать пункт в раскрывающемся списке). Обычно у этих манипуляций есть последствия — ведь элементы управления добавляются именно для того, чтобы программа реагировала на действия, связанные с этими элементами. Как же создать такую программу? Задачу можно разбить на две составляющие, или две подзадачи. Первая связана с созданием собственно графического интерфейса. Вторая подразумевает определение реакции программы на действия с элементами управления графического интерфейса. Каждую подзадачу мы рассмотрим отдельно.

Принцип создания графического интерфейса простой. Каждый элемент управления реализуется с помощью объекта. Для каждого типа элементов есть свой класс. Например, если нам нужна кнопка, то необходимо создавать объект класса `Button`. Если нам нужна метка для отображения текста, то создается объект класса `Label`. Само окно, в которое добавляются элементы управления, реализуется как объект класса `Form`. Соответствующие объекты создаются, определяются их характеристики, размеры и положение, элементы добавляются в контейнер (которым может быть, например, окно). В табл. 8.1 перечислены некоторые (далеко не все) классы из пространства имен `System.Windows.Forms`, которые позволяют создавать объекты для основных графических элементов управления.

Табл. 8.1. Некоторые классы для создания графических элементов

Класс	Описание
`Button`	Класс для создания элемента кнопки
`CheckBox`	Класс для создания элемента опционного типа (опция или кнопка с режимом вдавливания)
`ComboBox`	Класс для создания элемента «список выбора»
`Form`	Класс для создания объекта окна формы
`GroupBox`	Класс для создания элемента, используемого при группировке таких элементов управления, как переключатели

`Label`	Класс для создания объекта текстовой метки (элемент, который может содержать текст)
`ListBox`	Класс для создания объекта элемента, представляющего собой раскрывающийся список
`Panel`	Класс для создания объекта панели. Панель является контейнером для размещения других элементов управления
`RadioButton`	Класс для создания объекта элемента, являющегося переключателем
`TextBox`	Класс для создания объекта элемента, представляющего собой поле для ввода значений

Еще раз подчеркнем, что в действительности классов и элементов (в том числе и всевозможных меню) намного больше. Выше перечислены лишь те, что представляют наибольший интерес и будут задействованы в самой ближайшей перспективе.

Как именно создаются объекты для компонентов, настраиваются их свойства и выполняются прочие сопутствующие операции, мы рассмотрим чуть позже на конкретных примерах. Пикантность ситуации в том, что этого мало. Нужно еще «научить» программу реагировать на действия пользователя с элементами графического интерфейса. Здесь нужно учесть, что есть принципиальное отличие в способе выполнения консольных программ и программ с графическим интерфейсом. Если консольная программа сама обращается к операционной системе, то при выполнении программы с графическим интерфейсом операционная система отправляет сообщения программе. Понять это несложно. Представим, что имеется окно с двумя кнопками. Пользователь может нажать на любую из них. Между кнопками и пользователем есть «посредник» — операционная система. При щелчке по кнопке операционная система, образно выражаясь, сообщает программе, на какой кнопке произошло соответствующее событие. Программа, в свою очередь, должна «уметь» реагировать на него. Это в теории. На практике ситуация следующая. С каждый элементом управления связано определенное количество событий. Здесь событие понимается в программистском смысле. Например, у объектов класса `Button` (кнопка) есть событие `Click`. Данное событие генерируется, если по кнопке выполнен щелчок. Мы можем добавить в список обработчиков события ссылки на методы, которые будут вызываться при возникновении события. Собственно, это то, что нам нужно.

ПОДРОБНОСТИ

У класса кнопки `Button` событий много, не только событие `Click`. Например, есть события `KeyDown` и `KeyUp`, которые генерируются, если при активной кнопке (кнопке передан фокус) соответственно

нажимается и отпускается клавиша на клавиатуре. Событие `MouseEnter` генерируется, когда курсор мыши оказывается над областью кнопки, а событие `MouseLeave` генерируется, когда курсор мыши покидает область кнопки. И так далее. Полный список событий внушительный.

События обрабатываются экземплярами делегата `EventHandler`. Этот делегат определен в пространстве имен `System`. Делегат соответствует методам, не возвращающим результат, у которых два аргумента. Первый аргумент является объектной ссылкой класса `Object`. Через этот аргумент передается ссылка на объект элемента, на котором произошло событие (объект, сгенерировавший событие). Второй аргумент является объектной ссылкой класса `EventArgs` (пространство имен `System`). Через этот аргумент передается ссылка на объект, содержащий информацию о произошедшем событии.

Понятно, что создание приложения с графическим интерфейсом имеет и иные особенности. Но они во многом второстепенны, и соответствующие вопросы мы будем рассматривать по мере обсуждения примеров.

Создание пустого окна

Я за разделение труда, доктор.

из к/ф «Собачье сердце»

Знакомство с приемами создания приложений с графическим интерфейсом начнем с очень простой программы. При запуске программы на выполнение будет отображаться пустое *окно*. Окно представлено на рис. 8.1.

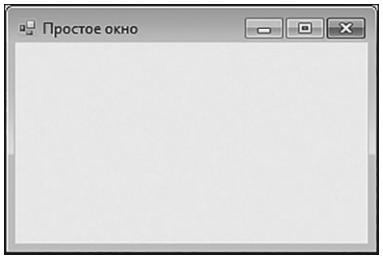


Рис. 8.1. При запуске программы на выполнение отображается пустое окно

Оно не содержит ничего, кроме строки названия, в которой отображается заголовок окна **Простое окно**. Размеры окна можно изменять с помощью мыши, его можно свернуть и развернуть, а для закрытия окна следует щелкнуть по системной пиктограмме с крестиком в правом верхнем углу окна. Программный код, выполнение которого приводит к отображению окна, представлен в листинге 8.1.

Листинг 8.1. Создание пустого окна

```
using System;
using System.Windows.Forms;
// Класс с главным методом:
class SimpleFormDemo{
   // Атрибут использования однопоточной модели:
   [STAThread]
   // Главный метод:
   static void Main(){
      // Создание объекта окна:
      Form mf=new Form();
      // Заголовок окна:
      mf.Text="Простое окно";
      // Высота окна:
      mf.Height=200;
      // Ширина окна:
      mf.Width=300;
      // Отображение окна:
      Application.Run(mf);
   }
}
```

Программа небольшая и содержит всего несколько команд в главном методе. Сначала командой `Form mf=new Form()` создается объект `mf` класса `Form`. Это объект окна. Свойство `Text` объекта окна «отвечает» за заголовок окна (отображается в строке названия окна). Командой `mf.Text="Простое окно"` свойству присваивается значение. Тем

самым мы определяем заголовок окна. Высота окна устанавливается равной 200 пикселей. Для этого значение 200 присваивается свойству Height объекта окна mf (команда mf.Height=200). Для определения ширины окна (в пикселях) командой mf.Width=300 присваиваем значение 300 свойству Width объекта окна mf.

Само по себе создание объекта окна не означает, что окно появится на экране. Мы просто создаем объект, содержащий информацию о свойствах и характеристиках окна. Для отображения окна используется команда Application.Run(mf). В ней мы из класса Application (пространство имен System) вызываем статический метод Run(), а в качестве аргумента методу передается объект окна. В результате окно становится видимым, а также запускается стандартный цикл обработки сообщений для приложения. Проще говоря, окно появляется на экране и пользователь может с ним взаимодействовать.

ПОДРОБНОСТИ

Принцип выполнения приложений с графическим интерфейсом немного отличается от способа выполнения консольных программ. Консольная программа выполняет команды, а если необходимо, то ожидает действий пользователя (при вводе данных, например). Работа приложения с графическим интерфейсом, по большому счету, связана с тем, что приложение реагирует на действия пользователя. Приложение как бы постоянно проверят, выполнил ли пользователь какие-то действия, и если так, то программа реагирует на эти действия. Такой режим «постоянной проверки» реализуется через цикл обработки сообщений для приложения.

В заголовке программы подключаются пространства имен System и System.Windows.Forms. Последнее необходимо для того, чтобы классы графических компонентов (в данном случае класс Form) и другие сопутствующие утилиты для реализации графического интерфейса были доступны в программе. Также стоит обратить внимание на наличие атрибута [STAThread] перед описанием метода Main(). Это инструкция, означающая, что при выполнении приложения необходимо использовать однопоточную модель. Программа у нас очень простая. Она будет работать и без этой инструкции. Но мы все же инструкцию [STAThread] используем, и будем ее использовать в следующих примерах.

ПОДРОБНОСТИ

Инструкция `[STAThread]` нужна для того, чтобы избежать потенциальных проблем при работе с COM-объектами (COM — это сокращение от Component Object Model). Большинство COM-объектов рассчитаны на работу в однопоточной модели STA (сокращение от Single-Threaded Apartment). Проще говоря, COM-объект должен создаваться и использоваться в едином потоке. Иначе могут возникнуть проблемы. По умолчанию в языке C# используется режим MTA (сокращение от Multi-Threaded Apartment). Атрибут `[STAThread]` указывается для главного метода программы `Main()` и представляет собой инструкцию использования однопоточной модели при работе с COM-объектами. Если COM-объекты в программе не используются, то наличие атрибута не имеет видимого эффекта. Создание приложений с графическим интерфейсом подразумевает, что для метода `Main()` указывается атрибут `[STAThread]`.

Уместно также будет вкратце описать способ (один из возможных), которым можно создать приложение с графическим интерфейсом — и в частности такое, как описано выше. Создание приложения начинается с команды **New Project** в меню **File** среды разработки Visual Studio Express. В окне **New Project** выбираем шаблон пустого проекта (**Empty Project**), как показано на рис. 8.2.

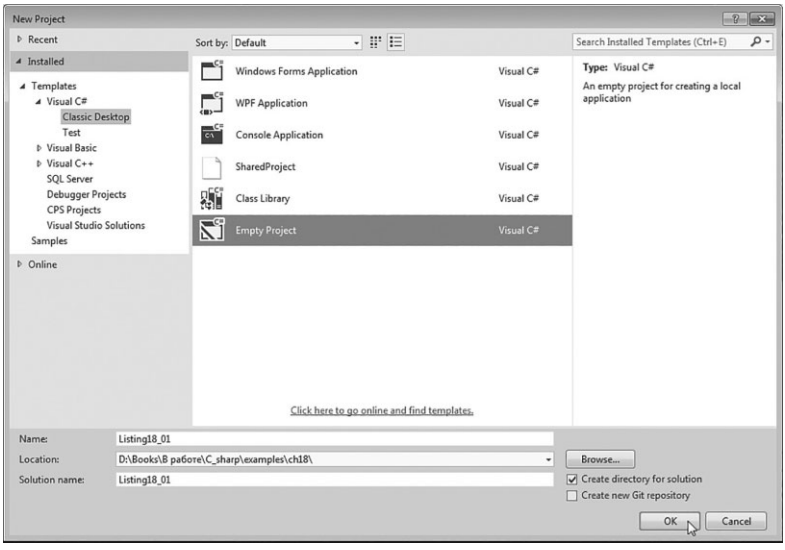


Рис. 8.2. Создание пустого проекта

В результате создается пустой проект. Нам необходимо добавить в него файл с описанием главного класса программы. Для этого можем в контекстном меню проекта в команде-меню **Add** выбрать пункт **Class**, как показано на рис. 8.3.

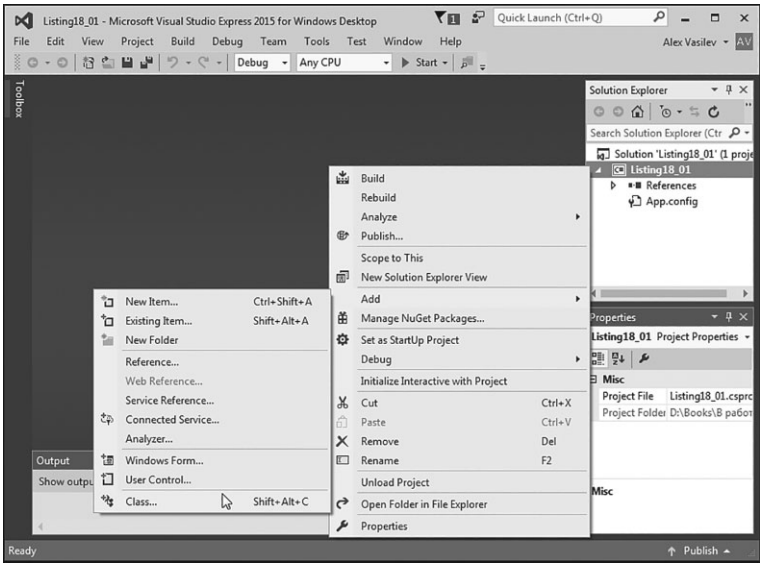


Рис. 8.3. Добавление класса в проект

Откроется диалоговое окно **Add New Item** (рис. 8.4), в котором выбирается тип добавляемого элемента (в данном случае это класс).

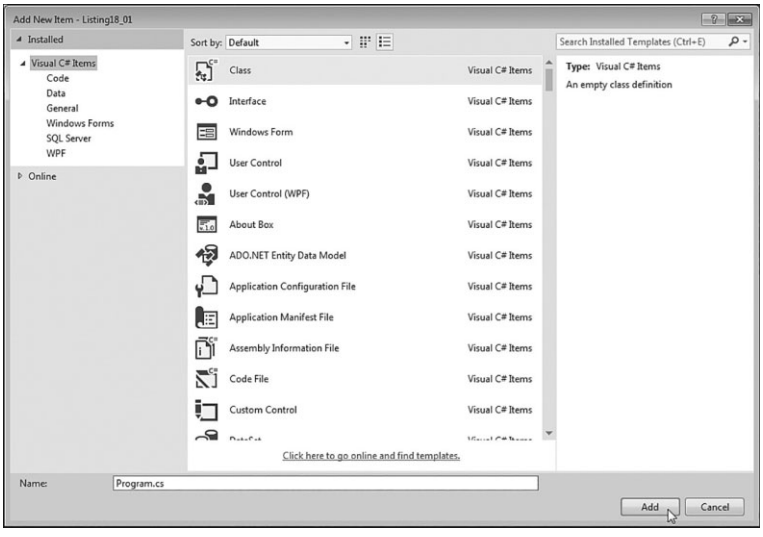


Рис. 8.4. Выбор добавляемого элемента в проект

Также в проект необходимо подключить ссылки (в нашем случае это `System` и `System.Windows.Forms`). Для этого в контекстном меню проекта в команде-меню **Add** выбираем команду **Reference** (рис. 8.5).

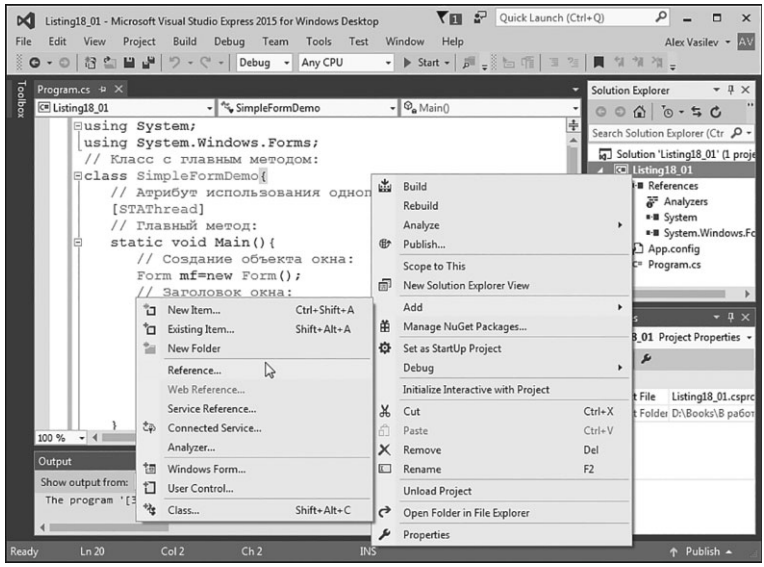


Рис. 8.5. Добавление ссылок в проект

В окне **Reference Manager** устанавливаем флажки для тех ссылок, которые должны быть включены в проект (рис. 8.6).

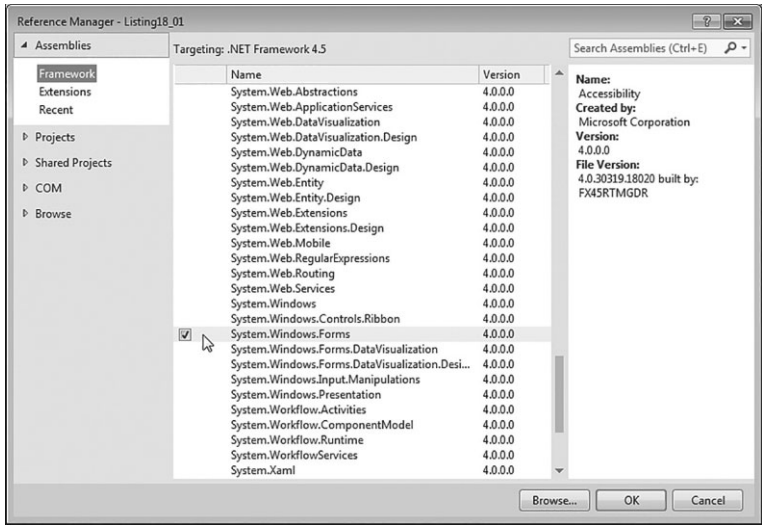


Рис. 8.6. Возле добавляемой ссылки следует установить флажок

По умолчанию для приложения устанавливается консольный режим. Последствия такие: одновременно с открытием окна (созданного в программе) отображается и консольное окно. Чтобы устранить этот «побочный эффект», необходимо в свойствах проекта указать, что приложение не консольное, а приложение Windows. Для этого в контекстном меню проекта выбираем команду **Properties**, как показано на рис. 8.7.

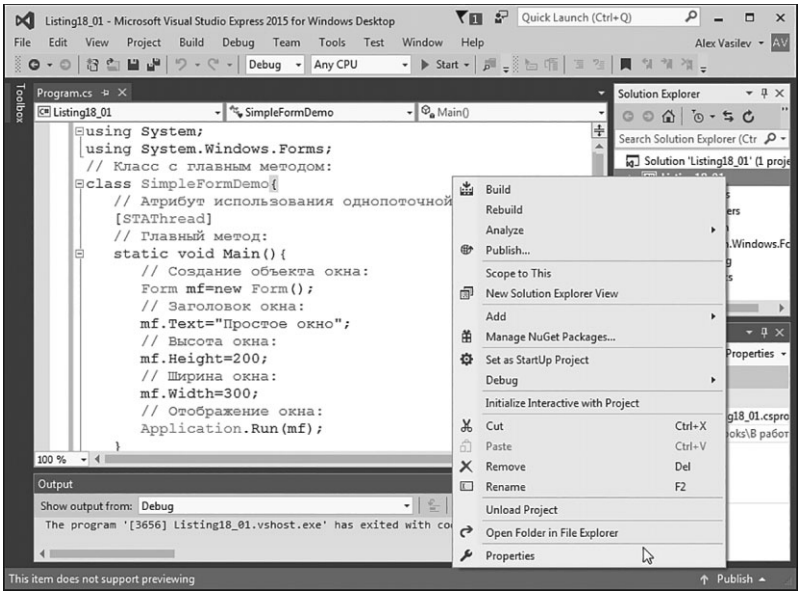


Рис. 8.7. Определение свойств проекта

Откроется вкладка свойств проекта. В этой вкладке в разделе **Application** в раскрывающемся списке **Output type** следует выбрать пункт **Windows Application** (рис. 8.8).

Описанные выше действия по настройке параметров приложения нужно будет выполнять и для прочих примеров, рассматриваемых далее в главе.

НА ЗАМЕТКУ

Выше мы в основном использовали команды контекстного меню проекта. Понятно, что аналогичные действия можно выполнять и через пункты главного меню.

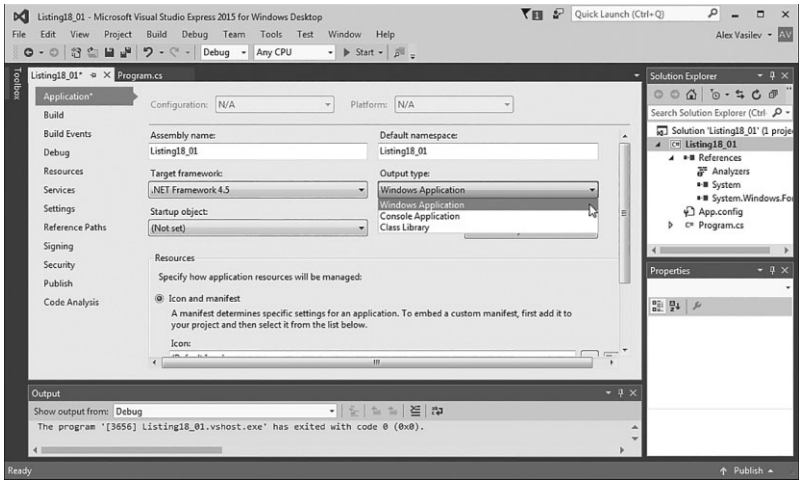


Рис. 8.8. Переход в режим приложения Windows

Окно и обработка событий

Пошли, Скрипач, в открытый космос.

из к/ф «Кин-дза-дза»

Еще один пример, который мы рассмотрим далее, также связан с созданием пустого окна. Но на этот раз окно будет более функциональным. Кроме того, если в предыдущем примере все команды по созданию и отображению окна размещались в главном методе программы, то теперь мы пойдем другим путем. Мы опишем класс, на основе которого будет создаваться окно, и этот класс будет наследовать класс `Form`. Для отображения окна в классе описан специальный статический метод. Также окно реагирует на действия мыши: при наведении курсора мыши на область окна фон окна становится красным. Когда курсор мыши перемещается за пределы области окна, цвет фона окна возвращается к первоначальному значению.

Рассмотрим программный код, представленный в листинге 8.2.

Листинг 8.2. Окно и обработка событий

```
using System;
using System.Drawing;
using System.Windows.Forms;
```

```
// Класс создается наследованием класса Form:
class MyForm:Form{
   // Закрытое поле для запоминания фонового цвета окна:
   private Color baseColor;
   // Конструктор:
   public MyForm(string txt):base(){
      // Заголовок окна:
      Text=txt;
      // Высота окна:
      Height=200;
      // Ширина окна:
      Width=300;
      // Окно неизменных размеров:
      FormBorderStyle=FormBorderStyle.Fixed3D;
      // Считывание значения для фонового цвета окна:
      baseColor=BackColor;
      // Обработчик для события, связанного с наведением
      // курсора мыши на область окна:
      MouseEnter+=(a,b)=>{
         // Цвет фона окна - красный:
         BackColor=Color.Red;
      };
      // Обработчик события, связанного с выходом курсора
      // мыши за пределы области окна:
      MouseLeave+=(a,b)=>{
         // Применение исходного цвета для фона окна:
         BackColor=baseColor;
      };
   }
   // Статический метод для отображения окна:
   public static void show(string txt){
      // Отображение окна:
```

```
      Application.Run(new MyForm(txt));
   }
}
// Главный класс:
class MoreSimpleFormDemo{
   // Атрибут использования однопоточной модели:
   [STAThread]
   // Главный метод:
   static void Main(){
      // Отображение первого окна:
      MyForm.show("Первое окно");
      // Отображение второго окна:
      MyForm.show("Второе окно");
   }
}
```

Класс `MyForm` создается наследованием класса `Form`. Конструктору класса `MyForm` при создании объекта передается текстовый аргумент (обозначен как `txt`). В теле конструктора командой `Text=txt` аргумент присваивается в качестве значения свойству `Text`. Таким образом, аргумент конструктора определяет заголовок окна, реализуемого через объект класса `MyClass`.

Высота окна определяется командой `Height=200`, а команда `Width=300` определяет ширину окна. Кроме этого, мы хотим, чтобы окно имело постоянный размер (то есть чтобы размер окна нельзя было изменить с помощью мыши). Для этого присваивается значение свойству `FormBorderStyle` объекта окна. Значением присваивается константа `Fixed3D` из перечисления `FormBorderStyle` (название такое же, как у свойства). Данная константа означает, что для окна используется объемная рамка, а само окно имеет фиксированные размеры.

Командой `baseColor=BackColor` значение свойства `BackColor` объекта окна присваивается полю `baseColor`. Поле объявлено как относящееся к типу `Color`. Это структура из пространства имен `System.Drawing`. Экземпляры этой структуры определяют цвет, который

можно применять, присваивая в том числе и в качестве значения свойству BackColor.

> **НА ЗАМЕТКУ**
>
> В программе, помимо прочего, подключается пространство имен System.Drawing. Также необходимо добавить соответствующую ссылку в проект.

Поскольку цвет фона окна может меняться, мы используем поле baseColor для того, чтобы запомнить, каким был цвет фона окна в самом начале, при его создании.

В конструкторе определяются обработчики для событий, связанных с наведением курсора мыши на область окна и с перемещением курсора мыши за пределы области окна.

У объектов класса Form (а значит, и у объектов класса MyForm) есть разные события, и среди них: событие MouseEnter и событие MouseLeave. Событие MouseEnter происходит, когда курсор оказывается над областью компонента, для которой определено событие. Событие MouseLeave происходит, когда курсор мыши перемещается за область компонента (окна). В обоих случаях для добавления в список обработчиков событий мы используем лямбда-выражения. Здесь нужно учесть, что для классов графических компонентов события (в том числе MouseEnter и MouseLeave) определены с типом делегата EventHandler (пространство имен System). Это означает, что в список обработчиков событий могут добавляться ссылки на экземпляры делегата EventHandler или ссылки на методы, соответствующие этому делегату. Делегат EventHandler соответствует методам, которые не возвращают результат и у которых два аргумента. Первым аргументом является ссылка класса Object. Подразумевается, что это ссылка на объект компонента, который вызвал событие (с которым событие произошло). Второй аргумент является объектной ссылкой класса EventArgs (пространство имен System). Этот аргумент нужен для передачи информации о параметрах события, которое произошло. Такая информация содержится в объекте, переданном методу в качестве второго аргумента. Что касается класса EventArgs, то он является базовым для классов, объекты которых реально передаются вторым аргументом методам-обработчикам.

НА ЗАМЕТКУ

Когда на компоненте происходит некоторое событие, то для его обработки вызываются методы из списка обработчиков события. Этим методам передается по два аргумента. Первый аргумент — это ссылка на объект компонента, на котором произошло событие. Второй аргумент — это объект, содержащий сведения о событии. Метод может использовать эти аргументы, а может не использовать. Но аргументы у метода должны быть.

Компоненты разные, и поэтому объекты компонентов относятся к разным классам. Но поскольку первый аргумент делегата `EventHandler` является ссылкой класса `Object`, то фактически этот аргумент может быть ссылкой на объект любого класса. Похожая ситуация имеет место со вторым аргументом. Реальные объекты, которые передаются в качестве второго аргумента при вызове методов из списка обработки события, относятся к классам, производным от класса `EventArgs`.

