Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

А.А. Волосевич

ИЗБРАННЫЕ ГЛАВЫ ИНФОРМАТИКИ

Курс лекций
для студентов специальности I-31 03 04 «Информатика»
всех форм обучения

Минск 2006

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В середине 2000 года корпорация Microsoft представила новую модель для создания приложений, основой которой является платформа .NET[1]. Платформа .NET образует каркас, который включает технологии разработки Windows-приложений, Web-приложений и Web-сервисов, технологии доступа к данным и межпрограммного взаимодействия. В состав платформы входит обширная библиотека классов. Основным инструментом для разработки является интегрированная среда MS Visual Studio.

Платформа .NET позволяет с легкостью создавать и интегрировать приложения, написанные на различных языках программирования. Специально для .NET был разработан язык программирования C#. Этот язык сочетает простой синтаксис, похожий на синтаксис языков C++ и Java, и полную поддержку всех современных объектно-ориентированных концепций и подходов. В качестве ориентира при разработке языка было выбрано безопасное программирование, нацеленное на создание надежного, простого в сопровождении кода.

Цель данного курса лекций – рассмотреть программирование для платформы .NET с использованием языка программирования C#.

Пособие содержит фрагменты кода и небольшие программы, иллюстрирующие теоретический материал. Примеры могут служить основой при написании лабораторных работ, связанных с объектно-ориентированным программированием с использованием C#.

[1] Произносится как «дот-нэт».

1. ЯЗЫК ПРОГРАММИРОВАНИЯ C#

1.1. ПЛАТФОРМА .NET – ОБЗОР АРХИТЕКТУРЫ

Задача *платформы .NET* (*.NET Framework*) – предоставить программистам более эффективную и гибкую среду разработки традиционных и Web-приложений. Одна из наиболее важных особенностей .NET Framework – способность обеспечить совместную работу кода, написанного на различных языках программирования. На рис. 1 показана структура платформы .NET на самом высоком уровне.

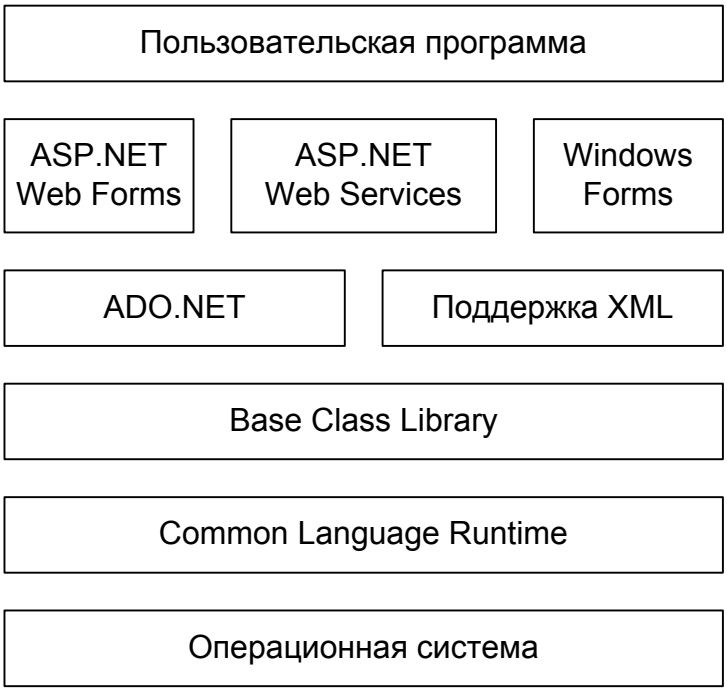

Рис. 1. Общая структура .NET Framework

Базой платформы является *общеязыковая среда исполнения* (*Common Language Runtime*, *CLR*). CLR является «прослойкой» между операционной системой и кодом приложений для .NET Framework. Такой код называется *управляемым* (*managed code*). Более подробно роль CLR обсуждается далее.

В состав платформы .NET входит библиотека классов *Framework Class Library* (*FCL*). Элементом этой библиотеки является базовый набор классов *Base Class Library* (*BCL*). В BCL входят классы для работы со строками, коллекциями данных, поддержки многопоточности и множество других классов. Частью FCL являются компоненты, поддерживающие различные технологии обработки данных и организации взаимодействия с пользователем. Это классы для работы с XML, базами данных (ADO.NET), создания Windows-приложений и Web-приложений (ASP.NET).

В стандартную поставку .NET Framework включены компиляторы для платформы. Это компиляторы языков C#, Visual Basic.NET, J#. Благодаря открытым спецификациям компиляторы для .NET предлагаются различными сто-

ронними производителями (не Microsoft). На данный момент количество компиляторов измеряется десятками.

Рассмотрим подробнее компоненты и роль CLR. Любой компилятор для .NET позволяет получить из исходного текста программы двоичный исполняемый файл или библиотеку кода. Однако эти файлы по своей структуре и содержанию не имеют ничего общего с традиционными исполняемыми файлами операционной системы. Двоичные файлы для платформы .NET называются *сборками* (*assembly*). Сборка состоит из следующих частей:

1. *Манифест* (*manifest*) – описание сборки: версия, ограничения безопасности, список внешних сборок и файлов, необходимых для работы данной сборки.

2. *Метаданные* – специальное описание всех пользовательских типов данных, размещенных в сборке.

3. *Код на промежуточном языке Microsoft Intermediate Language* (*MSIL* или просто *IL*). Данный код является независимым от операционной системы и типа процессора, на котором будет выполняться приложение. В процессе работы приложения он компилируется в машинно-зависимый код специальным компилятором (*Just-in-Time compiler*, *JIT compiler*).

Основная задача CLR – это манипулирование сборками: загрузка сборок, трансляция кода IL в машинно-зависимый код, создание окружения для выполнения сборок. Важной функцией CLR является управление размещением памяти при работе приложение и выполнение *автоматической сборки мусора*, то есть фонового освобождения неиспользуемой памяти. Кроме этого, CLR реализует в приложениях для .NET верификацию типов, управление политиками безопасности при доступе к коду и некоторые другие функции.

Кроме упомянутых элементов, выделим еще две части платформы .NET:

- *Система типов данных* (*Common Type System*, *CTS*) – базовые, не зависящие от языка программирования примитивные типы, которыми может манипулировать CLR.
- *Набор правил* для языка программирования (*Common Language Specification*, *CLS*), соблюдение которых обеспечивает создание на разных языках программ, легко взаимодействующих между собой.

В заключение хотелось бы подчеркнуть, что любой компилятор для .NET является верхним элементом архитектуры. Библиотека классов FCL, имена ее элементов не зависят от языка программирования. Специфичным элементом языка остается только синтаксис, но не работа с внешними классами. Это упрощает межъязыковое взаимодействие, перевод текста программы с одного языка на другой. С другой стороны, тесная связь с CLR неизбежно находит свое отражение в синтаксических элементах языка программирования.

1.2. ЯЗЫК C# - ОБЩИЕ КОНЦЕПЦИИ СИНТАКСИСА

Ключевыми структурными понятиями в языке C# являются *программы*, *пространства имен*, *типы*, *элементы типов* и *сборки*. Программа на языке C# размещается в одном или нескольких текстовых файлах, стандартное расширение которых – `.cs`. В программе объявляются пользовательские типы, которые

состоят из элементов. Примерами пользовательских типов являются классы и структуры, а примером элемента типа может служить метод класса. Типы могут быть логически сгруппированы в пространства имен. При компиляции программы получается сборка, представляющая собой файл с расширением `.exe` или `.dll`.

Исходный текст программы на языке C# содержит *операторы* и *комментарии*. Основными видами операторов в C# являются следующие.

- *Оператор-выражение*. Под выражением может пониматься вызов метода, присваивание, а также допустимые комбинации операндов и операций. Оператор-выражение завершается символом `;` (точка с запятой).
- *Операторы управления* ходом выполнения программы, такие как оператор условного перехода или операторы циклов.
- *Блок операторов*. Блок – это набор операторов, обрамленных фигурными скобками – `{` и `}`. Блоки использует там, где синтаксис языка требует одного оператора.
- *Операторы объявлений* пользовательских типов, элементов типов и локальных переменных и констант.

Программа может содержать комментарии, игнорируемые при компиляции. Различают следующие виды комментариев:

1. *Строчный комментарий* – это комментарий, начинающийся с последовательности `//` и продолжающийся до конца строки.

2. *Блочный комментарий* – все символы, заключенные между `/*` и `*/`.

3. *Комментарии для документации* – напоминают строчные комментарии, но начинаются с последовательности `///` и могут содержать специальные XML-тэги.

В языке C# различаются строчные и прописные символы при записи идентификаторов и ключевых слов. Количество пробелов в начале строки, в конце строки и между элементами строки значения не имеет. Это позволяет улучшить структуру исходного текста программы.

Программа «Hello, World» традиционно используется для первого знакомства с языком программирования. Вот пример этой программы на языке C#.

```
using System;
class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

Дадим некоторые пояснения. Программа представляет собой описание пользовательского типа – класса `Hello`. Любая исполняемая программа на C# должна иметь специальную *точку входа*, с которой начинается выполнение приложения. Такой точкой входа является статический метод `Main()`, объявленный в некотором классе программы (в данном случае – в классе `Hello`). Метод `Main()` содержит вызов метода `WriteLine()` класса `Console` из простран-

ства имен System. Ключевое слово using служит для подключения пространства имен System, содержащего базовые классы. Использование using позволяет вместо полного имени класса System.Console записать короткое имя Console.

Если программа содержится в файле hello.cs, то она может быть скомпилирована при помощи компилятора командной строки csc.exe.

```
csc hello.cs
```

После компиляции будет получена сборка hello.exe.

В заключение параграфа заметим, что большинство примеров в данном пособии представляет собой простые консольные приложения. В таких приложениях для вывода информации используются методы WriteLine() и Write() класса Console. Ввод данных осуществляется функцией Console.ReadLine(). Функция возвращает введенную строку, которая обычно преобразуется в значение требуемого типа.

1.3. СИСТЕМА ТИПОВ ЯЗЫКА C#

Основой C# является развитая система типов. Проведем ее классификацию. С точки зрения размещения переменных в памяти все типы можно разделить на *структурные типы* и *ссылочные типы*. Переменная структурного типа содержит непосредственно данные и размещается в стеке. Переменная ссылочного типа, далее называемая *объектом*, содержит ссылку на данные, которые размещены в управляемой динамической памяти. Структурными типами являются *примитивные типы*, *перечисления* и *структуры*. Ссылочные типы – это *классы*, *интерфейсы*, *массивы* и *делегаты*.

Числовые типы составляют подмножество примитивных типов. Информация о числовых типах содержится в табл. 1.

Таблица 1

Числовые типы языка C#

Категория	Размер (бит)	Имя типа	Диапазон/Точность
Знаковые целые	8	sbyte	–128...127
	16	short	–32 768...32 767
	32	int	–2 147 483 648...2 147 483 647
	64	long	-9 223 372 036 854 775 808...9 223 372 036 854 775 807
Беззнаковые целые	8	byte	0...255
	16	ushort	0...65535
	32	uint	0...4294967295
	64	ulong	0...18446744073709551615
Вещественные	32	float	Точность: от 1.5×10^{-45} до 3.4×10^{38}, 7 цифр
	64	double	Точность: от 5.0×10^{-324} до 1.7×10^{308}, 15 цифр
	128	decimal	Точность: от 1.0×10^{-28} до 7.9×10^{28}, 28 цифр

Отметим, что типы sbyte, ushort, uint, ulong не соответствуют Common Language Specification. Это означает, что данные типы не следует использовать

в интерфейсах многоязыковых приложений и библиотек. Тип `decimal` удобен для проведения финансовых вычислений.

Примитивный тип `bool` служит для представления булевых значений. Переменные данного типа могут принимать значения `true` или `false`.

При работе с символами и строками в C# используется кодировка Unicode. Тип `char` представляет символ в 16-битной Unicode-кодировке, тип `string` – это последовательность Unicode-символов. Заметим, что хотя тип `string` относится к примитивным, переменная этого типа хранит адрес строки в динамической памяти.

Имя примитивного типа в языке C# является синонимом соответствующего типа Framework Class Library. Например, типу `int` в C# соответствует тип `System.Int32`, типу `float` – тип `System.Single` и т. д.

Перечисления, структуры, классы, интерфейсы, массивы и делегаты составляют множество *пользовательских типов*. Элементы пользовательских типов должны быть описаны программистом при помощи особых синтаксических конструкций. Можно утверждать, что любая программа на языке C# представляет собой набор определенных пользовательских типов. Опишем функциональность, которой обладают пользовательские типы.

1. Класс – тип, поддерживающий всю функциональность объектно-ориентированного программирования, включая наследование и полиморфизм.
2. Структура – тип, обеспечивающий всю функциональность ООП, кроме наследования. Структура в C# очень похожа на класс, за исключением метода размещения в памяти и отсутствия поддержки наследования.
3. Интерфейс – абстрактный тип, реализуемый классами и структурами для обеспечения оговоренной функциональности.
4. Массив – пользовательский тип для представления упорядоченного набора значений некоторых (примитивных или пользовательских) типов.
5. Перечисление – тип, содержащий в качестве членов именованные целочисленные константы.
6. Делегат – пользовательский тип для представления ссылок на методы.

В .NET Framework сглажено различие между типами и классами. А именно, любой тип можно воспринимать как класс, который может быть связан с другими типами (классами) отношением наследования. Это позволяет рассматривать все типы .NET Framework (и языка C#) в виде *иерархии классов*. При этом существует базовый тип `System.Object` (в C# – `object`), являющийся общим предком всех типов. Все структурные типы наследуются от класса `System.ValueType`.

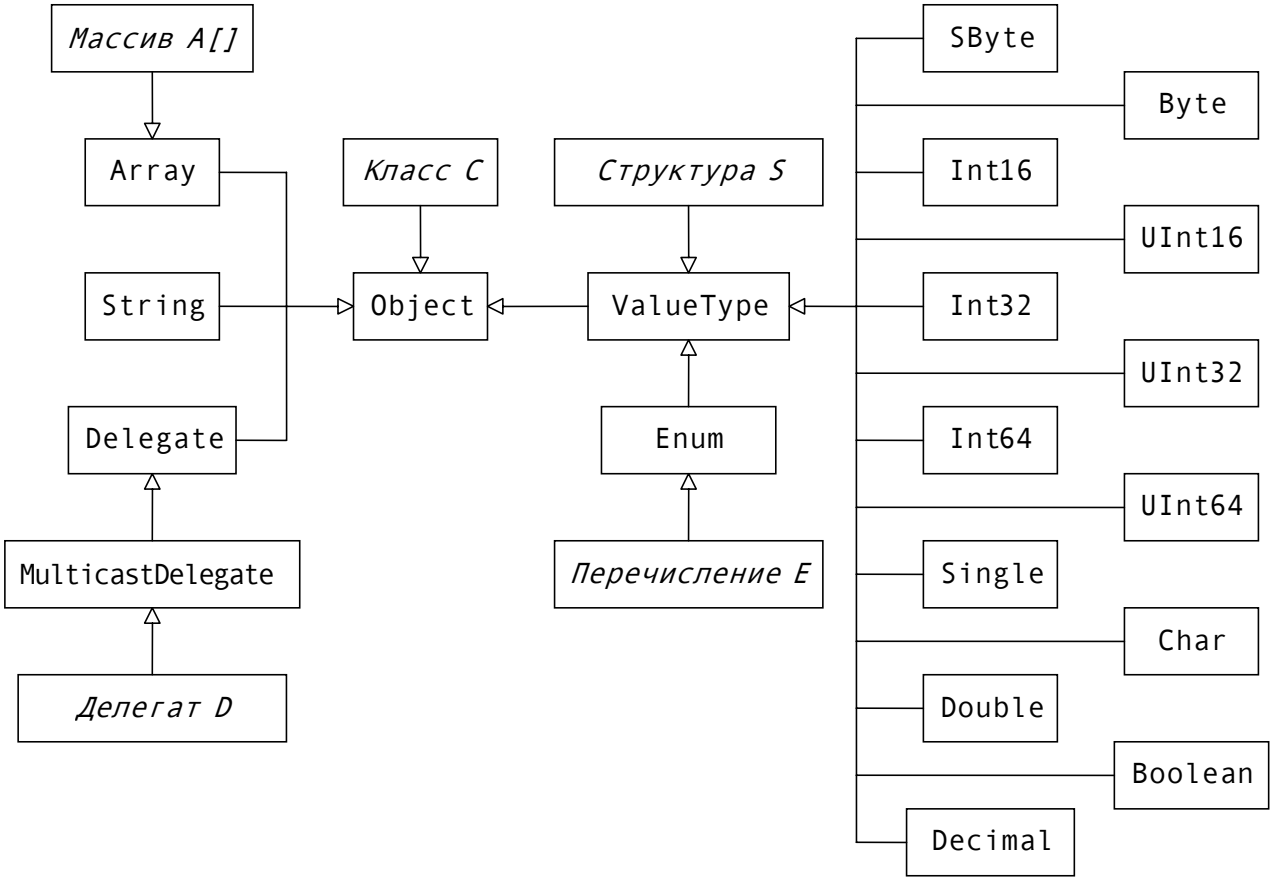

Рис. 2. Иерархия типов .NET Framework

В C# допускается рассмотрение значений структурных типов как переменных типа `object`. Преобразование в объект называется *операцией упаковки* (*boxing*), обратное преобразование – *операцией распаковки* (*unboxing*). При упаковке в динамической памяти создается объект, содержащий значение структурного типа. При распаковке проверяется фактический тип объекта, и значение из динамической памяти переписывается в соответствующую переменную в стеке. Операция распаковки требует явного указания целевого типа.

```
int i = 123;
object o = i;              // Упаковка
int j = (int)o;            // Распаковка
```

Возможность автоматического преобразование каждого типа в тип `object` позволяет создавать универсальные классы, работающие с любыми типами. Например, класс `ArrayList` может содержать коллекцию пользовательских объектов, или целых чисел или строк.

1.4. ПРЕОБРАЗОВАНИЯ ТИПОВ

Если при вычислении выражения операнды имеют разные типы, то возникает необходимость приведения их к одному типу. Такая необходимость возникает и тогда, когда операнды имеют один тип, но он несогласован с типом операции. Например, при выполнении сложения операнды типа `byte` должны быть приведены к типу `int`, поскольку сложение не определено над байтами. При выполнении присваивания `x=e` тип источника `e` и тип цели `x` должны быть со-

гласованы. Аналогично, при вызове метода также должны быть согласованы типы фактического и формального аргументов.

Рассмотрим преобразования при работе с числовыми типами. Заметим, что преобразование типов бывает неявным и явным. *Неявное преобразование* (*implicit conversion*) выполняется автоматически. При выполнении данного преобразования никогда не происходит потеря точности или переполнение, так как множество значений целевого типа включает множества значений приводимого типа. Для числовых типов неявное преобразование типа `A` в тип `B` возможно, если на схеме 3 существует путь из `A` в `B`.

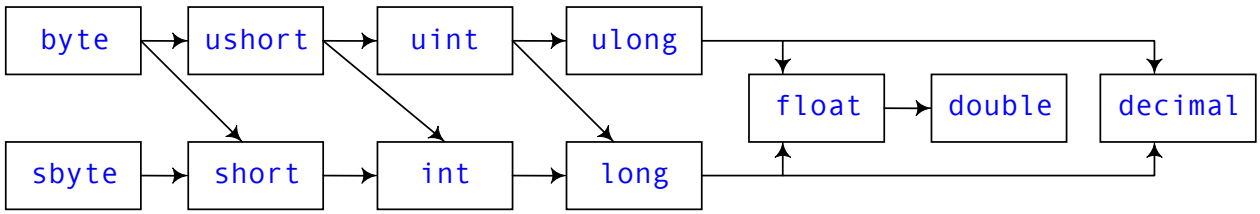

Рис. 3. Схема неявного преобразования числовых типов.

Для *явного преобразования* (*explicit conversion*) требуется применять *оператор приведения* в форме `(<целевой тип>)<выражение>`. При выполнении явного преобразования ответственность за его корректность возлагается на программиста.

```
int k = 100;
byte i;                 //тип byte «меньше» типа int
i = (byte) k;           //требуется явное преобразование типов
```

Для более гибкого контроля значений, получаемых при работе с числовыми выражениями, в языке C# предусмотрено использование контролируемого и неконтролируемого контекстов. *Контролируемый контекст* объявляется в форме `checked <программный блок>`, либо как оператор `checked(<выражение>)`. Если при преобразовании типов выражение в контролируемом контексте получает значение, выходящие за пределы целевого типа, то генерируется либо ошибка компиляции (для константных выражений), либо ошибка времени выполнения (для выражений с переменными).

При использовании *неконтролируемого контекста* выход за пределы целевого типа ведет к автоматическому «урезанию» результата либо путем отбрасывания бит (целые типы), либо путем округления (вещественные типы). *Неконтролируемый контекст* объявляется в форме `unchecked <программный блок>`, либо как оператор `unchecked(<выражение>)`.

Рассмотрим несколько примеров использования контекстов:

```
int i = 1000000;
int k = 1000000;
int n = i * k;
```

В данном примере `n` получит значение `-727379968`, то есть произойдет отбрасывание «лишних» бит числа, получившегося в результате умножения. Используем неконтролируемый контекст:

```
int i = 1000000;
```

```
int k = 1000000;
int n = unchecked(i * k);
```

Значение n осталось прежним (-727379968), таким образом, неконтролируемый контекст применяется по умолчанию.

Теперь проведем вычисления в контролируемом контексте:

```
int i = 1000000;
int k = 1000000;
int n = checked(i * k);
```

При выполнении последнего оператора произойдет генерация исключения System.OverflowException – переполнение.

Важным классом преобразований являются преобразования в строковый тип и наоборот. Преобразования в строковый тип всегда определены, поскольку все типы являются потомками базового класса object, а, следовательно, обладают методом ToString() этого класса. Для встроенных типов определена подходящая реализация этого метода. В частности, для всех числовых типов метод ToString() возвращает строку, задающую соответствующее значение типа. Метод ToString() можно вызывать явно, но, если явный вызов не указан, то он будет вызываться неявно, всякий раз, когда требуется преобразование к строковому типу. Преобразования из строкового типа в другие типы должны всегда выполняться явно при помощи методов встроенных или пользовательских классов. Например, класс System.Int32 обладает методом Parse(), позволяющим преобразовать строку в целое число.

```
System.Console.WriteLine("Input your age");
string s = System.Console.ReadLine();
int Age = System.Int32.Parse(s);
```

Тип char преобразуется в типы sbyte, short, byte явно, а в остальные числовые типы – неявно. Заметим, что преобразование любого числового типа в тип char может быть выполнено, но только в явной форме.

Преобразования пользовательских ссылочных типов выполняются при присваивании и вызове методов. При этом действует стандартное правило ООП: объект типа-потомка преобразуется в объект типа-предка автоматически. Все прочие преобразования должны быть выполнены при помощи оператора приведения. Ответственность за их правильность возлагается на программиста.

В пространстве имен System содержится класс Convert, методы которого поддерживают общий способ выполнения преобразований между типами. Класс Convert содержит набор статических методов вида To<Type>(), где Type – имя встроенного типа CLR (ToBoolean(),ToUInt64() и т. д.). Все методы To<Type>() класса Convert перегружены и каждый из них имеет, как правило, более десятка реализаций с аргументами разного типа. Так что фактически эти методы задают все возможные преобразования между всеми встроенными типами языка C#. Кроме методов, задающих преобразования типов, в классе Convert имеются и другие методы, например, задающие преобразования символов Unicode в однобайтную кодировку ASCII.

1.5. ИДЕНТИФИКАТОРЫ, КЛЮЧЕВЫЕ СЛОВА И ЛИТЕРАЛЫ

Идентификатор – это пользовательское имя для переменной, константы, метода или типа. В C# идентификатор – это произвольная последовательность букв, цифр и символов подчеркивания, начинающаяся с буквы, символа подчеркивания, либо с символа `@`. Идентификатор должен быть уникальным внутри области использования. Он не может совпадать с ключевым словом языка, за исключением того случая, когда используется специальный *префикс* `@`. Примеры допустимых идентификаторов: `Temp`, `_Variable`, `@class` (используется префикс `@`, `class` – ключевое слово).

Далее представлен список всех ключевых слов языка C#.

