13

Дополнительные приемы объектно-ориентированного программирования

В этой главе продолжается изучение языка C# и рассматриваются несколько механизмов и приемов, которые в принципе больше никуда особо не вписывались. Это вовсе не означает, что данные приемы не являются полезными, просто они не подпадают ни под какую из тех тем, которые рассматривались до этого.

В частности, в настоящей главе рассматриваются следующие основные темы.

- Операция `::` и спецификатор глобального пространства имен.
- Специальные исключения и связанные с ними рекомендации.
- События.
- Анонимные методы.

Помимо этого здесь еще будут внесены окончательные изменения в проект `CardLib`, который разрабатывался в последних нескольких главах, и даже показано как его можно использовать для создания карточной игры.

Операция `::` и спецификатор глобального пространства имен

Операция `::` предоставляет альтернативный способ для получения доступа к типам в пространствах имен. Таковой может быть необходим при желании использовать псевдоним пространства имен и существовании неоднозначности между этим псевдонимом и фактической иерархией пространства имен. В таком случае иерархия пространства получает преимущество над псевдонимом. Чтобы увидеть, что это означает, рассмотрим следующий код:

```
using MyNamespaceAlias = MyRootNamespace.MyNestedNamespace;
namespace MyRootNamespace
{
  namespace MyNamespaceAlias
  {
    public class MyClass
    {
    }
  }
  namespace MyNestedNamespace
  {
    public class MyClass
    {
    }
  }
}
```

В коде в `MyRootNamespace` для ссылки на класс мог бы применяться такой синтаксис:

```
MyNamespaceAlias.MyClass
```

Классом, на который ссылается данный код, является `MyRootNamespace.MyNamespaceAlias.MyClass`, а не `MyRootNamespace.MyNestedNamespace.MyClass`. То есть пространство имен `MyRootNamespace.MyNamespaceAlias` скрывает псевдоним, определенный в операторе `using` и ссылающийся на `MyRootNamespace.MyNestedNamespace`. Получать доступ к пространству имен `MyRootNamespace.MyNestedNamespace` и содержащемуся внутри него классу по-прежнему можно, но для этого требуется использовать другой синтаксис:

```
MyNestedNamespace.MyClass
```

В качестве альтернативного варианта можно применять операцию `::`, как показано ниже:

```
MyNamespaceAlias::MyClass
```

Применение этой операции вынудит компилятор использовать псевдоним, определенный в операторе `using` и тогда код станет ссылаться на `MyRootNamespace.MyNestedNamespace.MyClass`.

Вместе с операцией `::` еще можно использовать и ключевое слово `global`, которое превращает ее, по сути, в псевдоним, ссылающийся на корневое пространство имен наивысшего уровня. Делать подобное может быть удобно для того, чтобы было еще яснее, о каком именно пространстве имен идет речь:

```
global::System.Collections.Generic.List<int>
```

Здесь конечным классом является именно тот, который и требовался — обобщенный класс коллекции `List<T>`. Но классом, определенным с помощью приведенного ниже кода, он точно не является:

```
namespace MyRootNamespace
{
  namespace System
  {
    namespace Collections
    {
    namespace Generic
    {
      class List<T>
      {
      }
    }
  }
}
```

Разумеется, нужно стараться не присваивать собственным пространствам имен такие имена, которые уже есть у пространств имен .NET, но такая проблема все равно может возникать в больших проектах, особенно при работе над ними в составе очень многочисленной команды. В подобных случаях применение операции :: и ключевого слова global может быть единственным способом получения доступа к требуемым типам.

Специальные исключения

В главе 7 рассказывалось об исключения и объяснялось, как для работы с ними можно использовать блоки try...catch...finally. Еще там говорилось о нескольких стандартных исключениях .NET, а также базовом классе для всех исключений — System.Exception. Однако иногда вместо стандартных исключений бывает удобнее использовать в приложениях свои собственные классы исключений, наследуя от базового класса. Такой подход позволяет более точно задавать информацию, которая должна передаваться отвечающему за перехват исключений коду, и те исключения, которые он должен обрабатывать. Например, он позволяет добавлять в класс исключения новое свойство, обеспечивающее доступ к какой-то базовой информации, и тем самым предоставлять получателю исключения возможность вносить необходимые изменения или просто получать больше информации о причине возникновения данного исключения.

После определения класса исключения его можно добавить в список распознаваемых VS исключений, выбрав в меню Debug (Отладка) пункт Exceptions (Исключения) и выполнив в появившемся после этого окне щелчок на кнопке Add (Добавить), и затем конфигурировать его поведение так, как показывалось в главе 7.

Базовые классы исключений

Хотя классы специальных исключений и можно наследовать от базового класса System.Exception, как описывалось в предыдущем разделе, все-таки согласно наилучшим практическим методикам делать это не рекомендуется. Вместо этого лучше наследовать их от класса System.ApplicationException.

В пространстве имен System существуют два фундаментальных и унаследованных от Exception класса исключений — ApplicationException и SystemException. Класс SystemException используется в качестве базового класса для предопределенных исключений, которые предлагаются в .NET Framework. Класс ApplicationException предназначен специально для того, чтобы разработчики могли наследовать от него свои собственные классы исключений. Если вы будете следовать этой модели, то всегда сможете отличать предопределенные и специальные исключения при перехвате исключений, унаследованных от одного из этих двух классов.

Ни класс ApplicationException, ни класс SystemException никоим образом не расширяют функциональные возможности класса Exception. Они существуют исключительно для того, чтобы позволять разработчиками различать исключения описанным выше образом.

Добавление специальных исключений в CardLib

Легче всего проиллюстрировать способы применения специальных исключений, опять-таки, на примере модернизации проекта CardLib. В настоящее время метод Deck.GetCard() в этом проекте предусматривает генерацию стандартного исключения .NET при попытке получения доступа к карте с индексом меньше 0 и больше 51; предлагаем изменить его теперь так, чтобы в нем использовалось специальное исключение.

Чтобы сделать это, первым делом создайте новый проект типа библиотеки классов по имени Ch13CardLib и, как и раньше, скопируйте в него классы из Ch12CardLib, заменив, где требуется, пространство имен Ch12CardLib на Ch13CardLib. Далее определите исключение с помощью нового класса внутри нового файла CardOutOfRangeException.cs, который можете добавить в проект Ch13CardLib, выбрав в меню Project (Проект) пункт Add Class (Добавить класс), как показано ниже:

```
public class CardOutOfRangeException : ApplicationException
{
  private Cards deckContents;
  public Cards DeckContents
  {
    get
    {
      return deckContents;
    }
  }
  public CardOutOfRangeException(Cards sourceDeckContents) :
      base("There are only 52 cards in the deck.")
           // В колоде имеется лишь 52 карты
  {
    deckContents = sourceDeckContents;
  }
}
```

Экземпляр класса Cards является обязательным для конструктора данного класса. Он обеспечивает доступ к данному объекту Cards через свойство DeckContents и предоставляет подходящее сообщение об ошибке конструктору базового класса Exception, чтобы оно было доступно через его свойство Message.

