Васильев А.Н.

ПРОГРАММИРОВАНИЕ НА C#

ОСОБЕННОСТИ ЯЗЫКА

РОССИЙСКИЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Москва
2022

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, 2022. — 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

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

Страна происхождения: Российская Федерация
Шығарылған елі: Ресей Федерациясы

ООО «Издательство «Эксмо»
123308, Россия, город Москва, улица Зорге, дом 1, строение 1, этаж 20, каб. 2013.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru
Өндіруші: «ЭКСМО» АҚБ Баспасы,
123308, Ресей, қала Мәскеу, Зорге көшесі, 1 үй, 1 ғимарат, 20 қабат, офис 2013 ж.
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-магазин : www.book24.kz
Интернет-дукен : 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

Дата изготовления / Подписано в печать 21.12.2021.
Формат 70x100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 42,78.
Доп. тираж 2000 экз. Заказ

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
```