```
abstract   as         base          bool       break
byte       case       catch         char       checked
class      const      continue      decimal    default
delegate   do         double        else       enum
event      explicit   extern        false      finally
fixed      float      for           foreach    goto
if         implicit   in            int        interface
internal   is         lock          long       namespace
new        null       object        operator   out
override   params     private       protected  public
readonly   ref        return        sbyte      sealed
short      sizeof     stackalloc    static     string
struct     switch     this          throw      true
try        typeof     uint          ulong      unchecked
unsafe     ushort     using         virtual    void
volatile   while
```

В C# *литерал* – это последовательность символов, которая может интерпретироваться как значение одного из примитивных типов. Так как язык C# является языком со строгой типизацией, иногда необходимо явно указать, к какому типу относится последовательность символов, определяющая данные.

Рассмотрим правила записи некоторых литералов. В языке C# два булевых литерала: `true` и `false`. Целочисленные литералы могут быть записаны в десятичной или шестнадцатеричной форме. Признаком шестнадцатеричного литерала является префикс `0x`. Конкретный тип целочисленного литерала определяется следующим образом:

- Если литерал не имеет суффикса, то его тип – это первый из типов `int`, `uint`, `long`, `ulong`, который способен вместить значение литерала.
- Если литерал имеет суффикс `U` или `u`, его тип – это первый из типов `uint`, `ulong`, который способен вместить значение литерала.
- Если литерал имеет суффикс `L` или `l`, то его тип – это первый из типов `long`, `ulong`, который способен вместить значение литерала.
- Если литерал имеет суффикс `UL`, `Ul`, `uL`, `ul`, `LU`, `Lu`, `lU` или `lu`, то его тип – `ulong`.

Если в числе с десятичной точкой не указан суффикс, то подразумевается тип `double`. Суффикс `f` (или `F`) используется для указания на тип `float`, суффикс `d` (или `D`) используется для явного указания на тип `double`, суффикс `m` (или

M) определяет литерал типа decimal. Число с плавающей точкой может быть записано в научном формате: 3.5E-6, -7E10, .6E+7.

Символьный литерал обычно записывают как единичный символ в кавычках ('a'). Однако таким образом нельзя представить символы ' и \. Альтернативным способом записи символьного литерала является использование шестнадцатеричного значения кода Unicode, заключенного в одинарные кавычки ('\x005C' или '\u005C' – это символ \). Кроме этого, для представления некоторых специальных символов используются следующие пары:

- \' – одинарная кавычка;
- \" – двойная кавычка;
- \\ – обратный слеш;
- \0 – пустой символ (null);
- \a – оповещение;
- \b – забой;
- \f – новая страница;
- \n – новая строка;
- \r – возврат каретки;
- \t – горизонтальная табуляция;
- \v – ввертикальная табуляция.

Для строковых литералов в языке C# существуют две формы. Обычно строковый литерал записывается как последовательность символов в двойных кавычках. Среди символов строки могут быть и управляющие последовательности ("This is \t tabbed string"). *Дословная форма* (*verbatim form*) строкового литерала – это запись строки в кавычках с использованием префикса @ (@"There is \t no tab"). В этом случае управляющие последовательности воспринимаются как обычные пары символов.

1.6. ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ, ПОЛЕЙ И КОНСТАНТ

Объявление переменных в языке C# может использоваться в двух контекстах. В первом контексте объявление используется на уровне метода и описывает *локальную переменную*. Во втором контексте оно используется на уровне пользовательского типа (класса или структуры). В этом случае правильнее было бы говорить об объявлении *поля типа*.

Для объявления переменных и полей в C# используется оператор следующего формата:

```
<тип> <имя переменной или поля> [= <начальное значение>];
```

Здесь <имя переменной или поля> – допустимый идентификатор, <тип> – тип переменной, <начальное значение> – литерал или выражение, соответствующие типу переменной и задающие начальное значение. Если начальное значение переменной не задано, то поля получают значение 0 для числовых типов, false для типа bool, символ с кодом 0 для типа char. Любой ссылочный тип, включая строки и массивы, получает специальное значение null. Начальное значение может быть задано как для полей пользовательских типов, так и

для локальных переменных методов. Однако локальные переменные методов не могут использоваться без инициализации (она может быть выполнена как в момент объявления, так и позднее).

Если необходимо объявить несколько переменных или полей одного типа, то идентификаторы переменных можно перечислить через запятую после имени типа. При этом для каждой переменной можно выполнить начальную инициализацию.

```
int a;            //Простейший вариант объявления
int a = 20;       //Объявление с инициализацией
int a, b, c;      //Объявление нескольких однотипных переменных
int a = 20, b = 10;    //Объявление и инициализация переменных
```

Локальная переменная метода может быть объявлена в программном блоке. В этом случае *время жизни* переменной ограничено блоком:

```
{
    int i = 10;
    Console.WriteLine(i);
}
// ошибка компиляции – переменная i не доступна!!!
Console.WriteLine(i);
```

Если программные блоки вложены друг в друга, то внутренний блок не может содержать объявлений переменных, идентификаторы которых совпадают с переменными внешнего блока:

```
{
    int i = 10;
    {
        //ошибка компиляции – i существует во внешнем блоке
        int i = 20;
    }
}
```

Как и другие языки программирования, C# позволяет описать в пользовательском типе или в теле метода константы. Синтаксис объявления константы следующий:

```
const <тип константы> <имя константы> = <значение константы>;
```

Тип константы – это любой примитивный тип (за исключением `object`), `<значение константы>` может быть литералом или результатом действий с другими константами. Примеры объявления констант:

```
const double Pi = 3.1415926;
const double Pi2 = Pi + 2;
```

Для полей пользовательских типов возможно применение модификатора `readonly`, который фактически превращает их в константу. Однако в отличие от констант, тип такого поля может быть любым:

```
public readonly int Age;
```

Для полей классов, которые объявлены с модификатором `readonly`, начальное значение может устанавливать только конструктор (но оно может быть указано и при объявлении поля). Использование модификатора `readonly` для локальных переменных запрещено.

При написании программ часто возникает необходимость разграничить доступ к пользовательским типам или их элементам. Например, скрыть поля класса, ограничить доступ к методу, описанному в некотором классе и т. п. Для этих целей применяют *модификаторы доступа*, указываемые непосредственно перед объявлением типа или элемента типа. Возможно использование следующих модификаторов:

- `private`. Элемент с данным модификатором доступен только в том типе, в котором определен. Например, поле доступно только в содержащем его классе.
- `public`. Элемент доступен без ограничений как в той сборке, где описан, так и в других сборка, к которым подключается сборка с элементом.
- `internal`. Элемент доступен без ограничений, но только в той сборке, где описан.
- `protected`. Элемент с данным модификатором видим только в типе, в котором определен, и в наследниках данного типа (даже если эти наследники расположены в других сборках). Данный модификатор может применяться только в типах, поддерживающих наследование, то есть в классах.
- `protected internal`. Комбинация модификаторов `protected` и `internal`. Элемент виден в содержащей его сборке без ограничений, а вне сборки – только в наследниках типа.

Для локальных переменных методов и программных блоков модификаторы доступа не используются.

1.7. ВЫРАЖЕНИЯ И ОПЕРАЦИИ

Любое выражение в языке C# состоит из операндов и операций. В табл. 2 представлен список операций языка C#, в котором они расположены по убыванию приоритета.

Таблица 2

Операции языка C#

Категория	**Выражение**	**Описание**
Первичные	`x.m`	Доступ к элементу типа
	`x(...)`	Вызов методов и делегатов
	`x[...]`	Доступ к элемену массива и индексатора
	`x++`	Постинкремент
	`x--`	Постдекремент
	`new T(...)`	Создание объекта или делегата
	`new T[...]`	Создание массива
	`typeof(T)`	Получение для типа `T` объекта `System.Type`
	`checked(x)`	Вычисление в контролируемом контексте
	`unchecked(x)`	Вычисление в неконтролируемом контексте
Унарные	`+x`	Идентичность
	`-x`	Отрицание
	`!x`	Логическое отрицание
	`~x`	Битовое отрицание
	`++x`	Пре-инкремент
	`--x`	Пре-декремент
	`(T)x`	Явное преобразование `x` к типу `T`
Умножение	`x * y`	Умножение
	`x / y`	Деление
	`x % y`	Вычисление остатка
Сложение	`x + y`	Сложение, конкатенация строк
	`x - y`	Вычитание
Сдвиг	`x << y`	Битовый сдвиг влево
	`x >> y`	Битовый сдвиг вправо
Отношение и проверка типов	`x < y`	Меньше
	`x > y`	Больше
	`x <= y`	Меньше или равно
	`x >= y`	Больше или равно
	`x is T`	Возвращает `true`, если тип `x` это `T`
	`x as T`	Возвращает `x`, приведенный к типу `T`, или `null`
Равенство	`x == y`	Равно
	`x != y`	Не равно
Логическое AND	`x & y`	Целочисленное битовое AND, логическое AND
Логическое XOR	`x ^ y`	Целочисленное битовое XOR, логическое XOR
Логическое OR	`x | y`	Целочисленное битовое OR, логическое OR
Сокращенное AND	`x && y`	Вычисляется `y`, только если `x = true`
Сокращенное OR	`x || y`	Вычисляется `y`, только если `x = false`
Условие	`x ? y : z`	Если `x = true`, вычисляется `y`, иначе `z`
Присваивание	`x = y`	Присваивание
	`x` *op*`= y`	Составное присваивание, поддерживаются `*= /= %= += -= <<= >>= &= ^= |=`

Правила работы с операциями в C# в основном совпадают с аналогичными правилами в языке C++. Тип результата арифметических операций – это «больший» из типов операндов. Таким образом, `5/2 = 2` (так как операнды це-

лые, то и результат – целый тип), а `5/2d = 2.5`. Составное присваивание неявно включает приведение к типу переменной в левой части. Деление на `0` для вещественных типов не вызывает ошибку – результатом являются специальные значения `infinity` или `NaN` (то есть «бесконечность» при делении на ноль и «не число», если ноль делится на ноль).

1.8. ОПЕРАТОРЫ ЯЗЫКА C#

В языке C# описания типов, методов, свойств, синтаксические конструкции операторов ветвления и циклов образуют в тексте программные блоки. *Программный блок* – это последовательность операторов (возможно пустая), заключенная в фигурные скобки `{` и `}`.

Рассмотрим операторы языка C# для управления ходом выполнения программы. Оператор `break` используется для выхода из блоков операторов `switch`, `while`, `do`, `for` или `foreach`. Оператор `break` выполняет переход на оператор за блоком.

Оператор `continue` применяется для запуска новой итерации циклов `while`, `do`, `for` или `foreach`. Оператор располагается в теле цикла. Если циклы вложены, то запускается новая итерация того цикла, в котором непосредственно располагается `continue`.

Оператор `goto` передает управление на помеченный оператор. Обычно данный оператор употребляется в форме `goto <метка>`, где `<метка>` – это допустимый идентификатор. Метка должна предшествовать помеченному оператору и заканчиваться двоеточием, отдельно описывать метки не требуется:

```
goto label;
. . .
label:
A = 100;
```

Оператор `goto` и помеченный оператор должны располагаться в одном программном блоке. Возможно использование команды `goto` в одной из следующих форм:

```
goto case <константа>;
goto default;
```

Данные варианты обсуждаются при рассмотрении оператора `switch`.

Оператор условного перехода в языке C# имеет следующий формат:

```
if (<условие>)
    <блок1>
[else
    <блок2>]
```

Здесь `<условие>` – это некоторое булево выражение. Ветвь `else` является необязательной.

Оператор выбора `switch` выполняет одну из групп инструкций в зависимости от значения тестируемого выражения. Синтаксис оператора `switch`:

```
switch (<выражение>)
{
    case <константное выражение>:
       <оператор 1>
       . . .
       <оператор n>
       <оператор перехода>
    case <константное выражение 2>:
       <оператор 1>
       . . .
       <оператор n>
       <оператор перехода>
    . . .
    [default:
       <оператор 1>
       . . .
       <оператор n>
       <оператор перехода>]
}
```

Тестируемое `<выражение>` должно иметь целый числовой тип, символьный или строковый тип. При совпадении тестируемого и константного выражений выполняется соответствующая ветвь `case`. Если совпадения не обнаружено, то выполняется секция `default` (если она есть). `<оператор перехода>` – это один из следующих операторов: `break`, `goto`, `return`. Оператор `goto` используется с указанием либо ветви `default` (`goto default`), либо определенной ветви `case` (`goto case <константное выражение>`).

Приведем пример использования оператора `switch`:

```
Console.WriteLine("Input number");
int n = Int32.Parse(Console.ReadLine());
switch (n)
{
    case 0:
        Console.WriteLine("Null");
        break;
    case 1:
        Console.WriteLine("One");
        goto case 0;
    case 2:
        Console.WriteLine("Two");
        goto default;
    case 3:
        Console.WriteLine("Three");
        return;
    default:
        Console.WriteLine("I do not know");
        break;
}
```

Хотя после case может быть указано только одно константное выражение, при необходимости несколько ветвей case можно сгруппировать следующим образом:

```
switch (n)
{
    case 0:
    case 1:
    case 2:
        . . .
}
```

C# представляет разнообразный набор операторов организации циклов. Для циклов с определенным числом итераций используется оператор for. Его синтаксис:

```
for ([<инициализатор>]; [<условие>]; [<итератор>]) <блок>
```

<инициализатор> задает начальное значение счетчика (или счетчиков) цикла. В инициализаторе может использоваться существующая переменная для счетчика или объявляться новая переменнная, время жизни которой будет ограничено циклом. Цикл выполняется, пока булево <условие> истинно, а <итератор> определяет изменение счетчика цикла.

Простейший пример использования цикла for:

```
for (int i = 0; i < 10; i++)  // i доступна только в цикле for
    Console.WriteLine(i);     // вывод чисел от 0 до 9
```

В инициализаторе можно объявить и задать начальные значения для нескольких счетчиков одного типа. В этом случае итератор может представлять собой последовательность из нескольких операторов, разделенных запятой:

```
// цикл выполнится 5 раз, на последней итерации i=4, j=6
for (int i = 0, j = 10; i < j; i++, j--)
    Console.WriteLine("i = {0}, j = {1}", i, j);
```

Если число итераций цикла заранее не известно, можно использовать цикл while или цикл do/while. Данные циклы имеют схожий синтаксис объявления:

```
while (<условие>) <блок>
```

```
do
    <блок>
while (<условие>);
```

В обоих циклах тело цикла выполняется, пока булево <условие> истинно. В цикле while условие проверяется в начале очередной итерации, а в цикле do/while – в конце. Таким образом, цикл do/while всегда выполнится по крайней мере один раз. Обратите внимание, <условие> должно присутствовать обязательно. Для организации бесконечных циклов на месте условия можно использовать литерал true:

```
while (true) Console.WriteLine("Бесконечный цикл!");
```

Для перебора элементов массивов и коллекций в языке C# существует специальный цикл foreach:

```
foreach (<тип> <идентификатор> in <коллекция>) <блок>
```

В заголовке цикла объявляется переменная, которая будет последовательно принимать значения элементов коллекции. При этом присваивание переменной новых значений не отражается на элементах коллекции. Для выполнения цикла `foreach` над коллекцией необходимо, чтобы коллекция реализовывала интерфейс `IEnumerable`.

1.9. ОБЪЯВЛЕНИЕ И ВЫЗОВ МЕТОДОВ

Методы в языке C# являются неотъемлемой частью описания таких пользовательских типов как класс и структура. В C# не существует глобальных методов – любой метод должен быть членом класса или структуры.

Рассмотрим «облегченный» синтаксис описания метода (не используются некоторые модификаторы, в частности, модификаторы доступа):

```
<тип> <имя метода>([<список аргументов>]) <тело метода>
```

`<тип>` – это тип значения, которое возвращает метод. Допустимо использование любого примитивного или пользовательского типа. В C# формально не существует процедур – любой метод является функцией, возвращающей значение. Для «процедуры» в качестве типа указывается специальное ключевое слово `void`. `<имя метода>` – любой допустимый идентификатор, уникальный в описываемом контексте. После имени метода следует пара круглых скобок, в которых указывается список формальных параметров метода (если он не пуст).

Список формальных параметров метода – это набор элементов, разделенных запятыми. Каждый элемент списка имеет следующий формат:

```
[<модификатор>] <тип> <имя формального параметра>
```

Существуют четыре вида параметров, которые специфицируются модификатором:

1. *Параметры-значения* – объявляются без модификатора;
2. *Параметры, передаваемые по ссылке* – используют модификатор `ref`;
3. *Выходные параметры* – объявляются с модификатором `out`;
4. *Параметры-списки* – применяются модификатор `params`.

Параметры, передаваемые по ссылке и по значению, ведут себя аналогично тому, как это происходит в других языках программирования. *Выходные параметры* подобны ссылочным – при работе с ними в теле метода не создается копия фактического параметра. Компилятор отслеживает, чтобы в теле метода выходным параметрам обязательно было присвоено какое-либо значение.

Параметры-списки позволяют передать в метод любое количество фактических параметров. Метод может иметь не более одного параметра-списка, который обязательно должен быть последним в списке формальных параметров. Тип параметра-списка объявляется как тип-массив, и работа с таким параметром происходит в методе как с массивом. Каждый фактический параметр из передаваемого в метод списка ведет себя как параметр, переданный по значению.

Для выхода из метода служит оператор `return`. Если тип возвращаемого методом значения не `void`, то после `return` обязательно указывается возвращаемое методом значение (тип этого значения и тип метода должны совпа-

дать). Кроме этого, инструкция `return` должна встретиться в таком методе во всех ветвях кода по крайней мере один раз.

Рассмотрим несколько примеров объявления методов.

1. Простейшее объявление метода-процедуры без параметров:

```
void SayHello() {
    Console.WriteLine("Hello!");
}
```

2. Метод-функция без аргументов, возвращающая целое значение:

```
int SayInt() {
    Console.WriteLine("Hello!");
    return 5;
}
```

3. Функция `Add()` выполняет сложение двух аргументов, передаваемых как параметры-значения:

```
int Add(int a, int b) {
    return a + b;
}
```

4. Функция `ReturnTwo()` возвращает `10` как результат своей работы, кроме этого значение параметра `a` устанавливается равным `100`:

```
int ReturnTwo(out int a) {
    a = 100;
    return 10;
}
```

4. Метод `PrintList()` использует параметр-список:

```
void PrintList(params int[] List) {
    foreach(int i in List)
        Console.WriteLine(i);
}
```

Метод `PrintList()` можно вызвать несколькими способами. Можно передать методу произвольное количество аргументов целого типа или массив целых значений:

```
//передаем два аргумента
PrintList(10,20);
//а теперь передаем четыре аргумента
PrintList(1, 2, 3, 4);
//создаем и передаем массив целых чисел
PrintList(new int[] {10, 20, 30, 40});
//а можем вообще ничего не передавать
PrintList();
```

При вызове методов на месте формальных параметров помещаются фактические, совпадающие с формальными по типу или приводимые к этому типу. Если при описании параметра использовались модификаторы `ref` или `out`, то они должны быть указаны и при вызове. Кроме этого, фактические параметры с

такими модификаторами должны быть представлены переменными, а не литералами или выражениями.

Значение, возвращаемое методом-функцией, может использоваться в выражениях, а может быть проигнорировано:

```
SayInt(); //хотя это функция, результат ее работы игнорируется
```

C# позволяет использовать *перегрузку методов* в пользовательских типах. Перегруженные методы имеют одинаковое имя, но разную сигнатуру. *Сигнатура* – это набор из модификаторов и типов списка формальных параметров. В языке C# считается, что сигнатура включает модификаторы `ref` и `out`, но не включает модификатор `params`:

```
//Данный код не компилируется - методы Foo различить нельзя!
void Foo(params int[] a) {. . .}
void Foo(int[] a) {. . .}

//Следующий фрагмент кода корректен
void Foo(out int a) {. . .}
void Foo(ref int a) {. . .}
```

Тип метода также не является частью сигнатуры. Специальных ключевых слов для определения перегруженных методов указывать не требуется. При вызове одного из перегруженных методов компилятор выбирает подходящий метод по сигнатуре.

1.10. МАССИВЫ В C#

Объявление массива в языке C# схоже с объявлением переменной, но после указания типа размещается пара квадратных скобок – признак массива:

```
int[] Data;
```

Массив является ссылочным типом, поэтому перед началом работы любой массив должен быть создан в памяти. Для этого используется ключевое слово `new`, после которого указывается тип массива и в квадратных скобках – количество элементов массива.

```
int[] Data;
Data = new int[10];
```

Созданный массив автоматически заполняется значениями по умолчанию своего базового типа. Создание массива можно совместить с его объявлением:

```
int[] Data = new int[10];
```

Для доступа к элементу массива указывается имя массива и индекс в квадратных скобках: `Data[0] = 10`.

Элементы массива нумеруются с нуля, в C# не предусмотрено синтаксических конструкций для указания особого значения нижней границы массива.

В языке C# существует способ инициализации массива значениями при создании. Для этого используется список элементов в фигурных скобках. Инициализация может быть выполнена в *развернутой* и *короткой форме*, которые эквивалентны:

```
int[] Data = new int[10] {1, 2, 3, 5, 7, 11, 13, 17, 19, 23};
int[] Data = {1, 2, 3, 5, 7, 11, 13, 17, 19, 23};
```

Если инициализация выполняется в развернутой форме, то количество элементов в фигурных скобках должно быть равно числу, указанному в квадратных скобках.

При необходимости можно объявить массивы, имеющие несколько размерностей. Для этого в квадратных скобках после типа массива помещают запятые, «разделяющие» размерности:

```
// объявлен двумерный массив D
int[,] D;
// создаем массив D так:
D = new int[10,2];

// объявим трехмерный Cube и создадим его
int[,,] Cube = new int[3,2,5];
// установим элемент массива Cube:
Cube[1,1,0] = 1000;

// объявим маленький двумерный массив и инициализируем его
int[,] C = new int[2,4] {
                         {1, 2, 3, 4},
                         {10, 20, 30, 40}
                        };

// то же самое, немного короче:
int[,] C = {{1, 2, 3, 4}, {10, 20, 30, 40}};
```

В приведенных примерах объявлялись массивы из нескольких размерностей. Такие массивы всегда являются прямоугольными. Можно объявить *массив массивов*, используя следующий синтаксис:

```
int[][] Table;           // Table –массив одномерных массивов
Table = new int[3][];    // в Table будет 3 одномерных массива
Table[0] = new int[2];   // в первом будет 2 элемента
Table[1] = new int[20];  // во втором – 20 элементов
Table[2] = new int[12];  // в третьем – 12 элементов
// а вот так мы работаем с элементами массива Table:
Table[1][3] = 1000;
// совместим объявление и инициализацию массива массивов
int[][] T = {
              new int[2] {10, 20},
              new int[3] {1, 2, 3}
             };
```

При работе с массивами можно использовать цикл `foreach`, перебирающий все элементы. В следующем фрагменте производится суммирование элементов массива `Data`:

```
int[] Data = {1,3,5,7,9};
int Sum = 0;
foreach(int element in Data)
    Sum += element;
```

В цикле `foreach` возможно перемещение по массиву в одном направлении – от начала к концу, при этом попытки присвоить значение элементу массива игнорируются.

В качестве примера работы с массивами рассмотрим программу, выполняющую сортировку массива целых чисел.