Таким образом, метод, который добавляется в список обработчиков события, должен иметь два аргумента (даже если мы не собираемся эти аргументы использовать). Как отмечалось выше, методы для обработки событий мы определяем с помощью лямбда-выражений. У этих лямбда-выражений есть два формальных аргумента (обозначены как `a` и `b`), которые не используются. В теле метода, добавляемого обработчиком для события `MouseEnter`, выполняется команда `BackColor=Color.Red`, которой устанавливается красный цвет фона окна. В теле метода, добавляемом обработчиком для события `MouseLeave`, выполняется команда `BackColor=baseColor`, которой для фона окна устанавливается исходный цвет (который был у окна при создании).

Помимо конструктора и закрытого поля `baseColor`, в классе `MyForm` описан статический метод `Show()` с текстовым аргументом (обозначен как `txt`). При вызове метода выполняется команда `Application.Run(new MyForm(txt))`. В качестве аргумента метода `Run()` указана инструкция `new MyForm(txt)`, которой создается анонимный объект класса `MyForm`. В результате создается объект окна, и это окно отображается на экране. Выполнение метода `Run()` состоит в опросе на предмет событий, связанных с окном. Проще говоря, пока окно не закрыто, метод `Run()` продолжает работу.

В главном методе программы командой `MyForm.show("Первое окно")` отображается первое окно, представленное на рис. 8.9.

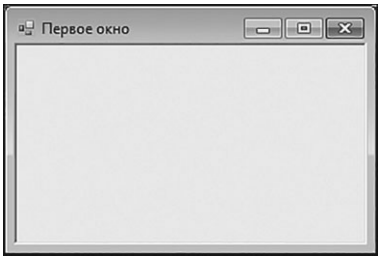


Рис. 8.9. Первое окно, которое отображается при запуске программы на выполнение

Заголовок окна определяется текстом, переданным в качестве аргумента методу `show()`. Размеры окна изменить нельзя (но при этом с помощью системной пиктограммы можно развернуть окно на весь экран), а если навести курсор мыши на область окна, то фон окна станет красным (рис. 8.10).

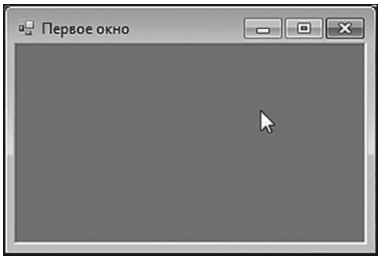


Рис. 8.10. При наведении курсора мыши на область окна фон окна становится красным

Если курсор переместить за пределы области окна, то фон окна станет прежним.

После закрытия окна появляется еще одно окно (результат выполнения команды `MyForm.show("Второе окно")`). Оно показано на рис. 8.11.

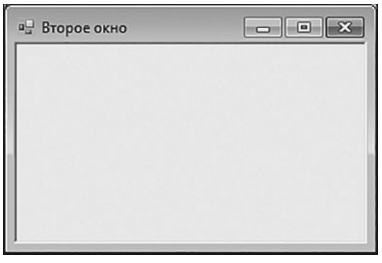


Рис. 8.11. После закрытия первого окна появляется еще одно (второе) окно

От первого окна оно отличается только заголовком. Как и первое окно, второе окно меняет цвет фона при наведении на него курсора мыши.

Кнопки и метки

— А нормального входа в этот универсам нет?
— За нормальный чатлами надо платить, родной!

из к/ф «Кин-дза-дза»

Мы продолжаем знакомиться с простыми программами (с графическим интерфейсом), но постепенно будем изучать разные графические компоненты. Далее рассмотрим программу, в которой используются наиболее простые компоненты — кнопки и текстовые метки. А именно, создадим программу, при запуске которой отображается диалоговое окно с сообщением и *кнопкой*. Щелчок по кнопке приводит к закрытию окна и завершению выполнения программы. Как выглядит окно, отображаемое при запуске программы на выполнение, показано на рис. 8.12.

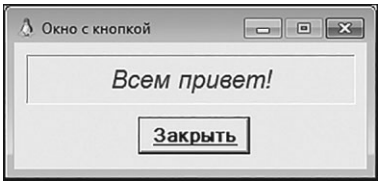


Рис. 8.12. Окно с сообщением и кнопкой

Мы раньше встречались с подобными окнами. От предыдущих случаев данная ситуация отличается тем, что окно с кнопкой «конструируем» мы сами. Программный код, выполнение которого приводит к желаемому результату, представлен в листинге 8.3.

Листинг 8.3. Окно с сообщением и кнопкой

```
using System;
using System.Drawing;
using System.Windows.Forms;
// Класс окна наследует класс Form:
class MyForm:Form{
   // Конструктор:
```

```
public MyForm(string msg,string title):base(){
   this.Text=title;        // Заголовок окна
   // Размеры окна:
   this.Size=new Size(300,140);
   // Окно постоянных размеров:
   this.FormBorderStyle=FormBorderStyle.FixedSingle;
   // Окно нельзя развернуть на весь экран:
   this.MaximizeBox=false;
   // Полный путь к файлу с пиктограммой для окна:
   string file="D:/Books/pictures/csharp/penguin.ico";
   // Пиктограмма для окна:
   this.Icon=new Icon(file);
   // Создание объекта метки:
   Label lbl=new Label();
   // Текст метки:
   lbl.Text=msg;
   // Положение метки в контейнере (окне):
   lbl.Location=new Point(10,10);
   lbl.Width=265;      // Ширина метки
   lbl.Height=40;      // Высота метки
   // Текст в метке выравнивается по центру:
   lbl.TextAlign=ContentAlignment.MiddleCenter;
   // Тип рамки для метки:
   lbl.BorderStyle=BorderStyle.Fixed3D;
   // Объект шрифта:
   Font F=new Font("Arial",15,FontStyle.Italic);
   // Применение шрифта для метки:
   lbl.Font=F;
   // Цвет шрифта для метки:
   lbl.ForeColor=Color.Blue;
   // Добавление метки в окно:
   this.Controls.Add(lbl);
```

```
      // Создание объекта кнопки:
      Button btn=new Button();
      // Текст (название) для кнопки:
      btn.Text="Закрыть";
      // Положение и размеры кнопки:
      btn.Bounds=new Rectangle(100,60,90,30);
      // Шрифт для кнопки:
      btn.Font=new Font(btn.Font.Name, // Название шрифта
                  btn.Font.Size+3,    // Размер шрифта
                  // Стиль шрифта:
                  FontStyle.Underline|FontStyle.Bold
      );
      // Добавление обработчика для события, связанного
      // со щелчком по кнопке:
      btn.Click+=delegate(object a,EventArgs b){
         // Завершение работы приложения:
         Application.Exit();
      };
      // Добавление кнопки в окно:
      this.Controls.Add(btn);
   }
}
// Главный класс:
class ButtonAndLabelDemo{
   [STAThread]
   // Главный метод:
   static void Main(){
      // Создание объекта окна:
      MyForm mf=new MyForm("Всем привет!",  // Заголовок
                           "Окно с кнопкой" // Текст в окне
      );
      // Отображение окна:
```

```
        Application.Run(mf);
    }
}
```

В программе описывается класс окна `MyForm`, который является производным от класса `Form`. Все самое интересное происходит в конструкторе этого класса. В качестве аргументов конструктору передаются два текстовых значения: первое `msg` определяет сообщение, отображаемое в окне, а второе `title` определяет заголовок окна. А именно, в теле конструктора командой `this.Text=title` свойству `Text` объекта окна присваивается значение `title`, чем определяется заголовок, отображаемый в строке названия окна.

НА ЗАМЕТКУ

В конструкторе класса `MyForm` используется три объекта для графических компонентов (объект окна, объект метки и объект кнопки), а также большое количество свойств, структур, классов, перечислений (и некоторые названия совпадают). Чтобы программный код читался легче, в теле конструктора использована явная ссылка `this` на объект окна (хотя необходимости в этом нет).

Размеры окна определяются командой `this.Size=new Size(300,140)`, которой свойству `Size` окна присваивается экземпляр структуры `Size`. При создании экземпляра структуры аргументами конструктору передаются размеры окна.

При выполнении команды `this.FormBorderStyle=FormBorderStyle.FixedSingle` свойству `FormBorderStyle` объекта окна в качестве значения присваивается константа `FixedSingle` (простая рамка постоянных размеров) из перечисления `FormBorderStyle`. Получается окно постоянных размеров, но при этом у окна будет пиктограмма, которая позволяет развернуть окно на весь экран. Мы используем команду `this.MaximizeBox=false`, чтобы заблокировать работу соответствующей пиктограммы для окна.

Мы используем собственную пиктограмму для окна (отображается в строке названия окна в левом верхнем углу). Для этого в текстовую переменную `file` записывается полный путь к файлу с изображением для пиктограммы (файл `penguin.ico` в каталоге `D:\Books\pictures\csharp`).

ПОДРОБНОСТИ

Обратный слеш \ используется в управляющих символах (таких, как \n или \t). Поэтому в текстовом значении, определяющем путь к файлу, используют или слеш /, или двойной обратный слеш \\.

Пиктограмма для окна определяется командой `this.Icon=new Icon(file)`, которой свойству `Icon` объекта окна присваивается ссылка на объект класса `Icon`. При создании объекта аргументом конструктору передается текстовая строка с полным путем к файлу с изображением для пиктограммы.

Объект текстовой *метки* создается командой `Label lbl=new Label()`. Командой `lbl.Text=msg` определяем текст, который будет отображаться в метке. Положение метки в контейнере (которым будет окно) определяем инструкцией `lbl.Location=new Point(10,10)`: свойству `Location` объекта метки `lbl` значением присваивается ссылка на объект класса `Point`. Размеры метки в пикселях задаем так: присваиваем значения свойствам `Width` (ширина) и `Height` (высота) объекта метки. Для выравнивания текста в области метки по центру по вертикали и горизонтали свойству `TextAlign` объекта метки значением присваивается константа `MiddleCenter` из перечисления `ContentAlignment`. Тип рамки для метки определяется командой `lbl.BorderStyle=BorderStyle.Fixed3D` (метка выделяется с использованием объемного эффекта — получается «вдавленная» в окно метка). Также мы хотим самостоятельно определить шрифт для отображения текста в метке. Для этого создаем объект шрифта (объект `F` класса `Font`). При создании объекта (команда `Font F=new Font("Arial",15,FontStyle.Italic)`) аргументами конструктору класса `Font` передаются: название шрифта `"Arial"`, размер шрифта `15`, а также константа `Italic` из перечисления `FontStyle` (данная константа означает, что используется курсивный шрифт). Для применения созданного шрифта для метки свойству `Font` объекта метки присваивается ссылка на объект шрифта (команда `lbl.Font=F`).

Мы хотим, чтобы в метке цвет текста был синим. Для этого свойству `ForeColor` в качестве значения присваивается константа `Blue` из структуры `Color` (команда `lbl.ForeColor=Color.Blue`). Для добавления метки в окно мы из свойства `Controls` объекта окна вызываем метод `Add()`, аргументом которому передается ссылка на объект метки.

Также в конструкторе класса окна создается объект *кнопки* (команда `Button btn=new Button()`). Название, которое отображается на кнопке, определяется значением свойства `Text` объекта кнопки. Поэтому при выполнении команды `btn.Text="Закрыть"` определяется название кнопки. Положение и размеры кнопки определяются, когда свойству `Bounds` объекта кнопки присваивается ссылка на объект класса `Rectangle`. Конструктору класса `Rectangle` при создании объекта передаются четыре целых числа. Первые два числа определяют положение кнопки в контейнере (в окне). Два других числа определяют ширину и высоту кнопки.

> **НА ЗАМЕТКУ**
>
> Обычно существуют разные способы определять характеристики графических компонентов. Это относится к способу определения положения графического компонента и его размеров. Мы в рассматриваемом примере используем разные способы для определения положения и размеров компонентов для окна, метки и кнопки, хотя могли бы использовать однотипные подходы. Данное разнообразие связано с желанием продемонстрировать разные подходы для решения однотипных задач.

Для кнопки мы также определяем шрифт. В этом случае свойству `Font` объекта кнопки `btn` значением присваивается ссылка на объект класса `Font`. При создании объекта шрифта первым аргументом конструктору передается значение `btn.Font.Name`. Это текущее название шрифта для кнопки. Проще говоря, мы используем тот шрифт, который предполагалось использовать по умолчанию. Вторым аргументом конструктору класса `Font` передается выражение `btn.Font.Size+3`. Данное значение — это увеличенный на `3` текущий размер шрифта, используемый по умолчанию для отображения текста в кнопке. Наконец, третьим аргументом конструктору класса `Font` передается выражение `FontStyle.Underline|FontStyle.Bold`. Это две константы `Underline` и `Bold` из перечисления `FontStyle`, объединенные оператором `|`. Данное выражение означает, что используется и подчеркивание (константа `Underline`), и выделение жирным стилем (константа `Bold`).

Для события `Click` кнопки, которое связано со щелчком по кнопке, добавляется обработчик, основанный на анонимном методе. В теле анонимного метода вызывается команда `Application.Exit()`, которой завершается работа приложения.

ПОДРОБНОСТИ

Напомним, что события для графических компонентов относятся к делегату `EventHandler`. Этот делегат подразумевает, что метод, используемый для обработки события, должен иметь два аргумента: объектная ссылка класса `Object` (ссылка на объект компонента, на котором произошло событие) и объектная ссылка класса `EventArgs` (ссылка на объект, содержащий информацию о произошедшем событии). Поэтому анонимный метод, который добавляется в список вызовов события `Click`, описывается с двумя аргументами.

Для добавления кнопки в окно из свойства `Controls` объекта окна вызывается метод `Add()`, аргументом которому передана ссылка на объект кнопки.

НА ЗАМЕТКУ

Для использования структур `Size` и `Color`, классов `Icon`, `Point`, `Rectangle` и `Font`, а также перечислений `ContentAlignment` и `FontStyle` в заголовке программы добавлена инструкция `using System.Drawing`. Кроме этого необходимо добавить соответствующую ссылку в проект.

В главном методе программы создается объект окна (объект `mf` класса `MyForm`). Этот объект передается в качестве аргумента методу `Run()`, который вызывается из класса `Application`. В результате отображается диалоговое окно (см. рис. 8.12). При закрытии окна приложение завершает работу.

ПОДРОБНОСТИ

Мы можем закрыть окно, либо щелкнув по кнопке **Закрыть**, либо щелкнув по системной пиктограмме в правом верхнем углу окна. Но действия эти не эквивалентные (хотя последствия одинаковые). При щелчке по кнопке **Закрыть** завершается выполнение приложения. При щелчке по системной пиктограмме закрывается окно. Но поскольку в главном методе других команд нет, то после этого завершается выполнение приложения.

Использование списков

Да, и что вы можете со своей стороны предложить?

из к/ф «Собачье сердце»

В следующей программе мы познакомимся с таким графическим компонентом, как *раскрывающийся список*. Раскрывающийся список реализуется через объект класса `ComboBox`. Но прежде чем приступить к рассмотрению программного кода, остановимся на том, как программа работает. Так будет легче понять программный код.

При запуске программы появляется диалоговое окно, представленное на рис. 8. 13.

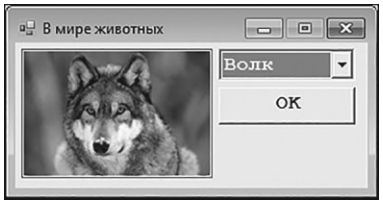


Рис. 8.13. Диалоговое окно с изображением волка

Половину окна занимает изображение. В правой части окна есть раскрывающийся список и кнопка **ОК**. В раскрывающемся списке выбирается название животного, а в области изображения появляется соответствующая картинка. При отображении окна в списке выбран элемент **Волк**, и соответственно в окне отображается картинка с волком. Какие еще элементы есть в раскрывающемся списке, показано на рис. 8.14.

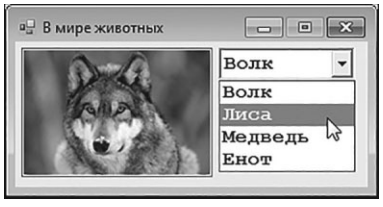


Рис. 8.14. В раскрывающемся списке можно выбрать элемент (название животного)

При выборе нового элемента изменения (смена картинки) наступают автоматически. На рис. 8.15 показано окно с изображением лисы.

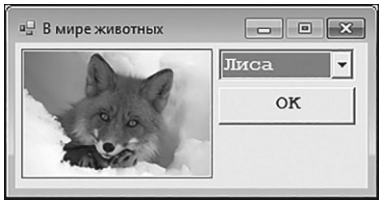


Рис. 8.15. Диалоговое окно с изображением лисы

Окно с изображением медведя представлено на рис. 8.16.

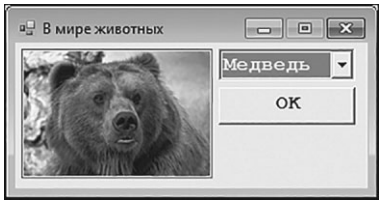


Рис. 8.16. Диалоговое окно с изображением медведя

Наконец, окно с изображением енота показано на рис. 8.17.

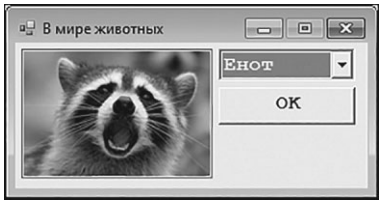


Рис. 8.17. Диалоговое окно с изображением енота

Если щелкнуть по кнопке **OK**, то окно закроется и программа завершит выполнение. Такая вот простая программа. Ее код представлен в листинге 8.4.

Листинг 8.4. Раскрывающийся список

```
using System;
using System.Drawing;
using System.Windows.Forms;
// Класс окна является производным от класса Form:
class MyForm:Form{
   // Закрытое поле является текстовым массивом
   // с названиями животных:
```

```
private string[] animals={"Волк","Лиса","Медведь","Енот"};
// Закрытое поле является текстовым массивом
// с названиями файлов с изображениями:
private string[] files={"wolf.jpg","fox.jpg","bear.jpg","raccoon.jpg"};
// Закрытое поле содержит путь к каталогу с файлами:
private string path="D:/Books/pictures/csharp/";
// Закрытое поле - значение индекса для выбранного
// элемента из массива:
private int index=0;
// Закрытое поле – ссылка на объект метки:
private Label pict;
// Закрытое поле – ссылка на объект
// раскрывающегося списка:
private ComboBox list;
// Конструктор:
public MyForm():base(){
   // Заголовок окна:
   this.Text="В мире животных";
   // Размеры окна:
   this.Size=new Size(300,155);
   // Окно фиксированных размеров:
   this.FormBorderStyle=FormBorderStyle.FixedSingle;
   // Окно нельзя развернуть на весь экран:
   this.MaximizeBox=false;
   // Создание объекта метки:
   pict=new Label();
   // Положение и размеры метки:
   pict.SetBounds(5,5,154,104);
   // Тип рамки вокруг метки:
   pict.BorderStyle=BorderStyle.FixedSingle;
   // Изображение для метки:
   pict.Image=Image.FromFile(path+files[index]);
```

```
// Добавление метки в окно:
this.Controls.Add(pict);
// Создание объекта для списка:
list=new ComboBox();
// Содержимое списка:
list.Items.AddRange(animals);
// Тип списка:
list.DropDownStyle=ComboBoxStyle.DropDownList;
// Индекс выбранного в списке элемента:
list.SelectedIndex=index;
// Положение списка:
list.Left=pict.Right+5;
list.Top=pict.Top;
// Размеры списка:
list.Size=new Size(110,50);
// Шрифт для списка:
list.Font=new Font("Courier New",12,FontStyle.Bold);
// Регистрация метода обработчиком для события,
// связанного с выбором нового элемента в списке:
list.SelectedIndexChanged+=OnItemChanged;
// Добавление списка в окно:
this.Controls.Add(list);
// Создание объекта кнопки:
Button btn=new Button();
// Название кнопки:
btn.Text="OK";
// Положение кнопки:
btn.Left=list.Left;
btn.Top=list.Bottom+10;
// Размеры кнопки:
btn.Width=list.Width;
btn.Height=30;
```

```
      // Шрифт для кнопки:
      btn.Font=list.Font;
      // Регистрация метода обработчиком события,
      // связанного со щелчком по кнопке:
      btn.Click+=OnButtonClick;
      // Добавление кнопки в окно:
      this.Controls.Add(btn);
   }
   // Метод для обработки события, связанного с выбором
   // в списке нового элемента:
   private void OnItemChanged(object obj,EventArgs ea){
      // Индекс выбранного в списке элемента:
      int index=list.SelectedIndex;
      // Новое изображение для метки:
      pict.Image=Image.FromFile(path+files[index]);
   }
   // Метод для обработки события, связанного со щелчком
   // по кнопке:
   private void OnButtonClick(object obj,EventArgs ea){
      // Завершение выполнения приложения:
      Application.Exit();
   }
}
// Главный класс:
class ComboBoxDemo{
   [STAThread]
   // Главный метод:
   static void Main(){
      // Отображение окна:
      Application.Run(new MyForm());
   }
}
```

Мы описываем класс `MyForm`, который является производным от класса `Form`. В классе есть закрытое поле `animals`, являющееся текстовым массивом с названиями животных. Элементы этого массива, как мы увидим далее, определяют содержимое раскрывающегося списка. Еще одно закрытое поле `files` является текстовым массивом с названиями файлов с изображениями животных. Между элементами массивов `animals` и `files` должно быть строгое взаимно-однозначное соответствие: название животного в массиве `animals` должно иметь такой же индекс, как название файла с изображением этого животного в массиве `files`.

Закрытое текстовое поле `path` содержит текст, определяющий путь к каталогу с файлами изображений (значение `"D:/Books/pictures/csharp/"`).

ПОДРОБНОСТИ

Для корректной работы программы в папку `D:\Books\pictures\csharp` должны быть помещены файлы с изображениями животных. В данном случае названия файлов `wolf.jpg`, `fox.jpg`, `bear.jpg` и `raccoon.jpg` (все файлы размером `150` на `100` пикселей). Именно эти названия указаны в массиве `files`. Если читатель планирует использовать другой каталог для хранения файлов и/или другие названия для файлов, то соответствующие изменения следует внести в программный код.

Закрытое целочисленное поле `index` с нулевым значением определяет индекс элемента в раскрывающемся списке, который будет выбран при начальном отображении диалогового окна.

Кроме этих полей, в классе `MyForm` есть еще несколько закрытых полей, используемых в качестве ссылок на объекты графических компонентов. Точнее, их два: поле `pict` класса `Label` (ссылка на объект метки) и поле `list` класса `ComboBox` (ссылка на объект раскрывающегося списка).

В конструкторе класса `MyForm()` в основном выполняются рутинные операции по настройке внешнего вида окна и его компонентов. Большинство из них уже должны быть знакомы читателю. Поэтому остановимся на наиболее важных и принципиально новых командах. Так, положение и размеры объекта метки задаются с помощью метода `SetBounds()`. Аргументы метода — это координаты метки и ее размеры (ширина и высота). Вокруг метки отображается рамка, тип которой определяется командой `pict.BorderStyle=BorderStyle.FixedSingle`. Хотя метка

обычно называется текстовой, в данном случае она содержит изображение. Изображение, связанное с меткой, определяется командой `pict.Image=Image.FromFile(path+files[index])`, которой свойству `Image` объекта метки присваивается ссылка на объект, который создается вызовом статического метода `FromFile()` из класса `Image`. Аргументом методу передается текст с полным путем к файлу с изображением (в данном случае такой текст получается объединением содержимого переменной `path` и элемента `files[index]`).

> **НА ЗАМЕТКУ**
>
> Размеры метки заданы на `4` пикселя больше размеров изображений, отображаемых в области метки.

Объект списка создается командой `list=new ComboBox()`. Чтобы заполнить список содержимым массива `animals`, используем команду `list.Items.AddRange(animals)`. Здесь из свойства `Items` объекта списка, возвращающего ссылку на коллекцию элементов списка, вызывается метод `AddRange()`. Аргументом методу передается ссылка на массив `animals`. В результате элементы массива добавляются в список.

Стиль, используемый по умолчанию для списка, подразумевает, что в поле выбора выбранное значение можно редактировать. Нас такой вариант не устраивает, поэтому мы используем команду `list.DropDownStyle=ComboBoxStyle.DropDownList`, которой свойству `DropDownStyle` значением присваивается константа `DropDownList` из перечисления `ComboBoxStyle`. В результате получаем раскрывающийся список без возможности редактировать выбранное значение. Присвоив значение свойству `SelectedIndex` (команда `list.SelectedIndex=index`), определяем элемент в списке, который будет выбран по умолчанию (свойству в качестве значения присваивается индекс выбранного по умолчанию элемента).

Положение списка в окне выбираем в соответствии с положением метки с изображением. Команда `list.Left=pict.Right+5` означает, что левый край списка сдвинут на `5` пикселей по отношению к правому краю метки. По вертикали список находится на том же уровне, что и метка (команда `list.Top=pict.Top`).

Размер списка определяется командой `list.Size=new Size(110,50)`, то есть присваиванием значения свойству `Size`. Шрифт для списка

определяем командой, в которой свойству Font объекта списка list значением присваивается выражение new Font("Courier New", 12,FontStyle.Bold) (объект шрифта). Но одна команда принципиально важная. Это инструкция list.SelectedIndexChanged+= OnItemChanged, которой в список обработчиков события SelectedIndexChanged объекта списка добавляется ссылка на метод OnItemChanged() (закрытый метод класса MyForm). Событие возникает, когда в списке выбирается новый элемент. В таком случае будет вызываться метод OnItemChanged().

Кнопка реализуется как объект btn класса Button. Название кнопки (текст в кнопке) определяем с помощью команды btn.Text="OK". Горизонтальная координата кнопки совпадает с левой координатой списка (команда btn.Left=list.Left). По вертикали кнопка размещена на 10 пикселей ниже нижней границы списка (команда btn.Top=list.Bottom+10). Ширина кнопки такая же, как ширина списка (команда btn.Width=list.Width). Высота кнопки равна 30 пикселям (команда btn.Height=30). Шрифт для кнопки применяется такой же, как и шрифт для списка (команда btn.Font=list.Font). Для обработки события, связанного со щелчком по кнопке, регистрируется закрытый метод OnButtonClick() класса MyForm.

НА ЗАМЕТКУ

Для добавления компонентов в окно используется метод Add(). Аргументом методу передается ссылка на объект компонента. Метод вызывается из свойства Controls объекта окна.

Кроме конструктора, в классе MyForm описано еще два закрытых метода. При вызове метода OnItemChanged() (используется для обработки событий, связанных с выбором нового элемента в списке) командой int index=list.SelectedIndex запоминается индекс выбранного в раскрывающемся списке элемента (значение этого индекса возвращается свойством SelectedIndex). После этого командой pict.Image=Image.FromFile(path+files[index]) для метки определяется новое изображение. Здесь мы приняли во внимание, что названия животных в массиве animals, на основе которого формируется содержимое списка, и название файлов с изображениями животных в массиве files размещены в строгом соответствии друг другу. Поэтому если мы знаем индекс, под которым название животного входит в массив

`animals` (а значит, и в список), то такой же индекс будет у элемента с названием файла с изображением этого животного в массиве `files`.