Далее добавьте код для генерации специального исключения в Deck.cs (заменив им прежний код, который предназначался для выдачи стандартного исключения):

```
public Card GetCard(int cardNum)
{
   if (cardNum >= 0 && cardNum <= 51)
     return cards[cardNum];
   else
     throw new CardOutOfRangeException(cards.Clone() as Cards);
}
```

Свойство DeckContents инициализируется после выполнения глубокого копирования текущего содержимого объекта Deck, которое имеет вид объекта Cards. Это дает возможность видеть, каким было это содержимое на момент генерации исключения, чтобы при последующем внесении изменений в содержимое колоды эта информация "не утрачивалась".

Протестировать все это можно с помощью следующего клиентского кода (который также доступен и загружаемом коде для этого главы, в проекте Ch13CardClient):

```
Deck deck1 = new Deck();
try
{
  Card myCard = deck1.GetCard(60);
}
catch (CardOutOfRangeException e)
{
  Console.WriteLine(e.Message);
  Console.WriteLine(e.DeckContents[0]);
}
Console.ReadKey();
```

Выполнение этого код приведет к получению вывода, показанного на рис. 13.1.

Рис. 13.1. Генерация специального исключения в проекте `CardLib`

Здесь перехватывающий код вывел на экран значение свойства `Message` объекта исключения. Еще была отображена первая карта в объекте `Cards`, полученном через свойство `DeckContents`, просто для подтверждения того, что к коллекции `Cards` можно получать доступ через объект, представляющий специальное исключение.

События

В этом разделе речь пойдет об одном из наиболее часто используемых механизмов ООП в .NET, а именно — о *событиях* (events). Сначала, как обычно, будет приведен краткий обзор того, что собой представляют события. Далее будут показаны некоторые простые события в действии и рассказано о том, что с ними можно делать. Напоследок вы узнаете о том, как создавать и использовать свои собственные события.

В самом конце главы будет предоставлена возможность завершить разработку проекта библиотеки классов `CardLib` путем добавления в него события, а также, в качестве последнего этапа перед переходом к рассмотрению более сложных тем, немного поразвлечься и создать использующее эту библиотеку приложение для игры в карты.

Что собой представляет событие

События похожи на исключения тем, что они тоже *генерируются*, т.е. выдаются объектами, и тем, что для них тоже можно предоставлять реагирующий на них выполнением какого-нибудь действия код. Однако существует и несколько отличий, наиболее важное из которых состоит в отсутствии для обработки событий структуры, эквивалентной `try...catch`. Вместо применения этой структуры на события нужно *подписываться* (subscribe). Под подпиской на событие подразумевается предоставление кода, который должен выполняться при генерации данного события, в виде *обработчика событий* (event handler).

На событие можно подписывать несколько обработчиков, которые тогда все будут вызываться при генерации этого события. Эти обработчики могут являться как частью того класса объекта, который генерирует данное событие, так и частью других классов.

Сами обработчики событий представляют собой просто функции. Единственным ограничением для такой функции является то, что ее возвращаемый тип и параметры должны обязательно соответствовать тем, которых требует событие. Это ограничение входит в состав определения события и задается *делегатом*.

> *Тот факт, что делегаты используются в событиях, как раз и делает их такими полезными. Именно поэтому им и было уделено некоторое внимание в главе 6. При необходимости можете перечитать приведенный там материал.*

Базовая последовательность обработки выглядит следующим образом: сначала приложение создает объект, который может генерировать событие. Например, рассмотрим приложение для мгновенного обмена сообщениями. Оно могло бы создавать объект, представляющий соединение с удаленным пользователем, и тогда этот объект мог бы генерировать событие при поступлении сообщения от удаленного пользователя по этому соединению (рис. 13.2).

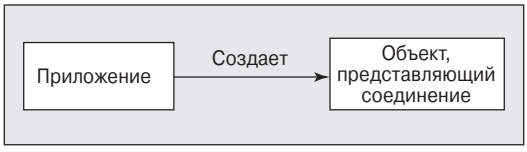

Рис. 13.2. Объект, генерирующий события

Далее приложение подписывается на событие. Приложение для мгновенного обмена сообщениями могло бы делать это путем определения функции, пригодной для использования с указанным в событии типом делегата, и передачи ссылки на эту функцию событию. Эта функция-обработчик событий могла бы представлять собой метод в другом объекте, например, объекте дисплея, и отображать мгновенные сообщения на экране по мере их поступления (рис. 13.3).

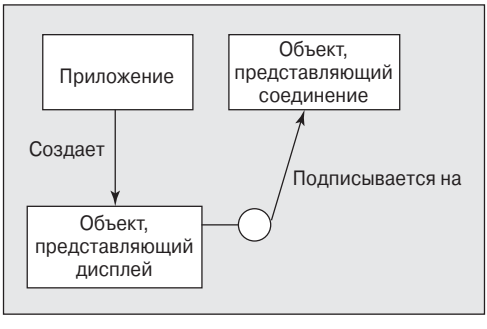

Рис. 13.3. Подписка на событие

И, наконец, последнее: при генерации события подписчику отправляется соответствующее уведомление. Во взятом для примера приложении это означает, что при поступлении мгновенного сообщения через объект соединения в объекте, представляющем дисплей, вызывался бы метод обработчика событий. Поскольку используется стандартный метод, объект, генерирующий событие, может передавать любую имеющую отношение к делу информацию через параметры и тем самым делать события очень разнообразными. В случае взятого в качестве примера приложения одним из таких параметров мог бы быть текст мгновенного сообщения, который обработчик событий тогда бы мог отображать в объекте, представляющем дисплей, как показано на рис. 13.4.

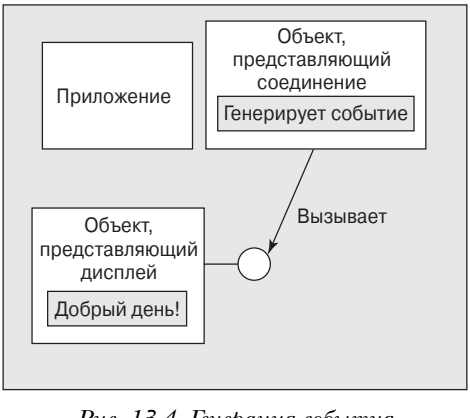

Рис. 13.4. Генерация события

Обработка событий

Как уже рассказывалось выше, для обработки события на него нужно подписываться, предоставляя функцию — обработчик событий, возвращаемый тип и параметры которой должны совпадать с возвращаемым типом и параметрами делегата, закрепленного для применения с этим событием. В следующем практическом занятии демонстрируется пример, в котором простой объект таймера используется для генерации событий, что приводит к вызову функции-обработчика.