```
using System;
class MainClass
{
    public static void Main()
    {
        Console.Write("Введите число элементов: ");
        int Size = Int32.Parse(Console.ReadLine());
        int[] M = new int[Size];

        for (int i = 0; i < Size; i++) {
            Console.Write("Введите {0} элемент массива: ", i);
            M[i] = Int32.Parse(Console.ReadLine());
        }

        Console.WriteLine("Исходный массив:");
        foreach(int i in M) Console.Write("{0,6}", i);
        Console.WriteLine();

        for(int i = 0; i < Size-1; i++)
            for(int j = i+1; j < Size; j++) {
                if (M[i] > M[j]) {
                    int dop = M[i];
                    M[i] = M[j];
                    M[j] = dop;
                }
            }
        Console.WriteLine("Отсортированный массив:");
        foreach(int i in M) Console.Write("{0,6}", i);
    }
}
```

Все массивы в .NET Framework могут рассматриваться как классы, являющиеся потомками класса `System.Array`. В табл. 3 описаны основные методы и свойства класса `System.Array`.

Таблица 3

Элементы класса `System.Array`

Имя элемента	Описание
`Rank`	Свойство только для чтения, возвращает размерность массива
`Length`	Свойство только для чтения, возвращает число элементов массива
`GetLength()`	Метод возвращает число элементов в указанном измерении
`GetLowerBound()`	Метод возвращает нижнюю границу для указанного измерения
`GetUpperBound()`	Метод возвращает верхнюю границу для указанного измерения
`GetValue()`	Метод возвращает значение элемента с указанными индексами

`SetValue()`	Метод устанавливает значение элемента с указанными индексами (значение – первый аргумент).
`Sort()`	Статический метод, который сортирует массив, переданный в качестве параметра. Тип элемента массива должен реализовывать интерфейс `IComparable`
`BinarySearch()`	Статический метод поиска элемента в отсортированном массиве. Тип элемента массива должен реализовывать интерфейс `IComparable`
`IndexOf()`	Статический метод, возвращает индекс первого вхождения своего аргумента в одномерный массив или `-1`, если элемента в массиве нет
`LastIndexOf()`	Статический метод. Возвращает индекс последнего вхождения своего аргумента в одномерный массив или `-1`, если элемента в массиве нет
`Reverse()`	Статический метод, меняет порядок элементов в одномерном массиве или его части на противоположный
`Copy()`	Статический метод. Копирует раздел одного массива в другой массив, выполняя приведение типов
`Clear()`	Статический метод. Устанавливает для диапазона элементов массива значение по умолчанию для типов элементов
`CreateInstance()`	Статический метод. Динамически создает экземпляр массива любого типа, размерности и длины

Теперь рассмотрим примеры использования описанных методов и свойств. В примерах выводимые данные записаны как комментарии. Вначале – использование нескольких простых элементов `System.Array`:

```
int[,] M = {{1, 3, 5}, {10, 20, 30}};
Console.WriteLine(M.Rank);                          // 2
Console.WriteLine(M.Length);                        // 6
Console.WriteLine(M.GetLowerBound(0));              // 0
Console.WriteLine(M.GetUpperBound(1));              // 2
```

Продемонстрируем сортировку и поиск в одномерном массиве:

```
int[] M = {1, -3, 5, 10, 2, 5, 30};
Console.WriteLine(Array.IndexOf(M, 5)); );          //2
Console.WriteLine(Array.LastIndexOf(M, 5));         //5

Array.Reverse(M);
foreach(int a in M)
    Console.WriteLine(a);     //30, 5, 2, 10, 5, -3, 1

Array.Sort(M);
foreach(int a in M)
    Console.WriteLine(a);     //-3, 1, 2, 5, 5, 10, 30

Console.WriteLine(Array.BinarySearch(M, 10));       //5
```

Опишем процесс динамического создания массива. Данный способ позволяет задать для массивов произвольные нижние и верхние границы. Допустим, нам необходим двумерный массив из элементов `decimal`, первая размерность которого представляет годы в диапазоне от 1995 до 2004, а вторая – кварталы в диапазоне от 1 до 4. Следующий код осуществляет создание массива и обращение к элементу массива:

```
//Назначение этих массивов понятно из их названий
int[] LowerBounds = {1995, 1};
int[] Lengths = {10, 4};

//"Заготовка" для будущего массива
decimal[,] Target;
Target = (decimal[,])Array.CreateInstance(typeof(decimal),
                                          Lengths,LowerBounds);

//Пример обращения к элементу
Target[2000, 1] = 10.3M;
```

Допустимо было написать код для создания массива без приведения типов. Однако в этом случае для установки и чтения элементов необходимо было бы использовать методы `SetValue()` и `GetValue()`:

```
Array Target;
Target = Array.CreateInstance(typeof(decimal), Lengths,
                              LowerBounds);
Target.SetValue(10.3M, 2000, 1);
Console.WriteLine(Target.GetValue(2000, 1));
```

Работа с элементами массива, созданного при помощи `CreateInstance()`, происходит медленнее, чем работа с «обычным» массивом.

1.11. РАБОТА С СИМВОЛАМИ И СТРОКАМИ В C#

Для представления отдельных символов в языке C# применяется тип `char`, основанный на структуре `System.Char` и использующий двухбайтную кодировку Unicode представления символов. Статические методы структуры `System.Char` представлены в табл. 4.

Таблица 4

Статические методы `System.Char`

Имя метода	Описание
`GetNumericValue()`	Возвращает численное значение символа, если он является цифрой, и `-1` в противном случае
`GetUnicodeCategory()`	Метод возвращает Unicode-категорию символа
`IsControl()`	Возвращает `true`, если символ является управляющим
`IsDigit()`	Возвращает `true`, если символ является десятичной цифрой
`IsLetter()`	Возвращает `true`, если символ является буквой
`IsLetterOrDigit()`	Возвращает `true`, если символ является буквой или цифрой
`IsLower()`	Возвращает `true`, если символ задан в нижнем регистре
`IsNumber()`	Возвращает `true`, если символ является десятичной или шестнадцатиричной цифрой
`IsPunctuation()`	Возвращает `true`, если символ является знаком препинания
`IsSeparator()`	Возвращает `true`, если символ является разделителем
`IsSurrogate()`	Некоторые символы Unicode представляются двумя 16-битными «суррогатными» символами. Метод возвращает `true`, если символ является суррогатным

`IsUpper()`	Возвращает `true`, если символ задан в верхнем регистре
`IsWhiteSpace()`	Возвращает `true`, если символ является «белым пробелом». К белым пробелам, помимо пробела, относятся и другие символы, например, символ конца строки и символ перевода каретки
`Parse()`	Преобразует строку в символ. Естественно, строка должна состоять из одного символа, иначе возникнет ошибка
`ToLower()`	Приводит символ к нижнему регистру
`ToUpper()`	Приводит символ к верхнему регистру

Большинство статических методов перегружены. Они могут применяться как к отдельному символу, так и к строке, для которой указывается номер символа для применения метода.

Из экземплярных методов `System.Char` стоит отметить метод `CompareTo()`, позволяющий проводить сравнение символов. Он отличается от метода `Equals()` тем, что для несовпадающих символов выдает «расстояние» между символами в соответствии с их упорядоченностью в кодировке Unicode.

Основным типом при работе со строками в C# является тип `string`, задающий строки переменной длины. Тип `string` относится к ссылочным типам. Объекты класса `string` объявляются как все прочие объекты простых типов – с явным или неявным вызовом конструктора класса. Чаще всего, при объявлении строковой переменной конструктор явно не вызывается, а инициализация задается строковым литералом. Но у класса `string` достаточно много конструкторов. Они позволяют сконструировать строку из:

- символа, повторенного заданное число раз;
- массива символов `char[]`;
- части массива символов.

Над строками определены следующие операции: присваивание (=), операции проверки эквивалентности (== и !=), *конкатенация* или сцепление строк (+), взятие индекса ([]).

Операция присваивания строк имеет важную особенность. Поскольку `string` – это ссылочный тип, то в результате присваивания создается ссылка на константную строку, хранимую в динамической памяти. С одной и той же строковой константой в динамической памяти может быть связано несколько переменных. Но когда одна из переменных получает новое значение, она связывается с новым константным объектом в динамической памяти. Остальные переменные сохраняют свои связи. Для программиста это означает, что семантика присваивания строк аналогична семантике присваивания структурных типов.

В отличие от других ссылочных типов операции, проверяющие эквивалентность строк, сравнивают значения строк, а не ссылки. Эти операции выполняются как над структурными типами.

Возможность взятия индекса при работе со строками отражает тот факт, что строку можно рассматривать как массив и получать каждый ее символ. Внимание: символ строки доступен только для чтения, но не для записи.

В языке C# существует понятие *неизменяемый класс* (*immutable class*). Для такого класса невозможно изменить значение объекта при вызове его методов. К неизменяемым классам относится и класс `System.String`. Ни один из мето-

дов этого класса не меняет значения существующих объектов. Конечно, некоторые из методов создают новые значения и возвращают в качестве результата новые строки. Рассмотрим статические методы и свойства класса `System.String`.

Таблица 5

Статические элементы класса `System.String`

Имя элемента	Описание
`Empty`	Возвращается пустая строка. Свойство со статусом `readonly`
`Compare()`	Сравнение двух строк. Реализации метода позволяют сравнивать как строки, так и подстроки. При этом можно учитывать регистр, особенности форматирования дат, чисел и т.д.
`CompareOrdinal()`	Сравнение двух строк. Реализации метода позволяют сравнивать как строки, так и подстроки. Сравниваются коды символов
`Concat()`	Конкатенация строк, метод допускает сцепление произвольного числа строк
`Copy()`	Создается копия строки
`Format()`	Выполняет форматирование строки в соответствии с заданными спецификациями формата
`Join()`	Конкатенация массива строк в единую строку. При конкатенации между элементами массива вставляются разделители. Операция, заданная методом `Join()`, является обратной к операции, заданной экземплярным методом `Split()`

Из описанных статических методов подробно рассмотрим метод `Join()` и «парный» ему экземплярный метод `Split()`. Метод `Split()` позволяет осуществить разбор текста на элементы. Статический метод `Join()` выполняет обратную операцию, собирая строку из элементов.

Метод `Split()` перегружен. Наиболее часто используемая реализация этого метода имеет следующий синтаксис:

```
public string[] Split(params char[])
```

На вход методу `Split()` передается один или несколько символов, интерпретируемых как разделители. Объект `string`, вызвавший метод, разделяется на подстроки, ограниченные этими разделителями. Из этих подстрок создается массив, возвращаемый в качестве результата метода.

Синтаксис статического метода `Join()` таков:

```
public static string Join(string delimiters, string[] items)
```

В качестве результата метод возвращает строку, полученную конкатенацией элементов массива `items`, между которыми вставляется строка разделителей `delimiters`. Как правило, строка `delimiters` состоит из одного символа.

В следующем примере строка разбивается на отдельные слова, затем производится обратная сборка текста:

```
string txt = "А это пшеница, "
             + "которая в темном чулане хранится, "
             + "в доме, который построил Джек!";
Console.WriteLine(txt);
```

```
string[] SimpleSentences, Words;

// делим сложное предложение на простые
SimpleSentences = txt.Split(',');
for(int i = 0;i < SimpleSentences.Length; i++)
    Console.WriteLine(SimpleSentences[i]);

// собираем сложное предложение
string txtjoin = string.Join(",",SimpleSentences);

// делим сложное предложение на слова
Words = txt.Split(',', ' ');
for(int i = 0;i < Words.Length; i++)
    Console.WriteLine("Words[{0}]= {1}",i, Words[i]);
```

Сводка экземплярных методов класса `System.String`, приведенная в таблице 6, дает достаточно полную картину широких возможностей, имеющихся при работе со строками в C#. Следует помнить, что класс `string` является неизменяемым. Поэтому `Replace()`, `Insert()` и другие методы представляют собой функции, возвращающие новую строку в качестве результата.

Таблица 6

Экземплярные методы класса `System.String`

Имя метода	Описание
`Insert()`	Вставляет подстроку в заданную позицию
`Remove()`	Удаляет подстроку в заданной позиции
`Replace()`	Заменяет подстроку в заданной позиции на новую подстроку
`Substring()`	Выделяет подстроку в заданной позиции
`IndexOf()`, `IndexOfAny()`, `LastIndexOf()`, `LastIndexOfAny()`	Определяются индексы первого и последнего вхождения заданной подстроки или любого символа из заданного набора
`StartsWith()`, `EndsWith()`	Возвращается `true` или `false`, в зависимости от того, начинается или заканчивается строка заданной подстрокой
`PadLeft()`, `PadRight()`	Выполняет набивку нужным числом пробелов в начале и в конце строки
`Trim()`, `TrimStart()`, `TrimEnd()`	Удаляются пробелы в начале и в конце строки, или только с одного ее конца
`ToCharArray()`	Преобразование строки в массив символов

В пространсве имен `System.Text` содержится класс `StringBuilder`. Этот класс также предназначен для работы со строками, но он принадлежит к изменяемым классам. Если в программе планируется активно изменять и анализировать строки, рекомендуется использовать именно объекты `StringBuilder`. Это позволит получить существенный (в разы) выйгрыш в производительности.

1.12. СИНТАКСИС ОБЪЯВЛЕНИЯ КЛАССА, ПОЛЯ И МЕТОДЫ КЛАССА

Класс является основным пользовательским типом в языке C#. Синтаксис объявления класса:

```
class <имя класса>
    [<члены класса>]
```

Здесь `<имя класса>` – любой уникальный идентификатор, `<члены класса>` объединены в программный блок. Допустимы следующие члены класса.

1. Поле. Поля класса описываются как обычные переменные, возможно с указанием модификатора доступа. Если для поля не указан модификатор доступа, то по умолчанию подразумевается модификатор `private`. Полям класса можно придавать начальные значения.

```
class CSomeClass {
    int Field1;
    private int Field2 = 10;
    public string Field3;
    . . .
}
```

2. Константа. Объявление константы обычно используется для того, чтобы сделать текст программы более читабельным. Модификатор доступа к константам по умолчанию – `private`. Если объявлена открытая (`public` или `internal`) константа, то для ее использования вне класса можно указывать как имя объекта, так и имя класса.

3. Метод. Методы описывают функциональность класса. Код методов записывается непосредственно в теле класса. Модификатором доступа для методов по умолчанию является `private`.

4. Свойство. Свойства класса призваны предоставить защищенный доступ к полям. Подробнее синтаксис и применение свойств обсуждаются ниже.

5. Индексатор. Индексатор – это свойство-коллекция, отдельный элеменнт которого доступен по индексу.

6. Конструктор. Задача конструктора – начальная инициализация объекта или класса.

7. Деструктор. Деструктор класса служит для уничтожения объектов класса. Так как язык C# является языком с автоматической сборкой мусора, в явном вызове деструкторов нет необходимости. Обычно они содержат некий *завершающий код* для объекта.

8. Событие. События представляют собой механизм рассылки уведомлений различным объектам.

9. Операция. Язык C# допускает перегрузку некоторых операций для объектов класса.

10. Вложенный пользовательский тип. Описание класса может содержать описание другого пользовательского типа – класса, структуры, интерфейса, делегата. Обычно вложенные типы выполняют вспомогательные функции и явно вне основного типа не используются.

При описании класса допустимо указать для него следующие модификаторы доступа – `public` или `internal` (применяется по умолчанию). Если класс является элементом другого пользовательского типа, то его можно объявить с любым модификатором доступа. Заметим, что если класс объявлен с модификатором `internal`, то его `public`-элементы не видны за пределами сборки.

Переменная класса – *объект* – объявляется как обычная переменная:

```
<имя класса> <имя объекта>;
```

Так как класс – ссылочный тип, то объекты должны быть инициализированы до непосредственного использования. Для инициализации объекта используется оператор `new`, совмещенный с вызовом конструктора класса. Если конструктор не описывался, используется предопределенный конструктор без параметров с именем класса:

```
<имя объекта> = new <имя класса>();
```

Инициализацию объекта можно совместить с его объявлением:

```
<имя класса> <имя объекта> = new <имя класса>();
```

Доступ к членам класса через объект осуществляется по синтаксису `<имя объекта>.<имя члена>`.

Приведем пример описания класса, который содержит два поля:

```
class CPet {
    public int Age;
    public string Name;
}
```

Проиллюстрируем описание и использование объектов класса `CPet`:

```
CPet Dog;                   //Просто объявление
CPet Cat = new CPet();      //Объявление с инициализацией
Dog = new CPet();           //Инициализация объекта
Dog.Age = 10;               //Доступ к полям
Cat.Name = "123Y";
```

Добавим в класс `CPet` методы. Заметим, что для устранения конфликта имен «имя члена класса = имя параметра метода» возможно использование ключевого слова `this` – это ссылка на текущий объект класса:

```
class CPet {
    public int Age;
    public string Name;
    void SetAge(int Age) {
        this.Age = Age;
    }
    string GetName() {
        return Name;
    }
}
```

Поля и методы, которые рассматривались в предыдущих примерах, были связаны с объектом класса. Подобные (связанные с объектом) элементы класса

называются *экземплярными*. *Статические* поля, методы и свойства предназначены для работы с классом, а не с объектом. Статические поля хранят информацию, общую для всех объектов, статические методы работают со статическими полями. Для того чтобы объявить статический член класса, используется ключевое слово `static`:

```
class CAccount {
    public static double Tax = 0.1;
    public static double getTax() {
        return Tax * 100;
    }
}
```

Для вызова статических элементов требуется использовать имя класса:

```
CAccount.Tax = 0.3;
Console.WriteLine(CAccount.getTax());
```

В качестве одно из примеров использования статических элементов класса опишем класс `Singleton`. Особенностью этого класса является то, что в приложении можно создать только один объект данного класса.

```
class Singleton {
    static Singleton Instance;
    public string Info;
    private Singleton() {}
    public static Singleton Create() {
        if (Instance == null) Instance = new Singleton();
        return Instance;
    }
}
```

Так как в классе `Singleton` конструктор объявлен как закрытый, то единственный способ создать объект этого класса – вызвать функцию `Create()`. Логика работы этой функции организована так, что `Create()` всегда возвращает ссылку на один и тот же объект. Обратите внимание: поле `Instance` хранит ссылку на объект и описано как статическое. Это сделано для того, чтобы с ним можно было работать в статическом методе `Create()`. Далее приводится пример кода, использующего класс `Singleton`:

```
Singleton A = Singleton.Create();
Singleton B = Singleton.Create();
A.Info = "Information";
Console.WriteLine(B.Info);
```

Объекты `A` и `B` представляют собой одну сущность, то есть на консоль выведется строка `"Information"`.

1.13. СВОЙСТВА И ИНДЕКСАТОРЫ

Свойства класса призваны предоставить защищенный доступ к полям. Как и в большинстве объектно-ориентированных языков, в C# непосредственная работа с полями не приветствуется. Поля класса обычно объявляются как `private`-элементы, а для доступа к ним используются свойства.

Рассмотрим синтаксис описания свойства:

```
<тип свойства> <имя свойства> {
    get {<блок кода>}
    set {<блок кода>}
}
```

Как видно, синтаксис описания свойства напоминает синтаксис описания обычного поля. Тип свойства обычно совпадает с типом того поля, для обслуживания которого свойство создается. У свойства присутствует специальный блок, содержащий методы для доступа к свойству. Данный блок состоит из get-части и set-части. Одна из частей может отсутствовать, так получается *свойство только для чтения* или *свойство только для записи*. Get-часть отвечает за возвращаемое свойством значение и работает как функция (обязательно наличие в блоке кода get-части оператора `return`). Set-часть работает как метод-процедура, устанавливающий значение свойства. Считается, что параметр, передаваемый в set-часть, имеет специальное имя `value`.

Добавим свойства в класс `CPet`, закрыв для использования поля:

```
class CPet {
    private int age;
    private string name;

    public int Age {
        get {
            return age;
        }
        set {
            // проверка корректности значения
            age = value < 0 ? age = 0 : age = value;
        }
    }

    public string Name {
        get {
            return "My name is " + name;
        }
        set { name = value; }
    }
}
```

Свойства транслируются при компиляции в вызовы методов. В скомпилированный код класса добавляются методы со специальными именами `get_`*`Name`* и `set_`*`Name`*, где *`Name`* – это имя свойства. Побочным эффектом данного преобразования является тот факт, что пользовательские методы с данными именами допустимы в классе, только если они имеют сигнатуру, отличающуюся от методов, соответствующих свойству.

В языках программирования VB.NET и Object Pascal наряду с обычными свойствами существовали свойства-массивы. Роль свойств-массивов в C# выполняют *индексаторы*. При помощи индексаторов осуществляется доступ к

коллекции данных, содержащихся в объекте класса, с использованием привычного синтаксиса для доступа к элементам массивы – пары квадратных скобок.

Объявление индексатора напоминает объявление свойства:

```
<тип индексатора> this[<аргументы>] { <get и set блоки> }
```

Аргументы индексатора служат для описания типа и имен индексов, применяемых для доступа к данным объекта. Обычно используется индексы целого типа, хотя это и не является обязательным. Аргументы индексатора доступны в блоках `get` и `set`. Если индексатор имеет более одного аргумента, то аргументы в описании индексатора перечисляются через запятую.

Рассмотрим пример класса, содержащего индексаторы. Пусть данный класс описывает студента с набором оценок:

```
class Student {
    private int[] marks = new int[5];
    public string Name;

    public int this[int i] {
        get {
            if ((i >= 1) && (i <= 5)) return marks[i-1];
            else return 0;
        }
        set {
            if ((i >= 1) && (i <= 5) && (value <= 10))
                marks[i-1] = value;
        }
    }
}
```

Данный класс и индексатор можно использовать следующим образом:

```
Student Ivan = new Student();
Ivan[1] = 8;
Ivan[3] = 4;
for(int i = 1; i <= 5; i++)
    Console.WriteLine(Ivan[i]);
```

Индексаторы всегда работают как *свойства по умолчанию*. Это значит, что в одном классе нельзя объявить два индексатора, у которых совпадают типы аргументов. Однако можно объявить в одном классе индексаторы, у которых аргументы имеют разный тип или количество аргументов различается.

Если свойства транслировались компилятором в методы со специальными именами `get_`*`Name`* и `set_`*`Name`*, то индексаторы транслируются в методы с именами `get_Item` и `set_Item`. Изменить имена методов для индексаторов можно, используя специальный атрибут:

```
class Student {
    . . .
     // методы будут называться get_ Mark и set_ Mark
    [System.Runtime.CompilerServices.IndexerName("Mark")]
    public int this[int i] {. . .}
}
```

1.14. КОНСТРУКТОРЫ КЛАССА И ЖИЗНЕННЫЙ ЦИКЛ ОБЪЕКТА

Конструкторы используются для начальной инициализации объектов. Конструкторы похожи на методы, но в отличие от методов конструкторы не наследуются, и вызов конструктора в виде `<имя объекта>.<имя конструктора>` после создания объекта запрещен. Различают несколько видов конструкторов – конструкторы по умолчанию, пользовательские конструкторы, статические конструкторы.

Конструктор по умолчанию автоматически создается компилятором, если программист не описал в классе собственный конструктор. Конструктор по умолчанию – это всегда конструктор без параметров. Можно считать, что конструктор по умолчанию содержит код начальной инициализации полей.

```
//Класс CPet не содержит конструктора
class CPet {
    public int Age;
    public string Name = "I'm a pet";
}
. . .
CPet Dog = new CPet();        //Вызов конструктора по умолчанию
Console.WriteLine(Dog.Age);   //Выводит 0
Console.WriteLine(Dog.Name); //Выводит I'm a pet
```

Пользовательский конструктор описывается в классе как метод с именем, совпадающим с именем класса. Тип возвращаемого значения для конструктора не указывается (отсутствует любое указание на тип, даже `void`). Пользовательский конструктор может получать параметры, необходимые для инициализации объекта. Класс может содержать несколько пользовательских конструкторов, однако они обязаны различаться сигнатурой.

Опишем два пользовательских конструктора в классе `CPet`:

```
class CPet {
    public int Age;
    public string Name = "I'm a pet";
    public CPet() {
        Age = 0;
        Name = "CPet";
    }
    public CPet(int x, string y) {
        Age = x;
        Name = y;
    }
}
```

При вызове пользовательского конструктора его фактические параметры указываются после имени конструктора в скобках. Если конструкторов у класса несколько, подходящий выбирается по сигнатуре:

```
CPet Cat = new CPet();              //Вызов первого конструктора
CPet Dog = new CPet(5,"Buddy");     //Вызов второго конструктора
```

Пользовательские конструкторы могут применяться для начальной инициализации `readonly`-полей. Напомним, что такие поля ведут себя как кон-

станты, однако могут иметь произвольный тип. Таким образом, `readonly`-поля доступны для записи, но только в конструкторе.

Обратите внимание: если в классе определен хотя бы один пользовательский конструктор, конструктор по умолчанию уже не создается. Если мы изменим класс `CPet`, оставив там только пользовательский конструктор с параметрами, то строка `CPet Cat = new CPet()` будет вызывать ошибку компиляции. Код, который выполняет присваивание полям значений по умолчанию, добавляется компилятором автоматически в начало любого пользовательского конструктора.

Конструктор класса может вызывать другой конструктор того же класса, но только в начале своей работы. Для этого при описании конструктора используется синтаксис, аналогичный приведенному в следующем примере:

```
public CPet() : this(10, "CPet") { . . . }
```

Статические конструкторы используются для начальной инициализации статических полей класса. Статический конструктор объявляется с модификатором `static` и без параметров. Область видимости у статических конструкторов не указывается.