Метод `OnButtonClick()` используется для обработки события, связанного со щелчком по кнопке. При вызове метода выполняется команда `Application.Exit()`, завершающая работу приложения.

ПОДРОБНОСТИ

Напомним, что события для графических компонентов относятся к типу делегата `EventHandler`. Этот делегат подразумевает, что соответствующий метод имеет два аргумента: первый — ссылка на объект класса `object` и второй — ссылка на объект класса `EventArgs`. Поэтому методы `OnItemChanged()` и `OnButtonClick()` описаны с двумя аргументами, хотя эти аргументы в методах не используются.

В главном методе командой `Application.Run(new MyForm())` создается объект окна, и окно отображается на экране.

Еще один вариант программы с выбором изображений животных реализован так, что вместо раскрывающегося списка используется *список выбора* (он полностью отображается в окне, так что для выбора элемента раскрывать ничего не нужно).

НА ЗАМЕТКУ

Если содержимое списка выбора не помещается в выделенную под список область, то по умолчанию появляется полоса прокрутки и список можно просматривать с помощью прокрутки содержимого.

При запуске программы на выполнение отображается окно, представленное на рис. 8.18.

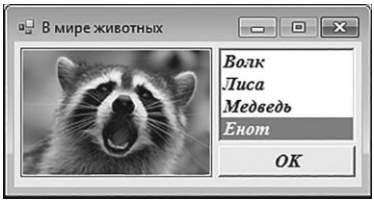


Рис. 8.18. Окно с изображением енота и списком выбора

Окно содержит список выбора из четырех элементов, и в самом начале выбран последний элемент в списке. Если выбрать другой элемент в списке, автоматически изменится и изображение в окне. На рис. 8.19 показано окно с изображением лисы.

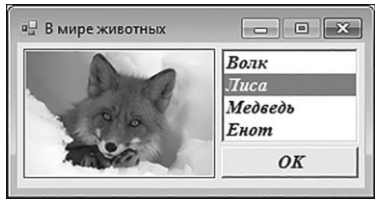


Рис. 8.19. Окно с изображением лисы

При щелчке по кнопке **OK** окно закрывается. Далее рассмотрим программный код, представленный в листинге 8.5.

Листинг 8.5. Использование списка выбора

```
using System;
using System.Drawing;
using System.Windows.Forms;
// Класс для окна:
class MyForm:Form{
   // Конструктор:
   public MyForm(string[,] data,string title,string path):base(){
      // Заголовок окна:
      this.Text=title;
      // Размер окна:
      this.Size=new Size(295,155);
      // Окно отображается в центре экрана:
      this.StartPosition=FormStartPosition.CenterScreen;
      // Окно фиксированных размеров:
      this.FormBorderStyle=FormBorderStyle.FixedSingle;
      // Окно нельзя развернуть на весь экран:
      this.MaximizeBox=false;
      // Шрифт для окна:
```

```
this.Font=new Font("Times New Roman",11,FontStyle.Italic|FontStyle.Bold);
// Объект метки:
Label pict=new Label();
// Положение и размеры метки:
pict.SetBounds(5,5,154,104);
// Рамка вокруг метки:
pict.BorderStyle=BorderStyle.FixedSingle;
// Добавление метки в окно:
this.Controls.Add(pict);
// Объект списка:
ListBox list=new ListBox();
// Размеры списка:
list.Size=new Size(110,80);
// Горизонтальная координата списка:
list.Left=pict.Right+5;
// Вертикальная координата списка:
list.Top=pict.Top;
// Добавление элементов в список:
for(int k=0;k<data.GetLength(1);k++){
   list.Items.Add(data[0,k]);
}
// Обработчик события, связанного с выбором
// нового элемента в списке:
list.SelectedIndexChanged+=(x,y)=>{
   // Индекс выбранного элемента:
   int index=list.SelectedIndex;
   // Новое изображение для метки:
   pict.Image=Image.FromFile(path+data[1,index]);
};
// Добавление списка в окно:
this.Controls.Add(list);
// Объект кнопки:
```

```
      Button btn=new Button();
      // Название кнопки:
      btn.Text="OK";
      // Размеры кнопки:
      btn.Width=list.Width;
      btn.Height=25;
      // Координаты кнопки:
      btn.Left=list.Left;
      btn.Top=pict.Bottom-btn.Height;
      // Обработчик события, связанного со щелчком
      // по кнопке:
      btn.Click+=(x,y)=>{
         Application.Exit();
      };
      // Добавление кнопки в окно:
      this.Controls.Add(btn);
      // В списке выбирается последний элемент:
      list.SetSelected(list.Items.Count-1,true);
   }
}
// Главный класс:
class ListDemo{
   [STAThread]
   // Главный метод:
   static void Main(){
      // Двумерный массив с названиями животных и
      // названиями файлов с изображениями:
      string[,] data={{"Волк","Лиса","Медведь","Енот"},
                      {"wolf.jpg","fox.jpg","bear.jpg","raccoon.jpg"}};
      // Заголовок для окна:
      string title="В мире животных";
      // Путь к каталогу с файлами изображений:
```

```
        string path="D:/Books/pictures/csharp/";
        // Создание объекта окна:
        MyForm wnd=new MyForm(data,title,path);
        // Отображение окна:
        Application.Run(wnd);
    }
}
```

Вообще, программа очень похожа на предыдущую (см. Листинг 8.4), но имеются некоторые новшества. Главное, конечно, связано с тем, что вместо объекта класса `ComboBox` используется объект класса `ListBox`.

Класс `MyForm` содержит только описание конструктора. Конструктору класса `MyForm` первым аргументом передается двумерный текстовый массив `data`. Первая строка массива должна содержать названия животных, а во второй строке должны содержаться названия файлов с изображениями. Вторым аргументом конструктору класса `MyForm` передается текст `title`, определяющий название (заголовок) окна. Третий аргумент конструктора — текстовое значение `path` с полным путем к каталогу, в котором файлы с изображениями животных.

Среди наиболее интересных команд в теле конструктора имеет смысл выделить такие:

- Для отображения окна в центре экрана использована команда `this.StartPosition=FormStartPosition.CenterScreen`. Для этого свойству `StartPosition` объекта окна в качестве значения присваивается константа `CenterScreen` из перечисления `FormStartPosition`. Свойство `StartPosition` определяет начальную позицию окна на экране.
- Командой `this.Font=new Font("Times New Roman",11,FontStyle.Italic|FontStyle.Bold)` для окна задается шрифт. Такой шрифт будет применяться ко всем компонентам в окне (если для этих компонентов шрифт не будет указан явно). Инструкция `FontStyle.Italic|FontStyle.Bold` означает, что используется курсивный жирный стиль начертания шрифта.
- Объект `list` для списка создается командой `ListBox list=new ListBox()`. То есть это объект класса `ListBox`.

- Элементы в список выбора добавляются индивидуально, для чего использован оператор цикла. В нем индексная переменная `k` перебирает значения второго индекса в массиве `data`. За каждую итерацию цикла выполняется команда `list.Items.Add(data[0,k])`, которой в список добавляется очередной элемент из первой строки двумерного массива `data`.
- Обработчиком для события, связанного с выбором нового элемента в списке, является анонимный метод, код которого состоит из двух команд. Командой `int index=list.SelectedIndex` определяется индекс выбранного в списке элемента, а командой `pict.Image=Image.FromFile(path+data[1,index])` для метки задается новое изображение (название файла с изображением выбирается из второй строки двумерного текстового массива `data`).
- Обработчик для события, связанного со щелчком по кнопке, также реализуется в виде анонимного метода.
- Командой `list.SetSelected(list.Items.Count-1,true)` в списке выбирается элемент, который последним добавлялся в список. Для выбора элемента в списке из объекта списка вызывается метод `SetSelected()`. Вторым аргументом методу передается значение `true`, означающее, что соответствующий элемент выделяется. Первый аргумент метода определяет индекс выбираемого в списке элемента. Мы использовали выражение `list.Items.Count-1`. Свойство `Items` возвращает ссылку на коллекцию элементов списка. Свойство `Count` возвращает количество элементов в коллекции. Индекс последнего элемента в коллекции на единицу меньше количества элементов в коллекции. При выделении элемента в списке происходит событие `SelectedIndexChanged`, и в результате вызова метода-обработчика пиктограмма получает изображение, соответствующее выбранному элементу.

В главном методе создается двумерный массив `data` с названиями животных и названиями файлов с изображениями, переменная `title` со значением для заголовка окна и переменная `path`, содержащая текст с путем к каталогу с файлами. Эти переменные передаются в качестве аргументов конструктору класса `MyForm` при создании объекта окна `wnd`. Затем окно отображается на экране.

Использование переключателей

Двум богам служить нельзя, дорогой доктор.

из к/ф «Собачье сердце»

В следующем примере мы прибегнем к помощи *переключателей* для того, чтобы показывать в диалоговом окне изображения животных. На рис. 8.20 показано, как выглядит окно при запуске программы на выполнение.

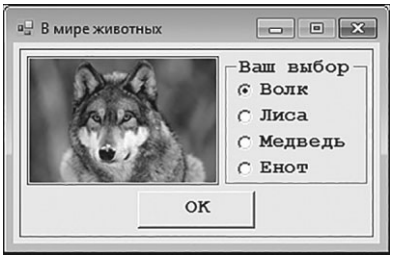


Рис. 8.20. Окно с группой переключателей и изображением волка

Теперь в правой части окна вместо списка размещена группа переключателей. Установлен (выбран) может быть один и только один из них. В соответствии с тем, какой переключатель установлен, в области метки отображается картинка с изображением животного. На рис. 8.21 показано окно с изображением медведя (в соответствии с выбранным переключателем).

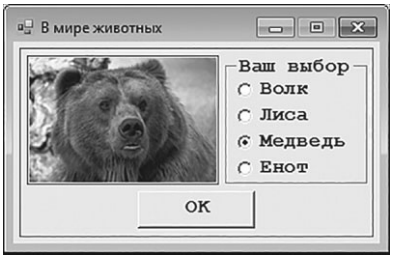


Рис. 8.21. Окно с группой переключателей и изображением медведя

Переключатель реализуется как объект класса `RadioButton`. Особенность переключателей в том, что они объединяются в группы и в группе может быть установлен только один переключатель. Поэтому мало собственно создать переключатели. Их еще нужно сгруппировать. Для этого создается объект для группы переключателей. Такой объект создается

на основе класса `GroupBox`. Все переключатели, добавленные в объект группы, образуют группу переключателей.

В программе, кроме переключателей, появляется еще один новый для нас компонент. Это *панель*. Панель реализуется через объект класса `Panel`. Вообще панель — это прямоугольная область, главное назначение которой состоит в том, чтобы быть контейнером для размещения графических компонентов. В рассматриваемой далее программе создается панель, на эту панель помещается метка с изображением, группа переключателей и кнопка, а затем панель добавляется в окно. Как это выглядит на практике, показано в листинге 8.6.

Листинг 8.6. Использование переключателей

```
using System;
using System.Drawing;
using System.Windows.Forms;
// Класс для кнопки:
class MyButton:Button{
   // Конструктор:
   public MyButton(string name):base(){
      // Название кнопки:
      this.Text=name;
      // Обработчик события, связанного со щелчком
      // по кнопке:
      this.Click+=(x,y)=>{
         Application.Exit();
      };
   }
}
// Класс окна:
class MyForm:Form{
   // Названия животных:
   private string[] animals={"Волк","Лиса","Медведь","Енот"};
   // Названия файлов с изображениями:
```

```
private string[] files={"wolf.jpg","fox.jpg","bear.jpg","raccoon.jpg"};
// Путь к каталогу с файлами изображений:
private string path="D:/Books/pictures/csharp/";
// Индекс переключателя, который установлен в начале:
private int index=0;
// Ссылка на метку:
private Label pict;
// Массив ссылок на переключатели:
private RadioButton[] radio;
// Конструктор:
public MyForm():base(){
   // Ширина окна:
   this.Width=310;
   // Высота окна:
   this.Height=200;
   // Режим явного определения начального
   // положения окна:
   this.StartPosition=FormStartPosition.Manual;
   // Координаты окна:
   this.Location=new Point(400,300);
   // Заголовок окна:
   this.Text="В мире животных";
   // Окно фиксированных размеров:
   this.FormBorderStyle=FormBorderStyle.FixedSingle;
   // Окно нельзя развернуть на весь экран:
   this.MaximizeBox=false;
   // Объект панели:
   Panel pnl=new Panel();
   // Ширина панели:
   pnl.Width=this.ClientSize.Width-10;
   // Высота панели:
   pnl.Height=this.ClientSize.Height-10;
```

```
// Горизонтальная координата панели:
pnl.Left=5;
// Вертикальная координата панели:
pnl.Top=5;
// Рамка вокруг панели:
pnl.BorderStyle=BorderStyle.FixedSingle;
// Объект метки:
pict=new Label();
// Положение и размеры метки:
pict.SetBounds(5,5,154,104);
// Рамка вокруг метки:
pict.BorderStyle=BorderStyle.FixedSingle;
// Начальное изображение для метки:
pict.Image=Image.FromFile(path+files[index]);
// Добавление метки на панель:
pnl.Controls.Add(pict);
// Создание массива ссылок на объекты переключателей:
radio=new RadioButton[animals.Length];
// Объект для группы переключателей:
GroupBox gb=new GroupBox();
// Координаты группы переключателей:
gb.Left=pict.Right+5;
gb.Top=pict.Top;
// Размеры группы переключателей:
gb.Width=115;
gb.Height=pict.Height;
// Заголовок для группы переключателей:
gb.Text="Ваш выбор";
// Шрифт для группы переключателей:
gb.Font=new Font("Courier New",12,FontStyle.Bold);
// Высота области для переключателя:
int h=17;
```

```
// Создание объектов для переключателей и добавление
// переключателей в группу:
for(int k=0;k<radio.Length;k++){
   // Объект переключателя:
   radio[k]=new RadioButton();
   // Подпись для переключателя:
   radio[k].Text=animals[k];
   // Состояние переключателя (установлен или нет):
   radio[k].Checked=(k==index);
   // Положение и размеры переключателя:
   radio[k].SetBounds(10,20+k*(h+4),100,h);
   // Обработчик для события, связанного с изменением
   // состояния переключателя:
   radio[k].CheckedChanged+=OnRadioClick;
   // Добавление переключателя в группу:
   gb.Controls.Add(radio[k]);
}
// Добавление группы переключателей на панель:
pnl.Controls.Add(gb);
// Объект кнопки:
MyButton btn=new MyButton("OK");
// Высота кнопки:
btn.Height=30;
// Ширина кнопки:
btn.Width=pnl.Width/3;
// Горизонтальная координата кнопки:
btn.Left=pnl.Width/3;
// Вертикальная координата кнопки:
btn.Top=pict.Bottom+5;
// Шрифт для кнопки:
btn.Font=gb.Font;
// Добавление кнопки на панель:
```

```
      pnl.Controls.Add(btn);
      // Добавление панели в окно:
      this.Controls.Add(pnl);
   }
   // Метод для обработки событий, связанных с изменением
   // состояния переключателя:
   private void OnRadioClick(object obj,EventArgs ea){
      // Перебор переключателей:
      for(int k=0;k<radio.Length;k++){
         // Проверка состояния переключателя:
         if(radio[k].Checked){
            // Новое изображение для пиктограммы:
            pict.Image=Image.FromFile(path+files[k]);
            // Завершение выполнения метода:
            return;
         }
      }
   }
}
// Главный класс:
class RadioButtonDemo{
   [STAThread]
   // Главный метод:
   static void Main(){
      // Отображение окна:
      Application.Run(new MyForm());
   }
}
```

Какие особенности этой программы? Их несколько. Начнем с того, что в программе для кнопки описывается специальный класс `MyButton`. Класс создается наследованием класса `Button`. Конструктору класса передается текстовое значение, определяющее название кнопки. Также

для кнопки регистрируется обработчик события (анонимный метод на основе лямбда-выражения), связанного со щелчком по кнопке. Обработка состоит в том, что завершается выполнение программы. Таким образом, кнопки, реализованные в виде объектов класса `MyButton`, сразу функциональные — щелчок по такой кнопке приводит к завершению выполнения программы.

В классе окна `MyForm` (подкласс класса `Form`) много знакомых «мотивов». Но есть и новые команды и инструкции. Так, в классе имеется поле `radio`, которое является ссылкой на массив объектных переменных класса `RadioButton`. В конструкторе класса `MyForm` командой `this.StartPosition=FormStartPosition.Manual` выполняется переход в режим явного определения начального положения окна. Это положение определяется командой `this.Location=new Point(400,300)` (от левого верхнего угла экрана окно отстоит по горизонтали на `400` пикселей и по вертикали — на `300` пикселей).

Объект панели `pnl` создается командой `Panel pnl=new Panel()`. Ширина панели определяется командой `pnl.Width=this.ClientSize.Width-10`. Это ширина клиентской области окна, уменьшенная на `10` пикселей. Высота панели определяется командой `pnl.Height=this.ClientSize.Height-10` (высота клиентской области окна минус `10`).

НА ЗАМЕТКУ

Клиентская область — это область внутри окна, доступная для размещения компонентов. Она немного меньше внешних параметров окна (с учетом того, что у окна есть рамка и строка названия).

Координаты панели определяются командами `pnl.Left=5` (горизонтальная координата) и `pnl.Top=5` (вертикальная координата). Это координаты компонента (в данном случае панели) внутри контейнера (которым будет окно). Рамка для панели определяется командой `pnl.BorderStyle=BorderStyle.FixedSingle`.

Объект метки `pict` создается на основе класса `Label`. Метка добавляется на панель командой `pnl.Controls.Add(pict)`. Координаты, заданные для метки, — это координаты внутри контейнера. Для метки контейнером является панель. Изображение, которое задается для метки, определяется файлом, название которого содержится в массиве `files` с индексом `index`.

Массив со ссылками на объекты переключателей создаем командой `radio=new RadioButton[animals.Length]`. Размер массива определяется размером массива `animals` с названиями животных. Также командой `GroupBox gb=new GroupBox()` создается объект `gb` класса `GroupBox` (объект для группы переключателей). Координаты группы переключателей определяются командами `gb.Left=pict.Right+5` и `gb.Top=pict.Top`. Таким образом, группа переключателей по высоте размещается на том же уровне, что и метка, а по горизонтали группа переключателей сдвинута на 5 пикселей по отношению к правой границе метки.

Размеры группы переключателей определяются командами `gb.Width=115` и `gb.Height=pict.Height` (высота группы переключателей такая же, как и высота метки). Заголовок для группы переключателей (отображается сверху над группой) определяется командой `gb.Text="Ваш выбор"`. Также мы задаем шрифт и его начертание для группы переключателей (команда `gb.Font=new Font("Courier New",12,FontStyle.Bold)`). Для добавления переключателей в группу используем оператор цикла. Индексная переменная `k` пробегает значения индексов элементов из массива `radio`. За каждую итерацию цикла командой `radio[k]=new RadioButton()` создается объект переключателя. Командой `radio[k].Text=animals[k]` определяется подпись для переключателя. Состояние переключателя (установлен или нет) определяется командой `radio[k].Checked=(k==index)`. Здесь свойству `Checked` переключателя значением присваивается выражение `k==index`. Значение выражения `k==index` рано `true`, если переменная `k` имеет такое же значение, что и поле `index`, и `false` в прочих случаях. Командой `radio[k].SetBounds(10,20+k*(h+4),100,h)` определяются положение и размеры переключателя (предварительно в переменную `h` записано значение высоты переключателя). Обработчик для события, связанного с изменением состояния переключателя, определяется командой `radio[k].CheckedChanged+=OnRadioClick`. Таким образом, при изменении статуса переключателя будет вызываться метод `OnRadioClick()`, описанный в классе `MyForm`. Наконец, с помощью команды `gb.Controls.Add(radio[k])` переключатель добавляется в группу переключателей.

После завершения оператора цикла группа переключателей добавляется на панель (команда `pnl.Controls.Add(gb)`).

Объект кнопки `btn` создается на основе класса `MyButton`. При создании объекта название кнопки передается аргументом конструктору

класса MyButton. Обработчик для этой кнопки уже определен, а параметры и координаты кнопки определяем явно. Ширина кнопки равна трети от ширины панели (команда btn.Width=pnl.Width/3). Высота кнопки определяется командой btn.Height=30. По горизонтали кнопка смещена от левой границы контейнера (панель) на расстояние, равное трети ширины панели (команда btn.Left=pnl.Width/3). По вертикали кнопка смещена вниз на 5 пикселей от нижней границы метки (команда btn.Top=pict.Bottom+5). Шрифт для кнопки устанавливается такой же, как и для группы переключателей (команда btn.Font=gb.Font). Кнопка добавляется на панель (команда pnl.Controls.Add(btn)), после чего панель добавляется в окно (команда this.Controls.Add(pnl)).

Закрытый метод OnRadioClick() предназначен для обработки событий, связанных с изменением состояния переключателей. В теле метода выполняется оператор цикла, в котором перебираются все переключатели из массива radio. Для каждого переключателя выполняется проверка состояния (значение свойства Checked объекта переключателя — true для установленного переключателя и false для неустановленного переключателя). Если переключатель установлен, то командой pict.Image=Image.FromFile(path+files[k]) для метки задается новое изображение (в соответствии с установленным переключателем), а командой return завершается работа метода.

В главном методе программы традиционно создается и отображается окно с изображением, переключателями и кнопкой (см. рис. 8.20).

Опция и поле ввода

Тот, кто нам мешает, тот нам и поможет.

из к/ф «Кавказская пленница»

Следующий проект также связан с отображением картинок. Но на этот раз мы используем такие компоненты, как *текстовое поле* и *опция*. Сначала появляется диалоговое окно с полем ввода. Окно показано на рис. 8.22.

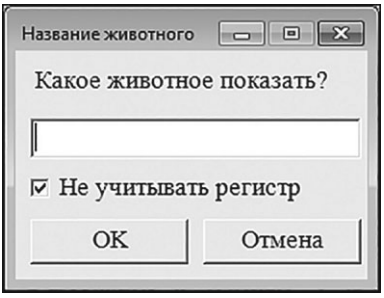


Рис. 8.22. Окно с полем для ввода названия животного

В поле нужно ввести название животного. Под полем ввода есть опция. Если она установлена (у опции есть флажок), то не имеет значения, какими буквами (строчными или заглавным) вводится название животного. Если опция не установлена, то название животного нужно вводить с заглавной буквы. На рис. 8.23 показано окно, в поле которого введено слово **Лиса**.

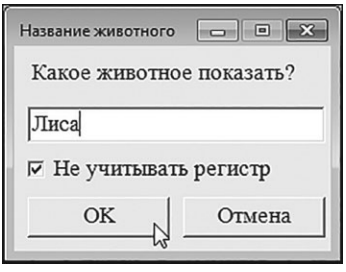


Рис. 8.23. В текстовое поле введено название животного

Если название животного введено корректно, то после щелчка по кнопке **OK** появится еще одно окно, в котором содержится изображение животного, а название животного станет заголовком окна. На рис. 8.24 показано окно с изображением лисы, которое появляется, если в поле ввода ввести слово **Лиса**.

Рис. 8.24. Окно с изображением лисы

Окно с изображением волка показано на рис. 8.25.

Рис. 8.25. Окно с изображением волка

На рис. 8.26 показано, какое окно появится, если пользователь введет в текстовое поле слово **Енот**.

Рис. 8.26. Окно с изображением енота

Окно с изображением медведя показано на рис. 8.27.

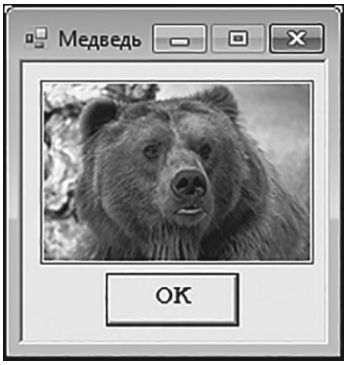


Рис. 8.27. Окно с изображением медведя

Если пользователь вместо кнопки **ОК** нажимает кнопку **Отмена** в окне с полем ввода (см. рис. 8.23), то работа программы завершается. Также возможен вариант, когда пользователь неправильно вводит название животного в поле ввода. В этом случае появляется окно с изображением жирафа. Окно показано на рис. 8.28.

Рис. 8.28. Окно с изображением жирафа

НА ЗАМЕТКУ

В данном случае файл с изображением жирафа помещается в тот же каталог, что и файлы других изображений. Размеры всех файлов одинаковы и составляют `150` пикселей в ширину и `100` пикселей в высоту. Файл с изображением жирафа сохраняется в формате `png`, который позволяет создавать в изображении прозрачный фон. Поэтому при отображении жирафа в окне создается впечатление, что картинка нарисована непосредственно в области окна.

Теперь проанализируем программный код примера. Сразу отметим несколько принципиальных моментов. Во-первых, как отмечалось выше, мы используем два новых компонента: текстовое поле и опцию. Объект для текстового поля создается на основе класса `TextBox`. Объект для опции создается на основе класса `CheckBox`. Во-вторых, мы используем внутренний класс.

НА ЗАМЕТКУ

Внутренним называется класс, который описан внутри другого класса. Если внутренний класс закрытый, то объекты этого внутреннего класса могут создаваться в программном коде внешнего класса (того, в котором описан внутренний класс). Если внутренний класс открытый, то его можно использовать и вне кода внешнего класса. В таком случае обращение к внутреннему классу выполняется так: указывается имя внешнего класса, точка и имя внутреннего класса.

Схема следующая: внешний класс `MyInputBox` используется для создания объекта окна с полем ввода. Закрытый внутренний класс `MyForm` используется для создания объекта окна с изображением. Программный код примера представлен в листинге 8.7.