Практическое занятие Обработка событий

1. Создайте новое консольное приложение по имени `Ch13Ex01` и сохраните его в каталоге `C:\BegVCSharp\Chapter13`.
2. Измените код в его файле `Program.cs` следующим образом:

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

namespace Ch13Ex01
{
  class Program
  {
    static int counter = 0;
    static string displayString =
                "This string will appear one letter at a time. ";
                // Строка, которая будет отображаться по одной букве за раз
    static void Main(string[] args)
    {
      Timer myTimer = new Timer(100);
      myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
      myTimer.Start();
      Console.ReadKey();
    }
    static void WriteChar(object source, ElapsedEventArgs e)
    {
      Console.Write(displayString[counter++ % displayString.Length]);
    }
  }
}
```

3. Запустите приложение (после запуска нажатие любой клавиши приводит к завершению приложения). Через некоторое время на экране появится результат, показанный на рис. 13.5.

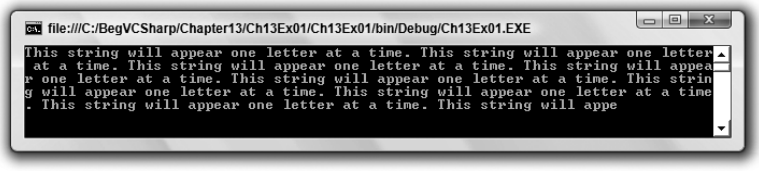

Рис. 13.5. Приложение `Ch13Ex01` в действии

Описание полученных результатов

Объект, который используется для генерации событий, представляет собой экземпляр класса `System.Timers.Timer`. Инициализируется он с указанием временного интервала (в миллисекундах). Когда он запускается с помощью метода `Start()`, начинают генерироваться события через соответствующие заданному промежутки времени. В `Main()` объект `Timer` инициализируется с временным промежутком, составляющим 100 миллисекунд, а это означает, что при запуске он будет генерировать события 10 раз в секунду:

```
static void Main(string[] args)
{
  Timer myTimer = new Timer(100);
```

Объект `Timer` инициирует событие `Elapsed`, и обработчик для этого события должен обязательно иметь такой же возвращаемый тип и параметры, как тип-делегат `System.Timers.ElapsedEventHandler`, который является одним из стандартных делегатов, поставляемых с .NET Framework. Возвращаемый тип и параметры этого делегата выглядят следующим образом:

```
void functionName(object source, ElapsedEventArgs e);
```

Объект `Timer` отправляет ссылку на самого себя в первом параметре и экземпляр объекта `ElapsedEventArgs` во втором параметре. Эти параметры вполне безопасно проигнорировать; более подробно они будут рассматриваться позже в главе.

Для этого в коде присутствует подходящий метод:

```
static void WriteChar(object source, ElapsedEventArgs e)
{
  Console.Write(displayString[counter++ % displayString.Length]);
}
```

Этот метод использует два статических поля `Class1` — `counter` и `displayString` — для отображения одного символа. При каждом вызове этого метода отображается очередной символ.

Следующим шагом является подключение этого обработчика к событию, т.е. подписку на него. Выполняется оно путем применения операции += для добавления обработчика в событие в виде нового экземпляра делегата, инициализируемого с методом обработки событий `WriteChar`:

```
static void Main(string[] args)
{
  Timer myTimer = new Timer(100);
  myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
```

Данная команда (в которой используется немного странно выглядящий синтаксис, являющийся специфическим синтаксисом делегатов) добавляет обработчик в список, который будет вызываться при генерации события `Elapsed`. В этот список можно добавлять сколько угодно обработчиков, главное, чтобы все они отвечали требуемым критериям. Каждый из этих обработчиков будет вызываться при генерации события по очереди.

Все, что остается для функции `Main()` — это запуск таймера:

```
myTimer.Start();
```

Нельзя, чтобы работа приложения завершалась до того, как будут обработаны те или иные события, поэтому функция `Main()` помещается в режим ожидания. Простейшим способом для этого является запрос ввода пользователя, поскольку такая

команда не будет завершать процесс обработки до тех пор, пока пользователь не нажмет какую-нибудь клавишу:

```
Console.ReadKey();
```

Хотя процесс обработки в `Main()` на этом этапе фактически прекращен, процесс обработки в объекте `Timer` продолжается. Когда он генерирует события, он вызывает метод `WriteChar()`, который работает параллельно с оператором `Console.ReadLine()`.

Определение событий

Теперь пришла пора научиться определять и использовать свои собственные события. В следующем практическом занятии демонстрируется пример реализации сценария мгновенного обмена сообщениями, приведенного ранее в этой главе, с созданием объекта `Connection`, генерирующего события, за обработку которых отвечает объект `Display`.

Практическое занятие Определение событий

1. Создайте новое консольное приложение по имени `Ch13Ex02` и сохраните его в каталоге `C:\BegVCSharp\Chapter13`.
2. Добавьте в него новый класс `Connection` и измените код в файле `Connection.cs` следующим образом:

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

namespace Ch13Ex02
{
  public delegate void MessageHandler(string messageText);
  public class Connection
  {
    public event MessageHandler MessageArrived;
    private Timer pollTimer;
    public Connection()
    {
      pollTimer = new Timer(100);
      pollTimer.Elapsed += new ElapsedEventHandler(CheckForMessage);
    }
    public void Connect()
    {
      pollTimer.Start();
    }
    public void Disconnect()
    {
      pollTimer.Stop();
    }
    private static Random random = new Random();
    private void CheckForMessage(object source, ElapsedEventArgs e)
    {
      Console.WriteLine("Checking for new messages.");
                        // Проверка поступления новых сообщений
```

```
      if ((random.Next(9) == 0) && (MessageArrived != null))
      {
        MessageArrived("Hello Mum!");
      }
     }
   }
}
```

3. Добавьте новый класс `Display` и измените его код в файле `Display.cs` следующим образом:

```
namespace Ch13Ex02
{
   public class Display
   {
      public void DisplayMessage(string message)
      {
         Console.WriteLine("Message arrived: {0}", message);
                          // Поступило новое сообщение
      }
   }
}
```

4. Измените код в файле `Program.cs` показанным ниже образом:

```
static void Main(string[] args)
{
   Connection myConnection = new Connection();
   Display myDisplay = new Display();
   myConnection.MessageArrived +=
      new MessageHandler (myDisplay.DisplayMessage);
   myConnection.Connect();
   Console.ReadKey();
}
```

5. Запустите приложение. На рис. 13.6 показан результат, который должен получиться.