```
class CAccount {
    public static double Tax;
    static CAccount(){
        Tax = 0.1;
    }
}
```

Статические конструкторы вызываются не программистом, а общеязыковой средой исполнения CLR в следующих случаях:

- перед созданием первого объекта класса или при первом обращении к элементу класса, не унаследованному от предка;
- в любое время перед первым обращением к статическому полю, не унаследованному от предка.

В теле статического конструктора возможна работа только со статическими полями и методами класса, статические конструкторы не могут вызывать экземплярные конструкторы класса.

Как уже отмечалось выше, все пользовательские типы в языке C# можно разделить на ссылочные и структурные. Переменные структурных типов создаются средой исполнения CLR в стеке. *Время жизни* (*lifetime*) переменных структурного типа обычно ограничено тем блоком кода, в котором они объявляются. Например, если переменная, соответствующая пользовательской структуре, объявлена в неком методе, то после выхода из метода память, занимаемая переменной, автоматически освободится.

Переменные ссылочного типа (объекты) размещаются в динамической памяти – «куче». Среда исполнения платформы .NET использует *управляемую кучу* (*managed heap*). Объясним значение этого понятия. Если при работе программы превышен некий порог расходования памяти, CLR запускает процесс, называемый *сборка мусора*. Среда исполнения отслеживает все используемые

объекты и определяет реально занимаемую этими объектами память. После этого вся оставшаяся память освобождается, то есть помечается как свободная для использования. Освобождая память, CLR заново размещает «уцелевшие» объекты в куче, чтобы уменьшить ее фрагментацию. Ключевой особенностью сборки мусора является то, что она осуществляется средой исполнения автоматически и независимо от основного потока выполнения приложения.

Обсудим автоматическую сборку мусора с точки зрения программиста, разрабатывающего некий класс. С одной стороны, такой подход имеет свои преимущества. В частности, практически исключаются случайные утечки памяти, которые могут вызвать «забытые» объекты. С другой стороны, объект может захватывать некоторые особо ценные ресурсы (например, подключения к базе данных), которые требуется освободить сразу после того, как объект перестает использоваться. В этой ситуации выходом является написание некого особого метода, который содержит код освобождения ресурсов.

Но как *гарантировать* освобождение ресурсов, даже если ссылка на объект была случайно утеряна? Класс `System.Object` содержит виртуальный метод `Finalize()` без параметров. Если пользовательский класс при работе резервирует некие ресурсы, он может переопределить метод `Finalize()` для их освобождения. Объекты классов, имеющих реализацию `Finalize()` при сборке мусора обрабатываются особо. Когда CLR распознаёт, что уничтожаемый объект имеет собственную реализацию метода `Finalize()`, она откладывает уничтожение объекта. Через некоторое время в отдельном программном потоке происходит вызов метода `Finalize()` и фактическое уничтожение объекта.

Язык C# не позволяет явно переопределить в собственном классе метод `Finalize()`. Вместо переопределения `Finalize()` в классе описывается специальный метод, синтаксис которого напоминает синтаксис *деструктора* языка C++. Имя метода образовывается по правилу `~<имя класса>`, метод не имеет параметров и модификаторов доступа. Считается, что модификатор доступа «деструктора» – `protected`, следовательно, явно вызвать его у объекта нельзя.

Рассмотрим пример класса с «деструктором»:

```
class ClassWithDestructor {
    public string name;
    public ClassWithDestructor(string name) {
        this.name = name;
    }
    public void doSomething() {
        Console.WriteLine("I'm working...");
    }
    //Это деструктор
    ~ClassWithDestructor() {
        Console.WriteLine("Bye!");
    }
}
```

Приведем пример программы, использующей описанный класс:

```
class MainClass {
    public static void Main() {
```

```
            ClassWithDestructor A = new ClassWithDestructor("A");
            A.doSomething();
            // Сборка мусора запуститься перед
            // завершением приложения
        }
    }
```

Данная программа выводит следующие строки:

```
I'm working...
Bye!
```

Проблема с использованием метода-«деструктора» заключается в том, что момент вызова этого метода сложно отследить. Программист может описать в классе некий метод, который *следует* вызывать «вручную», когда объект больше не нужен. Для унификации данного решения платформа .NET предлагает интерфейс `IDisposable`, содержащий единственный метод `Dispose()`, куда помещается завершающий код работы с объектом. Класс, объекты которого требуется освобождать «вручную», реализовывает интерфейс `IDisposable`.

Изменим приведенный выше класс, реализовав в нем интерфейс `IDisposable`:

```
class ClassWithDestructor : IDisposable {
    public string name;
    public ClassWithDestructor(string name) {
        this.name = name;
    }
    public void doSomething() {
        Console.WriteLine("I'm working...");
    }
    // Реализуем метод "освобождения"
    public void Dispose() {
        Console.WriteLine("Dispose called for " + name);
    }

    ~ClassWithDestructor() {
        // Деструктор вызывает Dispose "на всякий случай"
        Dispose();
        Console.WriteLine("Bye!");
    }
}
```

C# имеет специальную *обрамляющую конструкцию* `using`, которая *гарантирует* вызов метода `Dispose()` для объектов, использующихся в своем блоке. Синтаксис данной конструкции:

```
using (<имя объекта или объявление и создание объектов>)
    <программный блок>
```

Изменим программу с классом `ClassWithDestructor`, поместив туда обрамляющую конструкцию `using`:

```
class MainClass {
    public static void Main() {
```

```
            using(ClassWithDestructor A =
                                    new ClassWithDestructor("A")) {
                A.doSomething();
                // компилятор поместит сюда вызов A.Dispose()
            }
        }
    }
```

Что выведет на консоль данная программа? Ниже представлены выводимые строки с комментариями:

```
I'm working...              - это работа метода A.doSomething()
Dispose called for A        - вызывается Dispose() в конце using
Dispose called for A        - эта и следующая строка являются
Bye!                        - результатом работы деструктора
```

Сборщик мусора представлен классом `System.GC`. Метод `Collect()` данного класса вызывает принудительную сборку мусора в программе и может быть вызван программистом. Не рекомендуется пользоваться методом `Collect()` часто, так как сборка мусор требует расхода ресурсов.

1.15. НАСЛЕДОВАНИЕ КЛАССОВ

Язык C# полностью поддерживает объектно-ориентированную концепцию наследования. Чтобы указать, что один класс является наследником другого, используется следующий синтаксис:

```
class <имя наследника> : <имя базового класса> {<тело класса>}
```

Наследник обладает всеми полями, методами и свойствами предка, однако элементы предка с модификатором `private` не доступны в наследнике. Конструкторы класса-предка не переносятся в класс-наследник.

При наследовании нельзя расширить область видимости класса: `internal`–класс может наследоваться от `public`–класса, но не наоборот.

Для обращения к методам непосредственного предка класс-наследник может использовать ключевое слово `base` в форме `base.<имя метода базового класса>`. Если конструктор наследника должен вызвать конструктор предка, то для этого также используется `base`:

```
<конструктор наследника>([<параметры>]): base([<параметры_2>])
```

Для конструкторов производного класса справедливо следующее замечание: конструктор должен вначале совершить вызов другого конструктора своего или базового класса. Если вызов конструктора базового класса отсутствует, компилятор автоматически подставляет в заголовок конструктора вызов `:base()`. Если в базовом классе нет конструктора без параметров, происходит ошибка компиляции.

Наследование от двух и более классов в C# запрещено.

Для классов можно указать два модификатора, связанных с наследованием. Модификатор `sealed` задает класс, от которого запрещено наследование. Модификатор `abstract` задает *абстрактный класс*, у которого обязательно должны быть наследники. Объект абстрактного класса создать нельзя, хотя статиче-

ские члены такого класса можно вызвать, используя имя класса. Модификаторы наследования указываются непосредственно перед ключевым словом `class`:

```
sealed class FinishedClass { }
abstract class AbstractClass { }
```

Класс-наследник может дополнять базовый класс новыми методами, а также замещать методы базового класса. Для замещения достаточно указать в новом классе метод с прежним именем и, возможно, с новой сигнатурой:

```
class CPet {
    public void Speak() {
        Console.WriteLine("I'm a pet");
    }
}

class CDog : CPet {
    public void Speak() {
        Console.WriteLine("I'm a dog");
    }
}
. . .
CPet Pet = new CPet();
CDog Dog = new CDog();
Pet.Speak();
Dog.Speak();
```

При компиляции данного фрагмента будет получено предупреждающее сообщение о том, что метод `CDog.Speak()` закрывает метод базового класса `CPet.Speak()`. Чтобы подчеркнуть, что метод класса-наследника замещает метод базового класса, используется ключевое слово `new`:

```
class CDog : CPet
{
    new public void Speak() { //Компиляция без предупреждений
        Console.WriteLine("I'm a dog");
    }
}
```

Ключевое слово `new` может размещаться как до, так и после модификаторов доступа для метода. Данное ключевое слово применимо и к полям класса.

Замещение методов класса не является полиморфным по умолчанию. Следующий фрагмент кода печатает две одинаковые строки:

```
CPet Pet, Dog;
Pet = new CPet();
Dog = new CDog();      // Допустимо по правилам присваивания
Pet.Speak();           // Печатает "I'm a pet"
Dog.Speak();           // Так же печатает "I'm a pet"
```

Для организации полиморфного вызова методов применяется пара ключевых слов `virtual` и `override`: `virtual` указывается для метода базового класса, который мы хотим сделать полиморфным, `override` – для методов произ-

водных классов. Эти методы должны совпадать по имени и сигнатуре с перекрываемым методом класса-предка.

```
class CPet {
    public virtual void Speak() {
        Console.WriteLine("I'm a pet");
    }
}

class CDog : CPet {
    public override void Speak() {
        Console.WriteLine("I'm a dog");
    }
}
. . .
CPet Pet, Dog;
Pet = new CPet();
Dog = new CDog();
Pet.Speak();            // Печатает "I'm a pet"
Dog.Speak();            // Теперь печатает "I'm a dog"
```

Если на некоторой стадии построения иерархии классов требуется запретить дальнейшее переопределение виртуального метода в производных классах, этот метод помечается ключевым словом `sealed`:

```
class CDog : CPet {
    public sealed override void Speak() { . . . }
```

Для методов абстрактных классов (классов с модификатором `abstract`) возможно задать модификатор `abstract`, который говорит о том, что метод не реализуется в классе, а должен обязательно переопределяться в наследнике.

```
abstract class AbstractClass

{
    //Реализации метода в классе нет
    public abstract void AbstractMethod();
}
```

Отметим, что наряду с виртуальными методами в C# можно описать виртуальные свойства (свойство транслируется в методы). Статические элементы класса не могут быть виртуальными.

1.16. ПЕРЕГРУЗКА ОПЕРАЦИЙ

Язык C# позволяет организовать для объектов пользовательского класса или структуры *перегрузку операций*. Могут быть перегружены унарные операции +, -, !, ~, ++, --, `true`, `false` и бинарные операции +, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, <=.

При перегрузке бинарной операции автоматически перегружается соответствующая операция с присваиванием (например, при перегрузке операции + перегрузится и операция +=). Некоторые операции могут быть перегружены только парами: == и !=, > и <, >= и <=, `true` и `false`.

Для перегрузки операций используется специальный статический метод, имя которого образовано из ключевого слова `operator` и знака операции. Количество формальных параметров метода зависит от типа операции: унарная операция требует одного параметра, бинарная – двух. Метод обязательно должен иметь модификатор доступа `public`.

Рассмотрим перегрузку операций на примере. Определим класс для представления комплексных чисел с перегруженной операцией сложения:

```
class Complex {
    public double Re;
    public double Im;
    public Complex(double Re, double Im) {
        this.Re = Re;
        this.Im = Im;
    }
    public override string ToString() {
        return String.Format("Re = {0} Im = {1}", Re, Im);
    }

    public static Complex operator + (Complex A, Complex B) {
        return new Complex(A.Re + B.Re, A.Im + B.Im);
    }
}
```

Для объектов класса `Complex` возможно использование следующего кода:

```
Complex A = new Complex(10.0, 20.0);
Complex B = new Complex(-5.0, 10.0);
Console.WriteLine(A);           // Выводит Re = 10.0 Im = 20.0
Console.WriteLine(B);           // Выводит Re = -5.0 Im = 10.0
Console.WriteLine(A + B);       // Выводит Re =  5.0 Im = 30.0
```

Параметры метода перегрузки должны быть параметрами, передаваемыми по значению. Тип формальных параметров и тип возвращаемого значения метода перегрузки обычно совпадает с описываемым типом, хотя это и не обязательное условие. Более того, класс или структура могут содержать версии одной операции с разным типом формальных параметров. Однако не допускается существование двух версий метода перегрузки операции, различающихся только типом возвращаемого значения. Также класс не может содержать перегруженной операции, у которой ни один из формальных параметров не имеет типа класса.

Внесем некоторые изменения в класс `Complex`:

```
class Complex {
. . .
    public static Complex operator + (Complex A, Complex B) {
        return new Complex(A.Re + B.Re, A.Im + B.Im);
    }
    public static Complex operator + (Complex A, double B) {
        return new Complex(A.Re + B, A.Im + B);
    }
}
```

Новая перегруженная операция сложения позволяет прибавлять к комплексному числу вещественное число.

Любой класс может перегрузить операции `true` и `false`. Операции перегружаются парой, тип возвращаемого значения операций – булевский. Если в классе выполнена подобная перегрузка, объекты класса могут использоваться как условия в операторах условного перехода или циклов. При вычислении условий используется перегруженная версия операции `true`.

Рассмотрим следующий пример. Пусть в классе `Complex` перегружены операции `true` и `false`:

```
class Complex {
. . .
    public static bool operator true (Complex A) {
        return (A.Re > 0) || (A.Im > 0);
    }
    public static bool operator false (Complex A) {
        return (A.Re == 0) && (A.Im == 0);
    }
}
```

Теперь возможно написать такой код (обратите внимание на оператор `if`):

```
Complex A = new Complex(10.0, 20.0);
Complex B = new Complex(0, 0);
if (B)
    Console.WriteLine("Number is not zero");
else
    Console.WriteLine("Number is 0.0 + 0.0i");
```

Кроме перечисленных операций, любой класс может перегрузить операции для неявного и явного приведения типов. При этом используется следующий синтаксис:

```
public static implicit operator <целевой тип>(<привод. тип> <имя>)
public static explicit operator <целевой тип>(<привод. тип> <имя>)
```

Ключевое слово `implicit` используется при перегрузке неявного приведения типов, а ключевое слово `explicit` – при перегрузке операции явного приведения. Либо `<целевой тип>`, либо `<приводимый тип>` должены совпадать с типом того класса, где выполняется перегрузка операций.

Поместим две перегруженных операции приведения в класс `Complex`:

```
class Complex {
. . .
    public static implicit operator Complex (double a) {
        return new Complex(a, 0);
    }
    public static explicit operator double (Complex A) {
        return Math.Sqrt(A.Re * A.Re + A.Im * A.Im);
    }
}
```

Вот пример кода, использующего преобразование типов:

```
Complex A = new Complex(3.0, 4.0);
double x;
//Выполняем явное приведение типов
x = (double) A;
Console.WriteLine(x);        //Выводит 5

double y = 10;
//Выполняем неявное приведение типов
A = y;
Console.WriteLine(A);        //Выводит Re = 10 Im = 0
```

1.17. ДЕЛЕГАТЫ

Делегат в языке C# исполняет роль указателя на метод. Делегат объявляется с использованием ключевого слова `delegate`. При этом указывается имя делегата и сигнатура инкапсулируемого метода. Модификаторы доступа при необходимости указываются перед ключевым словом `delegate`:

```
delegate double Function(double x);
public delegate void IntegerSub(int i);
```

Делегат – самостоятельный пользовательский тип, он может быть как вложен в другой пользовательский тип (класс, структуру), так и объявлен отдельно. Так как делегат – это пользовательский тип, то нельзя объявить два или более делегатов с одинаковыми именами, но разной сигнатурой.

После объявления делегата можно объявить переменные этого типа:

```
Function Y;
IntegerSub SomeSub;
```

Переменные делегата инициализируются конкретными адресами методов при использовании *конструктора делегата* с одним параметром – именем метода (или именем другого делегата). Если делегат инициализируется статическим методом, требуется указать имя класса и имя метода, для инициализации экземплярным методом указывается объект и имя метода. При этом метод должен обладать подходящей сигнатурой:

```
Y1 = new Function(ClassName.MyStaticFunction);
Y1 = new Function(Obj1.MyInstanceFunction);
Y2 = new Function(Y1);
```

После того как делегат инициализирован, инкапсулированный в нем метод вызывается, указывая параметры метода непосредственно после имени переменной-делегата:

```
Y1(0.5);
```

Приведем пример использования делегатов. Опишем класс, содержащий метод вывода массива целых чисел.

```
class ArrayPrint {
    public static void print(int[] A, PrintMethod P) {
        foreach(int element in A)
```

```
                P(element);
        }
    }
```

PrintMethod является делегатом, который определяет способ печати отдельного числа. Он описан следующим образом:

```
delegate void PrintMethod(int x);
```

Теперь можно написать класс, который работает с классом ArrayPrint и делегатом PrintMethod:

```
class MainClass {
    public static void ConsolePrint(int i) {
        Console.WriteLine(i);
    }

    public void FormatPrint(int i) {
        Console.WriteLine("Element is {0}", i);
    }

    public static void Main() {
      int[] A = {1, 20, 30};

      PrintMethod D = new PrintMethod(MainClass.ConsolePrint);
      ArrayPrint.print(A, D);

      MainClass C = new MainClass();
      D = new PrintMethod(C.FormatPrint);
      ArrayPrint.print(A, D);
    }
}
```

В результате работы данной программы на экран будут выведены следующие строки:

```
1
20
30
Element is 1
Element is 20
Element is 30
```

Обратите внимание, что в данном примере переменная D инициализировалась статическим методом, а затем методом объекта. Делегат не делает различий между экземплярными и статическими методами класса.

Ключевой особенностью делегатов является то, что они могут инкапсулировать не один метод, а несколько. Подобные делегаты называются *групповыми делегатами*. При вызове группового делегата срабатывает вся цепочка инкапсулированных в нем методов.

Групповой делегат объявляется таким же образом, как и обычный. Затем создается несколько объектов делегата, и все они связывается с некоторыми методами. После этого используются перегруженные версии операций + или +=

класса System.Delegate для объединения делегатов в один групповой делегат. Для объединения можно использовать статический метод System.Delegate.Combine(), который получает в качестве параметров два объекта делегата (или массив объектов-делегатов) и возвращает групповой делегат, являющийся объединением параметров.

Модифицируем код из предыдущего примера следующим образом:

```
class MainClass {
. . .
    public static void Main() {
        int[] A = {1, 20, 30};
        PrintMethod first, second, result;
        first = new PrintMethod(MainClass.ConsolePrint);
        MainClass C = new MainClass();
        second = new PrintMethod(C.FormatPrint);
        result = first + second;
        ArrayPrint.print(A, result);
    }
}
```

Теперь результат работы программы выглядит следующим образом:

```
1
Element is 1
20
Element is 20
30
Element is 30
```

Если требуется удалить некий метод из цепочки группового делегата, то используются перегруженные операции - или -= (или метод System.Delegate.Remove()). Если из цепочки удаляют последний метод, результатом будет значение null. Следующий код удаляет метод first из цепочки группового делегата result:

```
result -= first;
```

Любой пользовательский делегат можно рассматривать как класс-наследник класса System.MulticastDelegate, который, в свою очередь, наследуется от класса System.Delegate. Именно на уровне класса System.Delegate определены перегрузкий операций + и -, использовавшихся для создания групповых делегатов. Полезным также может оказаться экземплярный метод GetInvocationList(). Он возвращает массив объектов, составляющих цепочку вызова группового делегата.

1.18. СОБЫТИЯ

События представляют собой способ описания связи одного объекта с другими по действиям. Родственным концепции событий является понятие функции обратного вызова.

Работу с событиями можно условно разделить на три этапа:

- объявление события (*publishing*);

- регистрация получателя события (*subscribing*);
- генерация события (*raising*).

Событие можно объявить в пределах класса, структуры или интерфейса. При объявлении события требуется указать делегат, описывающий процедуру обработки события. Синтаксис объявления события следующий:

```
event <имя делегата> <имя события>;
```

Ключевое слово `event` указывает на объявление события. Объявление события может предваряться модификаторами доступа.

Приведем пример класса с объявлением события:

```
//Объявление делегата для события
delegate void Proc(int val);

class CEventClass {
    int data;
    //Объявление события
    event Proc OnWrongData;
    . . .
}
```

Фактически, события являются полями типа делегатов. Объявление события транслируется компилятором в следующий набор объявлений в классе:

a. в классе объявляется `private`-поле с именем `<имя события>` и типом `<имя делегата>`;
b. в классе объявляются два метода с именами `add_<имя события>` и `remove_<имя события>` для добавления и удаления обработчиков события.

Методы для обслуживания события содержат код, добавляющий (`add_*`) или удаляющий (`remove_*`) процедуру обработки события в цепочку группового делегата, связанного с событием.

Для генерации события в требуемом месте кода помещается вызов в формате `<имя события>(<фактические аргументы>)`. Предварительно можно проверить, назначен ли обработчик события. Генерация события может происходить в одном из методов того же класса, в котором объявлено событие. Генерировать в одном классе события других классов нельзя.

Приведем пример класса, содержащего объявление и генерацию события. Данный класс будет включать метод с целым параметром, устанавливающий значение поля класса. Если значение параметра отрицательно, генерируется событие, определенное в классе:

```
delegate void Proc(int val);

class CExampleClass {
    int field;

    public event Proc onErrorEvent;

    public void setField(int i){
        field = i;
        if(i < 0) {
```

```
                if(onErrorEvent != null) onErrorEvent(i);
            }
        }
    }
```

Рассмотрим этап регистрации получателя события. Для того чтобы отреагировать на событие, его надо ассоциировать с *обработчиком события*. Обработчиком события может быть метод-процедура, совпадающий по типу с типом события (делегатом). Назначение и удаление обработчиков события выполняется при помощи перегруженных версий операторов `+=` и `-=`. При этом в левой части указывается имя события, в правой части – объект делегата, созданного на основе требуемого метода-обработчика.

Используем предыдущий класс `CExampleClass` и продемонстрируем назначение и удаление обработчиков событий:

```
class MainClass {
    public static void firstReaction(int i) {
        Console.WriteLine("{0} is a bad value!", i);
    }

    public static void secondReaction(int i) {
        Console.WriteLine("Are you stupid?");
    }

    public static void Main() {
        CExampleClass c = new CExampleClass();
        c.setField(200);
        c.setField(-200);   // Нет обработчиков, нет и реакции
        // Если бы при генерации события в CExampleClass
        // отсутствовала проверка на null, то предыдущая
        // строка вызвала бы исключительную ситуацию

        // Назначаем обработчик
        c.onErrorEvent += new Proc(firstReaction);

        // Теперь будет вывод "-10 is a bad value!"
        c.setField(-10);

        // Назначаем еще один обработчик
        c.onErrorEvent += new Proc(secondReaction);

        // Вывод: "-10 is a bad value!" и "Are you stupid?"
        c.setField(-10);
    }
}
```

Выше было указано, что методы добавления и удаления обработчиков события генерируются в классе автоматически. Если программиста по каким-либо причинам не устраивает подобный подход, он может описать собственную реализацию данных методов. Для этого при объявлении события указывается блок, содержащий секции `add` и `remove`:

```
event <имя делегата> <имя события> {
    add { }
    remove { }
};
```

Кроме этого, при наличии собственного кода для добавления/удаления обработчиков, требуется *явно* объявить поле-делегат для хранения списка методов обработки.

Исправим класс `CExampleClass`, использовав для события `onErrorEvent` секции `add` и `remove`:

```
class CExampleClass {
    int field;
    //Данное поле будет содержать список обработчиков
    private Proc handlerList;

    public event Proc onErrorEvent {
        add {
            Console.WriteLine("Handler added");
            // Обработчик поступает как неявный параметр value
            // Обратите внимание на приведение типов!
            handlerList += (Proc) value;
        }
        remove {
            Console.WriteLine("Handler removed");
            handlerList -= (Proc) value;
        }
    }

    public void setField(int i){
        field = i;
        if (i < 0) {
            // Проверяем на null не событие, а скрытое поле
            if (handlerList != null) handlerList(i);
        }
    }
}
```

В заключение отметим, что считается стандартным такой подход, при котором сигнатура делегата, отвечающего за обработку события, содержит параметр `sender` (типа `object`), указывающий на источник события, и объект класса `System.EventArgs` (или класса, производного от `System.EventArgs`). Задача второго параметра – инкапсулировать параметры обработчика события.

1.19. ИНТЕРФЕЙСЫ

В языке C# запрещено множественное наследование классов. Тем не менее, в C# существует концепция, позволяющая имитировать множественное наследование. Эта концепция *интерфейсов*. Интерфейс представляет собой набор объявлений свойств, индексаторов, методов и событий. Класс или структура могут *реализовывать* определенный интерфейс. В этом случае они берут на себя обязанность предоставить полную реализацию элементов интерфейса (хотя

бы пустыми методами). Можно сказать так: интерфейс – это контракт, пункты которого суть свойства, индексаторы, методы и события. Если пользовательский тип реализует интерфейс, он берет на себя обязательство выполнить этот контракт.

Объявление интерфейса схоже с объявлением класса. Для объявления интерфейса используется ключевое слово `interface`. Интерфейс содержит только заголовки методов, свойств и событий:

```
interface IBird {
    // Метод
    void Fly();

    // Свойство
    double Speed { get; set;}
}
```

Обратите внимание – в определении элементов интерфейса отсутствуют модификаторы уровня доступа. Считается, что все элементы интерфейса имеют `public` уровень доступа. Более точно, следующие модификаторы не могут использоваться при объявлении членов интерфейса: `abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override`, `static`. Для свойства, объявленного в интерфейсе, указываются только ключевые слова `get` и (или) `set`.

Если класс собирается реализовать интерфейс `IBird`, то он обязуется содержать метод-процедуру без параметров и свойство, доступное и для чтения и для записи, имеющее тип `double`. Имена при этом не существенны, главное – это реализация сигнатуры.

Чтобы показать, что класс реализовывает некий интерфейс, используется синтаксис `<имя класса> : <имя реализовываемого интерфейса>` при записи заголовка класса. Если класс является производным от некоторого базового класса, то имя базового класса указывается перед именем реализовываемого интерфейса: `<имя класса> : <имя базового класса>, <имя интерфейса>`. В простейшем случае, чтобы указать, что некий член класса соответствует элементу интерфейса, у них должны совпадать имена:

```
class CFalcon : IBird {
    private double FS;
    public void DoSomeThing() {
        Console.WriteLine("Falcon Flys");
    }
    public void Fly() {
        Console.WriteLine("Falcon Flys");
    }
    public double Speed {
        get { return FS; }
        set { FS = value; }
    }
}
```

При реализации в классе некоторого интерфейса запрещается использование модификатора `static` для соответствующих элементов класса, так как эле-

менты интерфейса должны принадлежать конкретному объекту, а не классу в целом. Для элементов класса, реализующих интерфейс, обязательным является использование модификатора доступа `public`.

В программе допускается использование переменной интерфейсного типа. Такой переменной можно присвоить значение объекта любого класса, реализующего интерфейс. Однако через такую переменную можно вызывать только члены соответствующего интерфейса:

```
// Объявим переменную интерфейсного типа
IBird Bird;

// Инициализация объектом подходящего класса
Bird = new CFalcon();
Bird.Fly();              // Фактически вызывается CFalcon.Fly()

// Строка вызовет ошибку компиляции! В IBird нет такого метода
Bird.DoSomeThing();
```

Если необходимо проверить, поддерживает ли объект `Obj` некоего класса интерфейс `Inter`, то можно воспользоваться операцией `is`:

```
//Результа равен true, если Obj реализует Inter
if (Obj is Inter) . . .
```

Один класс может реализовывать несколько интерфейсов, при этом имена интерфейсов перечисляются после имени класса через запятую:

```
interface ISwimable {
    void Swim();
}

class CDuck : IBird, ISwimable {
    public void Fly() {
        Console.WriteLine("Duck Flies");
    }
    public void Swim() {
        Console.WriteLine("Duck Swims");
    }
    public double Speed {
        get { return 0.0;}
        set {  }
    }
}
```

Если класс реализует несколько интерфейсов, которые имеют элементы с совпадающими именами, или имя одного из членов класса совпадает с именем элемента интерфейса, то при записи члена класса требуется указать имя в виде `<имя интерфейса>.<имя члена>`. Указание модификаторов доступа при этом запрещается.

Подобно классам, интерфейсы могут наследоваться от других интерфейсов. При этом, в отличие от классов, наследование интерфейсов может быть множественным.

1.20. СТРУКТУРЫ И ПЕРЕЧИСЛЕНИЯ

Структура – это пользовательский тип, поддерживающий всю функциональность класса, кроме наследования. Тип структуры, определенный в языке C#, в простейшем случае позволяет инкапсулировать несколько полей различных типов. Но элементами структуры в C# могут быть не только поля, а и методы, свойства, события, константы. Структуры, как и классы, могут реализовывать интерфейсы. Синтаксис определения структуры следующий:

```
struct <имя структуры> {
    <элементы структуры>
}
```

При описании полей структуры следует учитывать, что они не могут инициализироваться при объявлении. Как и класс, структура может содержать конструкторы. Однако в структуре можно объявить только пользовательский конструктор с параметрами.

Рассмотрим пример структуры для представления комплексных чисел:

```
struct Complex {
    public double Re, Im;
    public Complex(double X, double Y) {
        Re = X;
        Im = Y;
    }
    public Complex Add(Complex Z) {
        return new Complex(this.Re + Z.Re, this.Im + Z.Im);
    }
}
```

Переменные структур определяются как обычные переменные примитивных типов. Однако следует иметь в виду, что поля таких переменных будут не инициализированны. При определении переменной можно вызвать конструктор без параметров – тогда поля получат значения по умолчанию своих типов. Также допустим вызов пользовательского конструктора:

```
// Поля Z1 не инициализированы, их надо установить
Complex Z1;

// Поля Z2 инициализированы значениями 0.0
Complex Z2 = new Complex();

// Поля Z3 инициализированы значениями 2.0, 3.0
Complex Z3 = new Complex(2.0, 3.0);
```

Доступ к элементам структуры осуществляется так же, как к элементам объекта класса:

```
Z1.Re = 10.0;
Z1.Im = 5.0;
Z2 = Z3.Add(Z1);
```

Напомним, что переменные структур размещаются в стеке приложения. Структурные переменные можно присваивать друг другу, при этом выполняется копирование данных структуры на уровне полей.

Перечисление – это тип, содержащий в качестве элементов именованные целочисленные константы. Рассмотрим синтаксис определения перечисления:

```
enum <имя перечисления> [: <тип перечисления>] {
    <элемент перечисления 1> [= <значение элемента>],
    . . .
    <элемент перечисления N> [= <значение элемента>]
}
```

Перечисление может предваряться модификатором доступа. Если задан тип перечисления, то он определяет тип каждого элемента перечисления. Типами перечислений могут быть только `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` или `ulong`. По умолчанию принимается тип `int`. Для элементов перечисления область видимости указать нельзя. Значением элемента перечисления должна быть целочисленная константа. Если для какого-либо элемента перечисления значение опущено, то в случае, если это первый элемент, он принимает значение 0, иначе элемент принимает значение на единицу большее предыдущего элемента. Заданные значения элементов перечисления могут повторяться.

Приведем примеры определения перечислений:

```
enum Seasons {
    Winter,
    Spring,
    Summer,
    Autumn
}

public enum ErrorCodes : byte {
    First = 1,
    Second = 2,
    Fourth = 4
}
```

После описания перечисления можно объявить переменную соответствующего типа:

```
Seasons S;
ErrorCodes EC;
```

Переменной типа перечисления можно присваивать значения, как и обычной переменной:

```
S = Seasons.Spring;
Console.WriteLine(S);  // Выводит на печать Spring
```

Перечисления фактически являются наследниками типа `System.Enum`. При компиляции проводится простая подстановка соответствующих значений для элементов перечислений.

1.21. ПРОСТРАНСТВА ИМЕН

Пространства имен служат для логической группировки пользовательских типов. Применение пространств имен обосновано в крупных программных проектах для снижения риска конфликта имен и улучшения структуры библиотек кода.

Синтаксис описания пространства имен следующий:

```
namespace <имя пространства имен> {
    [<компоненты пространства имен>]
}
```

Компонентами пространства имен могут быть классы, делегаты, перечисления, структуры и другие пространства имен. Само пространство имен может быть вложено только в другое пространство имен.

Если в разных местах программы (возможно, в разных входных файлах) определены несколько пространств имен с одинаковыми именами, компилятор собирает компоненты из этих пространств в общее пространство имен. Для этого только необходимо, чтобы одноименные пространства имен находились на одном уровне вложенности в иерархии пространств имен.

Для доступа к компонентам пространства имен используется синтаксис `<имя пространства имен>.<имя компонента>`. Для компилируемых входных файлов имя пространства имен по умолчанию (если в файле нет обрамляющего пространства имен) можно задать специальной опцией компилятора.

Для использования в программе некоего пространства имен служит команда `using`. Ее синтаксис следующий:

```
using <имя пространства имен>;
```

или

```
using [<имя псевдонима> =] <имя пространства>[.<имя типа>];
```

Импортирование пространства имен позволяет сократить трудозатраты программиста при наборе текстов программ. *Псевдоним*, используемый при импортировании, это обычно короткий идентификатор для ссылки на пространство имен (или элемент из пространства имен) в тексте программы. Импортировать можно пространства имен из текущего проекта, а также из подключенных к проекту сборок.

1.22. ГЕНЕРАЦИЯ И ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Опишем возможности по обработке и генерации исключительных ситуаций в языке C#.

Рассмотрим синтаксис генерации исключительной ситуации. Для генерации исключительной ситуации используется команда `throw` со следующим синтаксисом:

```
throw <объект класса исключительной ситуации>;
```

Обратите внимание: объект, указанный после `throw`, должен обязательно быть объектом класса исключительной ситуации. Таким классом является класс `System.Exception` и все его наследники.

Рассмотрим пример программы с генерацией исключительной ситуации:

```
using System;
class CExample {
    private int fX;
    public void setFx(int x) {
        if (x > 0)
            fX = x;
        else
            // Объект исключит. ситуации создается "на месте"
            throw new Exception();
    }
}
class MainClass {
    public static void Main() {
        CExample A = new CExample();
        A.setFx(-3); // ИС генерируется, но не обрабатывается!
    }
}
```

Так как в данном примере исключительная ситуация генерируется, но никак не обрабатывается, при работе приложения появится стандартное окно с сообщением об ошибке.

Класс `System.Exception` является стандартным классом для представления исключительных ситуаций. Основными членами данного класса является свойство только для чтения `Message`, содержащее строку с описанием ошибки, и перегруженный конструктор с одним параметром-строкой, записываемой в свойство `Message`. Естественно, библиотека классов .NET Framework содержит большое число разнообразных классов, порожденных от `System.Exception` и описывающих конкретные исключительные ситуации.

Пользователь может создать собственный класс для представления информации об исключительной ситуации. Единственным условием является прямое или косвенное наследование этого класса от класса `System.Exception`.

Модифицируем пример с генерацией исключительной ситуации, описав для исключительной ситуации собственный класс:

```
class MyException : Exception {
    public int info;
}
class CExample {
    private int fX;
    public void setFx(int x) {
        if (x > 0)
            fX = x;
        else {
            MyException E = new MyException();
            E.info = x;
            throw E;
        }
    }
}
```

Опишем возможности по обработке исключительных ситуаций. Для перехвата исключительных ситуаций служит блок try – catch – finally. Синтаксис блока следующий:

```
try {
    [<команды, способные вызвать исключительную ситуацию>]
}
[<один или несколько блоков catch>]
[finally {
<операторы из секции завершения> }]
```

Операторы из части finally (если она присутствует) выполняются всегда, вне зависимости от того, произошла исключительная ситуация или нет. Если один из операторов, расположенных в блоке try, вызвал исключительную ситуацию, управление немедленно передается на блоки catch. Синтаксис отдельного блока catch следующий:

```
catch [(<тип ИС> [<идентификатор объекта ИС>])] {
    <команды обработки исключительной ситуации>
}
```

<идентификатор объекта ИС> – это некая временная переменная, которая может использоваться для извлечения информации из объекта исключительной ситуации. Отдельно описывать эту переменную нет необходимости.

Модифицируем программу, описанную выше, добавив в нее блок перехвата ошибки:

```
class MainClass
{
  public static void Main()
  {
    CExample A = new CExample();
    try {
      Console.WriteLine("Эта строка печатается");
      A.setFx(-3);
      Console.WriteLine("Строка не печатается, если ошибка ");
    }
    catch (MyException ex) {
      Console.WriteLine("Ошибка при параметре {0}", ex.Info);
    }
    finally {
      Console.WriteLine("Строка печатается - блок finally");
    }
  }
}
```

Если используется несколько блоков catch, то обработка исключительных ситуаций должна вестись по принципу «от частного – к общему», так как после выполнения одного блока catch управление передается на часть finally (при отсутствии finally – на оператор после try – catch). Компилятор C# не позволяет разместить блоки catch так, чтобы предыдущий блок перехватывал исключительные ситуации, предназначенные последующим блокам:

```
try {
. . .
}
//Ошибка компиляции, так как MyException – наследник Exception
catch (Exception ex) {
    Console.WriteLine("Общий перехват");
}
catch (MyException ex) {
    Console.WriteLine("Эта строка не печатается никогда!");
}
```

Запись блока `catch` в форме `catch (Exception) { }` позволяет перехватывать все исключительные ситуации, генерируемые CLR. Если записать блок `catch` в форме `catch { }`, то такой блок будет обрабатывать любые исключительные ситуации, в том числе и не связанные с исполняющей средой.

1.23. НОВОВВЕДЕНИЯ В ЯЗЫКЕ C# 2.0

В ноябре 2005 года корпорация Microsoft представила вторую версию платформы .NET. Эта версия содержит изменения, коснувшиеся как технологий и подходов, применяемых в рамках платформы, так и языков программирования для платформы. В данном и следующем параграфе рассмотрим нововведения во второй версии языка C#.

1. Статические классы. При объявлении класса возможно указать модификатор `static`. Таким образом определяется *статический класс*, включающий только статические элементы.

```
public static class AppSettings {
    public static string BaseDir { . . . }
    public static string GetRelativeDir() { . . . }
}
```

Экземпляр статического класса не может быть создан или даже объявлен в программе. Все элементы класса доступны только с использованием имени класса.

2. Частичные типы. Хорошей практикой программирования считается помещать описание каждого пользовательского типа в отдельный файл. Однако иногда классы и структуры получаются настолько большими, что указанный подход становиться непрактичным. Особенно это справедливо при использовании средств автоматического генерирования кода. *Частичные типы* (*partial types*) позволяют классам, структурам и интерфейсам быть разбитыми на несколько фрагментов, описанных в отдельных файлах с исходным текстом.

Для объявления частичного типа используется модификатор `partial`. Рассмотрим пример частичного типа:

```
// Файл part1.cs
partial class BrokenClass {
    private int someField;
    private string anotherField;
}
```

```
// Файл part2.cs
partial class BrokenClass {
    public BrokenClass() { }
    public void Method() { }
}
```

Подчеркнем, что все фрагменты частичного типа должны быть доступны во время компиляции, так как «сборку» типа выполняет компилятор. Еще одно замечание касается использования модификаторов, применяемых к типу. Модификаторы доступа должны быть одинаковыми у всех фрагментов. Если же к одному из фрагментов применяется модификатор `sealed` или `abstract`, то эти модификаторы считаются примененными ко всем фрагментам, то есть к типу в целом.

3. Модификаторы доступа для get и set частей свойств и индексаторов. Как правило, в пользовательском типе свойства открыты, имеют модификатор доступа `public`. Однако иногда логика типа требует, чтобы у свойства были отдельные «привилегии» для чтения и записи значений. Например, чтение позволено всем, а запись – только из методов того типа, где объявлено свойство. В C# 2.0 разрешено при описании свойства или индексатора указывать модификаторы доступа для get и set частей:

```
class SomeClass {

    public int Prop {
        get { . . . }
        private set { . . . }
    }
}
```

При указании модификаторов для get и set частей действуют два правила. Во-первых, модификатор может быть только у одной части. Во-вторых, он должен «понижать» видимость части по сравнению с видимостью всего свойства.

4. Безымянные методы. Назначение *безымянных методов* (*anonymous methods*) – сократить объем кода, который должен написать разработчик при работе с событиями. Рассмотрим пример, в котором назначаются обработчики событий для объектов класса `CExampleClass` (подробнее – в параграфе, посвященном работе с событиями).

```
class MainClass {
    public static void firstReaction(int i) {
        Console.WriteLine("{0} is a bad value!", i);
    }
    public static void secondReaction(int i) {
        Console.WriteLine("Are you stupid?");
    }

    public static void Main() {
        CExampleClass c = new CExampleClass();
        // Назначаем обработчик
        c.onErrorEvent += new Proc(firstReaction);
```

```
            // Назначаем еще один обработчик
            c.onErrorEvent += new Proc(secondReaction);

        }
    }
```

Мы видим, что для назначения даже простых обработчиков необходимо отдельно описать методы обработки событий и использовать конструкторы соответствующих делегатов. А вот как будет выглядеть тот же код при использовании безымянных методов:

```
class MainClass {
    public static void Main() {
        CExampleClass c = new CExampleClass();
        // Назначаем обработчик
        c.onErrorEvent += delegate(int i) {
              Console.WriteLine("{0} is a bad value!", i); };
        // Назначаем еще один обработчик
        c.onErrorEvent += delegate {
                    Console.WriteLine("Are you stupid?"); };
    }
}
```

Таким образом, код обработки события помещен непосредственно в место присваивания делегата событию `onErrorEvent`. Теперь компилятор сам создаст делегат, основываясь на указанной сигнатуре анонимного метода, поместит в этот делегат указатель на код анонимного метода, а сам объект делегата присвоит событию.

Обратите внимание в предыдущем фрагменте кода на второй безымянный метод. Он не использовал параметр события, что позволило не указывать сигнатуру метода после `delegate`. Компилятор автоматически выполняет соответствующие неявные преобразования для делегатов и безымянных методов. Точные правила совместимости делегата и безымянного метода выглядят следующим образом:

1. Список параметров делегата совместим с безымянным методом, если выполняется одно из двух условий:

a. безымянный метод не имеет параметров, а делегат не имеет out-параметров;

b. список параметров безымянного метода полностью совпадает со списком параметров делегата (число параметров, типы, модификаторы)

2. Тип, возвращаемый делегатом, совместим с типом безымянного метода, если выполняется одно из двух условий:

a. тип делегата – `void`, а безымянный метод не имеет оператора `return` или оператор `return` записан без последующего выражения;

b. выражение, записанное после `return` в безымянном методе, может быть неявно приведено к типу делегата.

В следующем примере безымянные методы используются для написания функций «на лету». Безымянный метод передается как параметр, тип которого `Function`.

```
using System;
delegate double Function(double x);
class Test {
    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++)
            result[i] = f(a[i]);
        return result;
    }

    static double[] MultAllBy(double[] a, double factor) {
      return Apply(a, delegate(double x){ return x*factor;});
    }

    static void Main() {
     double[] a = { 0.0, 0.5, 1.0 };
     double[] s = Apply(a, delegate(double x){ return x*x;});
     double[] doubles = MultAllBy(a, 2.0);
    }
}
```

Как описывалось выше, безымянные методы могут быть неявно приведены к типу соответствующего делегата. C# 2.0 позволяет проводить подобное преобразование и с использованием обычных методов. Рассмотрим фрагмент кода:

```
addButton.Click += new EventHandler(AddClick);
Apply(a, new Function(Math.Sin));
```

В C# 2.0 он может быть переписан в следующей короткой форме:

```
addButton.Click += AddClick;
Apply(a, Math.Sin);
```

Если используется короткая форма, компилятор автоматически устанавливает, конструктор какого тип делегата нужно вызвать.

5. Итераторы. Язык C# содержит удобную синтаксическую конструкцию `foreach` для перебора элементов пользовательского типа. Чтобы поддерживать перебор при помощи `foreach`, тип должен реализовывать интерфейс `IEnumerable`. Кодирование поддержки этого интерфейса упрощается с использованием итераторов. *Итератор* (*iterator*) – это блок кода, который порождает упорядоченную последовательность значений. Итератор отличает присутствие в блоке кода одного или нескольких операторов `yield`. Оператор `yield return` <выражение> возвращает следующее значение в последовательности, оператор `yield break` прекращает генерирование последовательности.

Итераторы могут использоваться в качестве тела функции, если тип возвращаемого значения функции – это тип, реализующий или наследованный от интерфейсов `IEnumerator`, `IEnumerator<T>`, `IEnumerable`, `IEnumerable<T>`.

Рассмотрим пример использования итераторов. Следующее приложение выводит на консоль таблицу умножения:

```
using System;
using System.Collections;
```

```
class Test {
    static IEnumerable FromTo(int from, int to) {
        while (from <= to) yield return from++;
    }

    static void Main() {
        IEnumerable e = FromTo(1, 10);
        foreach (int x in e) {
            foreach (int y in e) {
                Console.Write("{0,3} ", x * y);
            }
            Console.WriteLine();
        }
    }
}
```

6. Типы с поддержкой null-значений. Для разработчиков всегда была проблемой поддержка неопределенных, пустых значений в структурных типах. Иногда для указания на неопределенное значение использовалось дополнительное булево поле, иногда – некая специальная константа. Язык C# новой версии предлагает для решения этой проблемы *типы с поддержкой null-значений*.

Тип с поддержкой null-значений (далее для краткости – null-тип) объявляется с использованием модификатора `?`, записанного непосредственно после имени типа. Например, для типа `int` соответствующий null-тип объявляется как `int?`. Null-типы могут быть объявлены только для структурных типов (примитивных или пользовательских). В null-типе присутствует специальный булевский индикатор `HasValue`, указывающий на наличие значения, и свойство `Value`, содержащее значение. Попытка прочитать значение `Value` при `HasValue=false` ведет к генерации исключения.

Приведем фрагмент кода, использующий null-типы.

```
int? x = 123;
int? y = null;
if (x.HasValue) Console.WriteLine(x.Value);
if (y.HasValue) Console.WriteLine(y.Value);
```

Существует возможность неявного приведения структурного типа в соответствующий null-тип. Кроме этого, любой переменной null-типа может быть присвоено значение `null`. Если для структурного типа `S` возможно приведение к структурному типу `T`, то соответствующая возможность имеется и для типов `S?` и `T?`. Также возможно неявное приведение типа `S` к типу `T?` и явное приведение `S?` к `T`. В последнем случае возможна генерации исключительной ситуации – если значение типа `S?` не определено.

```
int x = 10;
int? z = x;               // неявное приведение int к int?
double? w = z;            // неявное приведение int? к double?
double y = (double)z;     // явное приведение int? к double
```

С поддержкой null-типов связано появление в C# новой операции ??. Результатом выражения a ?? b является a, если оно содержит некое значение, и b – в противном случае. Таким образом, b – это то значение, которое следует использовать, если a не определено. Тип результата выражения a ?? b определяется типом операнда b.

```
int? x = GetNullableInt();
int? y = GetNullableInt();
int? z = x ?? y;
int i = z ?? -1;
```

Операцию ?? можно применить и для ссылочных типов:

```
string s = GetStringValue();
Console.WriteLine(s ?? "Unspecified");
```

В этом фрагменте кода на консоль выводится значение строки s, или "Unspecified", если s=null.

1.24. ОБОБЩЕННЫЕ ТИПЫ (GENERICS)

Обобщенные (или *параметризованные*) *типы* (*generics*) позволяют при описании пользовательских классов, структур, интерфейсов, делегатов и методов указать как параметр тип данных для хранения и обработки.

Классы, описывающие структуры данных, обычно используют базовый тип object, чтобы хранить данные любого типа. Например, класс для стека может иметь следующее описание:

```
class Stack {
    object[] items;
    int count;
    public void Push(object item) {...}
    public object Pop() {...}
}
```

Класс Stack универсален, он позволяет хранить произвольные объекты:

```
Stack stack = new Stack();
stack.Push(new Customer());
Customer c = (Customer)stack.Pop();

Stack stack2 = new Stack();
stack2.Push(3);
int i = (int)stack2.Pop();
```

Однако универсальность класса Stack имеет и отрицательные моменты. При извлечении данных из стека необходимо выполнять приведение типов. Для структурных типов (таких как int) при помещении данных в стек и при извлечении выполняются операции упаковки и распаковки, что отрицательно сказывается на быстродействии. И, наконец, неверный тип помещаемого в стек элемента может быть выявлен только на этапе выполнения, но не компиляции.