Листинг 8.7. Использование опций и полей

```
using System;
using System.Drawing;
using System.Windows.Forms;
// Класс для окна с полем ввода:
class MyInputBox:Form{
   // Внутренний закрытый класс для окна с изображением:
   private class MyForm:Form{
      // Закрытое поле внутреннего класса (массив с
      // названиями животных):
      private string[] animals={"Волк","Лиса","Медведь","Енот"};
      // Закрытое поле внутреннего класса (массив с
      // названиями файлов с изображениями животных):
      private string[] files={"wolf.jpg","fox.jpg","bear.jpg","raccoon.jpg"};
```

```
// Закрытое поле внутреннего класса (путь к каталогу
// с файлами изображений):
private string path="D:/Books/pictures/csharp/";
// Конструктор внутреннего класса:
public MyForm(string t,bool state):base(){
   // Название животного и название файла,
   // используемые по умолчанию:
   string txt="Жираф",file="giraffe.png";
   // Перебор элементов массива с названиями
   // животных:
   for(int k=0;k<animals.Length;k++){
      // Сравнение текстовых значений
      if((t==animals[k])||((t.ToLower()==animals[k].ToLower())&state)){
         // Название животного:
         txt=animals[k];
         // Файл с изображением животного:
         file=files[k];
         // Завершение выполнения оператора цикла:
         break;
      }
   }
   // Заголовок для окна:
   this.Text=txt;
   // Шрифт для окна:
   this.Font=new Font("Courier New",12,FontStyle.Bold);
   // Окно постоянных размеров:
   this.FormBorderStyle=FormBorderStyle.Fixed3D;
   // Окно нельзя развернуть на весь экран:
   this.MaximizeBox=false;
   // Объект изображения:
   Image img=Image.FromFile(path+file);
   // Высота изображения:
```

```
        int h=img.Height;
        // Ширина изображения:
        int w=img.Width;
        // Объект метки:
        Label lbl=new Label();
        // Изображение для метки:
        lbl.Image=img;
        // Положение и размеры метки:
        lbl.SetBounds(10,10,w+4,h+4);
        // Рамка для метки:
        lbl.BorderStyle=BorderStyle.FixedSingle;
        // Объект кнопки:
        Button btn=new Button();
        // Название кнопки:
        btn.Text="OK";
        // Положение и размеры кнопки:
        btn.SetBounds(w/4+10,lbl.Bottom+5,w/2,30);
        // Обработчик для события, связанного со щелчком
        // по кнопке:
        btn.Click+=(x,y)=>{
           Application.Exit();
        };
        // Размеры окна:
        this.Size=new Size(w+40,btn.Bottom+50);
        // Начальное положение окна:
        this.StartPosition=FormStartPosition.CenterScreen;
        // Добавление метки в окно:
        this.Controls.Add(lbl);
        // Добавление кнопки в окно:
        this.Controls.Add(btn);
    }
}
```

```
// Закрытое поле внешнего класса (ссылка на опцию):
private CheckBox option;
// Закрытое поле внешнего класса (ссылка на кнопку):
private Button ok;
// Закрытое поле внешнего класса (ссылка на кнопку):
private Button cancel;
// Закрытое поле внешнего класса (ссылка на поле):
private TextBox tb;
// Конструктор внешнего класса:
public MyInputBox():base(){
   // Размеры окна:
   this.Size=new Size(250,190);
   // Положение окна:
   this.StartPosition=FormStartPosition.CenterScreen;
   // Заголовок окна:
   this.Text="Название животного";
   // Окно постоянных размеров:
   this.FormBorderStyle=FormBorderStyle.FixedDialog;
   // Окно нельзя развернуть на весь экран:
   this.MaximizeBox=false;
   // Шрифт для окна:
   this.Font=new Font("Times New Roman",13,FontStyle.Regular);
   // Объект метки:
   Label lbl=new Label();
   // Текст для метки:
   lbl.Text="Какое животное показать?";
   // Высота и ширина метки:
   lbl.Height=30;
   lbl.Width=this.Width-30;
   // Координаты метки:
   lbl.Left=10;
   lbl.Top=10;
```

```
// Добавление метки в окно:
this.Controls.Add(lbl);
// Объект текстового поля:
tb=new TextBox();
// Ширина поля:
tb.Width=this.Width-30;
// Координаты поля:
tb.Left=lbl.Left;
tb.Top=lbl.Bottom+5;
// Добавление поля в окно:
this.Controls.Add(tb);
// Объект опции:
option=new CheckBox();
// Подпись для опции:
option.Text="Не учитывать регистр";
// Координаты опции:
option.Left=tb.Left;
option.Top=tb.Bottom+5;
// Размеры опции:
option.Height=30;
option.Width=tb.Width;
// Состояние опции (по умолчанию - установлена):
option.Checked=true;
// Добавление опции в окно:
this.Controls.Add(option);
// Объект кнопки:
ok=new Button();
// Текст для кнопки:
ok.Text="OK";
// Размеры кнопки:
ok.Width=this.Width/2-20;
ok.Height=30;
```

```
    // Координаты кнопки:
    ok.Left=tb.Left;
    ok.Top=option.Bottom+5;
    // Обработчик для события, связанного со щелчком
    // по кнопке:
    ok.Click+=OnBtnsClick;
    // Объект кнопки:
    cancel=new Button();
    // Текст для кнопки:
    cancel.Text="Отмена";
    // Размеры кнопки:
    cancel.Size=ok.Size;
    // Координаты кнопки:
    cancel.Top=ok.Top;
    cancel.Left=tb.Right-cancel.Width;
    // Обработчик для события, связанного со щелчком
    // по кнопке:
    cancel.Click+=OnBtnsClick;
    // Добавление кнопок в окно:
    this.Controls.Add(ok);
    this.Controls.Add(cancel);
}
// Открытый метод (внешнего класса) для отображения
// окна с полем ввода:
public static void ShowInputBox(){
    // Создание и отображение окна:
    Application.Run(new MyInputBox());
}
// Закрытый метод (внешнего класса), используемый для
// обработки событий, связанных со щелчками по кнопкам:
private void OnBtnsClick(object obj,EventArgs ea){
    // Ссылка на объект кнопки, вызвавшей событие:
```

```
      Button btn=(Button)obj;
      // Идентификация кнопки:
      if(btn==ok){ // Если первая кнопка
         // Окно убирается с экрана:
         this.Visible=false;
         // Создается и отображается окно с изображением:
         new MyForm(tb.Text,option.Checked).ShowDialog();
      }else{ // Если вторая кнопка
         // Завершение работы программы:
         Application.Exit();
      }
   }
}
// Главный класс:
class TextBoxAndCheckBoxDemo{
   [STAThread]
   // Главный метод:
   static void Main(){
      // Отображение окна с полем ввода:
      MyInputBox.ShowInputBox();
   }
}
```

Класс `MyInputBox` для окна с полем ввода создается путем наследования класса `Form`. В классе есть закрытое поле `option` класса `CheckBox`. Это ссылка на объект опции. Закрытые поля `ok` и `cancel` класса `Button` являются ссылками на объекты кнопок **OK** и **Отмена**, которые отображаются в окне. Еще в классе есть закрытое поле `tb` класса `TextBox` (ссылка на объект текстового поля).

В конструкторе внешнего класса задаются размеры и положение окна, заголовок окна, тип рамки и шрифт. Также создается метка, которая содержит текст, размещаемый над полем ввода.

Объект текстового поля создается командой `tb=new TextBox()`. Мы задаем ширину поля (команда `tb.Width=this.Width-30`) и координаты поля (команды `tb.Left=lbl.Left` и `tb.Top=lbl.Bottom+5`), после чего поле добавляется в окно (команда `this.Controls.Add(tb)`).

> **НА ЗАМЕТКУ**
>
> Таким образом, ширина поля на 30 пикселей меньше ширины окна. Горизонтальная координата поля такая же, как и у метки. По вертикали поле смещено на 5 пикселей вниз по сравнению с нижней границей метки. Высота поля определяется размером шрифта, заданным для поля.

Объект опции создается командой `option=new CheckBox()`. Подпись для опции определяем командой `option.Text="Не учитывать регистр"`. Координаты и размеры опции задаем командами `option.Left=tb.Left` (горизонтальная координата опции совпадает с горизонтальной координатой текстового поля), `option.Top=tb.Bottom+5` (по вертикали опция сдвинута на 5 пикселей вниз по отношению к нижней границе текстового поля), `option.Height=30` (высота опции в пикселях равна 30) и `option.Width=tb.Width` (ширина опции равна ширине текстового поля). Состояние опции определяется командой `option.Checked=true`, что означает наличие «флажка» у опции (опция установлена). В окно опция добавляется командой `this.Controls.Add(option)`.

Объект `ok` для кнопки **OK** и объект `cancel` для кнопки **Отмена** создаются на основе класса `Button`. Задается текст кнопок, их размеры и положение.

> **НА ЗАМЕТКУ**
>
> Чтобы размер второй кнопки был таким же, как и первой, используем команду `cancel.Size=ok.Size`. Для выравнивания второй кнопки по правому краю текстового поля используем команду `cancel.Left=tb.Right-cancel.Width`. Здесь горизонтальная координата кнопки вычисляется как разность правой координаты поля и ширины кнопки.

Достоин внимания тот факт, что для обеих кнопок в качестве обработчика события, связанного со щелчком по кнопке, зарегистрирован метод

OnBtnsClick() (команды ok.Click+=OnBtnsClick и cancel.Click+=OnBtnsClick).

НА ЗАМЕТКУ

В классе MyInputBox описан открытый статический метод ShowInputBox(). При вызове этого метода выполняется команда Application.Run(new MyInputBox()). Командой создается объект класса MyInputBox для окна с полем ввода, и окно отображается на экране. В главном методе программы для отображения окна с полем ввода из класса MyInputBox запускается метод ShowInputBox().

Закрытый метод OnBtnsClick() внешнего класса MyInputBox, используемый для обработки событий, связанных со щелчками по кнопкам, получает два аргумента. Нас интересует первый аргумент obj (класса object), являющийся ссылкой на объект компонента, на котором произошло событие.

НА ЗАМЕТКУ

Проблема в том, что кнопки две, а метод-обработчик один. Поэтому при вызове метода необходимо определить, по какой из двух кнопок сделан щелчок.

Ссылка obj на объект, на котором произошло событие, передается как ссылка класса object. Мы знаем, что событие произошло на кнопке (объект класса Button). Поэтому выполняется команда Button btn=(Button)obj, которой ссылка на вызвавший событие объект преобразуется к типу Button (то есть мы получаем возможность работать с объектом как с объектом кнопки). Идентификация кнопки выполняется в условном операторе. Там проверяется условие btn==ok. Истинность условия означает, что событие произошло на кнопке ok. Если так, то командой this.Visible=false окно с полем ввода становится невидимым (убирается с экрана), а командой new MyForm(tb.Text,option.Checked).ShowDialog() создается анонимный объект внутреннего класса MyForm, и из этого объекта вызывается метод ShowDialog(). В результате вызова этого метода окно с изображением животного появляется на экране. Аргументами конструктору передаются текст из текстового поля (выражение tb.Text) и состояние опции (логическое значение option.Checked).

ПОДРОБНОСТИ

Метод `ShowDialog()` наследуется из класса `Form`. При вызове метода отображается окно, из объекта которого вызывается метод. Метод `Run()` из класса `Application` в данном случае не используется, поскольку этот метод вызывается только один раз.

Если условие `btn==ok` ложно, то, рассуждая методом исключения, получаем, что событие произошло на кнопке `cancel`. В этом случае командой `Application.Exit()` завершается выполнение программы.

Осталось расставить все точки на «i» в отношении внутреннего закрытого класса `MyForm` (создается наследованием класса `Form`). Напомним, что это класс используется для создания объекта окна с изображением животного. Большинство команд должны быть знакомы читателю. Мы, как и ранее, имеем дело с текстовыми массивами, содержащими названия животных и названия файлов с изображениями, и полем с путем к каталогу с изображениями. Конструктору класса передается два аргумента: текст (название животного, которое ввел пользователь в текстовое поле) и логическое значение (состояние опции в окне с полем ввода). В теле метода текстовой переменной `txt` присваивается значение `"Жираф"`, а текстовой переменной `file` присваивается значение `"giraffe.png"`. Это название животного и файл с его изображением — такие значения используются, если пользователь ввел некорректное имя для животного. Далее перебираются элементы массива `animals`, в котором хранятся названия животных. Элементы из массива сравниваются с текстовым значением `t`, переданным аргументом конструктору. В условном операторе проверяется условие `(t==animals[k])||((t.ToLower()==animals[k].ToLower())&state)`. Это сложное условие, которое истинно, если истинно хотя бы одно из двух условий: `t==animals[k]` (текстовые значения равны) или `(t.ToLower()==animals[k].ToLower())&state`. Второе условие истинно, если одновременно истинны условия `t.ToLower()==animals[k].ToLower()` и `state`. Истинность условия `t.ToLower()==animals[k].ToLower()` означает, что тексты равны без учета состояния регистра букв. Значение `true` для переменной `state` означает, что установлен флажок опции в окне с полем ввода.

НА ЗАМЕТКУ

Напомним, что вторым аргументом конструктору класса `MyForm` при создании объекта передается значение свойства `Checked` объекта

опции. Поэтому значение аргумента `state` отождествляется со значением свойства `Checked`.
Значение выражения `t.ToLower()==animals[k].ToLower()` вычисляется так: два текстовых значения (из аргумента `t` и элемента `animals[k]`) переводятся в нижний регистр, и полученные значения сравниваются на предмет равенства.

Таким образом, условие в условном операторе истинно, если текстовые значения из аргумента `t` и элемента `animals[k]` полностью совпадают или если они совпадают без учета состояния регистра букв, но при этом установлена опция в окне с полем ввода.

Если для какого-то индекса `k` условие истинно, то командой `txt=animals[k]` в переменную `txt` записывается название выбранного животного, командой `file=files[k]` в переменную `file` записывается название соответствующего файла, а командой `break` завершается выполнение оператора цикла.

После завершения оператора цикла определяется заголовок окна (команда `this.Text=txt`). Это название животного. Также задаются другие параметры окна (шрифт и рамка). Командой `Image img=Image.FromFile(path+file)` создается объект изображения (с учетом определенного названия файла с изображением). Это изображение используется как изображение для метки `lbl` (команда `lbl.Image=img`). Мы определяем высоту (инструкция `img.Height`) и ширину (инструкция `img.Width`) изображения. Эти параметры используются при определении размеров метки, а также кнопки, используемой для завершения работы приложения. Выполняются и другие операции: определяются размеры окна, начальное положение окна, а также добавляются компоненты (метка и кнопка) в окно. В главном методе из класса `MyInputBox` вызывается статический метод `ShowInputBox()`, в результате чего отображается первое окно с полем ввода и опцией.

Меню и панель инструментов

Это же вам не лезгинка, а твист!

из к/ф «Кавказская пленница»

Далее описывается еще одна программа, при выполнении которой появляется окно с изображением животных. Но теперь мы для выбора

изображения будем использовать *главное меню* и *панель инструментов*. Как выглядит в этом случае окно, показано на рис. 8.29.

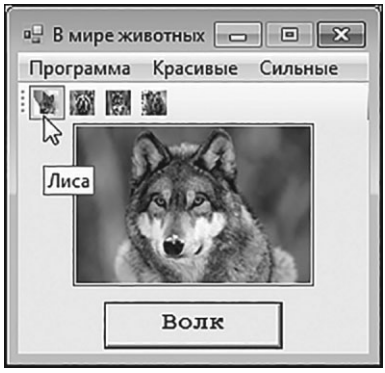


Рис. 8.29. Окно с главным меню и панелью инструментов

Окно в этом случае содержит изображение в центральной части. Под изображением есть кнопка с названием животного. Над изображением размещена панель инструментов из четырех кнопок. Каждая кнопка содержит миниатюрное изображение животного, а при наведении курсора мыши на кнопку появляется подсказка с названием животного (см. рис. 8.29). В окне также есть главное меню из трех пунктов: **Программа**, **Красивые** и **Сильные**.

Изображение в окне меняется при щелчке по кнопке на панели инструментов или при выборе команды из меню. Вместе с изображением меняется и название кнопки. При щелчке по кнопке окно закрывается.

На рис. 8.30 показано окно с изображением лисы. Также в окне раскрыт пункт меню **Программа**. В этом пункте всего одна команда **Выход**, при выборе которой завершается работа программы.

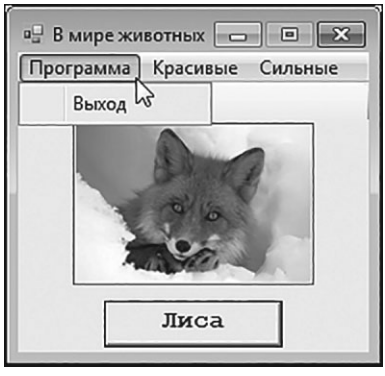


Рис. 8.30. Содержимое пункта меню ***Программа***

На рис. 8.31 показано окно программы с изображением енота. В окне открыт пункт меню **Красивые**. В этом пункте меню содержатся команды **Лиса** и **Енот**. Команды предназначены для выбора животного, изображение которого будет показано в окне.

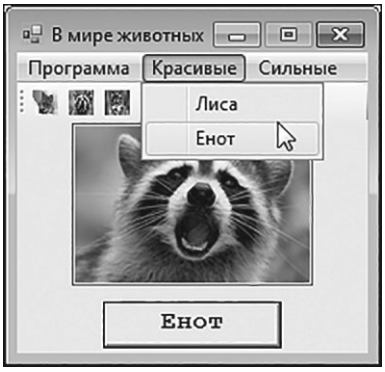


Рис. 8.31. Содержимое пункта меню ***Красивые***

На рис. 8.32 окно содержит изображение медведя, и при этом открыт пункт меню **Сильные**. В этом пункте меню есть две команды **Волк** и **Медведь**, предназначенные для выбора животного.

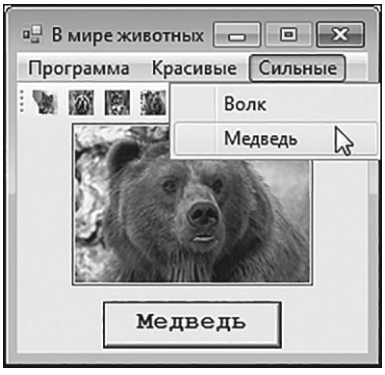


Рис. 8.32. Содержимое пункта меню ***Сильные***

Теперь рассмотрим программный код в листинге 8.8, выполнение которого приводит к появлению описанного выше окна.

Листинг 8.8. Использование меню и панели инструментов

```
using System;
using System.Drawing;
```

```
using System.Windows.Forms;
// Класс окна:
class MyForm:Form{
   // Путь к каталогу с изображениями:
   private string path="D:/Books/pictures/csharp/";
   // Ссылка на объект метки:
   private Label pict;
   // Ссылка на объект кнопки:
   private Button btn;
   // Конструктор:
   public MyForm():base(){
      // Размеры окна:
      this.Size=new Size(240,230);
      // Начальное положение окна:
      this.StartPosition=FormStartPosition.CenterScreen;
      // Заголовок окна:
      this.Text="В мире животных";
      // Окно постоянных размеров:
      this.FormBorderStyle=FormBorderStyle.FixedSingle;
      // Окно нельзя развернуть на весь экран:
      this.MaximizeBox=false;
      // Объект для главного меню:
      MainMenu menu=new MainMenu();
      // Объект для пункта меню "Программа":
      MenuItem prog=new MenuItem("Программа");
      // Добавление команды в пункт меню "Программа":
      prog.MenuItems.Add("Выход",OnButtonClick);
      // Объект для пункта меню "Красивые":
      MenuItem beautiful=new MenuItem("Красивые");
      // Объект для команды "Лиса":
      MenuItem fox=new MenuItem("Лиса");
      // Обработчик для команды "Лиса":
```

```
fox.Click+=(x,y)=>{
   set("Лиса","fox.jpg");
};
// Объект для команды "Енот":
MenuItem raccoon=new MenuItem("Енот");
// Обработчик для команды "Енот":
raccoon.Click+=(x,y)=>{
   set("Енот","raccoon.jpg");
};
// Добавление команд в меню "Красивые":
beautiful.MenuItems.Add(fox);
beautiful.MenuItems.Add(raccoon);
// Объект для пункта меню "Сильные":
MenuItem strong=new MenuItem("Сильные");
// Объект для команды "Медведь":
MenuItem bear=new MenuItem("Медведь");
// Обработчик для команды "Медведь":
bear.Click+=(x,y)=>{
   set("Медведь","bear.jpg");
};
// Объект для команды "Волк":
MenuItem wolf=new MenuItem("Волк");
// Обработчик для команды "Волк":
wolf.Click+=(x,y)=>{
   set("Волк","wolf.jpg");
};
// Добавление команд в меню "Сильные":
strong.MenuItems.Add(wolf);
strong.MenuItems.Add(bear);
// Добавление пунктов в главное меню:
menu.MenuItems.Add(prog);
menu.MenuItems.Add(beautiful);
```

```
menu.MenuItems.Add(strong);
// Назначение главного меню для окна:
this.Menu=menu;
// Объект для панели инструментов:
ToolStrip ts=new ToolStrip();
// Кнопки для панели инструментов:
ToolStripButton tsbA=new ToolStripButton(Image.FromFile(path+"fox.jpg"));
ToolStripButton tsbB=new ToolStripButton(Image.FromFile(path+"raccoon.jpg"));
ToolStripButton tsbC=new ToolStripButton(Image.FromFile(path+"wolf.jpg"));
ToolStripButton tsbD=new ToolStripButton(Image.FromFile(path+"bear.jpg"));
// Всплывающие подсказки для кнопок
// на панели инструментов:
tsbA.ToolTipText="Лиса";
tsbB.ToolTipText="Енот";
tsbC.ToolTipText="Волк";
tsbD.ToolTipText="Медведь";
// Обработчики для событий, связанных со щелчком
// по кнопкам на панели инструментов:
tsbA.Click+=(x,y)=>fox.PerformClick();
tsbB.Click+=(x,y)=>raccoon.PerformClick();
tsbC.Click+=(x,y)=>wolf.PerformClick();
tsbD.Click+=(x,y)=>bear.PerformClick();
// Добавление кнопок на панель инструментов:
ts.Items.Add(tsbA);
ts.Items.Add(tsbB);
ts.Items.Add(tsbC);
ts.Items.Add(tsbD);
// Добавление панели инструментов в окно:
this.Controls.Add(ts);
// Объект метки:
pict=new Label();
// Координаты и размеры метки:
```

```
    pict.SetBounds(35,25,154,104);
    // Рамка для метки:
    pict.BorderStyle=BorderStyle.FixedSingle;
    // Добавление метки в окно:
    this.Controls.Add(pict);
    // Объект кнопки:
    btn=new Button();
    // Координаты кнопки:
    btn.Left=pict.Left+20;
    btn.Top=pict.Bottom+10;
    // Размеры кнопки:
    btn.Width=pict.Width-40;
    btn.Height=30;
    // Шрифт для кнопки:
    btn.Font=new Font("Courier New",12,FontStyle.Bold);
    // Обработка события, связанного со щелчком
    // по кнопке в окне:
    btn.Click+=OnButtonClick;
    // Добавление кнопки в окно:
    this.Controls.Add(btn);
    // Программный щелчок на команде "Волк":
    wolf.PerformClick();
}
// Закрытый метод (вызывается при щелчке
// на команде из меню):
private void set(string name,string file){
    // Название для кнопки:
    btn.Text=name;
    // Изображение для метки:
    pict.Image=Image.FromFile(path+file);
}
// Закрытый метод (вызывается при щелчке
```

```
   // по кнопке в окне):
   private void OnButtonClick(object obj,EventArgs ea){
      Application.Exit();
   }
}
// Главный класс:
class MainMenuDemo{
   [STAThread]
   // Главный метод:
   static void Main(){
      // Отображение окна:
      Application.Run(new MyForm());
   }
}
```

Класс окна `MyForm` создается наследованием класса `Form`. В классе есть несколько закрытых полей. Текстовое поле `path` содержит полный путь к каталогу с файлами изображений. Поле `pict` является объектной ссылкой класса `Label` и предназначено для запоминания ссылки на объект метки с изображением. Также мы используем поле `btn`. Это ссылка на объект кнопки (создается на основе класса `Button`).

В конструкторе класса `MyForm()` задаются размеры окна (команда `this.Size=new Size(240,230)`), его начальное положение (команда `this.StartPosition=FormStartPosition.CenterScreen`), заголовок окна (команда `this.Text="В мире животных"`). Также мы используем окно постоянных размеров без пиктограммы максимизации окна (команды `this.FormBorderStyle=FormBorderStyle.FixedSingle` и `this.MaximizeBox=false`).

Объект для главного меню `menu` создается командой `MainMenu menu=new MainMenu()` на основе класса `MainMenu`. В главное меню будут добавляться пункты, и для каждого из них нужно создать объект. Объекты для пунктов меню создаются на основе класса `MenuItem`. Название для пункта меню указывается в качестве аргумента конструктора. Соответственно, объекты для пунктов главного меню создаются командами `MenuItem prog=new MenuItem("Программа")`, `MenuItem beautiful=new MenuItem("Красивые")`

и MenuItem strong=new MenuItem("Сильные"). Для добавления пунктов в главное меню используется метод Add(). Метод вызывается из свойства MenuItems объекта главного меню. Аргументом методу передается ссылка на объект пункта меню. Речь идет о командах menu.MenuItems.Add(prog), menu.MenuItems.Add(beautiful) и menu.MenuItems.Add(strong). Правда здесь есть еще один момент. Мы создали объект menu, который может быть главным меню. Но этого мало. Нужно зарегистрировать данный объект как главное меню окна. Для этого свойству Menu объекта окна в качестве значения присваивается ссылка на объект menu (команда this.Menu=menu).

В пункты главного меню добавляются команды. Для добавления команды в пункт меню можно из его свойства MenuItems (ссылка на коллекцию элементов из пункта меню) вызывать метод Add(). Аргументами методу передают название команды и название метода, который будет вызываться при выборе команды. Так мы поступаем, когда инструкцией prog.MenuItems.Add("Выход",OnButtonClick) в пункт меню **Программа** добавляем команду **Выход**. В итоге в пункте меню **Программа** появляется команда **Выход**, при выборе которой будет вызываться метод OnButtonClick() (закрытый метод класса MyForm). В теле метода всего одна команда Application.Exit(), которой завершается работа приложения.

Добавлять команду в пункт меню можно другим способом. В этом случае на основе класса MenuItem создается объект команды. Так происходит при выполнении инструкций MenuItem fox=new MenuItem("Лиса"), MenuItem raccoon=new MenuItem("Енот"), MenuItem bear=new MenuItem("Медведь") и MenuItem wolf=new MenuItem("Волк"). Для добавления команд в пункты меню используется уже знакомый нам метод Add(), который вызывается из объекта пункта меню, а аргументом методу передается ссылка на объект команды. Примером являются инструкции beautiful.MenuItems.Add(fox), beautiful.MenuItems.Add(raccoon), strong.MenuItems.Add(wolf) и strong.MenuItems.Add(bear).

Для определения обработчиков для команд, добавленных в пункты меню **Красивые** и **Сильные**, используются лямбда-выражения. Ссылка на анонимный метод, определенный с помощью лямбда-выражения, добавляется в список обработчиков события Click объекта команды меню. В теле анонимного метода вызывается закрытый метод set(). Метод set() получает два аргумента: текстовое значение name с названием для

кнопки (команда `btn.Text=name` в теле метода) и текстовое значение `file`, определяющее название файла с изображением животного. При выполнении команды `pict.Image=Image.FromFile(path+file)` в теле метода на основе указанного файла создается объект изображения, и это изображение применяется для метки.

Объект `ts` для панели инструментов создается на основе класса `ToolStrip` командой `ToolStrip ts=new ToolStrip()`. Для кнопок, которые предполагается размещать на панели инструментов, создаются объекты. Объекты создаются на основе класса `ToolStripButton`. Аргументом конструктору класса передается ссылка на объект изображения, которое применяется для кнопки. Всплывающие подсказки для кнопки на панели инструментов задаем, присвоив текстовое значение свойству `ToolTipText` объекта кнопки.