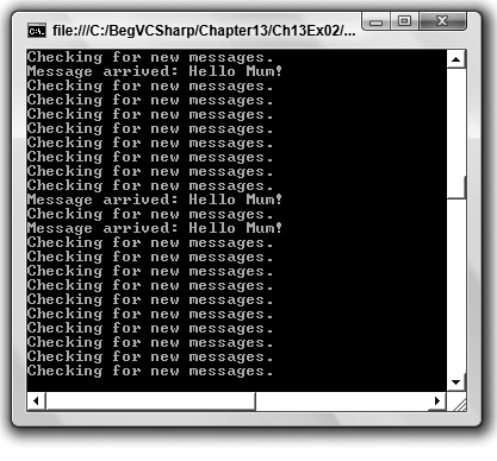

Рис. 13.6. Приложение `Ch13Ex02` в действии

Описание полученных результатов

Большую часть работы в этом приложении выполняет класс `Connection`. Экземпляры этого класса применяют объект `Timer`, во многом подобный тому, что показывался в первом примере этой главы, инициализируя его в конструкторе класса и предоставляя доступ к его состоянию (информации о том, включен он или выключен) через методы `Connect()` и `Disconnect()`:

```
public class Connection
{
  private Timer pollTimer;
  public Connection()
  {
    pollTimer = new Timer(100);
    pollTimer.Elapsed += new ElapsedEventHandler(CheckForMessage);
  }
  public void Connect()
  {
    pollTimer.Start();
  }
  public void Disconnect()
  {
    pollTimer.Stop();
  }
  ...
}
```

Еще в конструкторе регистрируется обработчик для события `Elapsed`, точно так же, как это делалось в первом примере. Метод этого обработчика — `CheckForMessage()` — предусматривает генерацию события в среднем один раз для каждых 10 раз его вызова. Его код еще будет поясняться, но сначала не помешает рассмотреть определение самого события.

Перед определением события требуется обязательно определить подлежащий использованию с ним тип делегата, т.е. тип делегата, возвращаемому типу и параметрам которого должен соответствовать метод обработки событий. Для выполнения этого используется стандартный синтаксис делегатов, с помощью которого необходимый делегат определяется как общедоступный (`pulbic`) внутри пространства имен `Ch13Ex02`, чтобы он был доступен внешнему коду, как показано ниже:

```
namespace Ch13Ex02
{
  public delegate void MessageHandler(string messageText);
```

Данный делегат, имеющий здесь имя `MessageHandler`, представляет собой функцию `void` с одним единственным параметром `string`. Этот параметр можно использовать для передачи мгновенного сообщения, получаемого объектом `Connection`, объекту `Display`. После определения делегата (или установления месторасположения подходящего существующего делегата) можно переходить к определению самого события, в виде члена класса `Connection`:

```
public class Connection
{
  public event MessageHandler MessageArrived;
```

Делается это просто выбором имени для события (каковым здесь является `MessageAarived`) и его объявлением с использованием ключевого слова `event` и указанием применяемого с ним типа-делегата (в данном случае — определенного ранее

типа-делегата MessageHandler). После объявления события подобным образом, его можно генерировать обращением к нему по имени так, будто бы оно является методом с возвращаемым типом и параметрами, заданными в делегате. Например, данное событие можно было бы сгенерировать следующим образом:

```
MessageArrived("This is a message.");
```

Если бы делегат был определен безо всяких параметров, тогда это можно было бы сделать так:

```
MessageArrived();
```

В качестве альтернативного варианта, делегат мог еще быть определен и с бо́льшим числом параметров, что тогда бы потребовало написания большего количества кода для генерации события. Что касается метода CheckForMessage(), то его код выглядит так:

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

private void CheckForMessage(object source, ElapsedEventArgs e)
{
  Console.WriteLine("Checking for new messages.");
  if ((random.Next(9) == 0) && (MessageArrived != null))
  {
     MessageArrived("Hello Mum!");
  }
}
```

Здесь применяется уже демонстрировавшийся в предыдущих главах класс Random для генерации случайного числа в диапазоне от 0 до 9 и, когда сгенерированным числом оказывается 0, что должно происходить в 10 процентах случаев, генерируется событие. Это имитирует опрос подключения для выяснения того, поступило ли новое сообщения, чего при каждой проверке происходить не будет. Для отделения таймера от экземпляра Connection используется приватный статический экземпляр класса Random.

Обратите внимание на предоставляемую дополнительную логику, согласно которой событие должно генерироваться только в случае, если выражение MessageArrived != null в результате дает true. Это выражение, в котором снова используется несколько необычный синтаксис делегатов, по сути, задает следующий вопрос: “Имеются ли у события какие-то подписчики?”. Если подписчиков нет, тогда MessageArrived вычисляется в null, и в генерации события нет никакого смысла.

Класс, который будет подписываться на событие, называется Display и содержит единственный метод DisplayMessage(), определенный следующим образом:

```
public class Display
{
  public void DisplayMessage(string message)
  {
     Console.WriteLine("Message arrived: {0}", message);
  }
}
```

Этот метод соответствует типу делегата (и является общедоступным, что представляет собой требование для всех обработчиков событий, которые используются в классах, отличных от класса, генерирующего событие), поэтому его можно использовать для реагирования на событие MessageArrived.

Теперь осталось только сделать так, чтобы код в Main() инициализировал экземпляры классов Connection и Display, подключал их и запускал весь процесс. Требуемый для этого код похож на тот, что был в первом примере:

```
static void Main(string[] args)
{
  Connection myConnection = new Connection();
  Display myDisplay = new Display();
  myConnection.MessageArrived += new MessageHandler(myDisplay.DisplayMessage);
  myConnection.Connect();
  Console.ReadKey();
}
```

Здесь после запуска процесса с помощью метода `Connect()` объекта `Connection` вызывается метод `Console.ReadKey()` для приостановки процесса обработки в `Main()`.

Универсальные обработчики событий

Делегат, который демонстрировался ранее для события `Timer.Elapsed`, содержал два параметра, которые являются параметрами очень часто встречающегося в обработчиках событий типа, а именно:

- `object source` — ссылка на объект, который сгенерировал событие;
- `ElapsedEventArgs e` — параметры, отправленные событием.

Причина, по которой тип `object` используется в данном событии и на самом деле во многих других событиях, состоит в том, что очень часто требуется использовать один обработчик событий для нескольких идентичных событий, генерируемых разными объектами, и при этом все равно знать, какой конкретно объект сгенерировал данное событие.

Чтобы лучше объяснить и проиллюстрировать это, в следующем практическом занятии предлагается немного расширить предыдущий пример.

Практическое занятие

Применение универсального обработчика событий

1. Создайте новое консольное приложение по имени `Ch13Ex03` и сохраните его в каталоге `C:\BegVCSharp\Chapter13`.
2. Скопируйте код из файлов `Program.cs`, `Connection.cs` и `Display.cs` проекта `Ch13Ex02` и замените названия пространств имен в каждом из них с `Ch13Ex02` на `Ch13Ex03`.
3. Добавьте новый класс `MessageArrivedEventArgs` и измените код в представляющем его файле `MessageArrivedEventArgs.cs` следующим образом:

```
namespace Ch13Ex03
{
  public class MessageArrivedEventArgs : EventArgs
  {
    private string message;
    public string Message
    {
      get
      {
        return message;
      }
    }
```

```
    public MessageArrivedEventArgs()
    {
      message = "No message sent.";
              // Сообщения не отправлялись
    }
    public MessageArrivedEventArgs(string newMessage)
    {
      message = newMessage;
    }
  }
}
```

4. Измените код в файле `Connection.cs` следующим образом:

```
namespace Ch13Ex03
{
  public delegate void MessageHandler(Connection source,
                                      MessageArrivedEventArgs e);
  public class Connection
  {
    public event MessageHandler MessageArrived;
    private string name;
    public string Name
    {
      get
      {
        return name;
      }
      set
      {
        name = value;
      }
    }
    ...
    private void CheckForMessage(object source, EventArgs e)
    {
      Console.WriteLine("Checking for new messages.");
                        // Проверка поступления новых сообщений
      if ((random.Next(9) == 0) && (MessageArrived != null))
      {
        MessageArrived(this, new MessageArrivedEventArgs("Hello Mum!"));
      }
    }
    ...
  }
}
```

5. Измените содержимое файла `Display.cs`, как показано ниже:

```
public void DisplayMessage(Connection source, MessageArrivedEventArgs e)
{
  Console.WriteLine("Message arrived from: {0}", source.Name);
                    // Поступило новое сообщение
  Console.WriteLine("Message Text: {0}", e.Message);
                    // Вывод текста сообщения
}
```

6. В завершение измените код в файле `Program.cs` следующим образом:

```
static void Main(string[] args)
{
  Connection myConnection1 = new Connection();
  myConnection1.Name = "First connection.";
                       // Первое подключение
  Connection myConnection2 = new Connection();
  myConnection2.Name = "Second connection.";
                       // Второе подключение
  Display myDisplay = new Display();
  myConnection1.MessageArrived += new MessageHandler(myDisplay.DisplayMessage);
  myConnection2.MessageArrived += new MessageHandler(myDisplay.DisplayMessage);
  myConnection1.Connect();
  myConnection2.Connect();
  Console.ReadKey();
}
```

7. Запустите приложение. На рис. 13.7 показан результат, который должен получиться.

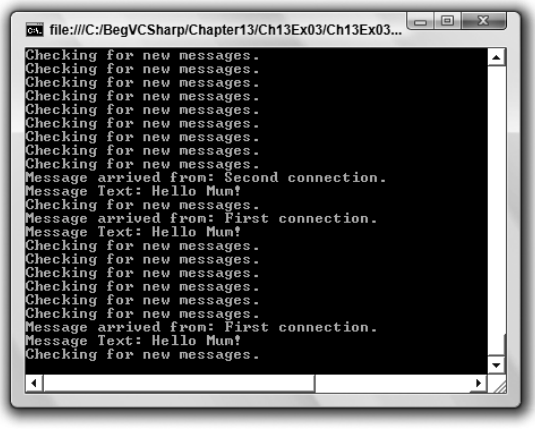

Рис. 13.7. Приложение `Ch13Ex03` в действии

Описание полученных результатов

Путем отправки ссылки на генерирующий событие объект в виде одного из параметров обработчика событий можно настраивать ответ этого обработчика отдельным объектам. Эта ссылка открывает доступ к исходному объекту и его свойствам включительно.

Отправкой параметров, содержащихся в классе, который наследуется от `System.EventArgs` (как это делает класс `ElapsedEventArgs`), можно предоставлять любую необходимую информацию (вроде параметра `Message` в классе `MessageArrivedEventArgs`).

Вдобавок эти параметры будут выигрывать от полиморфизма. Например, обработчик для события `MessageArrived` можно было бы определить так:

```
public void DisplayMessage(object source, EventArgs e)
{
  Console.WriteLine("Message arrived from: {0}",
                    ((Connection)source).Name);
  Console.WriteLine("Message Text: {0}",
                    ((MessageArrivedEventArgs)e).Message);
}
```

Тогда определение делегата в Connection.cs можно было бы изменить следующим образом:

```
public delegate void MessageHandler(object source, EventArgs e);
```

Данное приложение работало бы в точности так же, как и раньше, но его функция DisplayMessage() стала бы более разносторонней (по крайней мере, теоретически, поскольку на практике для достижения более высокого качества потребовалось бы больше реализации). Тот же самый обработчик смог бы работать и с другими событиями, такими как Timer.Elapsed, хотя пришлось бы изменить внутренние детали обработчика еще немного, чтобы параметры, отправляемые при генерации данного события, обрабатывались надлежащим образом (приведение их к типу объектов Connection и MessageArrivedEventArgs показанным способом приведет к генерации исключения; вместо этого должна использоваться операция as и выполняться проверка на предмет наличия в них значения null).

Возвращаемые значение и обработчики событий

У всех демонстрировавшихся до сих пор обработчиков событий возвращаемым типом был void. Предоставлять для события возвращаемый тип в принципе допускается, но это чревато возникновением проблем, поскольку каждое взятое событие может приводить к вызову нескольких обработчиков. Если все они возвращают значение, тогда может быть не ясно, какое именно значение было возвращено.

Система справляется с этой проблемой, позволяя получать доступ только к последнему значению, которое было возвращено обработчиком событий. Таковым всегда будет значение, возвращенное обработчиком, который последним подписался на данное событие. Хотя эта функциональность и может оказаться полезной в некоторых ситуациях, все-таки рекомендуется применять обработчики событий с возвращаемым типом void и избегать параметров out.

Анонимные методы

Вместо того чтобы определять методы обработки событий, можно выбирать и другой вариант, а именно — использовать *анонимные методы*. Анонимным методом называется такой, который фактически не существует как метод в традиционном смысле, т.е. не является методом какого-либо определенного класса. Вместо этого анонимный метод создается исключительно для применения в качестве целевого метода для делегата.

Синтаксис, необходимый для создания анонимного метода, выглядит следующим образом:

```
delegate(параметры)
{
  // Код анонимного метода.
};
```

На месте *параметры* указывается перечень параметров, соответствующих параметрам делегата, экземпляр которого создается, в том виде, в котором они должны использоваться в коде анонимного метода:

```
delegate(Connection source, MessageArrivedEventArgs e)
{
  // Код анонимного метода, соответствующего событию MessageHandler в Ch13Ex03.
};
```

Например, с помощью следующего кода можно было бы полностью пропустить метод Display.DisplayMessage() в Ch13Ex03:

```
myConnection1.MessageArrived +=
   delegate(Connection source, MessageArrivedEventArgs e)
{
  Console.WriteLine("Message arrived from: {0}", source.Name);
  Console.WriteLine("Message Text: {0}", e.Message);
};
```

Интересное свойство анонимных методов состоит в том, что они, по сути, являются локальными для того блока кода, в котором содержатся, и имеют доступ ко всем локальным переменным в этой области действия. В случае использования такой переменной она становится *внешней* (outer). Внешние переменные не уничтожаются при выходе за пределы области действия, как это происходит с другими локальными переменными; вместо этого они продолжают существовать до тех пор, пока не будут уничтожены методы, в которых они используются, что может случаться позже, чем ожидается, и потому обязательно требует внимательного отношения.

Расширение и использование `CardLib`

Теперь, когда было показано, как определять и использовать события, пришла пора попробовать применить полученные знания на практике в проекте `Ch13CardLib`. Событие, которое предлагается добавить, будет генерироваться при получении последнего объекта `Card` в объекте `Deck` с использованием `GetCard` и называться `LastCardDrawn` (Последняя вытянутая карта). Оно будет позволять подписчикам перемешивать колоду автоматически и тем самым сокращать объем подлежащей выполнению клиентом обработки. Делегат, определяемый для этого события (`LastCardDrawnHandler`), должен поставлять ссылку на объект `Deck` так, чтобы метод `Shuffle()` был доступен отовсюду, где бы ни находился обработчик. Это требует добавления в `Deck.cs` следующего кода:

```
namespace Ch13CardLib
{
  public delegate void LastCardDrawnHandler(Deck currentDeck);
```

Код для определения описанного события и его генерации выглядит просто:

```
public event LastCardDrawnHandler LastCardDrawn;
public Card GetCard(int cardNum)
{
  if (cardNum >= 0 && cardNum <= 51)
  {
    if ((cardNum == 51) && (LastCardDrawn != null))
      LastCardDrawn(this);
    return cards[cardNum];
  }
  else
    throw new CardOutOfRangeException((Cards)cards.Clone());
}
```

Это и есть весь код, необходимый для добавления события в определение класса `Deck`.

Клиентское приложение для игры в карты, использующее `CardLib`

Потратив столько времени на разработку библиотеки `CardLib`, было бы нелепо не воспользоваться ею. Поэтому напоследок в этом разделе, посвященном дополнительным механизмам ООП в C# и .NET Framework, предлагается немного развлечься

и написать базовый код клиентского приложения для игры в карты, основанного на использовании уже знакомых классов игральных карт.

Как и в предыдущих главах, сначала необходимо добавить в решение Ch13CardLib клиентское консольное приложение, назвать его Ch13CardClient, добавить в него ссылку на проект Ch13CardLib и сделать его стартовым проектом.

Потом нужно создать в нем новый класс по имени Player, размещаемый в новом файле Player.cs. Этот класс должен содержать приватное поле типа Cards по имени Hand, приватное строковое поле с именем name и два доступных только для чтения свойства Name и PlayHand. Эти свойства будут просто предоставлять доступ к соответствующим приватным полям. Хотя свойство PlayHand и должно быть доступным только для чтения, к возвращаемой им ссылке на поле hand доступ должен быть не только для чтения, но и для записи, чтобы с ее помощью можно было изменять имеющиеся на руках у игрока карты.

Еще необходимо скрыть конструктор по умолчанию, сделав его приватным, и предоставить общедоступный конструктор не по умолчанию, принимающий в качестве параметра первоначальное значение для свойства Name экземпляров Player.

И, наконец, последнее, что потребуется сделать — это предоставить метод типа bool с именем HasWon(), возвращающий значение true только в том случае, если все карты на руках игрока относятся к одной масти (что является простым условием для победы). Ниже приведен код, который нужно добавить в Player.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch13CardLib;

namespace Ch13CardClient
{
  public class Player
  {
    private Cards hand;
    private string name;
    public string Name
    {
      get
    {
      return name;
    }
  }
  public Cards PlayHand
  {
    get
      {
        return hand;
      }
    }
    private Player()
    {
    }
    public Player(string newName)
    {
      name = newName;
      hand = new Cards();
    }
```

```
    public bool HasWon()
    {
      bool won = true;
      Suit match = hand[0].suit;
      for (int i = 1; i < hand.Count; i++)
    {
      won & = hand[i].suit == match;
    }
    return won;
    }
  }
}
```

Далее необходимо определить класс, который будет отвечать за саму карточную игру. Этот класс называется `Game` и должен содержаться в проекте `Ch13CardClient` внутри файла `Game.cs`. У него должно быть четыре следующих приватных поля:

- `playDeck` — переменная типа `Deck`, содержащая используемую колоду карт;
- `currentCard` — значение типа `int`, применяемое в качестве указателя на следующую карту в используемой колоде;
- `players` — массив объектов `Player`, представляющих игроков игры;
- `discardedCards` — коллекция `Cards`, предназначенная для тех карт, которые были отброшены игроками, но не были перетасованы и помещены обратно в колоду.

Стандартный конструктор этого класса должен выполнять следующие действия: инициализировать и перетасовывать объект `Deck`, хранящийся в `playDeck`, устанавливать для переменной-указателя `currentCard` значение `0` (соответствующее первой карте в `playDeck`) и подключать обработчик событий по имени `Reshuffle()` к событию `playDeck.LastCardDrawn`. Этот обработчик должен просто перетасовывать колоду, инициализировать коллекцию `discardedCards` и снова устанавливать для `currentCard` значение `0` в знак готовности к считыванию карт из новой колоды.

Еще в классе `Game` должно содержаться два служебных метода: `SetPlayers()` для установки игроков для игры (в виде массива объектов `Player`) и `DealHands()` для раздачи карт на руки игрокам (по семь карт каждому). Допустимое количество игроков должно ограничиваться от 2 до 7, что гарантирует наличие достаточного числа карт для хождения по кругу.

И, наконец, еще в классе `Game` должен присутствовать метод `PlayGame()`, содержащий логику самой игры. Мы вернемся к нему чуть позже, после рассмотрения кода в файле `Program.cs`. В файле же `Game.cs` получается, что вся остальная часть кода должна выглядеть следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch13CardLib;

namespace Ch13CardClient
{
 public class Game
 {
  private int currentCard;
  private Deck playDeck;
  private Player[] players;
  private Cards discardedCards;
```

```
  public Game()
  {
    currentCard = 0;
    playDeck = new Deck(true);
    playDeck.LastCardDrawn += new LastCardDrawnHandler(Reshuffle);
    playDeck.Shuffle();
    discardedCards = new Cards();
  }
  private void Reshuffle(Deck currentDeck)
  {
    Console.WriteLine("Discarded cards reshuffled into deck.");
                      // Отброшенные карты перетасованы и помещены в колоду
    currentDeck.Shuffle();
    discardedCards.Clear();
    currentCard = 0;
  }
  public void SetPlayers(Player[] newPlayers)
  {
    if (newPlayers.Length > 7)
     throw new ArgumentException("A maximum of 7 players may play this" + " game.");
                                // В эту игру может играть не более 7 игроков
    if (newPlayers.Length < 2)
      throw new ArgumentException("A minimum of 2 players may play this" + " game.");
                                // В эту игру может играть не менее 2 игроков
    players = newPlayers;
  }
  private void DealHands()
  {
    for (int p = 0; p < players.Length; p++)
    {
      for (int c = 0; c < 7; c++)
      {
        players[p].PlayHand.Add(playDeck.GetCard(currentCard++));
      }
    }
  }
  public int PlayGame()
  {
    // Код, который должен выполняться дальше.
  }
 }
}
```

В файле `Program.cs` должна содержаться функция `Main()`, отвечающая за инициализацию и проведение игры. В частности, она должна выполнять перечисленные ниже шаги.

- Отображать вводную информацию.
- Приглашать пользователя указать количество игроков, каковых может быть не меньше двух, но и не больше семи.
- Создавать массив объектов `Player` соответствующим образом.
- Отображать каждому игроку приглашение ввести имя и использовать его для инициализации одного соответствующего объекта `Player` в массиве.
- Создавать объект `Game` и назначать игроков с помощью метода `SetPlayers()`.

- Запускать игру с помощью метода `PlayGame()`.
- Использовать значение `int`, возвращенное `PlayGame()`, для отображения сообщения о победителе (этим значением является индекс победившего игрока в массиве объектов `Player`).

Код, необходимый для выполнения всех этих шагов, выглядит следующим образом (и для удобства снабжен комментариями):