```
Stack stack = new Stack();    // планируем сделать стек чисел
stack.Push(1);
```

```
stack.Push(2);
stack.Push("three");          // вставили не число, а строку
for (int i = 0, i < 3, i++)
    // компилируется, но при выполнении на третьей итерации
    // будет сгенерирована исключительная ситуация
    int result = (int)stack.Pop();
```

Основной причиной появления обобщенных типов была необходимость устранения описанных недостатков универсальных классов.

Опишем класс `Stack` как обобщенный тип. Для этого используется следующий синтаксис: после имени класса в угловых скобках `<` и `>` указывается параметр типа. Этот параметр затем может использоваться при описании элементов класса `Stack`.

```
class Stack<T> {
    T[] items;
    int count;
    public void Push(T item) {...}
    public T Pop() {...}
}
```

При использовании обобщенного типа `Stack` требуется указать фактический тип вместо параметра `T`. В следующем фрагменте `Stack` применяется для хранения данных типа `int`.

```
Stack<int> stack = new Stack<int>();
stack.Push(3);
int x = stack.Pop();
```

Тип вида `Stack<int>` будем в дальнейшем называть *сконструированным типом* (*constructed type*). Обратите внимание: при работе с типом `Stack<int>` отпала необходимость в выполнении приведения типов при извлечении элементов из стека. Кроме этого, теперь компилятор отслеживает, чтобы в стек помещались только данные типа `int`. И еще одна менее очевидная особенность. Нет необходимости в упаковке и распаковке структурного элемента, а это приводит к увеличению быстродействия.

При объявлении обобщенного типа можно использовать несколько параметров. Приведем фрагмент описания класса для хранения пар «ключ-значение» с возможностью доступа к значению по ключу:

```
class Dict<K,V> {
    public void Add(K key, V value) {...}
    public V this[K key] {...}
}
```

Сконструированный тип для `Dict<K,V>` должен быть основан на двух конкретных типах:

```
Dict<string,Customer> dict = new Dict<string,Customer>();
dict.Add("Alex", new Customer());
Customer c = dict["Alex"];
```

Как правило, обобщенные типы не просто хранят данные указанного параметра, а еще и вызывают методы у объекта, чей тип указан как параметр. К примеру, в классе `Dict<K,V>` метод `Add()` может использовать метод `CompareTo()` для сравнения ключей:

```
class Dict<K,V> {
    public void Add(K key, V value) {
        . . .
        if(key.CompareTo(x) < 0) {...}  // Ошибка компиляции!
        . . .
    }
}
```

Так как тип параметра `K` может быть любым, у `key` можно вызывать только методы класса `object`, и приведенный выше код просто не компилируется. Конечно, проблему можно решить, используя приведение типов:

```
class Dict<K,V> {
    public void Add(K key, V value) {
        . . .
        if(((IComparable)key).CompareTo(x) < 0) {...}
        . . .
    }
}
```

Недостаток данного решения – необходимость приведения типов. К тому же, если тип `K` не поддерживает интерфейс `IComparable`, то при работе программы будет сгенерировано исключение `InvalidCastException`.

В C# для каждого параметра обобщенного типа может быть задан список *ограничений* (*constraints*). Только тип, удовлетворяющий ограничениям, может быть применен для записи сконструированного типа.

Ограничения объявляются с использованием ключевого слова `where`, после которого указывается параметр, двоеточие и список ограничения. Элементом списка ограничения могут являться интерфейсы, класс (только один) и ограничение на конструктор. Для класса `Dict<K,V>` можно установить ограничение на параметр `K`, гарантирующее, что тип `K` реализует `IComparable`.

```
class Dict<K,V> where K: IComparable
{
    public void Add(K key, V value) {
        . . .
        if(key.CompareTo(x) < 0) {...}
        . . .
    }
}
```

Компилятор будет проверять соблюдение ограничения при создании сконструированного типа. Кроме этого, отпадает необходимость в выполнении приведения типов в теле класса `Dict<K,V>`.

В следующем примере используется несколько ограничений на различные параметры типа:

```
class EntityTable<K,E>
    where K: IComparable<K>, IPersistable
    where E: Entity, new()
{
    public void Add(K key, E entity) {
        . . .
        if (key.CompareTo(x) < 0) {...}
        . . .
    }
}
```

Смысл ограничений, наложенных на параметр `E`: он должен быть приводим к классу `Entity` и иметь `public`-конструктор без параметров.

Порядок элементов в списке ограничений имеет значение в C#. Если есть элемент-класс, то он должен быть первым в списке. Если есть элемент-конструктор, то его надо помещать последним.

В некоторых случаях достаточно параметризовать не весь пользовательский тип, а только конкретный метод. *Обобщенные методы* (*generic methods*) объявляются с использованием параметра типа в угловых скобках после имени метода.

```
void PushMultiple<T>(Stack<T> stack, params T[] values) {
    foreach (T value in values) stack.Push(value);
}
```

Использование обобщенного метода `PushMultiple<T>` позволяет работать с любым сконструированным типом на основе `Stack<T>`.

```
Stack<int> stack = new Stack<int>();
PushMultiple<int>(stack, 1, 2, 3, 4);
```

Для обобщенных методов, подобных `PushMultiple<T>`, компилятор способен самостоятельно установить значение параметра типа на основе фактических параметров метода. Это позволяет записывать вызов метода без указания типа:

```
Stack<int> stack = new Stack<int>();
// Так как stack - объект Stack<int>, то используем тип int
PushMultiple(stack, 1, 2, 3, 4);
```

Как и при описании типов, обобщенные методы могут содержать ограничения на параметр-тип:

```
public T Max<T>(T val1, T val2) where T: IComparable {
    T retVal = val2;
    if (val2.CompareTo(val1) < 0) retVal = val1;
    return retVal;
}
```

В язык C# было введено новое ключевое слово `default` и соответствующая операция. Эта операция может применяться в тех методах, где возвращаемое значение задано как параметр типа. Как следует из названия, операция возвращает соответствующее значение по умолчанию для переменной некоторого типа.

```
class MyCache<K, V> {
    . . .
    // Метод для поиска элемента по ключу
    // Если элемент найден, то метод возвращается его
    // Иначе метод возвращает значение по умолчанию для V
    public V LookupItem(K key) {
        V retVal;
        if (ContainsKey(key))
            retVal = GetValue(key);
        else
            retVal = default(V);
        return retVal;
    }
}
```

Имеющиеся в .NET Framework обобщенные типы для представления структур данных обсуждаются ниже.

2. БАЗОВЫЕ ЭЛЕМЕНТЫ .NET FRAMEWORK

2.1. МЕТАДАННЫЕ И МЕХАНИЗМ ОТРАЖЕНИЯ

При программировании для платформы .NET в любую сборку помещаются *метаданные*, которые являются описанием всех типов данных сборки и их членов. Работа с метаданными происходит при помощи специального механизма, называемого *механизмом отражения* (*reflection*).

Главные элементы, которые необходимы для использования возможностей отражения – это класс `System.Type` и типы из пространства имен `System.Reflection`.

Класс `System.Type` служит для получения информации о типе. Для того чтобы получить объект данного класса, существует несколько возможностей.

1. Вызвать у объекта метод `GetType()`. Данный метод определен на уровне `System.Object`, а значит присутствует у любого объекта:

```
Foo A = new Foo();          //Foo – это некий класс
Type t = A.GetType();
```

2. Использовать статический метод `Type.GetType()`, которому передается текстовое имя интересующего нас типа:

```
Type t;
t = Type.GetType("Foo");
```

3. Использовать ключевое слово языка C# `typeof`:

```
Type t = typeof(Foo);
```

Чтобы продемонстрировать возможности класса `Type`, опишем вспомогательный класс `Foo`.

```
public class Foo {
    private int Field1;
    public float Field2;
    private int Method1() {
        Console.WriteLine("Method1 is called");
        return 0;
    }
    public void Method2(string s, ref int i) {
        Console.WriteLine("Method2 is called");
        Console.WriteLine("First parameter is " + s);
        Console.WriteLine("Second parameter is " + i);
    }
    public int Prop {
        get { return Field1; }
        set { Field1 = value; }
    }
}
```

А теперь рассмотрим код, который выводит информацию об элементах типа `Foo` (для использования типов, подобных `FieldInfo`, необходимо подключить пространство имен `System.Reflection`).

```
Type t = typeof(Foo);

//Это общая информация
Console.WriteLine("Full name = " + t.FullName);
Console.WriteLine("Base is = " + t.BaseType);
Console.WriteLine("Is abstract = " + t.IsAbstract);
Console.WriteLine("Is sealed = " + t.IsSealed);
Console.WriteLine("Is class = " + t.IsClass);
Console.WriteLine("******************************");

//Сейчас пройдемся по полям
FieldInfo[] fi = t.GetFields();
foreach(FieldInfo f in fi)
    Console.WriteLine("Field = " + f.Name);
Console.WriteLine("******************************");

//А теперь по свойствам
PropertyInfo[] pi = t.GetProperties();
foreach(PropertyInfo p in pi)
    Console.WriteLine("Property = " + p.Name);
Console.WriteLine("******************************");

//С методами поработаем подробнее
MethodInfo[] mi = t.GetMethods();
foreach(MethodInfo m in mi) {
    Console.WriteLine("Method Name = " + m.Name);
    Console.WriteLine("Method Return Type = " + m.ReturnType);

    //Изучим параметры метода
    ParameterInfo[] pri = m.GetParameters();
    foreach(ParameterInfo pr in pri) {
        Console.WriteLine("Parameter Name = " + pr.Name);
        Console.WriteLine("Type = " + pr.ParameterType);
    }
    Console.WriteLine("******************************");
}
```

Данный код выводит следующую информацию:

```
Full name = refl.Foo
Base is = System.Object
Is abstract = False
Is sealed = False
Is class = True
******************************
Field = Field2
******************************
Property = Prop
******************************
Method Name = GetHashCode
Method Return Type = System.Int32
******************************
Method Name = Equals
```

```
Method Return Type = System.Boolean
Parameter Name = obj
Parameter Type = System.Object
*******************************
Method Name = ToString
Method Return Type = System.String
*******************************
Method Name = Method2
Method Return Type = System.Void
Parameter Name = s
Parameter Type = System.String
Parameter Name = i
Parameter Type = System.Int32&
*******************************
Method Name = get_Prop
Method Return Type = System.Int32
*******************************
Method Name = set_Prop
Method Return Type = System.Void
Parameter Name = value
Parameter Type = System.Int32
*******************************
Method Name = GetType
Method Return Type = System.Type
*******************************
```

Обратите внимание, что была получена информация только об открытых членах класса `Foo`. Кроме этого, информация включала описание собственных и унаследованных элементов класса `Foo`. Пространство имен `System.Reflection` содержит специальное перечисление `BindingFlags`, которое позволяет управлять получаемой о типе информацией.

Таблица 7

Элементы перечисления `BindingFlags`

Элемент `BindingFlags`	**Описание**
`Default`	Поиск по умолчанию
`IgnoreCase`	Поиск, не чувствительный к регистру
`DeclaredOnly`	Игнорировать унаследованные члены
`Instance`	Поиск экземплярных членов
`Static`	Поиск статических членов
`Public`	Поиск открытых членов
`NonPublic`	Поиск внутренних членов
`FlattenHierarchy`	Поиск статических членов, заданных в базовых типах

Методы класса `Type`, подобные `GetFields()`, имеют перегруженные версии, в которых одним из параметров выступает набор флагов `BindingFlags`. Изменим приведенный выше код, чтобы получать информацию об открытых и внутренних методах, определенных в самом классе `Foo`:

```
BindingFlags bf = BindingFlags.DeclaredOnly |
                  BindingFlags.Public | BindingFlags.NonPublic |
```

```
                    BindingFlags.Static | BindingFlags.Instance;
MethodInfo[] mi = t.GetMethods(bf);
// Далее по тексту примера...
```

Некоторые типы пространства `System.Reflection`, которые могут быть полезны при работе с метаданными, перечисдены в таблице 8.

Таблица 8

Избранные типы пространства имен `System.Reflection`

Тип	Назначение
`Assembly`	Класс для загрузки сборки, изучения ее состава и выполнения операций со сборкой
`AssemblyName`	Класс для получения идентификационной информации о сборке
`EventInfo`	Хранит информацию о событии
`FieldInfo`	Хранит информацию о поле
`MemberInfo`	Абстрактный класс для классов вида `*Info`
`MethodInfo`	Хранит информацию о методе
`Module`	Позволяет обратиться к модулю в многофайловой сборке
`ParameterInfo`	Хранит информацию о параметре метода
`PropertyInfo`	Хранит информацию о свойстве

При помощи класса `Assembly` можно получить информацию обо всех модулях сборки, затем при помощи класса `Module` получить информацию обо всех типах модуля, далее вывести информацию для отдельного типа. Кроме этого, метод `Assembly.Load()` позволят динамически загрузить определенную сборку в память во время работы приложения. Подобный подход называется *поздним связыванием*.

Для демонстрации позднего связывания поместим класс `Foo` в динамическую библиотеку с именем `FooLib.dll`. Основное приложение будет загружать данную библиотеку и исследовать ее типы:

```
using System;
using System.Reflection;
using System.IO;          //Нужно для FileNotFoundException

class MainClass {
    public static void Main() {
        Assembly A = null;
        try {
            //Используется текстовое имя без расширения
            //Файл FooLib.dll находится в директории программы
            A = Assembly.Load("FooLib");
        } catch (FileNotFoundException e) {
            Console.WriteLine(e.Message);
            return;
        }
        foreach(Module M in A.GetModules())
            foreach(Type T in M.GetTypes())
                // Выводить особо нечего - одна строке Foo
                Console.WriteLine(T.Name);
    }
}
```

Позднее связывание не ограничивается загрузкой сборки и изучением состава ее элементов. При помощи позднего связывания можно создать объекты типов, определенных в сборке, а также работать с элементами созданных объектов (например, вызывать методы). Для создания определенного объекта используется метод `CreateInstance()` класса `System.Activator`. Существует несколько перегруженных версий данного метода. Можно использовать вариант, при котором в качестве параметра метода используется объект `Type` или строка-имя типа. Метод возвращает значение типа `object`. Класс `MethodInfo` имеет метод `Invoke()`, который позволяет вызвать метод объекта. Первый параметр метода `Invoke()` – это тот объект, у которого вызывается метод, второй параметр – массив объектов, представляющих параметры метода.

Представим листинг приложения, которое пытается интерактивно создавать объекты определенных типов и вызывать их методы (для простоты полностью отсутствует обработка исключительных ситуаций):

```
using System;
using System.Reflection;
using System.IO;

class MainClass {
    public static void Main() {
        //Просим пользователя ввести имя сборки и считываем ее
        Console.Write("Enter the name of assembly: ");
        string AssemblyName = Console.ReadLine();
        Assembly A = Assembly.Load(AssemblyName);

      //Перечисляем все типы в сборке
      //(вернее, проходим по модулям сборки и ищем типы в них)
   Console.WriteLine("Assembly {0} has types:", AssemblyName);
        foreach(Module M in A.GetModules())
            foreach(Type T in M.GetTypes())
                Console.WriteLine(T.Name);

        //Просим пользователя ввести имя типа
        Console.Write("Enter namespace.typename: ");
        string TypeName = Console.ReadLine();

        //Запрашиваем указанный тип и создаем его экземпляр
        Type UserType = A.GetType(TypeName);
        object obj = Activator.CreateInstance(UserType);

        //Перечисляем методы типа (с применением BindingFlags)
        Console.WriteLine("Type {0} has methods:", TypeName);
        BindingFlags bf = BindingFlags.DeclaredOnly |
                          BindingFlags.Public |
                          BindingFlags.NonPublic |
                          BindingFlags.Static |
                          BindingFlags.Instance;
        foreach(MethodInfo m in UserType.GetMethods(bf))
            Console.WriteLine("Method Name = " + m.Name);
```

```
            //Просим пользователя ввести имя метода
            Console.Write("Enter the name of method to call: ");
            string MethodName = Console.ReadLine();

            //Запрашиваем метод и выводим список его параметров
            MethodInfo Method = UserType.GetMethod(MethodName);
            Console.WriteLine("Method {0} has parameters:",
                                                    MethodName);
            foreach(ParameterInfo pr in Method.GetParameters()) {
                Console.WriteLine("Parameter Name = " + pr.Name);
                Console.WriteLine("Parameter Type = " +
                                                pr.ParameterType);
                Console.WriteLine("****************************");
            }

            // Создаем пустой массив для фактических параметров
            object[] paramArray =
                        new object[Method.GetParameters().Length];
            //Вызываем метод
            Method.Invoke(obj, paramArray);
        }
    }
```

2.2. ПОЛЬЗОВАТЕЛЬСКИЕ И ВСТРОЕННЫЕ АТРИБУТЫ

Помимо метаданных в платформе .NET представлена система атрибутов. *Атрибуты* позволяют определить дополнительную информацию, связанную с элементом метаданных и предоставляют механизм для обращения к этой информации в ходе выполнения программы. Следовательно, атрибуты позволяют динамически изменять поведение программы на основе анализа этой дополнительной информации.

Все атрибуты можно разделить на четыре группы:

1. *Атрибуты, используемые компилятором*. Информация, предоставляемая этими атрибутами, используется компилятором для генерации конечного кода (атрибуты выступают как своеобразные *директивы компилятора*).

2. *Атрибуты, используемые средой исполнения*.

3. *Атрибуты, используемые библиотекой классов*. Атрибуты этого вида используются в служебных целях классами, входящими в состав стандартной библиотеки.

4. *Пользовательские атрибуты*. Это атрибуты, созданные программистом.

Остановимся подробно на процессе создания пользовательского атрибута. Любой атрибут (в том числе и пользовательский) является классом. К классу атрибута предъявляются следующие требования: он должен быть потомком класса `System.Attribute`, имя класса должно заканчиваться суффиксом `Attribute`[1], атрибут должен иметь `public`-конструктор, на тип параметров

[1] Данные требования не продиктованы исполняющей средой, но заложены в компиляторы языков для .NET.

конструктора атрибута, а также на тип его открытых полей и свойств наложены ограничения (тип может быть не произвольным, а только типом из определенного набора).

Допустим, мы решили создать атрибут, предназначенный для указания автора и даты создания некоторого элемента кода. Для этого опишем следующий класс:

```
using System;
public class AuthorAttribute : Attribute {
    private string fName;
    private string fDate;
    public AuthorAttribute(string name) {
        fName = name;
    }
    public string Name {
        get { return fName; }
    }
    public string CreationDate {
        get { return fDate; }
        set { fDate = value;}
    }
}
```

Данный класс можно скомпилировать и поместить в отдельную сборку (оформленную в виде DLL) или отдельный модуль (`.netmodule`).

Далее мы можем применить созданный атрибут к произвольному классу:

```
using System;
//Применяем атрибут
//(Не забудьте добавить ссылку на сборку с атрибутом!)
[AuthorAttribute("Volosevich")]
class MainClass {
. . .
}
```

Рассмотрим *синтаксис использования атрибутов* подробнее. Атрибуты записываются в квадратных скобках. Можно записать несколько атрибутов через запятую в виде списка. Список атрибутов должен находиться перед тем элементом, к которому этот список применяется. Если возникает неоднозначность трактовки применения атрибута, то возможно использование специальных модификаторов – `assembly`, `module`, `field`, `event`, `method`, `param`, `property`, `return`, `type`. Например, запись вида `[assembly: Имя_атрибута]` означает применение атрибута к сборке. Если атрибут применяется к сборке или модулю, то он должен быть записан после секций импортирования `using`, но перед основным кодом. После имени атрибута указываются в круглых скобках параметры конструктора атрибута. Если конструктор атрибута не имеет параметров, круглые скобки можно не указывать. Для сокращения записи разрешено указывать имя атрибута без суффикса `Attribute`.

Наряду с параметрами конструктора при применении атрибута можно указать *именованные параметры*, предназначенные для задания значения открытого поля или свойства. При этом используется синтаксис

Имя_поля_или_свойства = Значение-константа. Именованные параметры всегда записываются в конце списка параметров. Если некое поле или свойство инициализируется через параметр конструктора и через именованный параметр, то конечное значение элемента – это значение именованного параметра.

Таким образом, описанный нами ранее атрибут может быть использован для класса в следующем виде:

```
[Author("Volosevich", CreationDate = "18.03.2005")]
class MainClass {
. . .
}
```

Ранее упоминалось, что на тип параметров конструктора атрибута, а также на тип его открытых полей и свойств наложены определенный ограничения. Тип параметров, указываемых при использовании атрибута, ограничен следующим набором: типы bool, byte, char, short, int, long, float, double, string; тип System.Type; перечисления; тип object (фактическим параметром в этом случае должна быть константа одного из типов, перечисленного выше); одномерные массивы перечисленных выше типов. Этот набор ограничивает типы для открытых свойств и полей класса атрибута. Закрытые элементы атрибута могут иметь произвольный тип.

При создании пользовательских атрибутов полезным оказывается использование специального класса AttributeUsageAttribute. Как видно из названия, это атрибут, который следует применить к пользовательскому классу атрибута. Конструктор класса AttributeUsageAttribute принимает единственный параметр, определяющий область действия пользовательского атрибута. Допустимые значения параметра перечислены в таблице 9.

Таблица 9

Значения параметра конструктора класса AttributeUsageAttribute

Значение параметра	Атрибут может применяться к …
AttributeTargets.All	любому элементу, указанному далее
AttributeTargets.Assembly	сборке
AttributeTargets.Class	классу
AttributeTargets.Constructor	конструктору
AttributeTargets.Delegate	делегату
AttributeTargets.Enum	перечислению
AttributeTargets.Event	событию
AttributeTargets.Field	полю
AttributeTargets.Interface	интерфейсу
AttributeTargets.Method	методу
AttributeTargets.Module	модулю
AttributeTargets.Parameter	параметру метода
AttributeTargets.Property	свойству
AttributeTargets.ReturnValue	возвращаемому значению функции
AttributeTargets.Struct	структуре

Свойство AllowMultiple класса AttributeUsageAttribute определяет, может ли быть атрибут применен к программному элементу более одного раза. Тип данного свойства – bool, значение по умолчанию – false.

Свойство Inherited класса AttributeUsageAttribute определяет, будет ли атрибут проецироваться на потомков программного элемента. Тип данного свойства – bool, значение по умолчанию – true.

Используем возможности класса AttributeUsageAttribute при описании пользовательского атрибута:

```
// Атрибут Author можно применить к классу или методу,
// причем несколько раз
[AttributeUsage(AttributeTargets.Class |
                AttributeTargets.Method,
                AllowMultiple = true)]
public class AuthorAttribute : Attribute {
. . .
}
```

Опишем возможности получения информации об атрибутах. Для этой цели можно использовать метод GetCustomAttribute() класса System.Attribute. Имеется несколько перегруженных версий данного метода. Мы рассмотрим только одну из версий:

```
public static Attribute GetCustomAttribute(MemberInfo element,
                                           Type attributeType)
```

При помощи параметра element задается требуемый элемент, у которого надо получить атрибут. Второй параметр – это тип получаемого атрибута. Возвращаемое функцией значение обычно приводится к типу получаемого атрибута. Рассмотрим следующий пример:

```
using System;

[Author("Volosevich", CreationDate = "18.03.2005")]
class SomeClass {
    . . .
}

class MainClass {
    public static void Main() {
        Attribute A = Attribute.GetCustomAttribute(
                                  typeof(SomeClass),
                                  typeof(AuthorAttribute));
        if (A != null)
            Console.WriteLine(((AuthorAttribute)A).Name);
    }
}
```

В данном примере к классу SomeClass был применен пользовательский атрибут AuthorAttribute. Затем в методе другого класса этот атрибут был прочитан, и из него извлеклась информация.

Следует иметь в виду, что объект, соответствующий классу атрибута, создается исполняющей средой только в тот момент, когда из атрибута извлекается информация. Задание атрибута перед некоторым элементом к созданию объекта не приводит. Количество созданных экземпляров атрибута равно количеству запросов к данным атрибута[1].

Метод `Attribute.GetCustomAttributes()` позволяет получить все атрибуты некоторого элемента в виде массива объектов. Одна из перегруженных версий данного метода описана следующим образом:

```
public static Attribute[] GetCustomAttributes(MemberInfo element);
```

Модифицируем предыдущий пример, использовав `GetCustomAttributes`:

```
using System;