Обработку событий, связанных со щелчком по кнопкам на панели инструментов, мы реализуем следующим образом: при щелчке по кнопке происходит программное генерирование события, связанного с выбором соответствующей команды из меню. Для этого в список обработчиков события `Click` объекта кнопки добавляется анонимный метод, определяемый лямбда-выражением. В теле лямбда-выражения использован метод `PerformClick()`, выполнение которого означает программный щелчок по команде из меню (метод вызывается из объекта этой команды). Для добавления кнопок на панель инструментов из свойства `Items` (ссылка на коллекцию элементов панели меню) объекта панели вызывается метод `Add()`. Ссылка на объект добавляемой кнопки указывается аргументом метода. Наконец, командой `this.Controls.Add(ts)` панель инструментов добавляется в окно.

Также в конструкторе создается объект метки, задаются ее размеры, положение и тип рамки. Метка добавляется в окно.

Также создается кнопка. Определяются ее размеры, положение, шрифт, а в качестве обработчика регистрируется метод `OnButtonClick()`. После добавления кнопки в окно выполняется команда `wolf.PerformClick()`, означающая программный щелчок на команде **Волк**. В результате для метки применяется изображение с волком, а также название **Волк** получает кнопка.

В главном методе программы командой `Application.Run(new MyForm())` отображается окно, описанное в самом начале раздела.

Контекстное меню

Плюк — чатланская планета, поэтому мы, пацаки, должны цаки носить.

из к/ф «Кин-дза-дза»

Далее мы познакомимся с *контекстным меню*. Это меню, которое открывается при щелчке правой кнопкой мыши на компоненте. В следующей программе создается и отображается окно традиционно с изображением животного. Окно представлено на рис. 8.33.

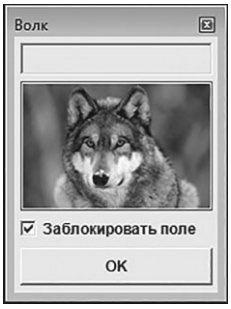


Рис. 8.33. При первом отображении окна оно содержит изображение волка, флажок опции установлен, а поле ввода заблокировано

В центральной части окна сначала появляется изображение волка. Если щелкнуть правой кнопкой мыши на изображении, появляется контекстное меню из четырех команд, названия которых совпадают с названиями животных. Ситуация проиллюстрирована на рис. 8.34.

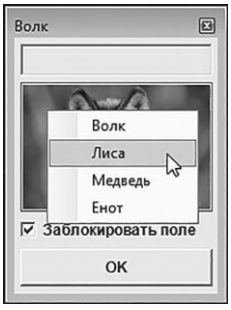


Рис. 8.34. Выбор команды в контекстном меню изображения

Выбрав команду в контекстном меню, мы меняем изображение в окне. Одновременно с изображением меняется и заголовок окна. На рис. 8.35 показан результат, который получим при выборе команды **Лиса** в контекстном меню.

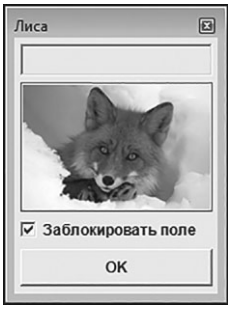


Рис. 8.35. В результате выбора команды в контекстном меню меняется изображение и заголовок окна

Кроме изображения, в окне под изображением есть опция **Заблокировать поле** и поле ввода над изображением. При установленном флажке опции поле заблокировано (выделено серым цветом и недоступно для ввода). Если отменить флажок для опции, то в поле можно вводить текст, как показано на рис. 8.36.

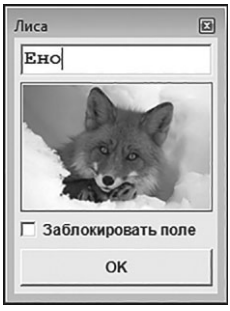


Рис. 8.36. Пока в поле ввода не указано корректное название животного, изображение в окне не меняется

Как только в поле ввода будет введено название животного (**Волк**, **Лиса**, **Медведь** или **Енот**), в окне изменится изображение животного и заголовок окна. На рис. 8.37 показано, как выглядит окно, если в текстовое поле ввести слово `Енот`.

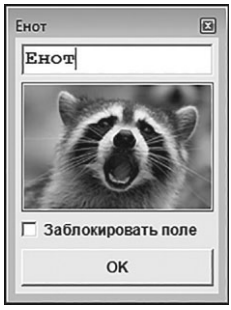


Рис. 8.37. При вводе в текстовое поле корректного названия животного, его изображение появляется в окне

Если установить флажок опции, то поле снова станет неактивным, а текст из поля будет удален. Как и ранее, мы можем выбрать команду из контекстного меню. На рис. 8.38 показано, как при активном текстовом поле в контекстном меню выбирается команда **Медведь**.

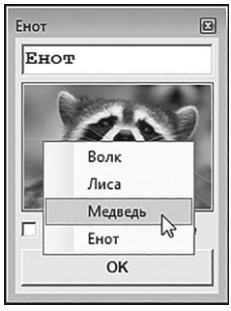


Рис. 8.38. Выбор команды из контекстного меню при активном поле ввода

В результате в окне появляется изображение медведя, меняется заголовок окна, у опции появляется флажок, из поля удаляется текст, и поле блокируется. Как будет выглядеть окно в этом случае, показано на рис. 8.39.

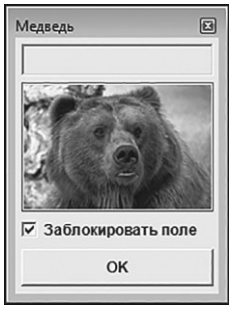


Рис. 8.39. После выбора команды из контекстного меню меняется изображение и заголовок окна, устанавливается флажок опции и блокируется поле ввода

Далее мы проанализируем программный код программы, который представлен в листинге 8.9.

Листинг 8.9. Контекстное меню

```
using System;
using System.Drawing;
using System.Windows.Forms;
// Класс окна:
class MyForm:Form{
   // Ссылка на текстовое поле:
   private TextBox tb;
   // Ссылка на метку:
   private Label lbl;
   // Закрытый метод для проверки названия животного:
   private bool checkIt(string name){
      switch(name){
         case "Волк":
         case "Лиса":
         case "Медведь":
         case "Енот":
            return true;
         default:
            return false;
       }
   }
   // Закрытый метод для определения по названию животного
   // названия файла с изображением:
   private string getFile(string name){
      string path="D:/Books/pictures/csharp/";
      string res="";
      switch(name){
         case "Волк":
```

```
            res="wolf.jpg";
            break;
         case "Лиса":
            res="fox.jpg";
            break;
         case "Медведь":
            res="bear.jpg";
            break;
         case "Енот":
            res="raccoon.jpg";
            break;
      }
      return path+res;
   }
   // Конструктор:
   public MyForm():base(){
      // Локальные переменные:
      int H=104;
      int W=154;
      // Размеры окна:
      this.Width=W+25;
      this.Height=240;
      // Шрифт для окна:
      this.Font=new Font("Arial",9,FontStyle.Bold);
      // Окно постоянных размеров:
      this.FormBorderStyle=FormBorderStyle.FixedToolWindow;
      // Начальное положение окна:
      this.StartPosition=FormStartPosition.CenterScreen;
      // Объект текстового поля:
      tb=new TextBox();
      // Положение текстового поля:
      tb.Location=new Point(5,5);
```

```
// Ширина текстового поля:
tb.Width=W;
// Шрифт для текстового поля:
tb.Font=new Font("Courier New",12,FontStyle.Bold);
// Обработчик события, связанного с отпусканием
// клавиши на клавиатуре:
tb.KeyUp+=(x,y)=>{
   // Если в поле название животного:
   if(checkIt(tb.Text)){
      // Заголовок окна:
      this.Text=tb.Text;
      // Изображение для метки:
      lbl.Image=Image.FromFile(getFile(tb.Text));
   }
};
// Добавление поля в окно:
this.Controls.Add(tb);
// Объект метки:
lbl=new Label();
// Положение и размеры метки:
lbl.Top=tb.Bottom+5;
lbl.Left=5;
lbl.Size=new Size(W,H);
// Рамка для метки:
lbl.BorderStyle=BorderStyle.FixedSingle;
// Добавление метки в окно:
this.Controls.Add(lbl);
// Объект опции:
CheckBox option=new CheckBox();
// Размеры и положение опции:
option.Width=W;
option.Height=20;
```

```
option.Left=5;
option.Top=lbl.Bottom+5;
// Подпись для опции:
option.Text="Заблокировать поле";
// Обработчик для события, связанного с изменением
// состояния опции:
option.CheckedChanged+=(x,y)=>{
   // Доступность поля для ввода текста:
   tb.Enabled=!option.Checked;
   // Очистка содержимого поля:
   tb.Text="";
};
// Добавление опции в окно:
this.Controls.Add(option);
// Объект контекстного меню:
ContextMenuStrip menu=new ContextMenuStrip();
// Добавление команд в контекстное меню:
menu.Items.Add("Волк");
menu.Items.Add("Лиса");
menu.Items.Add("Медведь");
menu.Items.Add("Енот");
// Перебор элементов из контекстного меню:
for(int k=0;k<menu.Items.Count;k++){
   // Обработчик для команды из контекстного меню:
   menu.Items[k].Click+=(x,y)=>{
      // Название команды:
      string txt=((ToolStripMenuItem)x).Text;
      // Изображение для метки:
      lbl.Image=Image.FromFile(getFile(txt));
      // Заголовок окна:
      this.Text=txt;
      // Устанавливается флажок опции:
```

```
            option.Checked=true;
         };
      }
      // Контекстное меню для метки:
      lbl.ContextMenuStrip=menu;
      // Объект кнопки:
      Button btn=new Button();
      // Название кнопки:
      btn.Text="OK";
      // Положение и размеры кнопки:
      btn.SetBounds(5,option.Bottom+5,W,30);
      // Обработчик для события, связанного со щелчком
      // по кнопке:
      btn.Click+=(x,y)=>{
         Application.Exit();
      };
      // Добавление кнопки в окно:
      this.Controls.Add(btn);
      // Программный щелчок на первой команде
      // из контекстного меню:
      menu.Items[0].PerformClick();
   }
}
// Главный класс:
class ContextMenuDemo{
   [STAThread]
   // Главный метод:
   static void Main(){
      // Отображение окна с полем ввода:
      Application.Run(new MyForm());
   }
}
```

В классе окна `MyForm` есть закрытое поле `tb` класса `TextBox` (ссылка на текстовое поле) и закрытое поле `lbl` класса `Label` (ссылка на метку). Также в классе описан закрытый метод `checkIt()`, аргументом которому передается текстовое значение, а результатом метод возвращает логическое значение `true` или `false` в зависимости от того, является ли переданный текст названием животного или нет (допускаются варианты `"Волк"`, `"Лиса"`, `"Медведь"` и `"Енот"`).

В классе есть закрытый метод `getFile()` для определения по названию животного (аргумент метода) названия файла с изображением (результат метода). Причем возвращается текст с полным путем к файлу.

> **НА ЗАМЕТКУ**
>
> Если аргументом методу `getFile()` передать текст, который не является названием животного (имеются в виду названия `"Волк"`, `"Лиса"`, `"Медведь"` или `"Енот"`), то результатом метод возвращает текстовую строку с путем к каталогу с файлами изображений. Но поскольку в программе этот метод вызывается только в случае, если аргумент является названием животного, то проблем не возникает.

В конструкторе класса задаются размеры и начальное положение окна, шрифт для окна, а также командой `this.FormBorderStyle=FormBorderStyle.FixedToolWindow` определяется тип окна. Это окно постоянных размеров без пиктограмм минимизации и максимизации окна (в правом верхнем углу есть только одна системная пиктограмма для закрытия окна).

Также в конструкторе создается объект для текстового поля. Определяется положение поля и его ширина, задается шрифт для поля (высота поля автоматически определяется на основе шрифта). Для поля регистрируется обработчик события, связанного с отпусканием клавиши на клавиатуре (событие происходит, если поле активно и пользователь отпускает ранее нажатую клавишу). Для этого в список обработчиков события `KeyUp` объекта поля `tb` добавляется ссылка на анонимный метод, реализованный с помощью лямбда-выражения. В теле метода в условном операторе проверяется условие `checkIt(tb.Text)`. Условие истинно, если в текстовом поле содержится название животного. Если так, то командой `this.Text=tb.Text` содержимое текстового поля становится заголовком окна. Затем командой `lbl.Image=Image.FromFile(getFile(tb.Text))` определяется изображение для

метки. В этом случае по названию животного в текстовом поле определяется название файла с изображением (инструкция `getFile(tb.Text)`), и оно передается аргументом методу `FromFile()`.

Помимо перечисленного выше, в конструкторе создается метка, определяются ее параметры, а также создается опция. Для опции определяется обработчик события, связанного с изменением состояния опции: в список обработчиков для события `CheckedChanged` объекта опции `option` добавляется ссылка на анонимный метод, реализованный с помощью лямбда-выражения. В теле метода командой `tb.Enabled=!option.Checked` определяется значение свойства `Enabled` текстового поля. Если значение свойства равно `true`, то поле доступно для ввода текста. Если значение свойства равно `false`, то поле заблокировано. В данном случае значение свойства `Enabled` противоположно значению свойства `Checked` опции.

НА ЗАМЕТКУ

То есть если опция установлена, то поле заблокировано, а если опция не установлена, то поле доступно для редактирования.

Также командой `tb.Text=""` содержимое поля «обнуляется»: в поле записывается пустой текст.

Для контекстного меню на основе класса `ContextMenuStrip` создается объект `menu`. Для добавления команд в контекстное меню из свойства `Items` объекта меню вызывается метод `Add()`, аргументом которому передается текст с названием команды. Для каждой команды определяется обработчик события, связанного со щелчком на команде меню. Для перебора элементов (команд) из контекстного меню используем оператор цикла. Команды контекстного меню образуют коллекцию, ссылку на которую получаем с помощью свойства `Items` объекта меню `menu`. У свойства `Items` есть свойство `Count`, возвращающее в качестве значения количество элементов (команд) в коллекции. В операторе цикла индексная переменная `k` пробегает значения индексов элементов в коллекции. На каждой итерации цикла в список обработчиков события `Click` элемента `Items[k]` добавляется ссылка на анонимный метод, реализованный с помощью лямбда-выражения. В лямбда-выражении командой `string txt=((ToolStripMenuItem)x).Text` в текстовую переменную `txt` записывается название команды, на которой произошло событие (на которой был выполнен щелчок).

ПОДРОБНОСТИ

В лямбда-выражении первый аргумент (обозначен как `x`) обозначает объект компонента, на котором произошло событие. В соответствии с определением делегата `EventHandler` этот аргумент относится к классу `object`. Мы знаем, что этот объект — команда контекстного меню. Поэтому объект приводится к типу `ToolStripMenuItem` (инструкция `(ToolStripMenuItem)x`). Свойство `Text` этого объекта — это название команды, что нам и нужно.

Командой `lbl.Image=Image.FromFile(getFile(txt))` задается изображение для метки. Заголовок окна определяем командой `this.Text=txt`. С помощью команды `option.Checked=true` устанавливается флажок опции (при этом выполняется обработка события, связанного с изменением статуса опции). Чтобы созданное меню стало контекстным меню метки с изображением, используем команду `lbl.ContextMenuStrip=menu` (свойству `ContextMenuStrip` в качестве значения присваивается ссылка на объект контекстного меню).

Также мы создаем объект кнопки, задаем ее параметры и определяем обработчик события, связанного со щелчком по кнопке. Компоненты (поле, метка, опция и кнопка) добавляются в окно, после чего командой `menu.Items[0].PerformClick()` выполняется программный щелчок на первой команде из контекстного меню. В результате в окне появляется изображение волка и выполняются прочие сопутствующие настройки. В главном методе создается и отображается окно.

Координаты курсора мыши

> Меня не интересуют ваши координаты. «Беда» догоняет регату, скоро финиш. Товар не должен вернуться назад.
>
> *из м/ф «Приключения капитана Врунгеля»*

Еще один пример, который мы рассмотрим в этой главе, связан с использованием параметров курсора мыши. Кроме этого, мы увидим, как можно выполнять рисование в области графического компонента. Программа выполняется следующим образом. Вначале отображается окно с большой областью желтого цвета и меткой в верхней части окна. Как выглядит окно в начале выполнения программы, показано на рис. 8.40.

Рис. 8.40. При запуске программы отображается окно с меткой и большой желтой панелью в центре

Если курсор мыши оказывается над желтой областью, то он приобретает вид ладони, а через точку размещения курсора проходят две толстые линии (горизонтальная и вертикальная) синего цвета. Также в метке отображаются координаты курсора в системе координат, связанной с желтой областью. Точка начала координат находится в левом верхнем углу области, координаты определяются в пикселях как расстояние по горизонтали и вертикали от точки начала координат до точки в области (для которой определяются координаты). Как выглядит окно при наведенном на желтую область курсоре мыши, показано на рис. 8.41.

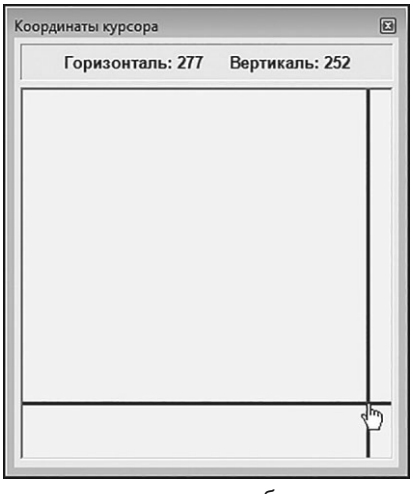


Рис. 8.41. При наведении курсора на область панели отображаются две линии и координаты курсора

При перемещении курсора мыши в области автоматически меняются значения координат в метке над областью, а также автоматически, вслед за курсором, перемещаются синие линии. Ситуацию иллюстрирует рис. 8.42.

Рис. 8.42. При перемещении курсора над панелью синхронно изменяются значения координат курсора и расположение линий

В общих чертах так функционирует интересующая нас программа. Теперь проанализируем код программы, представленный в листинге 8.10.

Листинг 8.10. Координаты курсора

```
using System;
using System.Drawing;
using System.Windows.Forms;
// Класс окна:
class MyForm:Form{
   // Ссылка на объект для рисования:
   Pen p;
   // Ссылка на объект графического контекста:
   Graphics g;
   // Конструктор:
   public MyForm():base(){
      // Размеры окна:
```

```
this.Width=325;
this.Height=380;
// Окно фиксированных размеров:
this.FormBorderStyle=FormBorderStyle.FixedToolWindow;
// Заголовок окна:
this.Text="Координаты курсора";
// Объект метки:
Label lbl=new Label();
// Размеры метки:
lbl.Width=300;
lbl.Height=30;
// Положение метки:
lbl.Location=new Point(5,5);
// Рамка для метки:
lbl.BorderStyle=BorderStyle.Fixed3D;
// Шрифт для метки:
lbl.Font=new Font("Arial",10,FontStyle.Bold);
// Способ выравнивания текста в метке:
lbl.TextAlign=ContentAlignment.MiddleCenter;
// Добавление метки в окно:
this.Controls.Add(lbl);
// Объект панели:
Panel pnl=new Panel();
// Размеры панели:
pnl.Width=lbl.Width;
pnl.Height=300;
// Положение панели:
pnl.Top=lbl.Bottom+5;
pnl.Left=lbl.Left;
// Рамка для панели:
pnl.BorderStyle=BorderStyle.Fixed3D;
// Цвет панели:
```

```
    pnl.BackColor=Color.Yellow;
    // Вид курсора над панелью:
    pnl.Cursor=Cursors.Hand;
    // Объект для рисования:
    p=new Pen(Color.Blue,3);
    // Объект графического контекста для панели:
    g=pnl.CreateGraphics();
    // Обработчик события, связанного с движением курсора
    // над областью панели:
    pnl.MouseMove+=(obj,ea)=>{
       // Перерисовка панели:
       pnl.Refresh();
       // Текст для метки:
       lbl.Text="Горизонталь: "+ea.X;
       lbl.Text+="     ";
       lbl.Text+="Вертикаль: "+ea.Y;
       // Отображение линий:
       g.DrawLine(p,new Point(0,ea.Y),new Point(pnl.ClientSize.Width,ea.Y));
       g.DrawLine(p,new Point(ea.X,0),new Point(ea.X,pnl.ClientSize.Height));
    };
    // Обработчик события, связанного с перемещением
    // курсора за пределы панели:
    pnl.MouseLeave+=(obj,ae)=>{
       // Текст для метки:
       lbl.Text="";
       // Перерисовка панели:
       pnl.Refresh();
    };
    // Добавление панели в окно:
    this.Controls.Add(pnl);
}
// Деструктор:
```

```
   ~MyForm(){
      // Освобождение ресурсов, задействованных
      // для рисования:
      g.Dispose();
      p.Dispose();
   }
}
// Главный класс:
class CursorDemo{
   [STAThread]
   // Главный метод:
   static void Main(){
      // Отображение окна:
      Application.Run(new MyForm());
   }
}
```

В классе окна `MyForm` есть поле `p` класса `Pen` и поле `g` класса `Graphics` (оба класса из пространства имен `System.Drawing`). Это ссылки на объекты, которые нам понадобятся при рисовании линий.

В конструкторе класса создается метка (объект `lbl` класса `Label`) и панель (объект `pnl` класса `Panel`). Для них и для объекта окна задаются их размеры и положение, а также ряд других параметров (назначение соответствующих команд должно быть понятно читателю). Для панели командой `pnl.BackColor=Color.Yellow` задается желтый цвет фона. Команда `pnl.Cursor=Cursors.Hand` используется для того, чтобы задать вид курсора мыши, когда он находится над областью панели. В данном случае свойству `Cursor` объекта панели `pnl` в качестве значения присваивается константа `Hand` из перечисления `Cursors`, и поэтому над областью панели курсор будет принимать вид ладони.

Поскольку мы собираемся рисовать линии, то нам понадобится два объекта. Первый объект создается на основе класса `Pen`. Объект создается командой `p=new Pen(Color.Blue,3)`. Он содержит информацию о цвете и толщине линий в процессе рисования. Командой `g=pnl.CreateGraphics()` в поле `g` записывается ссылка на объект

графического контекста для панели. Этот объект необходим для получения доступа к ресурсам, задействованным при рисовании в области графического компонента (в данном случае панели).

Для панели определяются обработчики для события `MouseMove`, связанного с перемещением курсора над областью панели, а также для события `MouseLeave`, связанного с перемещением курсора за пределы панели. В частности, в список обработчиков события `MouseMove` объекта панели `pnl` добавляется ссылка на анонимный метод, определенный лямбда-выражением. В теле метода командой `pnl.Refresh()` выполняется перерисовка панели. Это необходимо, чтобы стереть линии, нарисованные для предыдущего положения курсора. Командами `lbl.Text="Горизонталь: "+ea.X`, `lbl.Text+="  "` и `lbl.Text+="Вертикаль: "+ea.Y` определяется текст метки (команд три, хотя можно было воспользоваться и одной). Здесь использованы выражения `ea.X` и `ea.Y`, которые возвращают координаты (горизонтальную и вертикальную, соответственно) курсора мыши в области панели.

ПОДРОБНОСТИ

Делегат `EventHandler`, которому должны соответствовать методы-обработчики для событий графических компонентов, подразумевает наличие у метода второго аргумента (объявлен как переменная класса `EventArgs`), содержащего информацию о событии. В данном случае второй аргумент обозначен как `ea`. Его свойства `X` и `Y` позволяют определить координаты курсора мыши в момент события.

Для отображения линий использованы команды `g.DrawLine(p,new Point(0,ea.Y),new Point(pnl.ClientSize.Width,ea.Y))` и `g.DrawLine(p,new Point(ea.X,0),new Point(ea.X,pnl.ClientSize.Height))`. В обоих случаях из объекта графического контекста `g` вызывается метод `DrawLine()`, предназначенный для отображения линии. Первым аргументом методу передается объект `p` класса `Pen`, который определяет цвет и толщину линии. Два других аргумента метода — объекты класса `Point`, определяющие начальную и конечную точки, которые соединяет линия.

ПОДРОБНОСТИ

Выражения `pnl.ClientSize.Width` и `pnl.ClientSize.Height` определяют, соответственно, ширину и высоту клиентской (внутренней, доступной для использования) области панели.

В случаях, когда курсор выходит за пределы области панели, обработка выполняется так: командой `lbl.Text=""` выполняется очистка содержимого метки, а командой `pnl.Refresh()` выполняется перерисовка панели.

> **НА ЗАМЕТКУ**
>
> В классе `MyForm` определен еще и деструктор. В деструкторе выполняются команды `g.Dispose()` и `p.Dispose()`, которыми освобождаются ресурсы, задействованные для рисования. Правда в данном случае особого смысла в этих командах нет — при удалении объекта окна из памяти эти ресурсы и так освободятся. Главная причина, по которой описан деструктор, — подчеркнуть, что с ресурсами следует обращаться экономно.

Для отображения окна в главном методе используется команда `Application.Run(new MyForm())`.

Резюме

Были демоны — мы этого не отрицаем. Но они самоликвидировались.

из к/ф «Иван Васильевич меняет профессию»

- Для реализации окна формы и прочих графических компонентов существуют специальные классы. Создание графического компонента отождествляется с созданием объекта соответствующего класса.
- Объекты классов для графических компонентов имеют специальные методы и свойства, позволяющие задавать основные характеристики компонентов (размеры, положение, шрифт, цвет фона и многое другое).
- Для обработки действий с графическими компонентами в классах компонентов предусмотрены события.
- Характеристики методов, используемых для обработки событий, определяются делегатом `EventHandler`. Делегат подразумевает, что соответствующий метод не возвращает результат и у него два аргумента: ссылка типа `object` на объект компонента, вызвавшего событие, и объектная ссылка типа `EventArgs` на объект, содержащий описание события.

Задания для самостоятельной работы

Сеня, по-быстрому объясни товарищу, почему Володька сбрил усы.

из к/ф «Бриллиантовая рука»

1. Напишите программу, отображающую окно, в котором внутренняя часть закрашена желтым цветом. При наведении курсора на область окна цвет фона меняется на зеленый. При щелчке мышью размеры окна должны увеличиваться на 10%.

2. Напишите программу, в которой отображается окно с текстовой меткой и тремя кнопками. В текстовой метке содержится число (начальное значение — нулевое). Щелчок по одной из меток приводит к увеличению значения числа на единицу. Щелчок по другой кнопке приводит к уменьшению значения числа на единицу. Щелчок по третьей кнопке приводит к закрытию окна.

3. Напишите программу, в которой открывается окно с раскрывающимся списком. Список содержит названия цветов (красный, желтый, зеленый и так далее). Также окно содержит область, закрашенную тем цветом, который выбран в списке. При выборе в списке нового цвета область закрашивается этим цветом автоматически.

4. Напишите программу, в которой открывается окно с полем ввода. При вводе текста в окно этот текст автоматически дублируется в текстовой метке. В окне должны быть две опции, которые позволяют применять к тексту в метке жирный и курсивный стили.