```
static void Main(string[] args)
{
  // Отображение вводной информации.
  Console.WriteLine("KarliCards: a new and exciting card game.");
                    // KarliCards: новая и увлекательная карточная игра
  Console.WriteLine("To win you must have 7 cards of the same suit in" +
                 "your hand.");
                    // Для выигрыша необходимо, чтобы на руках оказалось
                    // 7 карт одной масти
  Console.WriteLine();
  // Отображение приглашения указать количество игроков.
  bool inputOK = false;
  int choice = -1;
  do
  {
    Console.WriteLine("How many players (2-7)?");
                      // Ввод количества игроков (2-7)
    string input = Console.ReadLine();
    try
    {
      // Попытка преобразовать введенные данные
      // в допустимое число игроков.
       choice = Convert.ToInt32(input);
       if ((choice >= 2) && (choice <= 7))
         inputOK = true;
    }
    catch
    {
      // Игнорирование неудачных попыток преобразования
      // и продолжение отображения приглашения.
    }
  } while (inputOK == false);
  // Инициализация массива объектов Player.
  Player[] players = new Player[choice];
  // Получение имен игроков.
  for (int p = 0; p < players.Length; p++)
  {
    Console.WriteLine("Player {0}, enter your name:", p + 1);
                      // Ввод имен игроков
    string playerName = Console.ReadLine();
    players[p] = new Player(playerName);
  }
  // Запуск игры.
  Game newGame = new Game();
  newGame.SetPlayers(players);
  int whoWon = newGame.PlayGame();
  // Отображение сообщения о победившем игроке.
  Console.WriteLine("{0} has won the game!", players[whoWon].Name);
}
```

Теперь дошло дело и до кода метода `PlayGame()`, представляющего собой основное тело всего приложения. Большинство деталей этого метода можно прояснить по комментариям в коде. Ничего сложного в нем нет; просто метод довольно велик.

Игра продолжается просмотром каждым игроком карт на своих и руках и перевернутой карты на столе. Они могут либо брать эту карту, либо вытаскивать новую из колоды. После вытаскивания карты каждый игрок должен отбрасывать одну карту, либо заменяя карту на столе другой, если та была взята, либо помещая отбрасываемую карту поверх той, что лежит на столе (и тем самым также добавляя ее в коллекцию `discardedCards`).

При просмотре этого кода, который, кстати, приведен ниже, нужно помнить о том, каким образом осуществляется манипулирование объектами `Card`. Причина того, почему эти объекты определяются как ссылочные типы, а не типы-значения (с помощью структуры), к этому моменту должна стать совершенно ясной. Любой взятый объект `Card` может существовать одновременно в нескольких местах, поскольку ссылка не него может присутствовать и в объекте `Deck`, и в полях `hand` объектов `Player`, и в коллекции `discardedCards`, и даже в объекте `playCard` (представляющем карту, выложенную на стол в текущий момент). Это упрощает отслеживание карт и, в частности, применяется в коде, отвечающем за вытаскивание новой карты из колоды. Карта принимается только в том случае, если ее нет ни на руках у игроков, ни в коллекции `discardedCards`.

Выглядит весь необходимый для этого код следующим образом:

```
public int PlayGame()
{
  // Проводить игру только в том случае, если существуют игроки.
  if (players == null)
  return -1;

  // Выполнение первой раздачи карт на руки игрокам.
  DealHands();

  // Инициализация имеющих отношение к игре переменных вместе с переменной,
  // представляющей первую карту, которая должна выкладываться
  // на стол, то есть переменной playCard.
  bool GameWon = false;
  int currentPlayer;
  Card playCard = playDeck.GetCard(currentCard++);
  discardedCards.Add(playCard);

  // Главный цикл игры, который должен продолжать выполняться до тех пор,
  // пока не будет соблюдено условие GameWon == true.
  do
  {
    // Проход по игрокам в каждом раунде игры.
    for (currentPlayer = 0; currentPlayer < players.Length; currentPlayer++)
    {
      // Считывание информации о текущем игроке, о том, какие карты имеются
      // у него на руках, и том, какая карта в текущей момент выложена на столе.
      Console.WriteLine("{0}'s turn.", players[currentPlayer].Name);
      Console.WriteLine("Current hand:");
      foreach (Card card in players[currentPlayer].PlayHand)
      {
        Console.WriteLine(card);
      }
      Console.WriteLine("Card in play: {0}", playCard);
      // Вывод игроку приглашения взять карту со стола или вытащить новую.
      bool inputOK = false;
```

```
do
{
  Console.WriteLine("Press T to take card in play or D to " + "draw:");
                    // Нажмите T, чтобы взять разыгранную карту,
                    // или D, чтобы вытащить новую

  string input = Console.ReadLine();

  if (input.ToLower() == "t")
  {
    // Добавление карты со стола на руки пользователю.
    Console.WriteLine("Drawn: {0}", playCard);

    // Избавление от отброшенных карт, если возможно (в случае
    // перемешивания колоды их там больше быть не должно).
    if (discardedCards.Contains(playCard))
    {
      discardedCards.Remove(playCard);
    }
    players[currentPlayer].PlayHand.Add(playCard);
    inputOK = true;
  }

  if (input.ToLower() == "d")
  {
    // Добавление на руки пользователю новой карты из колоды.
    Card newCard;

    // Добавление карты только в том случае, если ее нет
    // ни на руках у игроков, ни в стопке отброшенных карт.
    bool cardIsAvailable;
    do
    {
      newCard = playDeck.GetCard(currentCard++);

      // Выполнение проверки на предмет того, не находится
      // ли данная карта в стопке отброшенных карт.
      cardIsAvailable = !discardedCards.Contains(newCard);
      if (cardIsAvailable)
      {
        // Просмотр карт на руках у всех игроков для выяснения того,
        // не находится ли карта newCard у кого-нибудь из них.
        foreach (Player testPlayer in players)
        {
          if (testPlayer.PlayHand.Contains(newCard))
          {
            cardIsAvailable = false;
            break;
          }
        }
      }
    } while (!cardIsAvailable);

    // Выдача вытащенной карты на руки игроку.
    Console.WriteLine("Drawn: {0}", newCard);
    players[currentPlayer].PlayHand.Add(newCard);
     inputOK = true;
  }
} while (inputOK == false);
```