[Author("Volosevich"), Author("Ivanov")]
class SomeClass {
    . . .
}

class MainClass {
    public static void Main() {
        Attribute[] A;
        A = Attribute.GetCustomAttributes(typeof(SomeClass));
         for(int i = 0; i < A.Length; i++)
            if(A[i] is AuthorAttribute)
              Console.WriteLine(((AuthorAttribute)A[i]).Name);
   }
}
```

Платформа .NET предоставляет для использования обширный набор атрибутов, некоторые из которых представлены в таблице 10:

Таблица 10

Некоторые атрибуты, применяемые в .NET Framework

Имя атрибута	Область применения	Описание
`AttributeUsage`	Класс	Задает область применения класса-атрибута
`Conditional`	Метод	Компилятор может игнорировать вызовы помеченного метода при заданном условии
`DllImport`	Метод	Указывает DLL, содержащую реализацию метода
`MTAThread`	Метод (`Main`)	Для приложения используется модель COM Multithreaded apartment
`NonSerialized`	Поле	Указывает, что поле не будет сериализовано
`Obsolete`	Любая, исключая `assembly`, `module`, `param`, `return`	Информирует, что в будущих реализациях данный элемент может отсутствовать
`ParamArray`	Параметр	Позволяет одиночному параметру быть обработанным как набор параметров `params`

[1] Если бы создавался только один объект, соответствующий атрибуту, то изменение данных в нем отразилось бы на всех запросах к атрибуту.

Serializable	Класс, структура, перечисление, делегат	Указывает, что все поля типа могут быть сериализованы
STAThread	Метод (Main)	Для приложения используется модель COM Single-threaded apartment
StructLayout	Класс, структура	Задает схему размещения данных класса или структуры в памяти (Auto, Explicit, Sequential)
ThreadStatic	Статическое поле	В каждом потоке будет использоваться собственная копия данного статического поля

Атрибут DllImport предназначен для импортирования функций из библиотек динамической компоновки, написанных на «родном» языке процессора. В следующем примере данный атрибут используется для импортирования системной функции MessageBoxA():

```
using System;
using System.Runtime.InteropServices;

class MainClass {
    [DllImport("user32.dll")]
    public static extern int MessageBoxA(int h, string m,
                                         string c, int type);
    public static void Main() {
        MessageBoxA(0, "Hello World", "nativeDLL", 0);
    }
}
```

Обратите внимание, что для использования атрибута DllImport требуется подключить пространство имен System.Runtime.InteropServices. Кроме это, необходимо объявить импортируемую функцию статической и пометить ее модификатором extern. Атрибут DllImport допускает использование дополнительных параметров, подробное описание которых можно найти в документации.

Исполняемая среда .NET выполняет корректную передачу параметров примитивных типов между управляемым и неуправляемым кодом. Для правильной передачи параметров-структур требуется использование специального атрибута StructLayout при объявлении пользовательского типа, соответствующего структуре. Например, пусть выполняется экспорт системной функции GetLocalTime():

```
[DllImport("kernel32.dll")]
public static extern void GetLocalTime(SystemTime st);
```

В качестве параметра используется объект класса SystemTime. Этот класс должен быть описан следующим образом:

```
[StructLayout(LayoutKind.Sequential)]
public class SystemTime {
    public ushort wYear;
    public ushort wMonth;
    public ushort wDayOfWeek;
    public ushort wDay;
    public ushort wHour;
    public ushort wMinute;
```

```
    public ushort wSecond;
    public ushort wMilliseconds;
}
```

Атрибут `StructLayout` указывает, что поля объекта должны быть расположены в памяти в точности так, как это записано в объявлении класса (`LayoutKind.Sequential`). В противном случае при работе с системной функцией возможно возникновение ошибок.

Атрибут `Conditional` (описан в пространстве имен `System.Diagnostics`) может быть применен к любому методу, возвращающему значение `void`. Параметром атрибута является строковый литерал. Применение атрибута указывает компилятору, что вызовы помеченного метода следует опускать, если в проекте не определен строковый литерал. Задание литерала производится либо директивой препроцессора `#define` в начале текста программы (`#define D`), либо параметром компилятора `/define:<symbol list>`, либо в диалогах настройки интегрированной среды.

Рассмотрим пример использования атрибута `Conditional`:

```
using System;
using System.Diagnostics;

class MainClass {
    [Conditional("D")]
    public static void SomeDebugFunc() {
        Console.WriteLine("SomeDebugFunc");
    }
    public static void Main() {
        SomeDebugFunc();
        Console.WriteLine("Hello!");
    }
}
```

Если в проекте не определен параметр `D` (а так, естественно, и будет по умолчанию), вызова метода `SomeDebugFunc()` не произойдет. В принципе, подобный эффект достигается использованием *директив препроцессора*:

```
public static void Main() {
    #if D
    SomeDebugFunc();
    #endif
    Console.WriteLine("Hello!");
}
```

В отличие от директив препроцессора, которыми требуется обрамлять каждый вызов метода, атрибут `Conditional` достаточно указать в одном месте.

Атрибут `Obsolete` используется для того, чтобы пометить элемент как *устаревший*. При записи атрибута может указывается строка-сообщение, выводимая компилятором, если устаревший элемент применяется в коде. Рассмотрим следующий пример:

```
using System;
class MainClass {
```

```
    [Obsolete(@"Old Function - Don't use!")]
    public static void SomeFunc() {
        Console.WriteLine("SomeFunc");
    }
    public static void Main() {
        SomeFunc();
        Console.WriteLine("Hello!");
    }
}
```

При компиляции данного проекта будет выведено предупреждение:

```
'MainClass.SomeFunc()' is obsolete:'Old Function - Don't use!'
```

Перегруженная версия конструктора атрибута `Obsolete` может кроме строки принимать булевый параметр. Если значение параметра `true`, то при компиляции кода с использованием устаревшего элемента генерируется не просто предупреждение, а ошибка:

```
[Obsolete("Old Function - Do not use it!", true)]
public static void SomeFunc() {Console.WriteLine("SomeFunc");}

public static void Main() {
    //Теперь эта строка просто не компилируется!
    SomeFunc();
    Console.WriteLine("Hello!");
}
```

Для пользовательского проекта некоторые среды программирования (MS Visual Studio .NET, SharpDevelop) генерируют специальный файл `AssemblyInfo.cs`. Этот файл содержит код, автоматически встраиваемый в сборку. Фактически, файл представляет собой набор атрибутов сборки. Вот пример данного файла (с удаленными комментариями):

```
using System.Reflection;
using System.Runtime.CompilerServices;

[assembly: AssemblyTitle("")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCopyright("")]
[assembly: AssemblyTrademark("AlexV")]
[assembly: AssemblyCulture("")]
[assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("")]
```

Автор проекта может записать собственные значения для параметров атрибутов сборки. В этом случае при просмотре свойств файла в Windows эти значения будут отображаться в виде строк на соответствующей закладке.

2.3. ПРОСТРАНСТВО ИМЕН SYSTEM.COLLECTIONS

Пространство имен `System.Collections` содержит классы и интерфейсы, которые служат для поддержки наборов (или *коллекций*) данных, организованных в виде стандартных структур данных – список, хэш-таблица, стек и т. д. Кроме этого, некоторые интерфейсы из описываемого пространства имен удобны для использования при сортировках, перечислении элементов массивов и структур. В данном параграфе будут описаны базовые приемы работы с элементами из пространства имен `System.Collectons`, а также пространства имен `System.Collections.Specialized` и `System.Collectons.Generics`.

Рассмотрим интерфейсы из пространства имен `System.Collections` и дадим их краткое описание.

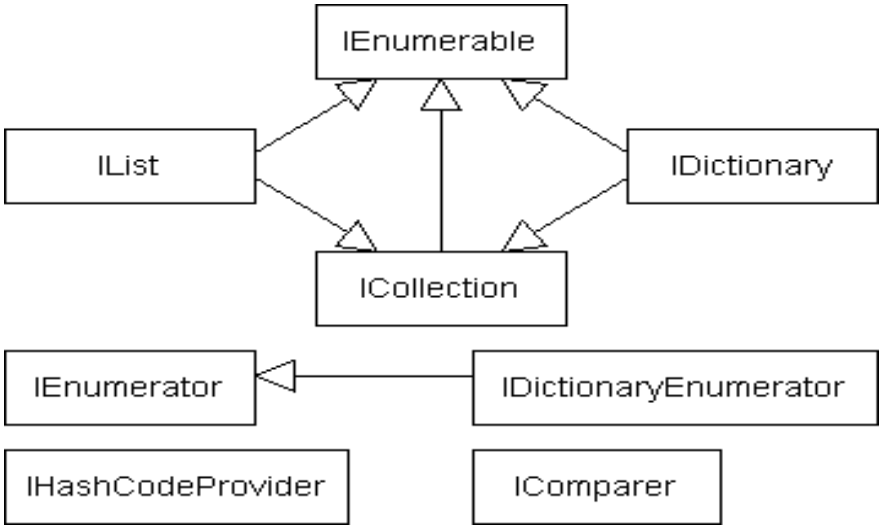

Рис. 4. Интерфейсы из `System.Collections`

Интерфейс `IEnumerator` представляет поддержку для набора объектов возможности перебора в цикле `foreach`.

Интерфейс `ICollection` служит для описания некоторого набора объектов. Этот интерфейс имеет свойство `Count` для количества элементов в наборе, а также метод `CopyTo()` для копирования набора в массив `Array` (`CopyTo`).

Интерфейс `IList` описывает набор данных, которые проецируются на масив. Приведем описание данного интерфейса, назначение его методов и свойств очевидно:

```
interface IList : ICollection, IEnumerable {
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object this[int index] { get; set; }
    int Add(object value);
    void Clear();
    bool Contains(object value);
    int IndexOf(object value);
    void Insert(int index, object value);
    void Remove(object value);
    void RemoveAt(int index);
}
```

Интерфейс IDictionary служит для описания коллекций, представляющих словари, то есть таких наборов данных, где доступ осуществляется с использованием произвольного ключа.

```
interface IDictionary : ICollection, IEnumerable {
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    ICollection Keys { get; }
    object this[object key]{ set; get; }
    ICollection Values { get; }
    void Add(object key, object value);
    void Clear();
    bool Contains(object key);
    IDictionaryEnumerator GetEnumerator();
    void Remove(object key);
}
```

Интерфейс IEnumerator предназначен для организации перебора элементов коллекции, а интерфейс IDictionaryEnumerator кроме перебора элементов позволяет обратиться к ключу и значению текущего элемента. Назначение интерфейсов IHashCodeProvider и IComparer разъясняется далее по тексту.

На протяжении параграфа будем использовать следующий вспомогательный класс, хранящий информацию о студенте:

```
class CStudent {
    public string name;
    public int course;
    public double ball;

    public CStudent(string name, int course, double ball) {
        this.name = name;
        this.course = course;
        this.ball = ball;
    }

    public override string ToString() {
       return String.Format("Name={0,10}; Course={1,3}; Ball={2,4}",
                            name, course, ball);
    }
}
```

Имея в своем распоряжении класс CStudent, мы можем организовать массив из объектов данного класса. Однако методы сортировки из класса Array для такого массива работать не будут. Для того чтобы выполнялась сортировка, необходимо, чтобы класс реализовывал интерфейс System.IComparable.

Интерфейс IComparable определен следующим образом:

```
interface IComparable {
    int CompareTo(object o);
}
```

Метод CompareTo() сравнивает текущий объект с объектом o. Метод должен возвращать ноль, если объекты «равны», любое число больше нуля, если

текущий объект «больше», и любое число меньше нуля, если текущий объект «меньше» сравниваемого.

Пусть мы хотим сортировать массив студентов по убыванию среднего балла. Внесем поддержку `IComparable` в класс `CStudent`:

```
class CStudent : IComparable {
. . .
    public int CompareTo(object o) {
        CStudent tmp = (CStudent) o;
        if(ball > tmp.ball) return 1;
        if(ball < tmp.ball) return -1;
        return 0;
    }
}
```

Класс `CStudent` теперь можно использовать таким образом:

```
CStudent[] Group = {new CStudent("Alex", 1, 7.0),
                    new CStudent("Ivan", 1, 9.0),
                    new CStudent("Anna", 1, 8.0),
                    new CStudent("Peter", 1, 5.0)};
Console.WriteLine("Unsorted group");
foreach (CStudent s in Group) {
    Console.WriteLine(s);
}

Array.Sort(Group);
Array.Reverse(Group);

Console.WriteLine("Sorted group");
foreach (CStudent s in Group) {
    Console.WriteLine(s);
}
```

Данная программа выводит следующее:

```
Unsorted group
Name =       Alex; Course =   1; Ball =    7
Name =       Ivan; Course =   1; Ball =    9
Name =       Anna; Course =   1; Ball =    8
Name =      Peter; Course =   1; Ball =    5
Sorted group
Name =       Ivan; Course =   1; Ball =    9
Name =       Anna; Course =   1; Ball =    8
Name =       Alex; Course =   1; Ball =    7
Name =      Peter; Course =   1; Ball =    5
```

Интерфейс `IComparer` из пространства имен `System.Collections` предоставляет стандартизированный способ сравнения любых двух объектов:

```
interface IComparer {
    int Compare(object o1, object o2);
}
```

Использование IComparer позволяет осуществить сортировку по нескольким критериям. Для этого каждый критерий описывается вспомогательным классом, реализующим IComparer и определенный способ сравнения.

Возьмем за основу первоначальный вариант класса CStudent и опишем два вспомогательных класса следующим образом:

```
class SortStudentsByBall : IComparer {
    public int Compare(object o1, object o2) {
        CStudent t1 = (CStudent) o1;
        CStudent t2 = (CStudent) o2;
        if(t1.ball > t2.ball) return 1;
        if(t1.ball < t2.ball) return -1;
        return 0;
    }
}
class SortStudentsByName : IComparer {
    public int Compare(object o1, object o2) {
        CStudent t1 = (CStudent) o1;
        CStudent t2 = (CStudent) o2;
        return String.Compare(t1.name, t2.name);
    }
}
```

Теперь для сортировки по разным критериям можно использовать перегруженную версию метода Array.Sort(), принимающую в качестве второго параметра объект класса, реализующего интерфейс IComparer.

```
// Сортировка массива студентов по баллу
Array.Sort(Group, new SortStudentsByBall());
// Сортировка массива студентов по имени
Array.Sort(Group, new SortStudentsByName());
```

Язык C# имеет команду для перебора элементов коллекций – foreach. Пусть мы создали класс, хранящий некий набор элементов:

```
class Room {
    private CStudent[] H;
    // Создаем в конструкторе "комнату" - не лучшее решение!
    public Room() {
        H = new CStudent[3];
        H[0] = new CStudent("Alex", 1, 7.0);
        H[1] = new CStudent("Ivan", 1, 9.0);
        H[2] = new CStudent("Peter", 1, 5.0);
    }
}
```

Было бы удобно для перебора элементов в объектах нашего класса использовать команду foreach. Для этого необходимо организовать в классе поддержку интерфейса IEnumerable[1]. Интерфейс IEnumerable устроен очень просто. Он содержит единственный метод, задача которого – вернуть интерфейс IEnumerator.

[1] В C# 2.0 поддержка перебора реализуется при помощи итераторов.

```
interface IEnumerable {
    IEnumerator GetEnumerator();
}
```

В свою очередь, интерфейс `IEnumerator` имеет следующее описание:

```
interface IEnumerator {
    // Передвинуть курсор данных на следующую позицию
    bool MoveNext();

    // Свойство для чтения – текущий элемент
    object Current { get; }

    // Установить курсор перед началом набора данных
    void Reset();
}
```

Очень часто интерфейсы `IEnumerable` и `IEnumerator` поддерживает один и тот же класс. Добавим поддержку данных интерфейсов в класс `Room`:

```
class Room : IEnumerable, IEnumerator {
    private CStudent[] H;

    // Внутреннее поле для курсора данных
    private int pos = -1;

    // Конструктор – описан выше
    public Room(){
    . . .
    }

    public IEnumerator GetEnumerator() {
        // Класс сам реализует IEnumerator
        return (IEnumerator)this;
    }

    public bool MoveNext() {
        if(pos < H.Length - 1) {
            pos++;
            return true;
        }
        else return false;
    }

    public void Reset() {
        pos = -1;
    }

    public object Current {
        get { return H[pos]; }
    }
}
```

Теперь для перебора элементов класса Room мы можем использовать такой код:

```
Room R_1003 = new Room();
foreach(CStudent S in R_1003) {
    Console.WriteLine(S);
}
```

В таблице 11 перечислен набор основных классов, размещенных в пространстве имен System.Collections.

Таблица 11

Классы из System.Collections

Имя класса	Класс представляет…	Класс реализует интерфейсы…
ArrayList	Массив изменяемого размера с целочисленным индексом	IList, ICollection, IEnumerable,ICloneable
BitArray	Компактный массив бит с поддержкой битовых операций	ICollection, IEnumerable,ICloneable
CaseInsensitiveComparer	Класс с реализацией интерфейса IComparer, которая игнорирует регистр в случае сравнения строк	IComparer
CaseInsensitive-HashCodeProvider	Класс с методом генерации хэш-кодов, игнорирующим регистр в случае генерации кодов для строк	IHashCodeProvider
CollectionBase	Абстрактный базовый класс для построения пользовательских типизированных коллекций	IList, ICollection, IEnumerable
Comparer	Класс с реализацией интерфейса IComparer, основанной на вызове методов CompareTo() сравниваемых объектов	IComparer
DictionaryBase	Абстрактный класс для построения пользовательских типизированных хэш-таблиц	IDictionary, ICollection, IEnumerable
Hashtable	Таблицу объектных пар «ключ-значение», оптимизированную для быстрого поиска по ключу	IDictionary, ICollection, IEnumerable, ICloneable, IDeserializationCallback, ISerializable
Queue	Очередь (FIFO)	ICollection, IEnumerable,ICloneable
ReadOnlyCollection-Base	Абстрактный класс для построения пользовательских типизированных коллекций с доступом только для чтения	ICollection, IEnumerable
SortedList	Таблицу отсортированных по ключу пар «ключ-значение» (доступ к элементу – по ключу или по индексу)	IDictionary, ICollection, IEnumerable,ICloneable
Stack	Стек (LIFO)	ICollection, IEnumerable,ICloneable

Особенностью классов-коллекций является их *слабая типизация*. Элементами любой коллекции являются переменные типа `object`. Это позволяет коллекциям хранить данные любых типов (так как тип `object` – базовый для любого типа), однако вынуждает выполнять приведение типов при изъятии элемента из коллекции. Кроме этого, при помещении в коллекцию элемента структурного типа выполняется операция упаковки, что ведет к снижению производительности.

Класс `ArrayList` предназначен для представления динамических массивов, то есть массивов, размер которых изменяется при необходимости. Создание динамического массива и добавление в него элементов выполняется просто:

```
ArrayList list = new ArrayList();
list.Add("John");
list.Add("Paul");
list.Add("George");
list.Add("Ringo");
```

Метод `Add()` добавляет элементы в конец массива, автоматически увеличивая память, занимаемую объектом класса `ArrayList` (если это необходимо). Метод `Insert()` вставляет элементы в массив, сдвигая исходные элементы. Методы `AddRange()` и `InsertRange()` добавляют или вставляют в динамический массив произвольную коллекцию.

Для получения элементов динамического массива используется индексатор. Индексы – целые числа, начинающиеся с нуля. С помощью индексатора можно изменить значения существующего элемента[1]:

```
string st = (string)list[0];
list[0] = 999;
```

Свойство `Count` позволяет узнать количество элементов динамического массива. Для перебора элементов массива можно использовать цикл `foreach`:

```
for(int i=0; i<list.Count; i++)
    Console.WriteLine(list[i]);

// Можно так (правда, это примерно на 25% медленнее)
foreach(int i in list)
    Console.WriteLine(i);
```

Количество элементов динамического массива, которое он способен содержать без увеличения своего размера, называется *емкостью* массива. Емкость массива доступна через свойство `Capacity`, причем это свойство может быть как считано, так и установлено. По умолчанию создается массив емкостью 16 элементов. При необходимости `ArrayList` увеличивает емкость, удваивая ее. Если приблизительная емкость динамического массива известна, ее можно указать как параметр конструктора `ArrayList`. Это увеличит скорость вставки элементов.

---

[1] Попытка работать с несуществующим элементом генерирует исключение `ArgumentOutOfRangeException`.

Для удаления элементов массива предназначены методы `Remove()`, `RemoveAt()`, `Clear()`, `RemoveRange()`. Удаление элемента автоматически изменяет индексы оставшихся элементов. При удалении элементов емкость массива не уменьшается. Для того чтобы память массива соответствовала количеству элементов в нем, требуется использовать метод `TrimToSize()`:

```
// Создаем массив и добавляем элементы
ArrayList list = new ArrayList(1000);
for(int i=0; i<1000; i++)
    list.Add(i);

// Удаляем первые 500 элементов
list.RemoveRange(0, 500);

// Емкость массива по-прежнему 1000
Console.WriteLine(list.Capacity);

// Подгоняем размер массива под количество элементов
list.TrimToSize();

// Теперь его емкость 500
Console.WriteLine(Lst.Capacity)
```

Класс `Hashtable` является классом, реализующим хэш-таблицу. Следующий пример кода показывает использование этого класса для организации простого англо-русского словаря. В паре «ключ-значение» ключом является английское слово, а значением – русское:

```
Hashtable table = new Hashtable();
table.Add("Sunday",    "Воскресенье");
table.Add("Monday",    "Понедельник");
table.Add("Tuesday",   "Вторник");
table.Add("Wednesday", "Среда");
table.Add("Thursday",  "Четверг");
table.Add("Friday",    "Пятница");
table.Add("Saturday",  "Суббота");
```

Если хэш-таблица инициализирована указанным образом, то поиск русского эквивалента английского слова может быть сделан так (обратите внимание на приведение типов):

```
string s;
s = (string)table["Friday"];
```

Элементы могут быть добавлены в хэш-таблицу, используя индексатор:

```
Hashtable table = new Hashtable();
table["Sunday"] = "Воскресенье";
table["Monday"] = "Понедельник";
```

Если указываемый при добавлении ключ уже существует в таблице, метод `Add()` генерирует исключительную ситуацию. При использовании индексатора старое значение просто заменяется новым.

Элементы хэш-таблицы являются объектами класса DictionaryEntry. Этот класс имеет свойства для доступа к ключу (Key) и значению (Value). Класс Hashtable поддерживает интерфейс IEnumerable, что делает возможным перебор элементов с использованием foreach:

```
foreach (DictionaryEntry entry in table)
    Console.WriteLine("Key = {0}, Value = {1}",
                      entry.Key, entry.Value);
```

Класс Hashtable имеет методы для удаления элементов (Remove()), удаления всех элементов (Clear()), проверки существования элемента (ContainsKey() и ContainsValue()) и другие. Свойство Count содержит количество элементов хэш-таблицы, свойства Keys и Values позволяют перечислить все ключи и значения таблицы соответственно.

На скорость работы с элементам хэш-таблицы влияют два фактора: размер таблицы и уникальность хэш-кодов, продуцируемых ключами таблицы. Размер хэш-таблицы изменяется динамически, однако изменение размера требует затрат ресурсов. Класс Hashtable имеет конструктор, который в качестве параметра принимает предполагаемый размер таблицы в элементах.

Хэш-таблица захватывает дополнительную память, если ее заполненность превышает определенный предел, который по умолчанию равен 72% от емкости. Перегруженный конструктор класса Hashtable позволяет указать *множитель загрузки*. Это число в диапазоне от 0.1 до 1.0, на которое умножается 0.72, определяя тем самым новую максимальную заполненность:

```
// Будем захватывать память при заполнении 58% (72*0.8)
Hashtable tbl = new Hashtable(1000, 0.8);
```

По умолчанию хэш-таблица генерирует хэш-коды ключей, применяя метод GetHashCode() ключа. Все объекты наследуют данный метод от System.Object. Если объекты-ключи производят повторяющиеся хэши, это приводит к *коллизиям* в таблице, а, следовательно, к снижению производительности. В этом случае можно:

- переписать метод GetHashCode() для класса-ключа с целью усиления уникальности;
- создать тип, реализующий интерфейс IHashCodeProvider и передавать экземпляр такого типа конструктору таблицы. Интерфейс IHashCodeProvider содержит метод GetHashCode(), генерирующий хэши по входным объектам.

Для сравнения ключей таблица по умолчанию применяет метод Equals() объектов. Если сравнение реализовано не эффективно, требуется переписать метод Equals() или передать конструктору таблицы экземпляр класса, реализующего интерфейс IComparer.

В пространстве имен System.Collections определены два класса, которые можно использовать как основу для создания собственных типизированных коллекций-списков и коллекций-словарей. Это классы DictionaryBase и CollectionBase. Каждый из этих классов предоставляет ряд виртуальных методов, вызываемых при модификации коллекции. Например, OnClean() вызы-

вается, когда коллекция очищается; `OnInsert()` – перед добавлением элемента; `OnInsertComplete()` – после добавления; `OnRemove()` – перед удалением элемента и т. п. Переопределяя эти методы можно создать дополнительные проверки и вызвать исключение в случае ошибок типизации. Например, следующий класс представляет собой коллекцию, которая допускает хранение только строк из четырех символов:

```
using System;
using System.Collections;

class MyStringCollection : CollectionBase
{
    // Этот метод будет добавлять элементы
    // Он просто вызывает метод Add() внутреннего списка List
    // Метод не производит проверки типов!
    void Add(object value) {
        List.Add(value);
    }

    // Перекрываем виртуальный метод OnInsert()
    // В нем выполняем проверку
    protected override void OnInsert(int index, object value)
    {
        if ((value is string) && ((string)value).Length == 4))
        {
            base.OnInsert(index, value);
        }
        else throw new ArgumentException("Length 4 allowed!");
    }

    // Метод для получения элементов
    // В нем выполняем приведение типов
    string GetItem(int index) {
        return (string)List[index];
    }
}

class MainClass {

    static void Main() {
        MyStringCollection msc = new MyStringCollection();
        msc.Add("Alex");
        msc.Add("QWER");
        string s = msc.GetItem(1);
        Console.WriteLine(s);

        // Любая из следующих строк генерирует исключение
        msc.Add("asldalda");
        msc.Add(5);
    }
}
```

Для иллюстрации использования коллекций приведем пример приложения, подсчитывающего уникальные слова в текстовом файле. Имя текстового файла передается консольному приложению в качестве параметра. Вопросы, связанные с обработкой файлов, будут рассмотрены далее.