5. Напишите программу, в которой отображается окно с двумя текстовыми полями. Предполагается, что в эти текстовые поля вводятся целочисленные значения. Кроме полей, в окне размещена метка, в которой содержится информация о том, какое из двух чисел больше/меньше или что числа равны друг другу. Информация в метке обновляется автоматически при изменении содержимого полей. Если хотя бы в одном из полей указано не число, метка должна содержать информацию об этом.

6. Напишите программу, в которой отображается окно с изображением и двумя кнопками. Имеется несколько изображений, которые последовательно циклически отображаются в окне при щелчке по кнопкам. При щелчке по одной кнопке появляется следующее изображение

в последовательности, а при щелчке по другой кнопке отображается предыдущее изображение в последовательности.

7. Напишите программу, в которой отображается окно со списком выбора. В списке выбора представлены названия шрифтов. Также окно содержит раскрывающийся список с названиями цветов (красный, зеленый, синий и так далее). Окно содержит область с текстом. При выборе цвета или названия шрифта этот цвет или шрифт применяются для отображения текста.

8. Напишите программу, в которой отображается окно с закрашенной областью. Для этой области есть контекстное меню с названиями цветов (красный, желтый, зеленый и так далее). При выборе команды из контекстного меню область закрашивается соответствующим цветом.

9. Напишите программу, в которой отображается окно с главным меню и областью с текстом. Текст содержит информацию о названии, стиле и размере шрифта, которым отображается текст. В меню есть пункты для выбора названия шрифта, стиля шрифта и размера шрифта. При выборе команды из меню соответствующая характеристика применяется для отображения текста, а также с учетом новых параметров шрифта меняется сам текст.

10. Напишите программу, в которой отображается окно, представляющее собой арифметический калькулятор.

Глава 9
НЕМНОГО О РАЗНОМ

Однако не следует думать, что здесь какое-то колдовство или чудо.

из к/ф «Собачье сердце»

В этой главе рассматриваются вопросы, которые по различным причинам не вошли в предыдущие главы (первой и второй части книги), но которые, тем не менее, важны для понимания принципов программирования на C#. Большинство из представленных далее тем рассматриваются в «факультативном» режиме. Они не связаны между собой концептуально. Главная цель, которую мы преследуем здесь — дать читателю общее представление о том или ином механизме или подходе.

Далее мы кратко рассмотрим такие вопросы:

- работа с диалоговыми окнами;
- использование пространств имен;
- работа с датой и временем;
- операции с файлами;
- использование коллекций.

Разумеется, приведенные далее теоретические сведения подкрепляются примерами.

Работа с диалоговым окном

Это не ты говоришь, это кричит твой вакуум!

из к/ф «Покровские ворота»

В самом начале книги (в первой части) мы рассматривали способы вывода сообщений в *диалоговое окно*. Для этого использовался метод `Show()` класса `MessageBox`. На самом деле, окно, которое отображается в таком

случае, можно использовать не только для отображения сообщения, но и для получения ответа от пользователя. Для этого в окне с сообщением отображается несколько кнопок, и при щелчке пользователя по одной из них программа определяет, какую именно кнопку нажал пользователь.

Все дело в том, что метод `Show()` класса `MessageBox` возвращает результат. Просто раньше мы его не использовали. Результатом метода является одна из констант из перечисления `DialogResult`. Эта константа определяет кнопку, которую нажал пользователь в диалоговом окне (например, `Abort`, `Cancel`, `Ignore`, `No`, `OK`, `Retry` или `Yes` — название константы соответствует названию кнопки, по которой щелкает пользователь). Как все это используется на практике, иллюстрирует программа в листинге 9.1.

Листинг 9.1. Работа с диалоговым окном

```
using System.Windows.Forms;
// Главный класс:
class DialogResultDemo{
   // Главный метод:
   static void Main(){
      // Текстовые переменные:
      string msg,title;
      // Переменная для определения пиктограммы:
      MessageBoxIcon icn;
      // Переменная для определения результата:
      DialogResult res;
      // Отображение диалогового окна:
      res=MessageBox.Show("Вам нравится C#?",// Сообщение
         "Риторический вопрос",              // Заголовок
         MessageBoxButtons.YesNo,            // Кнопки
         MessageBoxIcon.Question             // Пиктограмма
      );
      // Проверка результата:
      if(res==DialogResult.Yes){ // Если ответ "Да"
```

```
            // Текст сообщения:
            msg="Добро пожаловать в команду!";
            // Заголовок окна:
            title="Наш человек";
            // Пиктограмма:
            icn=MessageBoxIcon.Information;
        }else{ // Если ответ "Нет"
            // Текст сообщения:
            msg="У Вас нет сердца!";
            // Заголовок окна:
            title="Не может быть";
            // Пиктограмма:
            icn=MessageBoxIcon.Error;
        }
        // Отображение окна с сообщением:
        MessageBox.Show(msg,title,MessageBoxButtons.OK,icn);
    }
}
```

В этой программе объявляется несколько переменных. Текстовые переменные `msg` и `title` нам нужны для того, чтобы запомнить текст сообщения и заголовок окна (который отображается последним в программе). В переменную `icn` (типа `MessageBoxIcon`) будет записано значение константы из перечисления `MessageBoxIcon`, определяющее тип пиктограммы для окна. Переменная `res` типа `DialogResult` используется для определения результата при нажатии кнопки в диалоговом окне. В переменную `res` записывается результат вызова статического метода `Show()` из класса `MessageBox`. Третьим аргументом методу передается константа `YesNo` из перечисления `MessageBoxButtons`. Эта константа означает, что у окна будет две кнопки (**Yes** и **No**, или **Да** и **Нет**).

НА ЗАМЕТКУ

Четвертый аргумент `MessageBoxIcon.Question` метода `Show()` означает, что в окне отображается пиктограмма с вопросительным знаком.

На рис. 9.1 показано, как выглядит первое окно (с вопросом), которое отображается при запуске программы на выполнение.

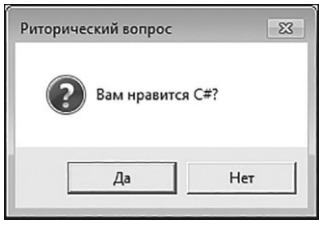


Рис. 9.1. В начале выполнения программы появляется диалоговое окно с кнопками *Да* и *Нет*

Если пользователь нажимает в окне кнопку **Да**, то значением переменной `res` будет константа `Yes` из перечисления `DialogResult`. Если пользователь нажмет кнопку **Нет**, то в переменную `res` будет скопировано значение константы `No` из перечисления `DialogResult`. Поэтому условие `res==DialogResult.Yes` в условном операторе истинно, если пользователь нажал в диалоговом окне кнопку **Да**. В этом случае выполняются команды `msg="Добро пожаловать в команду!"`, `title="Наш человек"` и `icn=MessageBoxIcon.Information`. В противном случае (если условие `res==DialogResult.Yes` ложно) выполняются команды `msg="У Вас нет сердца!"`, `title="Не может быть"` и `icn=MessageBoxIcon.Error`. Но при любом исходе переменные `msg`, `title` и `icn` получают значения и после условного оператора выполняется команда `MessageBox.Show(msg,title, MessageBoxButtons.OK,icn)`, которой отображается еще одно окно с сообщением. Параметры данного окна (текст сообщения, заголовок и пиктограмма) зависят от того, какую кнопку пользователь нажал в предыдущем окне. На рис. 9.2 показано окно, которое отображается в случае, если пользователь в первом окне нажал кнопку **Да**.

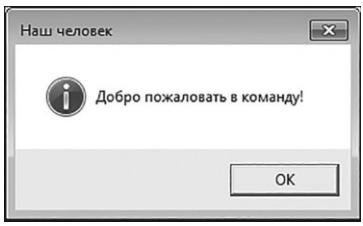


Рис. 9.2. Такое окно появляется, если пользователь нажал кнопку *Да*

Если пользователь нажмет в первом окне кнопку **Нет**, то появится окно, показанное на рис. 9.3.

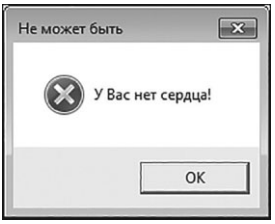


Рис. 9.3. Такое окно появляется, если пользователь нажмет кнопку **Нет**

НА ЗАМЕТКУ

Напомним, что тип пиктограммы в окне определяется константой (`Asterisk`, `Error`, `Exclamation`, `Hand`, `Information`, `None`, `Question`, `Stop` и `Warning`) из перечисления `MessageBoxIcon`. Количество и название кнопок в диалоговом окне определяется константой (`AbortRetryIgnore`, `OK`, `OKCancel`, `RetryCancel`, `YesNo` и `YesNoCancel`) из перечисления `MessageBoxButtons`.

Использование пространства имен

> Надежа-царь говорит, что я князь Милославский. Устраивает это вас?
>
> *из к/ф «Иван Васильевич меняет профессию»*

Мы многократно использовали самые разные *пространства имен*. Теперь пришло время познакомиться с концепцией пространства имен поближе. Для начала отметим, что пространство имен используется для определения области уникальности имен: названия классов, структур, делегатов, интерфейсов и перечислений, объявленных в одном пространстве имен, должны быть уникальными. Другими словами, пространство имен — это некоторое множество классов и других утилит, и в пределах пространства имен названия этих утилит не могут совпадать. Например, два класса в одном и том же пространстве не могут иметь одинаковые названия. Зато если классы относятся к разным пространствам имен, то названия классов могут и совпадать.

НА ЗАМЕТКУ

Пространства имен могут содержать в себе другие пространства имен. Обращение к таким внутренним пространствам имен выполняется с использованием точечного синтаксиса: указывается вся цепочка вложенных пространств имен, а названия пространств разделяются точкой. Например, если пространство имен `Third` находится в пространстве имен `Second`, а пространство `Second` находится в пространстве имен `First`, то для обращения к пространству имен `Third` используется выражение `First.Second.Third`.

Мы уже знаем, что если в программе нужно использовать пространство имен, то в начале файла с программой размещается инструкция `using`, после которой указывается название пространства имен. В результате классы, структуры и прочие утилиты, описанные в пространстве имен, станут доступны в программе. Если пространство имен не подключено с помощью инструкции `using`, то для обращения к утилитам из этого пространства необходимо перед именем утилиты (например, класса, структуры, интерфейса) указывать название пространства имен. Например, если класс `MyClass` описан в пространстве имен `MySpace`, то обращение к такому классу выполняется инструкцией вида `MySpace.MyClass`.

Мы можем не только подключать уже существующие пространства имен, но и описывать собственные. Делается это просто: после ключевого слова `namespace` указывается название пространства имен, и в блоке из фигурных скобок описываются классы, структуры, интерфейсы, перечисления и делегаты, которые входят в это пространство имен.

НА ЗАМЕТКУ

Описание пространства имен может быть разбито на несколько блоков. Другими словами, мы можем описывать пространство имен не все сразу, а «по частям».

Следовательно, шаблон описания пространства имен следующий:

```
namespace имя{
   // Содержимое пространства имен
}
```

В листинге 9.2 представлен небольшой пример, иллюстрирующий процесс описания пространства имен. В этой программе описаны два пространства имен `First` и `Second`, и каждое пространство имен содержит описание класса `MyClass` (то есть пространства имен содержат разные классы с совпадающими названиями).

Листинг 9.2. Объявление пространства имен

```
using System;
// Первое пространство имен:
namespace First{
   // Описание класса:
   class MyClass{
      public int num;
      public MyClass(int n){
         num=n;
      }
      public void show(){
         Console.WriteLine("Класс из First: "+num);
      }
   }
}
// Второе пространство имен:
namespace Second{
   // Описание класса:
   class MyClass{
      public string txt;
      public MyClass(string t){
         txt=t;
      }
      public void show(){
         Console.WriteLine("Класс из Second: "+txt);
      }
   }
```

```
}
// Главный класс:
class NameSpaceDemo{
   // Главный метод:
   static void Main(){
      // Объект класса из первого пространства имен:
      First.MyClass A=new First.MyClass(123);
      // Объект класса из второго пространства имен:
      Second.MyClass B=new Second.MyClass("text");
      // Использование объектов:
      A.show();
      B.show();
   }
}
```

Результат выполнения программы такой, как показано ниже:

Результат выполнения программы (из листинга 9.2)

```
Класс из First: 123
Класс из Second: text
```

Как отмечалось выше, оба пространства (`First` и `Second`) содержат описание класса. Классы имеют одинаковые названия `MyClass`, они однотипны, но это разные классы. Поскольку они относятся к разным пространствам имен, то совпадение их названий проблемой не является. Мы в программе для подключения пространств имен `First` или `Second` инструкцию `using` не используем, поэтому обращение к классу из пространства имен `First` выполняется в формате `First.MyClass`, а обращение к классу из второго пространства имен выполняется инструкцией вида `Second.MyClass`. На основе этих классов создаются объекты (команды `First.MyClass A=new First.MyClass(123)` и `Second.MyClass B=new Second.MyClass("text")`). Затем из этих объектов вызывается метод `show()`, что дает возможность проверить значения полей созданных объектов (в соответствии с описанием классов, у одного объекта поле целочисленное, а у другого — текстовое).

Инструкция `using`, кроме известного нам способа использования, позволяет создавать еще и псевдонимы. С псевдонимами мы тоже имели дело. Например, мы использовали инструкцию `string`, которая является псевдонимом выражения `System.String`. Инструкция `object` является псевдонимом для выражения `System.Object`. Список можно продолжить.

При создании псевдонима инструкция `using` используется в таком формате:

```
using имя=выражение;
```

В данном случае идентификатор `имя` становится эквивалентом для инструкции, заданной `выражением`. Чтобы прояснить ситуацию, рассмотрим пример в листинге 9.3.

Листинг 9.3. Использование инструкции using

```
using System;
// Используется первое пространство имен:
using First;
// Используется псевдоним для класса
// из второго пространства имен:
using myclass=Second.MyClass;
// Описание первого пространства имен:
namespace First{
   class MyClass{
      public int num;
      public MyClass(int n){
         num=n;
      }
      public void show(){
         Console.WriteLine("Класс из First: "+num);
      }
   }
}
// Описание второго пространства имен:
```

```
namespace Second{
   class MyClass{
      public string txt;
      public MyClass(string t){
         txt=t;
      }
      public void show(){
         Console.WriteLine("Класс из Second: "+txt);
      }
   }
}
// Главный класс:
class UsingNameSpaceDemo{
   static void Main(){
      // Объект класса из первого пространства имен:
      MyClass A=new MyClass(123);
      // Объект класса из второго пространства имен:
      myclass B=new myclass("text");
      A.show();
      B.show();
   }
}
```

Данная программа практически совпадает с предыдущей (см. Листинг 9.2), за исключением лишь нескольких моментов. Как и ранее, мы описываем два пространства имен `First` и `Second`, и в каждом из них есть класс `MyClass`. Но на этот раз в программе есть инструкция `using First`. Поэтому в программе мы можем обращаться к классу `MyClass` из пространства имен `First` просто по имени `MyClass`, не указывая явно пространство имен `First`.

НА ЗАМЕТКУ

Если в программе, кроме инструкции `using First`, добавить еще и инструкцию `using Second`, то возникнет ошибка и программа

не скомпилируется. Причина ошибки в том, что в таком случае по имени класса `MyClass` нельзя однозначно определить, о каком именно классе (из какого пространства имен) идет речь.

Кроме этого, мы использовали инструкцию `using myclass=Second.MyClass`. В результате к классу `MyClass` из пространства имен `Second` можно обращаться по псевдониму `myclass`. Результат выполнения этой программы такой же, как и в предыдущем случае (см. Листинг 9.2 и результат выполнения программы).

НА ЗАМЕТКУ

Для использования статических членов класса без явного обозначения названия класса можно применить статический импорт. Для этого после инструкции `using` указывается ключевое слово `static`, а затем — полная ссылка (с указанием пространства имен) на класс. Например, если воспользоваться инструкцией `using static System.Math`, то к статическим методам и полям класса `Math` можно обращаться, не указывая имя класса. То есть вместо `Math.PI` или `Math.Cos()` можно соответственно использовать инструкции `PI` и `Cos()`. Аналогично, воспользовавшись инструкцией `using static System.Console`, получаем возможность вместо обращения к методу `Console.WriteLine()` использовать выражение `WriteLine()`.

Работа с датой и временем

Прием окончен. Обеденный перерыв!

из к/ф «Иван Васильевич меняет профессию»

Нередко приходится иметь дело с таким специфическим типом информации, как *дата* и *время*. Для работы с датой и временем в языке C# предназначена структура `DateTime`. Экземпляры этой структуры позволяют представлять дату и время в диапазоне от 00:00:00 (ноль часов, ноль минут, ноль секунд) 1 января 1 года новой эры до 23:59:59 (без одной секунды полночь) 31 декабря 9999 года (даты здесь указаны по григорианскому календарю). Технически дата (и время) реализуется в виде целого числа, равного количеству тактов, которые прошли до данной даты, начиная с полночи 1 января 1 года. Длительность одного такта равна 100 наносекундам (или 1 миллисекунда — это 10 000 тактов).

НА ЗАМЕТКУ

«Наибольшую» и «наименьшую» дату, которые можно реализовать с помощью экземпляра структуры, можно узнать с помощью статических полей `MaxValue` и `MinValue` соответственно. Значением полей (доступных только для чтения) являются экземпляры структуры `DateTime`, соответствующие финальной и начальной датам, которые могут быть реализованы с помощью этой структуры.
Если нас интересует только время (без даты), то в этом случае можно использовать экземпляр структуры `TimeSpan`.

Экземпляр структуры `DateTime` реализует определенную дату (и время). Существуют разные способы создать экземпляр структуры. Например, аргументами конструктору можно передать год, месяц и день для даты, реализуемой экземпляром структуры. Можно дополнительно указать часы, минуты и секунды. Или даже миллисекунды. Предусмотрена возможность явно указывать календарь, а также использовать мировое время (вместо местного, используемого по умолчанию). Можно определить дату и время, передав конструктору количество тактов.

НА ЗАМЕТКУ

Экземпляр структуры `DateTime` содержит информацию о годе, месяце, дне в месяце, часах, минутах, секундах и миллисекундах. Здесь и далее под датой мы будем подразумевать совокупность всех перечисленных выше параметров.

В структуре `DateTime` есть много свойств и методов. Часто используется статическое свойство `Now`. Результатом оно возвращает экземпляр структуры, соответствующий текущей дате и времени (определяется по текущему времени компьютера). Если нужно получить текущую дату и время по мировому времени (формат UTC), используют свойство `UtcNow`.

У структуры `DateTime` есть и другие полезные свойства:

- свойство `Date` результатом возвращает экземпляр структуры `DateTime`, содержащий дату из исходного экземпляра (без времени — время устанавливается нулевое);
- значением свойства `TimeOfDay` возвращается экземпляр структуры `TimeSpan`, определяющий время для исходной даты (то есть это время без дня, месяца и года);

- значением свойства Today является экземпляр структуры DateTime, реализующий текущую дату (имеется в виду только дата, без времени — время устанавливается нулевое);
- значением свойства Year является целое число, соответствующее году для данной даты;
- значением свойства Month является целое число, определяющее месяц для данной даты;
- значение свойства Day — это целое число, определяющее день в месяце для данной даты;
- значением свойства Hour является целое число, определяющее количество часов для данной даты;
- значением свойства Minute является целое число, определяющее количество минут для данной даты;
- значением свойства Second является целое число, определяющее количество секунд для данной даты;
- свойство Millisecond возвращает целочисленное значение, определяющее миллисекунды для даты;
- свойство Ticks возвращает целое число, равное количеству тактов в данной дате;
- значением свойства DayOfWeek является константа из одноименного перечисления DayOfWeek, определяющая день недели для даты;
- свойство DayOfYear возвращает результатом целое число, определяющее день в году для данной даты;
- значением свойства Kind является константа из перечисления DateTimeKind, определяющая характер времени (местное или мировое).

Кроме многочисленных свойств, у структуры DateTime имеется много методов, позволяющих выполнять различные операции с датой и временем. Есть группа методов, предназначенных для добавления к дате, реализованной экземпляром структуры DateTime, определенного интервала времени (годы, дни, часы, секунды и так далее):

- Методу Add() аргументом передается экземпляр структуры TimeSpan (время в часах, минутах, секундах и миллисекундах). Результатом возвращается новый экземпляр структуры DateTime, который

описывает результат добавления данного временного интервала к исходной дате.

- Метод `AddDays()` позволяет добавить к дате дни (`double`-аргумент метода). Результат — экземпляр структуры `DateTime`.
- Метод `AddHours()` используется для добавления к дате часов (`double`-аргумент метода), результатом возвращается экземпляр структуры `DateTime`.
- Метод `AddMilliseconds()` аргументом получает `double`-значение, определяющее количество миллисекунд, добавляемых к дате. Результатом возвращается экземпляр структуры `DateTime`, соответствующий новой дате.
- Метод `AddMinutes()` позволяет вычислить новую дату (экземпляр структуры `DateTime`), получающуюся прибавлением к исходной дате определенного количества минут (`double`-аргумент метода).
- Метод `AddMonths()` аргументом получает количество месяцев, прибавляемых к дате (целое число). Результатом является экземпляр структуры `DateTime` с новой датой.
- Методу `AddSeconds()` передается `double`-значение, определяющее количество добавляемых к дате секунд. Результатом возвращается экземпляр структуры `DateTime` с новой датой.
- Метод `AddTicks()` позволяет добавить к дате указанное количество тактов (целочисленный аргумент метода). Результатом возвращается экземпляр структуры `DateTime` с новой датой.
- Метод `AddYears()` позволяет добавить к дате определенное количество лет (целочисленный аргумент метода). Результатом является экземпляр структуры `DateTime` с новой датой.
- Метод `DaysInMonth()` аргументами получает два целочисленных значения (год и месяц в году), а результатом возвращает количество дней в месяце.
- Метод `GetDateTimeFormats()` может вызываться с разными аргументами (в том числе и без них), а результатом является текстовый массив с поддерживаемыми форматами для текстовых представлений даты.
- Метод `IsDaylightSavingTime()` позволяет проверить, попадает ли данная дата в диапазон летнего времени (для текущего часового пояса).

- Статический метод `IsLeapYear()` используется для проверки того, является ли год високосным. Год в виде целого числа передается аргументом методу.
- Статический метод `Parse()` используется для преобразования текста (аргумент метода) в экземпляр структуры `DataTime`. Существуют разные способы передачи аргументов методу `Parse()`. Аналогичная задача решается с помощью статического метода `ParseExact()`, но в этом случае требуется строгое соответствие текстовой строки с датой заданному формату. Для решения аналогичных задач также могут использоваться статические методы `TryParse()` и `TryParseExact()`.
- Аргументом методу `Subtract()` передается экземпляр структуры `DateTime` (дата) или `TimeSpan` (время). Из исходной даты вычитается дата/время, переданные аргументом методу Результатом возвращается экземпляр структуры `TimeSpan` и `DateTime` соответственно.
- Для перевода даты в местное время используют метод `ToLocalTime()`.
- Для получения текстового представления даты используют метод `ToString()`. Метод может вызываться без аргументов (в том числе при автоматическом преобразовании типов) или с передачей аргументов, определяющих формат текстового представления.
- Для перевода даты в мировое время UTC используют метод `ToUniversalTime()`.
- Для сравнения дат используют методы `Compare()`, `CompareTo()` и `Equals()`.

НА ЗАМЕТКУ

К экземплярам структуры `DateTime` применяются операторы сравнения, а также сложение (к экземпляру структуры `DateTime` можно прибавлять экземпляр структуры `TimeSpan`) и вычитание (из экземпляра структуры `DateTime` можно вычитать экземпляр структуры `DateTime` или `TimeSpan`).

Скромный пример, иллюстрирующий методы работы с датой и временем, представлен в листинге 9.4.