```
    // Отображение новой раскладки на руках у игрока с нумерацией карт.
    Console.WriteLine("New hand:");
    for (int i = 0; i < players[currentPlayer].PlayHand.Count; i++)
    {
      Console.WriteLine("{0}: {1}", i + 1, players[currentPlayer].PlayHand[i]);
    }

    // Отображение игроку приглашения отбросить какую-нибудь карту.
    inputOK = false;
    int choice = -1;
    do
    {
      Console.WriteLine("Choose card to discard:");
                        // Выберите карту для отбрасывания
      string input = Console.ReadLine();
      try
      {
        // Попытка преобразовать введенные
        // данные в допустимый номер карты.
        choice = Convert.ToInt32(input);
        if ((choice > 0) && (choice <= 8))
          inputOK = true;
      }
      catch
      {
        // Игнорирование неудачных попыток преобразования
        // и продолжение вывода приглашения.
      }
    } while (inputOK == false);

    // Помещение ссылки на удаляемую карту в playCard (выкладывание
    // карты на стол), затем изъятие карты из рук игрока
    // и добавление ее в стопку отброшенных карт.
    playCard = players[currentPlayer].PlayHand[choice - 1];
    players[currentPlayer].PlayHand.RemoveAt(choice - 1);
    discardedCards.Add(playCard);
    Console.WriteLine("Discarding: {0}", playCard);
                      // Отбрасывание карты

    // Вывод пустой строки для удобства.
    Console.WriteLine();

    // Выполнение проверки на предмет того, выиграл ли игрок
    // в этой игре, и если да, осуществление выхода из цикла.
    GameWon = players[currentPlayer].HasWon();
    if (GameWon == true)
       break;
   }
  } while (GameWon == false);

  // Завершение игры с указанием выигравшего игрока.
  return currentPlayer;
}
```

На рис. 13.8 показана эта игра в действии.

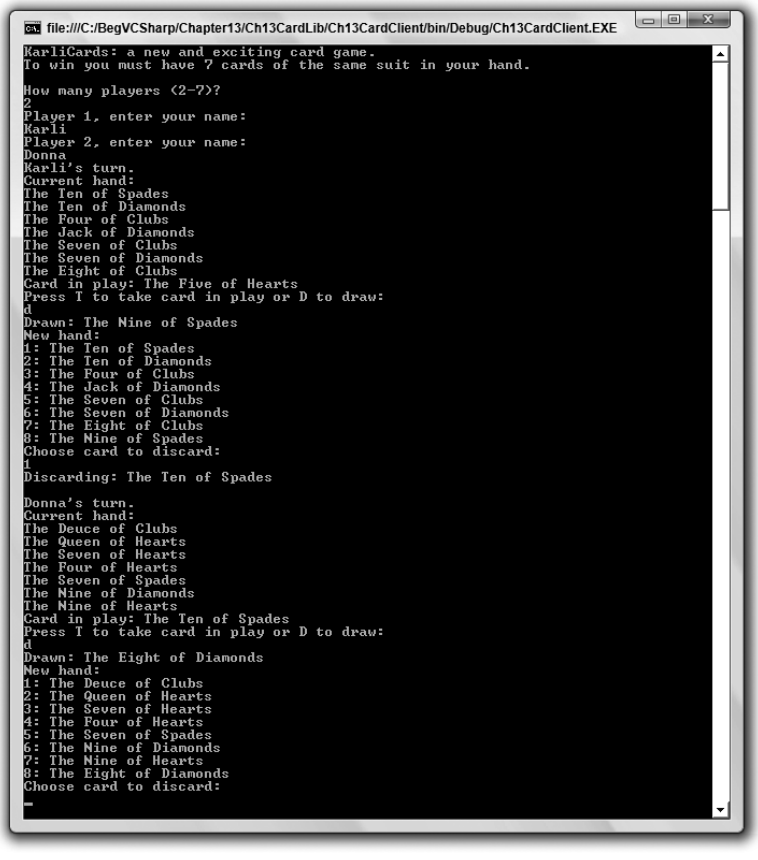

Рис. 13.8. Карточная игра в действии

Поиграйте в эту игру и не поленитесь детально изучить ее. Попробуйте разместить в методе `Reshuffle()` точку останова и сыграть в игру с участием семи игроков. Вытаскивание и отбрасывание вытянутых карт будет быстро приводить к перетасовке колоды, потому что при наличии семи игроков будет оставаться только три карты для избавления. Так вы сможете удостовериться в том, что все работает правильно, просто обращая внимание на то, когда эти три карты будут появляться снова.

Резюме

В этой главе были описаны некоторые более сложные приемы, расширяющие знания по языку C#. Далее перечислены ключевые моменты, с которыми вы ознакомились в этой главе.

- Квалификация наименований типов в пространствах имен (с более детальным описанием, чем приводилось в предыдущих главах).
- Применение операции `::` и ключевого слова `global` для гарантии того, что ссылки на типы являются ссылками именно на те типы, которые требуются.
- Реализация собственных объектов исключений и передача более детальной информации обработчикам исключений.

- Использование специального исключения в коде для `CardLib` — библиотеки карточной игры, которая разрабатывалась в последних нескольких главах.
- События и обработка событий, что является очень важной темой. Требуемый код, хотя и довольно замысловат и непрост для изучения в начале, на самом деле является достаточно простым; к тому же, с обработчиками событий в настоящей книге придется встретиться еще не раз.
- Некоторые наглядные примеры событий и способы их обработки.

В главе снова были внесены изменения в проект `CardLib`, после чего он был использован для создания простого приложения карточной игры. Это приложение позволило проиллюстрировать практически все из тех приемов, которые рассматривались в этой книге до сих пор.

Вместе с этой главой подошло к концу не только описание всех приемов ООП, применяемых в программировании на C#, но и описание всей версии языка C# 2.0. В следующей главе речь пойдет о новых функциональных возможностях C#, которые появились в версии C# 3.0.

Упражнения

1. Напишите с использованием универсального синтаксиса `(object sender, EventArgs e)` код обработчика событий, способного принимать от приведенного ранее в этой главе кода либо событие `Timer.Elapsed`, либо событие `Connection.MessageArrived`. Этот обработчик должен выводить на экран строку, сообщающую о том, событие какого типа было получено, вместе со свойством `Message` параметра `MessageArrivedEventArgs` или свойством `SignalTime` параметра `ElapsedEventArgs` в зависимости от того, какое событие произошло.
2. Модифицируйте пример приложения карточной игры так, чтобы в нем выполнялась проверка на предмет соблюдения такого же более интересного условия выигрыша, как и в популярной карточной игре “рамми”. В соответствие с правилами, выигравшим считается игрок, у которого на руках окажется два “набора” карт — один, состоящий из трех карт, и второй, состоящий из четырех карт. Под набором подразумевается либо последовательность карт одной масти (например, 3 червей, 4 червей, 5 червей, 6 червей), либо несколько карт одного достоинства (например, 2 червей, 2 пик, 2 бубен).