Листинг 9.4. Работа с датой и временем

```
using System;

// Главный класс:
```

```
class DateTimeDemo{
   // Главный метод:
   static void Main(){
      // Текущая дата и время:
      DateTime today=DateTime.Now;
      // Текущая дата:
      Console.WriteLine("Текущая дата: {0}",DateTime.Today);
      // Использование свойств и методов
      // структуры DateTime:
      Console.WriteLine("Дата и время: {0}",today);
      Console.WriteLine("Год: {0}",today.Year);
      Console.WriteLine("Високосный: {0}",DateTime.IsLeapYear(today.Year));
      Console.WriteLine("Месяц: {0}",today.Month);
      Console.WriteLine("День недели: {0}",today.DayOfWeek);
      Console.WriteLine("День в году: {0}",today.DayOfYear);
      Console.WriteLine("Часы: {0}",today.Hour);
      Console.WriteLine("Минуты: {0}",today.Minute);
      Console.WriteLine("Секунды: {0}",today.Second);
      Console.WriteLine("Миллисекунды: {0}",today.Millisecond);
      Console.WriteLine("Такты: {0}",today.Ticks);
      // Новая дата и время:
      DateTime date=new DateTime(1974,10,22,6,15,12);
      Console.WriteLine(date.ToString("Новая дата: dd MMMM yyyy; время – HH:mm:ss"));
      // Интервал времени:
      TimeSpan time=new TimeSpan(17,71,125);
      // Операции с датой и временем:
      Console.WriteLine("Интервал времени: {0}",time);
      Console.WriteLine("Еще одна дата: {0}",date.Add(time));
      Console.WriteLine("Разность дат (дней): {0}",(today-date).Days);
      Console.WriteLine("Разность дат (тактов): {0}",(today-date).Ticks);
   }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 9.4)

```
Текущая дата: 29.10.20170:00:00
Дата и время: 29.10.201713:41:13
Год: 2017
Високосный: False
Месяц: 10
День недели: Sunday
День в году: 302
Часы: 13
Минуты: 41
Секунды: 13
Миллисекунды: 958
Такты: 636448812739586786
Новая дата: 22 октября 1974; время — 06:15:12
Интервал времени: 18:13:05
Еще одна дата: 23.10.19740:28:17
Разность дат (дней): 15713
Разность дат (тактов): 13576299619586786
```

Код исключительно простой, поэтому прокомментируем лишь некоторые его моменты. Так, результатом выражения `DateTime.Now` является текущая (на момент вызова команды) дата и время. Результатом выражения `DateTime.Today` является текущая дата, но время в этом случае нулевое. Командой `DateTime date=new DateTime(1974,10,22, 6,15,12)` создается экземпляр для реализации даты. Год, месяц, день месяца, часы, минуты и секунды для даты передаются аргументами конструктору.

При вызове из экземпляра `date` структуры `DateTime` метода `ToString()` с текстовым аргументом инструкция `"dd"` в тексте означает, что в соответствующем месте должен отображаться день месяца (двумя цифрами). Инструкция `"MMMM"` означает отображение названия месяца. Инструкция `"yyyy"` означает, что в соответствующем месте год должен

отображаться с помощью четырех цифр. Инструкция "HH:mm:ss" определяет формат для отображения времени (часы, минуты, секунды).

ПОДРОБНОСТИ

Инструкция "d" означает, что для обозначения даты используются числа от 1 до 31. Инструкция "dd" означает, что день обозначается числами от 01 до 31 (то есть всегда используется две цифры). Инструкция "ddd" используется, если день должен отображаться как день недели (в сокращенном написании). Если используется инструкция "dddd", то день отображается как день недели (полное написание).

Месяц отображается с помощью инструкций "M" (число от 1 до 12), "MM" (число от 01 до 12 — то есть всегда две цифры), "MMM" (сокращенное название месяца), "MMMM" (полное название месяца).

Для отображения года можно использовать инструкции "y" (число от 0 до 99), "yy" (число от 00 до 99 — то есть всегда две цифры), "yyy" (год отображается числом минимум из трех цифр), "yyyy" (год отображается числом из четырех цифр).

Инструкции "h" и "hh" используются для отображения количества часов числами от 1 до 12 и от 01 до 12 соответственно. А инструкции "H" и "HH" используются для отображения количества часов числами от 0 до 23 и от 00 до 23.

Инструкции "m" и "mm" используют для отображения количества минут числами от 0 до 59 и от 00 до 59. Для отображения секунд используют инструкции "s" (диапазон значений от 0 до 59) и "ss" (диапазон значений от 00 до 59).

Экземпляр структуры TimeSpan, который создается командой TimeSpan time=new TimeSpan(17,71,125), определяет интервал времени. Аргументы конструктора — это часы, минуты и секунды.

НА ЗАМЕТКУ

Обратите внимание, что 71 минута — это 1 час и 11 минут, а 125 секунд — это 2 минуты и 5 секунд. Таким образом, экземпляру time структуры TimeSpan отвечает интервал времени протяженностью 18 часов, 13 минут и 5 секунд.

Результатом выражения date.Add(time) является экземпляр структуры DateTime, через который реализуется такая дата: к дате из экземпляра date прибавляется временной интервал из экземпляра time. Результатом разности today-date является экземпляр структуры TimeSpan. В нем реализован интервал времени между датами из экземпляров

today и date. Свойства Days и Ticks позволяют узнать количество дней в этом интервале и количество тактов в этом интервале (напомним, один такт равен 100 наносекундам).

Еще один небольшой пример, связанный с использованием дат, представлен в листинге 9.5. В этой простой программе для указанного диапазона лет вычисляются дни, когда 13-й день месяца попадает на пятницу.

Листинг 9.5. Использование дат

```
using System;
// Главный класс:
class FindFridayDemo{
   // Главный метод:
   static void Main(){
      // Годы начала и окончания поиска и день месяца:
      int start=2015,end=2020,day=13;
      // Переменная для записи даты:
      DateTime date;
      Console.WriteLine("Пятница:");
      // Перебираются годы:
      for(int year=start;year<=end;year++){
         // Перебираются месяцы:
         for(int month=1;month<=12;month++){
            // Дата:
            date=new DateTime(year,month,day);
            // Если это пятница:
            if(date.DayOfWeek==DayOfWeek.Friday){
               // Отображение даты:
               Console.WriteLine(date.ToString("dd MMMM yyyy года"));
            }
         }
      }
   }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 9.5)

```
Пятница:
13 февраля 2015 года
13 марта 2015 года
13 ноября 2015 года
13 мая 2016 года
13 января 2017 года
13 октября 2017 года
13 апреля 2018 года
13 июля 2018 года
13 сентября 2019 года
13 декабря 2019 года
13 марта 2020 года
13 ноября 2020 года
```

В целочисленную переменную `start` записывается год, начиная с которого выполняется поиск (значение `2015`), в переменную `end` записывается год окончания поиска (значение `2020`), а в переменную `day` записывается номер дня в месяце (значение `13`). Переменная `date` типа `DateTime` нужна для запоминания даты. В программе использованы вложенные операторы цикла. Во внешнем операторе цикла переменная `year` (год для даты) принимает значения от `start` до `end` включительно. Во внутреннем операторе цикла с помощью переменной `month` перебираются месяцы. Командой `date=new DateTime(year,month,day)` создается экземпляр для даты. В условном операторе проверяется условие `date.DayOfWeek==DayOfWeek.Friday`. Условие истинно, если дата, записанная в `date`, соответствует пятнице. Если так, то дата отображается в консольном окне.

Работа с файлами

> Такие вопросы, дорогой посол, с кондачка не решаются. Нам надо посоветоваться с товарищами, зайдите на недельке.
>
> *из к/ф «Иван Васильевич меняет профессию»*

Очень часто данные в программу передаются с помощью *файлов*. Вывод результатов в файл также является удобным и популярным способом реализации программы. В этом разделе речь пойдет о том, как в C# реализуется работа с файлами. Более конкретно, мы обсудим вопросы, связанные со считыванием информации из файла и записью информации в файл.

Важным элементом в концепции файлового ввода и вывода данных является такое понятие, как *поток*.

НА ЗАМЕТКУ

Не следует путать потоки данных с потоками, которые мы рассматривали в главе, посвященной многопоточному программированию. В английском языке термин thread используется для обозначения кода, выполняемого одновременно с другими блоками кода. Термин stream используется для обозначения последовательности данных. Оба термина на русский язык обычно переводят как поток, но это разные потоки. В данном случае мы говорим о потоках, которым соответствует англоязычный термин stream.

Под потоком мы будем подразумевать некоторую упорядоченную последовательность данных. Технически такая последовательность может рассматриваться как набор битов (нулей и единиц). Обрабатываются такие последовательности блоками. Как мы знаем, блок из восьми битов называется байтом. Соответственно, байтовый поток — это поток, который обрабатывается на уровне байтов (то есть байтовыми блоками). Есть и символьные потоки — в них данные обрабатываются блоками по два байта, и каждый такой блок интерпретируется как символ.

НА ЗАМЕТКУ

Напомним, что в языке C# для хранения символьных значений выделяется 16 битов, или 2 байта.

Для работы с потоками существуют специальные классы. Абстрактный класс `Stream` из пространства имен `System.IO` является базовым для других классов, предназначенных для работы с потоками. В частности, для реализации файлового ввода/вывода используется класс `FileStream`. Этот класс предназначен для работы с байтовыми потоками. У класса есть несколько конструкторов, которые позволяют создать поток, связанный с файлом. Например, первым аргументом конструктору класса можно передать текст с названием файла (полным путем к файлу), а второй аргумент при этом определяет режим доступа к файлу. Для определения режима доступа к файлу используют одну из констант перечисления `FileMode`:

- Константа `Append` означает режим, при котором новые данные добавляются в конец файла.
- Константа `Create` означает, что будет создан новый файл. Если файл с указанным именем уже существует, то он будет заменен новым.
- Константа `CreateNew` используется, если нужно создать новый файл. При этом не допускается, чтобы файл с указанным именем уже существовал.
- Константа `Open` означает, что открывается уже существующий файл.
- Если использована константа `OpenOrCreate`, то открывается существующий файл, а если файла нет, то он создается.
- Константа `Truncate` означает, что открывается существующий файл, при этом для файла устанавливается нулевой размер.

НА ЗАМЕТКУ

Попытка открыть файл может привести к ошибке. Например, если файл отсутствует, генерируется ошибка класса `FileNotFoundException`. При отсутствии прав доступа к файлу генерируется исключение класса `SecurityException`. При ошибке ввода/вывода генерируется исключение `IOException` и так далее. Поэтому при создании потока на основе файла обычно применяется обработка исключительных ситуаций.

Создание потока на основе файла открывает возможность для записи данных в файл и считывания данных из файла. В принципе, можно ограничиться лишь одной операцией (считывание данных или запись данных). Сделать это можно с помощью третьего аргумента, который

передается конструктору класса FileStream. Это константа из перечисления FileAccess: Read (только чтение), Write (только запись) или ReadWrite (чтение и запись).

Объекты класса FileStream имеют полезные свойства и методы. Например, длину потока в байтах можно узнать с помощью свойства Length. Свойства CanRead и CanWrite позволяют определить доступность потока для чтения и записи соответственно. Позиция в потоке определяется с помощью свойства Position. Есть и другие свойства. Среди методов можно выделить такие:

- Метод ReadByte() позволяет считать байт из потока. Считанный байт возвращается результатом метода. Если достигнут конец файла, результатом метода возвращается значение -1.
- Метод Read() считывает байты из потока и записывает их в байтовый массив. Первым аргументом метод получает байтовый массив, в который записываются байты. Второй аргумент метода определяет позицию в массиве, начиная с которого копируются байты. Третий аргумент метода определяет количество байтов, считываемых из потока.
- Метод WriteByte() позволяет записать байт (аргумент метода) в файловый поток (то есть в файл) в текущую позицию.
- Метод Write() позволяет записать байты из массива (первый аргумент метода) в файловый поток (то есть в файл). Вторым аргументом методу передается индекс элемента в массиве, начиная с которого копируются байты. Третий аргумент метода — количество байтов, записываемых в поток.
- Метод Close() используется для закрытия потока. При завершении работы с потоком необходимо вызывать этот метод.
- Метод CopyTo() позволяет копировать содержимое одного потока (из объекта которого вызывается метод) в другой поток (аргумент метода).
- Метод Flush() вызывается в случае, если необходимо записать буферизированные данные в поток.
- Метод Seek() используется для установки текущего положения в потоке для ввода/вывода данных. Вторым аргументом методу передается константа из перечисления SeekOrigin (возможные значения Begin, Current и End), которая определяет «отправную точку»

для определения позиции в потоке. Позиция в потоке определяется смещением по отношению к «отправной точке». Значение смещения (в байтах) передается первым аргументом методу.

В листинге 9.6 представлена программа, при выполнении которой осуществляется побайтовое копирование файла: содержимое исходного (существующего) файла считывается байт за байтом и копируется в новый файл, который создается программой.

Листинг 9.6. Байтовые файловые потоки

```
using System;
using System.IO;
// Главный класс:
class FileStreamDemo{
   // Главный метод:
   static void Main(){
      // Каталог с исходным файлом:
      string pathFrom="D:/Books/pictures/";
      // Каталог с новым файлом:
      string pathTo="D:/Books/files/";
      // Название исходного (копируемого) файла:
      string file="bear.jpg";
      // Название для нового файла:
      string name="animal.jpg";
      // Название и полный путь для файлов:
      string f=pathFrom+file;
      string F=pathTo+name;
      // Контролируемый код:
      try{
         // Создание потока для считывания файла:
         FileStream fin=new FileStream(f,FileMode.Open);
         Console.WriteLine("Исходный файл: \"{0}\"",f);
         // Создание потока для записи в файл:
         FileStream fout=new FileStream(F,FileMode.Create);
```

```
            Console.WriteLine("Новый файл: \"{0}\"",F);
            // Целочисленная переменная:
            int val;
            // Считывается первый байт из потока:
            val=fin.ReadByte();
            // Чтение и запись байтов:
            while(val!=-1){
               // Запись байта в поток:
               fout.WriteByte((byte)val);
               // Считывается байт из потока:
               val=fin.ReadByte();
            }
            // Закрываются потоки:
            fout.Close();
            fin.Close();
         } // Обработка исключений:
         catch(Exception e){
            Console.WriteLine("Произошла ошибка!");
            Console.WriteLine(e.Message);
            return; // Завершение выполнения программы
         }
         Console.WriteLine("Копирование успешно завершено");
      }
}
```

При успешном выполнении программы в консольном окне появляются такие сообщения:

Результат выполнения программы (из листинга 9.6)

```
Исходный файл: "D:/Books/pictures/bear.jpg"
Новый файл: "D:/Books/files/animal.jpg"
Копирование успешно завершено
```

Но главный результат выполнения программы иной: в каталоге D:\Books\files появляется файл animal.jpg, который является копией файла bear.jpg из каталога D:\Books\pictures.

НА ЗАМЕТКУ

Разумеется, перед запуском программы на выполнение в каталог D:\Books\pictures нужно поместить файл bear.jpg. Альтернатива: указать в программе другой каталог и другой исходный файл. Но если, несмотря на все усилия, при выполнении программы возникает ошибка (связанная, например, с отсутствием исходного файла в указанном каталоге), то в процессе обработки исключения отображается сообщение о том, что возникла ошибка, и описание этой ошибки (значение свойства Message объекта исключения).

В самой программе в текстовые переменные f и F записывается полный путь к исходному файлу и файлу, который должен быть создан в процессе копирования. Текстовая переменная f передается в качестве аргумента конструктору класса FileStream при создании объекта потока fin. Это поток используется для считывания содержимого исходного файла. Второй аргумент конструктора FileMode.Open означает, что речь идет об открытии существующего файла. Текстовая переменная F передается первым аргументом конструктору класса FileStream при создании объекта потока fout. Поток предназначен для записи байтовых данных в файл, который создается при создании объекта потока (поэтому вторым аргументом конструктору передана константа FileMode.Open).

Для запоминания считанных байтов мы используем переменную val типа int. Командой val=fin.ReadByte() в эту переменную записывается первый байт из потока, связанного с исходным файлом. После этого запускается оператор цикла while, в котором проверяется условие val!=-1. Здесь мы учли, что если достигнут конец файла, то при попытке прочитать очередной байт с помощью метода ReadByte() возвращается значение -1. Поэтому истинность условия val!=-1 означает, что конец файла не достигнут. Если так, то командой fout.WriteByte((byte)val) этот байт записывается в поток fout, после чего командой val=fin.ReadByte() считывается очередной байт из потока fin.

ПОДРОБНОСТИ

Метод ReadByte() считывает байт, но возвращает значение типа int. Значение типа byte не может быть отрицательным. Значение

> -1, возвращаемое методом при достижении конца файла, является целочисленным (типа int). Поэтому результат метода заносится в переменную типа int. Но при записи байта в поток это int-значение необходимо преобразовать в byte-значение.

После завершения оператора цикла, когда копирование выполнено, командами fout.Close() и fin.Close() оба потока закрываются.

В рассмотренной выше программе мы проиллюстрировали процесс побайтового считывания информации из файла и записи данных в файл. Что касается собственно задачи копирования файлов, то ее можно было реализовать проще, воспользовавшись методом CopyTo(). Соответствующая программа представлена в листинге 9.7.

Листинг 9.7. Копирование файла

```
using System;
using System.IO;
// Главный класс:
class CopyToDemo{
   // Главный метод:
   static void Main(){
      // Название и полный путь для файлов:
      string f="D:/Books/pictures/bear.jpg";
      string F="D:/Books/files/animal.jpg";
      try{
         // Создание потока для считывания файла:
         FileStream fin=new FileStream(f,FileMode.Open);
         Console.WriteLine("Исходный файл: \"{0}\"",f);
         // Создание потока для записи в файл:
         FileStream fout=new FileStream(F,FileMode.Create);
         Console.WriteLine("Новый файл: \"{0}\"",F);
         // Копирование файла:
         fin.CopyTo(fout);
         // Закрываются потоки:
         fout.Close();
```

```
            fin.Close();
        }
        catch(Exception e){
            Console.WriteLine("Произошла ошибка!");
            Console.WriteLine(e.Message);
            return;
        }
        Console.WriteLine("Копирование успешно завершено");
    }
}
```

Результат выполнения программы такой же, как и в предыдущем случае. В данном случае в программе, после создания потоков для считывания и записи данных, из объекта потока `fin` вызывается метод `CopyTo()`, а аргументом методу передается объект потока `fout`. В результате выполняется копирование файла, связанного с потоком `fin`, в файл, связанный с потоком `fout`.

Еще раз подчеркнем, что особенность байтового потока в том, что данные считываются блоками размером в байт. На практике нередко приходится иметь дело с символьными данными. Символ реализуется в виде блока размером в два байта. Поэтому при работе с символьной информацией удобнее выполнять ввод и вывод данных блоками размером по два байта — то есть посимвольно. В таком случае используют символьные потоки. Объекты символьных потоков создаются на основе классов `StreamReader` (поток для считывания) и `StreamWriter` (поток для записи). Но в основе символьного потока все равно лежит байтовый поток. Разница лишь в том, какими блоками считывается и записывается информация. Это отражается и на процессе создания символьных потоков: аргументом конструктору класса, на основе которого создается символьный поток, передается объект байтового потока. Вместе с тем существует и другой способ создать символьный поток: аргументом конструктору класса `StreamReader` или `StreamWriter` достаточно передать текст с именем файла, на основе которого создается поток.

Объект символьного потока для считывания данных создается на основе класса `StreamReader`. Для считывания одного символа используется метод `Read()`. Результатом метода возвращается `int`-значение

с кодом символа или значение –1, если достигнут конец файла. Метод ReadLine() позволяет считать текстовую строку. Текстовая строка возвращается результатом метода. Если достигнут конец файла, то метод возвращает пустую ссылку null.

Объект символьного потока для записи данных в файл создается на основе класса SreamWriter. Для записи данных в файл используются методы Write() и WriteLine(). Разница между методами в том, что метод WriteLine() после занесения данных в файл добавляет инструкцию перехода к новой строке. Пример создания и использования символьных потоков представлен в листинге 9.8.

Листинг 9.8. Символьные потоки

```
using System;
using System.IO;
// Главный класс:
class ReaderWriterDemo{
   // Главный метод:
   static void Main(){
      // Название и полный путь для файла:
      string file="D:/Books/files/MyText.txt";
      // Текстовая переменная:
      string str;
      // Ключевое слово:
      string word="exit";
      // Целочисленная переменная:
      int k=1;
      try{   // Контролируемый код
         // Символьный поток для записи в файл:
         StreamWriter fw=new StreamWriter(file);
         Console.WriteLine("Для завершения введите \"{0}\"",word);
         // Считывание текстовых строк с консоли:
         do{
            // Приглашение ввести строку:
            Console.Write("{0}-я строка: > ",k);
```

```
            k++;
            // Считывание текстовой строки с консоли:
            str=Console.ReadLine();
            if(str==word){ // Если введено ключевое слово
               // Завершение оператора:
               break;
            }
            else{ // Если не введено ключевое слово
               // Запись текстовой строки в файл:
               fw.WriteLine(str);
            }
         }while(true);
         // Закрывается поток:
         fw.Close();
         Console.WriteLine("Файл создан!");
         // Создание потока для считывания из файла:
         StreamReader fr=new StreamReader(file);
         Console.WriteLine("Содержимое файла:");
         // Считывание строки из файла:
         str=fr.ReadLine();
         // Построчное считывание содержимого файла:
         while(str!=null){
            // Отображение считанной из файла строки:
            Console.WriteLine(str);
            // Считывание строки из файла:
            str=fr.ReadLine();
         }
         // Закрывается поток:
         fr.Close();
      } // Обработка исключений:
      catch(Exception e){
         Console.WriteLine("Произошла ошибка!");
```

```
            Console.WriteLine(e.Message);
        }
    }
}
```

Программа выполняется следующим образом: пользователь последовательно вводит текстовые строки, которые считываются и построчно записываются в текстовый файл. Файл создается программой. Процесс ввода строк завершается, когда пользователь вводит ключевое слово `"exit"`. После этого содержимое созданного файла построчно считывается и построчно отображается в консольном окне.

В программе в текстовую переменную `file` записывается полный путь к текстовому файлу, который создается программой, а также название файла. Текстовая переменная `str` используется для запоминания текстовых строк, которые будет вводить пользователь. В текстовую переменную `word` записывается ключевое слово, ввод которого означает завершение процесса формирования содержимого текстового файла. Целочисленная переменная `k` используется для нумерации строк.

Объект `fw` символьного потока для записи данных в файл создается командой `StreamWriter fw=new StreamWriter(file)`. Затем запускается формально бесконечный цикл с оператором `do-while` (в нем проверяемым условием указано значение `true`). В теле оператора цикла командой `str=Console.ReadLine()` считывается строка, введенная пользователем. Далее в условном операторе проверяется условие `str==word`. Условие истинно, если пользователь ввел ключевое слово. Если так, то командой `break` завершается выполнение оператора цикла. В противном случае (если условие ложно) командой `fw.WriteLine(str)` текстовая строка записывается в файл.

По завершении выполнения оператора цикла командой `fw.Close()` закрывается символьный поток. После этого командой `StreamReader fr=new StreamReader(file)` создается объект символьного потока для считывания данных из файла, который был создан и заполнен на предыдущем этапе. Командой `str=fr.ReadLine()` из файла считывается первая строка (если она есть), после чего запускается оператор цикла `while`. В операторе цикла проверяется условие `str!=null`. Истинность условия означает, что конец файла не достигнут и переменная `str` содержит ссылку на текст. В этом случае командой `Console.`

`WriteLine(str)` считанная из файла строка отображается в консольном окне. Затем командой `str=fr.ReadLine()` из файла считывается следующая строка. После того как содержимое файла прочитано, командой `fr.Close()` поток закрывается.

Результат выполнения программы может быть таким (жирным шрифтом выделен текст, который вводит пользователь):

Результат выполнения программы (из листинга 9.8)

```
Для завершения введите "exit"
1-я строка: > Кто понял жизнь, тот больше не спешит,
2-я строка: > Смакует каждый миг и наблюдает,
3-я строка: > Как спит ребенок, молится старик,
4-я строка: > Как дождь идет, и как снежинка тает.
5-я строка: > exit
Файл создан!
Содержимое файла:
Кто понял жизнь, тот больше не спешит,
Смакует каждый миг и наблюдает,
Как спит ребенок, молится старик,
Как дождь идет, и как снежинка тает.
```

Также стоит отметить, что в результате выполнения программы в каталоге `D:\Books\files` создается файл с названием `MyText.txt`. Содержимое этого файла показано на рис. 9.4.

Несложно догадаться, что именно это содержимое отображалось в консольном окне.

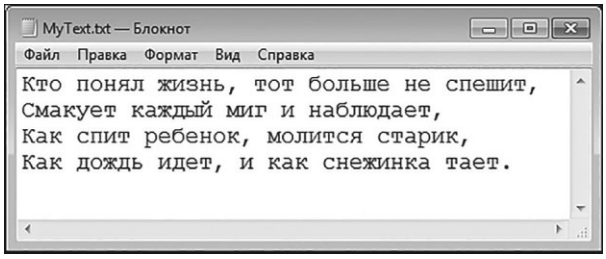


Рис. 9.4. Содержимое текстового файла, созданного в результате выполнения программы

В некоторых случаях удобно хранить данные в бинарных файлах (последовательность произвольных байтов). Для работы с бинарными файлами используют бинарные потоки. Объекты таких потоков создаются на основе классов `BinryReader` (поток для считывания данных из бинарного файла) и `BinaryWriter` (поток для записи данных в бинарный файл). Пример создания и использования бинарных потоков представлен в листинге 9.9.

Листинг 9.9. Бинарные потоки

```
using System;
using System.IO;
// Главный класс:
class BinarySreamsDemo{
   // Главный метод:
   static void Main(){
      // Данные для записи в файл:
      int num=123;                // Целое число
      char symb='A';              // Символ
      double x=2.5;               // Действительное число
      string txt="Hello, World!"; // Текст
      // Путь к файлу и его название:
      string file="D:/Books/files/MyData.dat";
      Console.WriteLine("Запись данных в файл...");
      try{ // Контролируемый код
         // Бинарный поток для записи данных в файл:
         BinaryWriter bw=new BinaryWriter(new FileStream(file,FileMode.Create));
         // Запись данных в файл:
         bw.Write(num);    // Целое число
         bw.Write(symb);   // Символ
         bw.Write(x);      // Действительное число
         bw.Write(txt);    // Текст
```

```
            // Поток закрывается:
            bw.Close();
        } // Обработка исключений:
        catch(Exception e){
            Console.WriteLine("Ошибка записи в файл!");
            Console.WriteLine(e.Message);
            return;
        }
        Console.WriteLine("Создан файл \"{0}\"",file);
        Console.WriteLine("Считывание данных из файла...");
        try{ // Контролируемый код
            // Бинарный поток для считывания данных из файла:
            BinaryReader br=new BinaryReader(new FileStream(file,FileMode.Open));
            // Считывание данных из файла и отображение в
            // консольном окне:
            Console.WriteLine(br.ReadInt32());  // Целое число
            Console.WriteLine(br.ReadChar());   // Символ
            Console.WriteLine(br.ReadDouble()); // Число
            Console.WriteLine(br.ReadString()); // Текст
            // Поток закрывается:
            br.Close();
        } // Обработка исключений:
        catch(Exception e){
            Console.WriteLine("Ошибка чтения файла!");
            Console.WriteLine(e.Message);
        }
        Console.WriteLine("Завершение программы...");
    }
}
```

При выполнении программы в консольном окне появляются следующие сообщения:

Результат выполнения программы (из листинга 9.9)

```
Запись данных в файл...
Создан файл "D:/Books/files/MyData.dat"
Считывание данных из файла...
123
A
2,5
Hello, World!
Завершение программы...
```

При выполнении программы в каталоге `D:\Books\files` создается файл `MyData.dat`. В этот файл записывается целочисленное значение, символ, действительное число и текст. После этого файл открывается для чтения, и записанные ранее в файл значения отображаются в консольном окне.

НА ЗАМЕТКУ

Если открыть файл, то его содержимое будет представлять собой последовательность странных символов и текста.

Данные, предназначенные для записи в файл, заносятся в переменные `num`, `symb`, `x` и `txt`. Путь к файлу и его название записываются в текстовую переменную `file`.

Бинарный поток для записи данных в файл создается на основе байтового потока. Если точнее, то при создании объекта бинарного потока `bw` на основе класса `BinaryWriter` в качестве аргумента конструктору класса передается инструкция `new FileStream(file,FileMode.Create)`. Эта инструкция означает создание байтового потока, который открывается в режиме создания нового файла. Созданный таким образом поток `bw` предназначен для записи данных в файл. Для этого из объекта потока вызывается метод `Write()`, аргументом которому передается записываемое в файл значение. При завершении записи данных в поток он закрывается командой `bw.Close()`.

После того как файл создан и в него занесены данные, создается еще один бинарный поток `br`. Поток создается на основе класса `BinaryReader` и предназначен для считывания данных из файла. Аргументом конструктору класса `BinaryReader` передается анонимный объект байтового потока, который создается инструкцией `new FileStream(file,FileMode.Open)` (поток создается в режиме открытия уже существующего файла). Для считывания значений из файла используются методы `ReadInt32()` (считывается целое число), `ReadChar()` (считывается символ), `ReadDouble()` (считывается действительное число) и `ReadString()` (считывается текст). Методы вызываются из объекта потока `br`, а по завершении процесса считывания данных поток закрывается командой `br.Close()`.

Знакомство с коллекциями

> История, которую мы хотим рассказать, не опирается на факты. Она настолько невероятна, что в нее просто нельзя не поверить.
>
> *из к/ф «О бедном гусаре замолвите слово»*

Далее мы познакомимся с тем, как в языке C# реализуются *коллекции* (иногда их еще называют контейнерами). Вообще, под коллекцией подразумевают группу объектов. Такая группа объектов реализуется как некий объект определенного класса, содержащий в себе коллекцию объектов другого класса. Технически это можно реализовать с помощью класса, у которого есть поле, представляющее собой ссылку на массив объектов, формирующих коллекцию. То есть в прикладном плане мы могли бы сами без особых проблем описать подобный класс. То есть сама по себе возможность «спрятать» в одном объекте группу других объектов уникальной не является. Удобство и эффективность использования коллекций во многом связаны с наличием методов, облегчающих работу с группами объектов, содержащихся в коллекции. Другая важная особенность коллекций, выделяющая их на фоне обычных массивов, — возможность добавлять элементы в коллекцию и удалять элементы из коллекции.

НА ЗАМЕТКУ

После создания массива его размер изменить нельзя. Но мы можем поступить так: создаем новый массив нужного размера, заполняем

его правильным образом и ссылку на этот массив записываем в переменную массива. Создается иллюзия, что массив изменил размер. Хотя это, конечно, не так. При работе с коллекциями процесс «изменения» размера массива, который лежит в основе коллекции, автоматизирован, что довольно удобно.

В C# существуют классы для реализации самых разных коллекций. С некоторыми из них мы познакомимся поближе.

Коллекции могут быть самыми разными, но у всех них есть определенные общие характеристики. Такое «единство стандарта» поддерживается за счет реализации классами коллекций специальных интерфейсов. Интерфейсы, реализованные в классе коллекции, определяют, какими методами обладает коллекция. Одним из фундаментальных интерфейсов при работе с коллекциями является `ICollection` (из пространства имен `System.Collections`). Кроме этого интерфейса, можно упомянуть такие, как `IComparer`, `IDictionary`, `IEnumerator`, `IList` и `INumerable`. Например, интерфейс `ICollection` содержит объявление доступного только для чтения свойства `Count`, определяющего количество элементов коллекции. Метод `CopyTo()` предназначен для копирования содержимого коллекции в массив, переданный методу в качестве аргумента. Интерфейс `IList` содержит объявление таких методов, как `Add()` (добавление элемента в коллекцию), `Clear()` (очистка коллекции), `Contains()` (проверка на наличие элемента в коллекции), `Insert()` (вставка элемента в определенную позицию в коллекции), `Remove()` (удаление элемента из коллекции), `IndexOf()` (определение индекса элемента в коллекции) и `RemoveAt()` (удаление из коллекции элемента с указанным индексом). Аналогичные методы объявлены и в других интерфейсах.

Существуют стандартные классы, реализующие перечисленные выше интерфейсы. Один из них — класс `ArrayList`. Класс `ArrayList` реализует интерфейсы `ICollection`, `IList`, `IEnumerable` и `ICloneable`. Этот класс предназначен для реализации динамических массивов, размер которых может меняться в процессе выполнения программы. Пример создания и использования коллекции на основе класса `ArrayList` представлен в листинге 9.10.

Листинг 9.10. Использование класса ArrayList

```
using System;
using System.Collections;
```

```
// Главный класс:
class ArrayListDemo{
   // Главный метод:
   static void Main(){
      // Целочисленная переменная:
      int n=10;
      // Создание объекта коллекции (без элементов):
      ArrayList fibs=new ArrayList();
      // Добавление элементов в коллекцию:
      fibs.Add(1);
      fibs.Add(1);
      for(int k=2;k<=n;k++){
         // Вычисление нового элемента и добавление
         // в коллекцию:
         fibs.Add((int)fibs[fibs.Count-1]+(int)fibs[fibs.Count-2]);
      }
      // Отображение содержимого коллекции:
      foreach(object obj in fibs){
         Console.Write("|"+obj);
      }
      Console.WriteLine("|");
      // Удаление элементов из коллекции:
      fibs.Remove(1);
      fibs.Remove(5);
      // Вставка элемента в начало коллекции:
      fibs.Insert(0,100);
      // Обратный порядок элементов в коллекции:
      fibs.Reverse();
      // Преобразование коллекции в массив:
      int[] nums=(int[])fibs.ToArray(typeof(int));
      // Отображение содержимого массива:
      for(int k=0;k<nums.Length;k++){
```

```
            Console.Write("|"+nums[k]);
        }
        Console.WriteLine("|");
    }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 9.10)

```
|1|1|2|3|5|8|13|21|34|55|89|
|89|55|34|21|13|8|3|2|1|100|
```

В данном случае для работы с коллекциями мы подключаем пространство имен `System.Collections`. В главном методе программы объект коллекции `fibs` создается командой `ArrayList fibs=new ArrayList()`. Созданный таким образом объект будет пустым (он не содержит ни одного элемента).

НА ЗАМЕТКУ

При создании объекта коллекции в качестве аргумента конструктору класса `ArrayList` можно передать, например, массив. В этом случае будет создана коллекция на основе массива. В любом случае в коллекцию можно добавлять элементы и удалять элементы из коллекции. Размер коллекции при этом регулируется автоматически. Также следует различать размер коллекции и количество элементов в коллекции. Размер коллекции определяется количеством мест в базовом массиве коллекции, предназначенном для хранения элементов. Обычно размер коллекции больше количества содержащихся в ней элементов. Узнать количество элементов в коллекции можно с помощью свойства `Count`. Узнать или задать размер коллекции можно с помощью свойства `Capacity`.

Для добавления элементов в коллекцию используем метод `Add()` (элемент добавляется в конец коллекции). То есть первые два элемента в коллекции — единицы. Затем запускается оператор цикла, и за каждую итерацию цикла в коллекцию добавляется новый элемент. Предварительно этот элемент вычисляется как сумма двух предыдущих элементов. Для этого используется выражение `(int)fibs[fibs.Count-1]+(int)fibs[fibs.Count-2]`.

ПОДРОБНОСТИ

Для получения значения элемента в коллекции можно использовать индекс, как при работе с массивами. Поэтому доступ к первому элементу в коллекции `fibs` получаем с помощью инструкции `fibs[0]` (можно прочитать значение элемента или присвоить значение элементу). Количество элементов в коллекции определяем с помощью свойства `Count`. Поскольку индексация элементов в коллекции, как и в массиве, начинается с нуля, то индекс последнего элемента в коллекции на единицу меньше количества элементов в коллекции. Индекс последнего элемента в коллекции `fibs` вычисляется как `fibs.Count-1`, а индекс предпоследнего элемента вычисляется как `fibs.Count-2`.

При создании коллекции на основе класса `ArrayList` элементы коллекции реализуются на основе класса `Object`. Поэтому, с одной стороны, в коллекцию можно заносить элементы любого типа. С другой стороны, при считывании значений элементов из коллекции приходится выполнять явное приведение типа.

Для отображения элементов из коллекции использован оператор цикла по коллекции `foreach`. При этом мы также учли, что элементы в коллекции сохраняются в виде объектных ссылок класса `Object` (в программе использован псевдоним `object` для выражения `System.Object`).

Для удаления элементов используется метод `Remove()`. Командой `fibs.Remove(1)` из коллекции удаляется элемент со значением `1` (первый из двух), а командой `fibs.Remove(5)` из коллекции удаляется элемент со значением `5`. А вот командой `fibs.Insert(0,100)` в позицию с индексом `0` в коллекции `fibs` добавляется элемент со значением `100`. Командой `fibs.Reverse()` порядок элементов в коллекции `fibs` изменяется на обратный.

Коллекцию можно преобразовать в массив (точнее, создать массив на основе коллекции). Для этого используют метод `ToArray()`. Пример использования этого метода дается командой `int[] nums=(int[]) fibs.ToArray(typeof(int))`, которой на основе коллекции `fibs` создается целочисленный массив `nums`. Но и здесь не все так просто. Метод `ToArray()` результатом возвращает ссылку на массив из элементов класса `Object`. То есть результатом метода является значение типа `Object[]`. Чтобы записать такую ссылку в переменную `nums`, необходимо выполнить явное приведение к типу `int[]`. Аргументом методу `ToArray()` передается выражение `typeof(int)`. Этот аргумент

определяет тип элементов массива, который создается на основе коллекции.

ПОДРОБНОСТИ

Результатом выражения на основе `typeof`-инструкции является объект класса `Type`, который содержит описание типа, идентификатор которого указан в `typeof`-инструкции.

Есть и другие классы, предназначенные для работы с коллекциями. Например, классы `Stack` и `Queue` предназначены для создания *стека* и *очереди* соответственно.

НА ЗАМЕТКУ

Стек — это последовательность элементов, которые обрабатываются по принципу «последний зашел — первый вышел»: первым обрабатывается элемент, добавленный последним в стек. Очередь — это последовательность элементов, которые обрабатываются по принципу «первый зашел — первый вышел»: первым обрабатывается элемент, который добавлялся в очередь первым.

Классы `Stack` и `Queue` реализуют интерфейсы `ICollection`, `IEnumerable` и `ICloneable`. При работе со стеком для добавления элемента в стек используется метод `Push()`. С помощью метода `Pop()` из стека извлекается (удаляется) последний элемент, а значение элемента возвращается результатом метода. При работе с очередью новый элемент в очередь можно добавить с помощью метода `Enqueue()`. Метод `Dequeue()` извлекает (удаляет) первый элемент из очереди и результатом возвращает значение этого элемента. У классов `Stack` и `Queue` также есть метод `Peek()`, который результатом возвращает последний элемент для стека и первый элемент для очереди. При этом элемент из стека/очереди не удаляется.

НА ЗАМЕТКУ

Методы `Peek()`, `Pop()` и `Dequeue()` результатом возвращают объектные ссылки класса `Object`.

Пример использования классов `Stack` и `Queue` представлен в листинге 9.11.

Листинг 9.11. Стек и очередь

```
using System;
using System.Collections;
// Главный класс:
class StackQueueDemo{
   // Метод для отображения содержимого коллекции:
   static void show(ICollection s){
      Console.WriteLine("Содержимое коллекции:");
      foreach(object obj in s){
         Console.Write("|"+obj);
      }
      Console.WriteLine("|");
   }
   // Главный метод:
   static void Main(){
      // Создание стека:
      Stack mystack=new Stack();
      // Добавление элементов в стек:
      mystack.Push("Первый");
      mystack.Push("Второй");
      mystack.Push("Третий");
      // Отображение содержимого стека:
      show(mystack);
      // Считывание элемента из стека:
      Console.WriteLine("Прочитано: "+mystack.Peek());
      // Отображение содержимого стека:
      show(mystack);
      // Извлечение элемента из стека:
      Console.WriteLine("Извлечено: "+mystack.Pop());
      // Отображение содержимого стека:
      show(mystack);
```

```
        // Создание очереди:
        Queue myqueue=new Queue();
        // Добавление элементов в очередь:
        myqueue.Enqueue("Один");
        myqueue.Enqueue("Два");
        myqueue.Enqueue("Три");
        // Отображение содержимого очереди:
        show(myqueue);
        // Считывание элемента из очереди:
        Console.WriteLine("Прочитано: "+myqueue.Peek());
        // Отображение содержимого очереди:
        show(myqueue);
        // Извлечение элемента из очереди:
        Console.WriteLine("Извлечено: "+myqueue.Dequeue());
        // Отображение содержимого очереди:
        show(myqueue);
    }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 9.11)

```
Содержимое коллекции:
|Третий|Второй|Первый|
Прочитано: Третий
Содержимое коллекции:
|Третий|Второй|Первый|
Извлечено: Третий
Содержимое коллекции:
|Второй|Первый|
Содержимое коллекции:
|Один|Два|Три|
```

```
Прочитано: Один
Содержимое коллекции:
|Один|Два|Три|
Извлечено: Один
Содержимое коллекции:
|Два|Три|
```

В программе описан статический метод `show()`, который предназначен для отображения содержимого коллекции, переданной аргументом методу. Аргумент метода описан как относящийся к типу `ICollection`. В этом случае мы учли, что классы `Stack` и `Queue` реализуют интерфейс `ICollection`, и поэтому объекты этих классов могут передаваться методу в качестве аргумента.

В методе `Main()` создаются стек (команда `Stack mystack=new Stack()`) и очередь (команда `Queue myqueue=new Queue()`). В стек и очередь сначала добавляются элементы, после чего отображается содержимое соответствующей коллекции. Затем элемент считывается с помощью метода `Peek()`. При этом элемент из коллекции не удаляется. При извлечении элемента с помощью метода `Pop()` для стека и `Dequeue()` для очереди элемент удаляется из коллекции.

> **НА ЗАМЕТКУ**
>
> При попытке получить элемент из стека или очереди, в которой нет элементов, генерируется исключение класса `InvalidOperationException`.

Существует еще один примечательный тип коллекций, которые иногда называют *ассоциативными*. В такой коллекции доступ к элементу выполняется по *ключу*. Ключ — это некоторое значение (не обязательно целочисленное), которое играет роль индекса. То есть элемент в коллекции сохраняется в виде пары ключ-значение.

Реализовать ассоциативную коллекцию можно, например, на основе класса `Hashtable` или класса `SortedList`. На основе класса `Hashtable` создается контейнер, в котором пары ключ-значение хранятся в виде хэш-таблицы (для кодирования и реализации значений разных типов используется специальный числовой код, или хэш-код). В коллекции, созданной на основе класса `SortedList`, пары ключ-значение

хранятся в упорядоченном виде (упорядочивание выполняется по ключу). Небольшая иллюстрация к использованию классов `Hashtable` и `SortedList` представлена в листинге 9.12.

Листинг 9.12. Ассоциативные коллекции

```
using System;
using System.Collections;
// Главный класс:
class HashTableDemo{
   // Главный метод:
   static void Main(){
      // Создание коллекции:
      Hashtable ht=new Hashtable();
      // Добавление элементов в коллекцию:
      ht.Add("первый",100);
      ht.Add("второй",200);
      ht["третий"]=300;
      // Коллекция индексов:
      ICollection keys=ht.Keys;
      Console.WriteLine("Содержимое коллекции:");
      // Содержимое коллекции:
      foreach(string s in keys){
         Console.WriteLine("{0} -> {1}",s,ht[s]);
      }
      // Создание коллекции:
      SortedList sl=new SortedList();
      // Добавление элементов в коллекцию:
      sl.Add("один",1);
      sl.Add("два",2);
      sl["три"]=3;
      // Коллекция ключей:
      keys=sl.Keys;
```

```
      Console.WriteLine("Содержимое коллекции:");
      // Содержимое коллекции:
      foreach(string s in keys){
         Console.WriteLine("{0} -> {1}",s,sl[s]);
      }
   }
}
```

Ниже показано, каким будет результат выполнения программы:

Результат выполнения программы (из листинга 9.12)

```
Содержимое коллекции:
третий -> 300
первый -> 100
второй -> 200
Содержимое коллекции:
два -> 2
один -> 1
три -> 3
```

В этой программе создается две коллекции: одна на основе класса `Hashtable` (команда `Hashtable ht=new Hashtable()`) и другая на основе класса `SortedList` (команда `SortedList sl=new SortedList()`). Для добавления элементов в коллекции используется метод `Add()`, первым аргументом которому передается ключ элемента, а вторым аргументом методу передается значение элемента. Также можно индексировать объект коллекции, указав в качестве индекса ключ элемента.

У объектов коллекций есть свойства `Keys` и `Values`. Эти свойства результатом возвращают коллекции, соответственно, из ключей и значений элементов. Поэтому для получения коллекции ключей используем выражения `ht.Keys` и `sl.Keys`. Ссылки на коллекции ключей последовательно записываются в переменную `keys`, которая объявлена как относящаяся к интерфейсному типу `ICollection`. Перебирая ключи в коллекции ключей, получаем доступ к значениям элементов.

НА ЗАМЕТКУ

Обратите внимание, что в коллекции, созданной на основе класса `SortedList`, элементы сортируются по значениям ключей. Также стоит заметить, что классы коллекций, в том числе и классы `Hashtable` и `SortedList`, содержат всевозможные методы и свойства, с помощью которых решаются самые различные задачи, связанные с использованием коллекций.

Резюме

— Позвольте узнать, что вы можете сказать по поводу прочитанного?
— Да не согласен я.

из к/ф «Собачье сердце»

- Метод `Show()` из класса `MessageBox` возвращает результат. Этим результатом является одна из констант перечисления `DialogResult`. Значение константы позволяет определить кнопку, которую нажал пользователь в диалоговом окне.
- Для определения области уникальности имен используется пространство имен. Для подключения уже существующего пространства имен используют инструкцию `using`, после которой указывается подключаемое пространство имен. Если пространство имен не подключено, то при использовании классов и других утилит этого пространства перед их именем следует через точку указывать название пространства имен.
- Чтобы описать собственное пространство имен, используют ключевое слово `namespace`, после которого следует блок (выделяется фигурными скобками) с описанием классов, структур, интерфейсов и прочих утилит, которые входят в пространство имен. Описание пространства имен можно разбивать на несколько блоков.
- Инструкцию `using` можно использовать для создания псевдонимов. В этом случае после ключевого слова `using` указывается идентификатор (псевдоним), оператор присваивания и выражение, для которого создается псевдоним.
- Для работы с датой и временем используют структуры `DateTime` (экземпляр структуры реализует дату и время) и `TimeSpan` (экземпляр

структуры реализует время). Эти структуры имеют большое количество свойств и методов, позволяющих получать и обрабатывать информацию, связанную с датой и временем. В частности, статическое свойство Now структуры DateTime результатом возвращает экземпляр этой же структуры, реализующий текущую дату и время.

- Для записи данных в файл и считывания данных из файла используют потоки. Существует несколько типов потоков в зависимости от того, каким образом обрабатываются данные: символьные потоки, байтовые потоки и бинарные потоки. Основные классы, используемые при работе с потоками, относятся к пространству имен System.IO.

- Для создания объекта байтового потока используется класс FileStream. При создании объекта потока аргументом конструктору передается имя файла, связанного с этим потоком. Второй аргумент конструктора определяет режим открытия файла. Для считывания байта из потока может использоваться метод ReadByte(), а для записи байта в поток может использоваться метод WriteByte(). Есть и другие методы, предназначенные для работы с потоками.

- Для создания символьного потока используют классы StreamReader и StreamWriter. Объекты символьных потоков могут создаваться на основе объекта байтового потока, но проще сразу указать имя файла, связанного с символьным потоком. Для обработки данных есть много методов. Среди них можно выделить такие, как ReaddLine() (считывает текстовую строку из файла) и WriteLine() (записывает данные в файл).

- Для создания бинарных потоков используют классы BinryReader и BinaryWriter. Объект бинарного потока можно создать на основе объекта байтового потока (ссылка на этот объект передается аргументом конструктору). Для записи данных в поток можно использовать метод Write(). Для считывания данных из файла используются такие методы, как ReadInt32(), ReadChar(), ReadDouble(), ReadString(), и ряд других.

- Существуют классы для реализации коллекций. Коллекции представляют собой набор или группу объектов. Среди этих классов можно выделить ArrayList, Stack и Queue. Для создания ассоциативных коллекций можно использовать классы Hashtable и SortedList.

Задания для самостоятельной работы

Все построено на силах природы с разрешения месткома и культпросветкомиссии.

из к/ф «Собачье сердце»

1. Напишите программу, в которой генерируется случайное целое число (например, в диапазоне от 1 до 10), а пользователю необходимо его угадать. Если пользователь не угадал число, программа выдает запрос о том, хочет ли он попробовать еще раз. Процесс заканчивается, если пользователь угадывает число или отказывается угадывать снова. Для ввода чисел использовать окно с полем ввода, а для запроса о желании продолжить угадывание использовать окно с кнопками **Да** и **Нет**.

2. Напишите программу, в которой есть производный класс. Класс создается на основе базового класса и реализует интерфейс. Производный класс, базовый класс и интерфейс описываются в разных пространствах имен.

3. Напишите программу, в которой пользователь вводит дату своего рождения, а программа вычисляет, сколько прошло полных лет, месяцев и дней от указанной даты до текущей.

4. Напишите программу, в которой для указанного интервала времени (в годах) определяются годы, первый день которых (1 января) попадает на понедельник.

5. Напишите программу, содержащую статический метод, которому передаются два аргумента, определяющих некоторые даты. Метод сравнивает эти даты на предмет совпадения. Даты считаются совпадающими, если они отличаются не более чем на определенный интервал времени. Интервал времени задается третьим аргументом метода.

6. Напишите программу, в которой считывается содержимое текстового файла и создается новый текстовый файл. В новый текстовый файл заносится текст из исходного текстового файла, но все пробелы заменяются подчеркиваниями, а заглавные буквы заменяются строчными.

7. Напишите программу, в которой описывается класс с тремя полями (целое число, символ и текст). Создайте объект этого класса, прочитав значения для его полей из бинарного файла. Файл предварительно создается программой.

8. Напишите программу, в которой на основе класса `ArrayList` создается коллекция и заполняется текстовыми элементами. Текстовые значения — это слова из текста, записанного в текстовую переменную. Исходите из того, что слова в тексте разделяются пробелами.

9. Напишите программу, содержащую статический метод с аргументом, являющимся целочисленным массивом. Результатом метод возвращает стек, состоящий из тех элементов массива, значение которых не меньше среднего значения по массиву.

10. Напишите программу, в которой создается ассоциативная коллекция. Значения элементов в коллекции — это случайные буквы. Значения ключей определяются так: первый ключ равен `1`, второй ключ равен `2`, а каждый следующий ключ определяется как сумма двух предыдущих.

Заключение
ИТОГИ И ПЕРСПЕКТИВЫ

Опять электричества нет.

из к/ф «Собачье сердце»

В книге мы рассмотрели большинство тем, критически важных при изучении языка C#. Конечно, кое-что осталось за бортом. Но это не должно смущать. Во-первых, объем материала, предназначенного для изучения, и так достаточно большой. Во-вторых, технологии не стоят на месте и средства программирования идут в ногу с прогрессом. Поэтому процесс изучения тонкостей программирования и совершенствования навыков написания программ — многогранный, живой и фактически нескончаемый. Ведь жизнь — это движение. Важно не останавливаться и двигаться в правильном направлении.

Темы, примеры и программы, которые рассматривались в книге, должны заложить основы в понимании базовых, наиболее важных принципов, без которых невозможно эффективное создание программ на языке C#. Здесь уместно вспомнить принцип Парето. Согласно этому принципу, в деятельности любого рода есть 20% усилий, которые дают 80% результата. Нельзя сказать, что работа с материалом книги — это 20% необходимых усилий. Пожалуй, объем проделанной читателем работы намного превышает означенную планку. И даже если какие-то программы или приемы, описанные в книге, оказались не совсем понятными — не страшно. Достаточно понять принципы, на основании которых разные программные механизмы, объекты и конструкции объединяются в одно целое и взаимодействуют друг с другом. Если такое понимание пришло — значит книга свою задачу выполнила и читатель не зря потратил время и усилия.

Закончить хочется где-то банальным, но очень важным советом. Состоит он в том, чтобы систематически и неустанно совершенствовать навыки программирования. Ведь лучший способ научиться программировать — это программировать. Ошибаться, исправлять ошибки, критически анализировать свои и чужие программы, задавать вопросы и искать оригинальные решения. Именно так выглядит путь к успеху.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Научно-популярное издание

РОССИЙСКИЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Васильев Алексей Николаевич

ПРОГРАММИРОВАНИЕ НА C# ДЛЯ НАЧИНАЮЩИХ
Особенности языка

Главный редактор *Р. Фасхутдинов*
Ответственный редактор *Е. Истомина*
Младший редактор *Е. Минина*
Художественный редактор *В. Брагина*

В оформлении обложки использована иллюстрация:
pavel7tymoshenko / Shutterstock.com
Используется по лицензии от Shutterstock.com

ООО «Издательство «Эксмо»
123308, Москва, ул. Зорге, д. 1. Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru
Өндіруші: «ЭКСМО» АҚБ Баспасы, 123308, Мәскеу, Ресей, Зорге көшесі, 1 үй.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru.
Тауар белгісі: «Эксмо»
Интернет-магазин : www.book24.ru
Интернет-дүкен : www.book24.kz
Импортёр в Республику Казахстан ТОО «РДЦ-Алматы».
Қазақстан Республикасындағы импорттаушы «РДЦ-Алматы» ЖШС.
Дистрибьютор и представитель по приему претензий на продукцию,
в Республике Казахстан: ТОО «РДЦ-Алматы»
Қазақстан Республикасында дистрибьютор және өнім бойынша арыз-талаптарды
қабылдаушының өкілі «РДЦ-Алматы» ЖШС,
Алматы қ., Домбровский көш., 3«а», литер Б, офис 1.
Тел.: 8 (727) 251-59-90/91/92; E-mail: RDC-Almaty@eksmo.kz
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта: www.eksmo.ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить на сайте Издательства «Эксмо»
www.eksmo.ru/certification
Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Подписано в печать 26.10.2018. Формат 70x100 1/16.
Печать офсетная. Усл. печ. л. 42,78.
Тираж экз. Заказ

ISBN 978-5-04-092520-9

КОГДА ВЫ ДАРИТЕ КНИГУ, ВЫ ДАРИТЕ ЦЕЛЫЙ МИР

ХОТИТЕ ЗНАТЬ БОЛЬШЕ?

Заходите на сайт:
https://eksmo.ru/b2b/

Звоните по телефону:
+7 495 411-68-59, доб. 2261

ВАШ ЛОГОТИП
НА ОБЛОЖКЕ

ВАШ ЛОГОТИП НА КОРЕШКЕ

ОБРАЩЕНИЕ
К КЛИЕНТАМ
НА ОБЛОЖКЕ

Вторая книга Алексея Васильева, ведущего российского автора самоучителей по программированию, посвященная особенностям языка C#. Продвиньтесь дальше в изучении одного из самых популярных языков семейства C и приступите к самостоятельной разработке проектов на нем. Язык книги прост и доступен для новичка, автор приводит множество примеров и подробно разбирает каждый из них, шаг за шагом приближая читателя к овладению необходимыми для программирования на C# навыками. Изучайте классы и методы, интерфейсы и переменные, пишите собственные программы и закрепляйте пройденный материал!

Дополнительные материалы можно скачать по адресу: **https://eksmo.ru/files/vasilyev_csharp2.zip**

Самое главное:

- Главные особенности и структурные единицы языка C#
- Подробный разбор каждой главы с примерами и выводами
- Все примеры актуальные и могут применяться в работе
- Доступный язык изложения, понятный новичкам
- Использована методика обучения, многократно проверенная на практике

Об авторе

Алексей Николаевич Васильев — доктор физико-математических наук, профессор кафедры теоретической физики физического факультета Киевского национального университета имени Тараса Шевченко. Автор более 15 книг по программированию на языках C++, Java, JavaScript, Python и математическому моделированию.

«Книга подойдет для тех читателей, которые уже знакомы с основами программирования на C# и хотят сделать следующий шаг в освоении этого языка. По каждой теме приводится большое количество примеров и задач с детальным разбором их решений и комментариями автора. Я давно знаком с книгами А. Н. Васильева и считаю, что его методический подход, системность и доступность изложения, прекрасно подходят для самостоятельного изучения языков программирования, и в частности языка C#».

Дмитрий Златопольский,
кандидат технических наук, доцент, автор книги
«Программирование: типовые задачи,
алгоритмы, методы»

ISBN 978-5-04-092520-9