```
using System;
using System.IO;
using System.Collections;

class MyApp {
    static void Main (string[] args) {
        // Проверка параметров командной строки
        if (args.Length == 0) {
            Console.WriteLine ("Ошибка: нет имени файла");
            return;
        }

        // Создаем объект для доступа к строкам файла
        // и объект для хранения уникальных слов
        StreamReader reader = null;
        Hashtable table = new Hashtable();

        try {
            reader = new StreamReader(args[0]);

            // Перебираем все строки файла
            for (string line = reader.ReadLine();
                        line != null; line = reader.ReadLine())
            {
                // Этот массив хранит слова строки
                string[] words = GetWords(line);

                // Перебираем все слова
                foreach(string word in words) {
                    string iword = word.ToLower();
                    if(table.ContainsKey(iword))
                        // Если слово уже есть, счетчик + 1
                        table[iword] = (int)table[iword] + 1;
                    else
                        // Иначе помещаем слово в таблицу
                        table[iword] = 1;
                }
            }

            // Сортируем хэш-таблицу, используя SortedList
            SortedList list = new SortedList(table);

            // Выводим результат работы
            Console.WriteLine("Уникальных слов {0}",
                                              table.Count);
            foreach(DictionaryEntry ent in list)
                Console.WriteLine("{0} ({1})",
```

```
                                                ent.Key, ent.Value);
            }
            catch (Exception e) {
                Console.WriteLine(e.Message);
            }
            finally {
                if (reader != null)
                    reader.Close ();
            }
        }

        // Метод для выделения слов в строке
        static string[] GetWords(string line) {
            // Создадим ArrayList для промежуточных результатов
            ArrayList al = new ArrayList();

            // Разбираем строку на слова
            int i = 0;
            string word;
            char[] chars = line.ToCharArray();
            while ((word=GetNextWord(line, chars, ref i)) != null)
                al.Add(word);

            // Возвращаем статический массив,
            // эквивалентный динамическому
            string[] words = new string[al.Count];
            al.CopyTo(words);
            return words;
        }

        // Метод для получения следующего слова в строке
        static string GetNextWord(string line, char[] chars,
                                                ref int i) {
          // Находим начало следующего слова
          while(i<chars.Length && !Char.IsLetterOrDigit(chars[i]))
                i++;

            if (i == chars.Length)
                return null;

            int start = i;

            // Находим конец слова
           while(i<chars.Length && Char.IsLetterOrDigit(chars[i]))
                i++;

            // Возвращаем найденное слово
            return line.Substring(start, i - start);
        }
    }
```

Наряду с пространством имен System.Collections, .NET Framework предоставляет пространство имен System.Collections.Specialized. Оно содержит классы коллекций, которые подходят для решения специальных задач.

Таблица 12

Классы и структуры из System.Collections.Specialized

Имя класса (структуры)	Для чего служит	Реализует интерфейсы
BitVector32	Структура фиксированного размера (4 байта) для хранения массива булевых значений	
CollectionsUtil	Класс, разделяемые методы которого позволяют создавать коллекции, в которых игнорируется регистр строк	
HybridDictionary	Реализует интерфейс IDictionary с использованием ListDictionary для малых коллекций; автоматический переключается на использование Hashtable, когда размер коллекции увеличивается	IDictionary, ICollection, IEnumerable
ListDictionary	Реализует интерфейс IDictionary с использованием односвязного списка. Рекомендуется для коллекций с количеством элементов меньше 10	IDictionary, ICollection, IEnumerable
NameObjectCollectionBase	Абстрактный базовый класс для сортированных таблиц «ключ-значение», ключом которых является строка (доступ к элементу – по ключу или по индексу)	
NameValueCollection	Класс для сортированных таблиц «ключ-значение», ключом и значением в которых является строка (доступ к элементу – по ключу или по индексу)	
StringCollection	Представляет коллекцию строк	
StringDictionary	Класс для хэш-таблиц, ключом которых является строка	IEnumerable
StringEnumerator	Итератор для StringCollection	

Рассмотрим некоторые из классов, упомянутых в таблице. Класс StringCollection представляет реализацию класса ArrayList, которая может хранить только строки. Это обеспечивает дополнительный контроль при помещении объекта в коллекцию и избавляет от необходимости выполнять приведения типов. Класс StringDictionary представляет реализацию хэш-таблицы, в которой ключ и значение всегда имеют тип string. Кроме этого, ключ таблицы не зависит от регистра символов.

```
StringDictionary sd = new StringDictionary();
sd["Leonid"] = "Minchenko";
sd["Alexey"] = "Volosevich";
Console.WriteLine(sd["LEONID"]);       // выводит Minchenko
```

Появление во второй версии .NET Framework обобщенных типов позволило создать классы для представления коллекций с их использованием. Данные классы (а также сопутствующие им типы) сосредоточены в пространстве имен `System.Collectons.Generics`. В таблице 13 описаны некоторые типы из этого пространства имен.

Таблица 13

Типы пространства имен `System.Collections.Generics`

Имя типа	Аналог из `System.Collections`	Описание
`Collection<T>`	`CollectionBase`	Произвольная коллекция
`Comparer<T>`	`Comparer`	Стандартизированный способ сравнения двух объектов
`Dictionary<K, V>`	`Hashtable`	Таблица пар «ключ-значение»
`List<T>`	`ArrayList`	Динамический массив
`Queue<T>`	`Queue`	Очередь
`SortedDictionary<K,V>`	`SortedList`	Отсортированное множество пар «ключ-значение»
`Stack<T>`	`Stack`	Стек
`LinkedList<T>`		Двусвязный список
`ReadOnlyCollection<T>`	`ReadOnlyCollectionBase`	Коллекция с доступом только для чтения

В `System.Collectons.Generics` определены обобщенные версии интерфейсов из `System.Collectons`:

- `ICollection<T>`
- `IComparer<T>`
- `IDictionary<K, V>`
- `IEnumerable<T>`
- `IEnumerator<T>`
- `IList<T>`

Описываемое пространство имен содержит несколько вспомогательных классов и структур. Так, тип `LinkedListNode<T>` представляет отдельный узел в списке `LinkedList<T>`, исключение `KeyNotFoundException` возникает при попытке обращения к элементу по несуществующему ключу, и так далее.

Далее представлено описание одного из обобщенных классов-коллекций – `List<T>`.

```
public class List<T> : IList<T>, ICollection<T>,
                       IEnumerable<T>, IList, ICollection,
                       IEnumerable
{
. . .
    public void Add(T item);
    public IList<T> AsReadOnly();
    public int BinarySearch(T item);
    public bool Contains(T item);
    public void CopyTo(T[] array);
    public int FindIndex(System.Predicate<T> match);
    public T FindLast(System.Predicate<T> match);
```

```
        public bool Remove(T item);
        public int RemoveAll(System.Predicate<T> match);
        public T[] ToArray();
        public bool TrueForAll(System.Predicate<T> match);
        public T this[int index] { get; set; }
    }
```

Классы коллекций, предоставляемые .NET Framework, подходят для решения большинства типичных задач, встречающихся при написании приложений. Если же стандартных классов не достаточно, программист может воспользоваться сторонними библиотеками или разработать собственный класс для некой структуры данных.

2.4. РАБОТА С ФАЙЛАМИ И ДИРЕКТОРИЯМИ

Пространство имен `System.IO` содержит четыре класса, предназначенные для работы с физическими файлами и каталогами на диске. Классы `Directory` и `File` выполняют операции в файловой системе при помощи статических членов, классы `DirectoryInfo` и `FileInfo` обладают схожими возможности, однако для работы требуется создании соответствующих объектов.

Рассмотрим работу с классами `DirectoryInfo` и `FileInfo`. Данные классы являются наследниками абстрактного класса `FileSystemInfo`. Этот класс содержит следующие основные элементы, перечисленные в таблице 14.

Таблица 14

Элементы класса `FileSystemInfo`

Имя элемента	Описание
`Attributes`	Свойство позволяет получить или установить атрибуты объекта файловой системы (тип – перечисление `FileAttributes`)
`CreationTime`	Свойство для чтения или установки времени создания объекта файловой системы
`Exists`	Свойство для чтения, проверка существования объекта файловой системы
`Extension`	Свойство для чтения, расширение файла
`FullName`	Свойство для чтения, полное имя объекта файловой системы
`LastAccessTime`	Свойство обеспечивает чтение или установку времени последнего доступа для объекта файловой системы
`LastWriteTime`	Свойство обеспечивает чтение или установку времени последней записи для объекта файловой системы
`Name`	Свойство для чтения, которое возвращает имя файла или каталога
`Delete()`	Метод удаляет объект файловой системы
`Refresh()`	Метод обновляет информацию об объекте файловой системы

Конструктор класса `DirectoryInfo` принимает в качестве параметра строку с именем того каталога, с которым будет производиться работа. Для указания текущего каталога используется точка (строка `"."`). При попытке работать с данными несуществующего каталога, генерируется исключительная ситуация. Работу с методами и свойствами класса `DirectoryInfo` продемонстрируем в следующем примере:

```
using System;
using System.IO;
class MainClass {
    public static void Main() {
        // Создали объект для директории
        DirectoryInfo dir = new DirectoryInfo(@"C:\Temp\D");
        // Выводим некоторые свойства директории
        Console.WriteLine("Full Name: {0}", dir.FullName);
        Console.WriteLine("Parent: {0}", dir.Parent);
        Console.WriteLine("Root: {0}", dir.Root);
        Console.WriteLine("Creation: {0}", dir.CreationTime);
        // На экран будет выведена следующая информация:
        // Full Name: C:\Temp\D
        // Parent: Temp
        // Root: C:\
        // Creation: 01.01.1601 3:00:00

        // Создаем новую поддиректорию в нашей
        dir.CreateSubdirectory("Dir2");
        // Создаем еще одну новую поддиректорию
        dir.CreateSubdirectory(@"Dir2\SubDir2");
        // Получаем массив объектов, описывающих поддиректории
        DirectoryInfo[] subdirs = dir.GetDirectories();
        // Получаем массив объектов,
        // описывающих все файлы в директории
        FileInfo[] files = dir.GetFiles();
        //Можно использовать маску файлов
        FileInfo[] files1 = dir.GetFiles("*.er1");
    }
}
```

Класс `FileInfo` описывает файл на жестком диске и позволяет производить операции с этим файлом. Наиболее важные члены класса представлены в таблице 15:

Таблица 15

Элементы класса `FileInfo`

Имя элемента	Описание
`AppendText()`	Создает объект `StreamWriter` для добавления текста к файлу
`CopyTo()`	Копирует существующий файл в новый
`Create()`	Создает файл и возвращает объект `FileStream` для работы с файлом
`CreateText()`	Создает объект `StreamWriter` для записи текста в новый файл
`Directory`	Свойство для чтения, каталог файла
`DirectoryName`	Свойство для чтения, полный путь к файлу
`Length`	Свойство для чтения, размер файла в байтах
`Open()`	Открывает файл с указанными правами доступа на чтение, запись или совместное использование
`OpenRead()`	Создает объект `FileStream`, доступный только для чтения
`OpenText()`	Создает объект `StreamReader` для чтения информации из существующего текстового файла
`OpenWrite()`	Создает объект `FileStream`, доступный для чтения и записи

Примеры, иллюстрирующие работу с объектами-потоками, будут рассмотрены ниже. Рассмотрим пример, показывающий создание и удаление файла:

```
using System;
using System.IO;
class MainClass {
    public static void Main() {
        // Создаем объект
        FileInfo file = new FileInfo(@"C:\Test.txt");
        // Создаем файл (с потоком делать ничего не будем)
        FileStream fs = file.Create();
        // Выводим информацию
        Console.WriteLine("Full Name: {0}", file.FullName);
        Console.WriteLine("Atts: {0}",
                          file.Attributes.ToString());
        Console.WriteLine("Press any key to delete file");
        Console.Read();
        // Закрываем поток, удаляем файл
        fs.Close();
        file.Delete();
    }
}
```

Для работы с файлом можно использовать метод `FileInfo.Open()`, который обладает большим числом возможностей, чем метод `Create()`. Рассмотрим перегруженную версию метода `Open()`, которая содержит три параметра. Первый параметр определяет режим запроса на открытие файла. Для него используются значения из перечисления `FileMode`:

- `Append` – Открывает файл, если он существует, и ищет конец файла. Если файл не существует, то он создается. Этот режим может использоваться только с доступом `FileAccess.Write`;
- `Create` – Указывает на создание нового файла. Если файл существует, он будет перезаписан;
- `CreateNew` – Указывает на создание нового файла. Если файл существует, генерирует исключение `IOException`;
- `Open` – Операционная система должна открыть существующий файл;
- `OpenOrCreate` – Операционная система должна открыть существующий файл или создать новый, если файл не существует;
- `Truncate` – Система должна открыть существующий файл и обрезать его до нулевой длины.

Второй параметр метода `Open()` определяет тип доступа к файлу как к потоку байтов. Для него используются элементы перечисления `FileAccess`:

- `Read` – Файл будет открыт только для чтения;
- `ReadWrite` – Файл будет открыт и для чтения, и для записи;
- `Write` – Файл открывается только для записи, то есть добавления данных.

Третий параметр определяет возможность совместного доступа к открытому файлу и представлен значениями перечисления FileShare:

- None – Совместное использование запрещено, на любой запрос на открытие файла будет возвращено сообщение об ошибке;
- Read – Файл могут открыть и другие пользователи, но только для чтения;
- ReadWrite – Другие пользователи могут открыть файл для чтения и записи;
- Write – Файл может быть открыт другими пользователями для записи.

Вот пример кода, использующего метод Open():

```
// Файл создается (или открывается) для чтения и записи,
// без возможности совместного использования
FileInfo file = new FileInfo(@"C:\Test.txt");
FileStream fs = file.Open(FileMode.OpenOrCreate,
                          FileAccess.ReadWrite,
                          FileShare.None);
```

2.5. ИСПОЛЬЗОВАНИЕ ПОТОКОВ ДАННЫХ

Для поддержки операций, связанных с вводом и выводом информации, библиотека классов платформы .NET предоставляет пространство имен System.IO. Основное понятие, связанное с элементами данного пространства имен, – это поток. *Поток* – абстрактное представление данных в виде последовательности байт. Потоки (в отличие от файлов) могут быть ассоциированы с файлами на диске, памятью, сетью. В пространстве имен System.IO поток представлен абстрактным классом Stream. От данного абстрактного класса порождены классы System.IO.FileStream (работа с файлами как с потоками), System.IO.MemoryStream (поток в памяти), System.Net.Sockets.NetworkStream (работа с сокетами как с потоками), System.Security.Cryptography.CryptoStream (потоки зашифрованных данных).

Рассмотрим основные методы и свойства класса Stream. Свойства для чтения CanRead, CanWrite и CanSeek определяют, поддерживает ли поток чтение, запись и поиск. Если поток поддерживает поиск, перемещаться по потоку можно при помощи метода Seek(). На текущую позицию в потоке указывает свойство Position (нумерация с нуля). Свойство Length возвращает длину потока, которая может быть установлена при помощи метода SetLength(). Методы Read() и ReadByte(), Write() и WriteByte() служат для чтения и записи блока байт или одиночного байта. Метод Flush() записывает данные из буфера в связанный с потоком источник данных. При помощи метода Close() поток закрывается и все связанные с ним ресурсы освобождаются.

Класс Stream вводит поддержку асинхронного ввода/вывода. Для этого служат методы BeginRead() и BeginWrite(). Уведомление о завершении асинхронной операции возможно двумя способами: или при помощи делегата тип System.AsyncCallback, передаваемого как параметр методов BeginRead() и BeginWrite(), или при помощи вызова методов EndRead() и EndWrite(), которые приостанавливают текущий поток управления до окончания асинхронной операции.

Использование методов и свойств класса Stream продемонстрируем в фрагменте кода с классом FileStream. Объект класса FileStream возвращается некоторыми методами классов FileInfo и File, кроме этого, данный объект можно создать при помощи конструктора с параметрами, включающими имя файла и режимы доступа к файлу.

```
// Создаем файл test.dat в текущем каталоге
FileStream fs = new FileStream("test.dat",
                               FileMode.OpenOrCreate,
                               FileAccess.ReadWrite);
// В цикле записываем туда 100 байт
for (byte i = 0; i < 100; i++) {
    fs.WriteByte(i);
}
// Мы можем записывать информацию из массива байт
byte[] info = {1, 2, 3, 4, 5, 6, 7, 8, 9};
// Первый параметр – массив, второй – смещение в массиве,
// третий – количество записываемых байт
fs.Write(info, 2, 4);
// Возвращаемся на начало файла
fs.Position = 0;
// Читаем все байты и выводим на экран
while (fs.Position <= fs.Length - 1) {
    Console.Write(fs.ReadByte());
}
// Закрываем поток (и файл), освобождая ресурсы
fs.Close();
```

Класс MemoryStream предоставляет возможность организовать поток в оперативной памяти. Свойство Capacity этого класса позволяет получить или установить количество байтов, выделенных под поток. Метод ToArray() записывает все содержимое потока в массив байт. Метод WriteTo() переносит содержимое потока в памяти в другой поток, производный от класса Stream.

Классы-потоки представляют поток как последовательность неформатированных байт. Однако в большинстве приложений удобнее читать и записывать в поток данные примитивных типов или строк. Библиотека классов .NET Framework содержит набор парных классов вида XXXReader/XXXWriter, которые инакапсулируют поток и предоставляют к нему высокоуровневый доступ.

Классы BinaryReader и BinaryWriter позволяют при помощи своих методов читать и записывать в поток данные примитивных типов, строк и массивов байт или символов. Вся информация записывается в поток как последовательность байт. Рассмотрим работу с данными классами на следующем примере. Пусть имеется класс, который хранит информацию о студенте:

```
class Student {
    public string Name;
    public int Age;
    public double MeanScore;
}
```

Методы, которые осуществляют запись и чтение объекта этого класса в поток в виде последовательности байт, могут иметь следующий вид:

```
void SaveToStream(Stream stm, Student s) {
    // Конструктор класса позволяет "обернуть"
    // BinaryWriter вокруг потока
    BinaryWriter bw = new BinaryWriter(stm);
    // BinaryWriter содержит 18 перегруженных версий
    // метода Write()
    bw.Write(s.Name);
    bw.Write(s.Age);
    bw.Write(s.MeanScore);
    // Убеждаемся, что буфер BinaryWriter пуст
    bw.Flush();
}

void ReadFromStream(Stream stm, Student s) {
    BinaryReader br = new BinaryReader(stm);
    // Для чтения каждого примитивного типа есть свой метод
    s.Name = br.ReadString();
    s.Age = br.ReadInt32();
    s.MeanScore = br.ReadDouble();
}
```

Абстрактные классы `TextReader` и `TextWriter` позволяю читать и записывать данные в поток как последовательность символов. От этих классов наследуются классы `StreamReader` и `StreamWriter`. Перепишем методы для сохранения данных класса `Student` с использованием `StreamReader` и `StreamWriter`:

```
void SaveToStream(Stream stm, Student s) {
    StreamWriter sw = new StreamWriter(stm);
    // Запись напоминает вывод на консоль
    sw.WriteLine(s.Name);
    sw.WriteLine(s.Age);
    sw.WriteLine(s.MeanScore);
    sw.Flush();
}

void ReadFromStream(Stream stm, Student s) {
    StreamReader sr = new StreamReader(stm);
    // Читаем данные как строки, требуется их преобразовать
    s.Name = sr.ReadLine();
    s.Age = Int32.Parse(sr.ReadLine());
    s.MeanScore = Double.Parse(sr.ReadLine());
}
```

Классы `StringReader` и `StringWriter` – это наследники классов `TextReader` и `TextWriter`, которые представляют доступ к строке или к объекту класса `StringBuilder` как к потоку. Это может оказаться полезным, если текстовая информация добавляется в специальный буфер в оперативной информации. Работу с данными классами иллюстрирует следующий пример:

```
StringWriter sw = new StringWriter();
// Пишем информацию в поток
sw.WriteLine("Hello!");
sw.WriteLine("This is an example...");
sw.Close();

// Выводим все информацию
Console.WriteLine(sw.ToString());

string s = "Big\n Big string\n 10";
// Создаем StringReader на основе строки
StringReader sr = new StringReader(s);

// Последовательно читаем "кусочки" строки
string input;
while((input = sr.ReadLine()) != null)
    Console.WriteLine(input);
sr.Close();
```

2.6. СЕРИАЛИЗАЦИЯ

Под *сериализацией* понимается действие, при котором данные объекта в памяти переносятся в байтовый поток для сохранения или передачи. *Десериализация* – это обратное действие, заключающееся в восстановлении состояния объекта по данным из байтового потока. При выполнении сериализации следует учитывать несколько нетривиальных моментов, например: сохранение полей объекта некоторого класса требует сохранения всех данных базовых классов; если объект содержит ссылки на другие объекты, то требуется сохранить данные всех объектов, на которые имеются ссылки.

Среда .NET Framework обладает развитым механизмом поддержки сериализации, включая поддержку сериализации в различных форматах. Основные классы, связанные с сериализацией, размещены в наборе пространств имен вида `System.Runtime.Serialization.*` и `System.Xml.Serialization.*`. Так, пространство имен `System.Runtime.Serialization.Formatters.Binary` обеспечивает поддержку сериализации в двоичном формате. Класс `BinaryFormatter` из этого пространства имен способен выполнить сериализацию *графа объектов* в поток при помощи метода `Serialize()` и десериализацию при помощи метода `Deserialize()`.

Рассмотрим сериализацию на примере. Пусть имеется класс с информацией о студенте (имя, возраст, средний балл), а также класс с информацией о группе (список студентов и средний балл группы). Эти классы могут быть описаны следующим образом:

```
class Student {
    public string Name;
    public int Age;
    public double MeanScore;
    public Student(string Name, int Age, double MeanScore) {
        this.Name = Name;
        this.Age = Age;
```

```
            this.MeanScore = MeanScore;
        }
    }

    class Group {
        public ArrayList GL = new ArrayList();
        public double MSG;
        public Student BestStudent;
        public double CalcMSG() {
            double sum = 0;
            foreach(Student s in GL)
                sum += s.MeanScore;
            MSG = sum / GL.Count;
            return MSG;
        }
        public Student FindTheBest() {
            BestStudent = (Student)GL[0];
            foreach(Student s in GL)
                if (s.MeanScore > BestStudent.MeanScore)
                    BestStudent = s;
            return BestStudent;
        }
    }
```

Допустим, что планируется осуществлять сериализацию объектов класса `Group`. Чтобы реализовать сериализацию пользовательского типа, он должен быть помечен специальным атрибутом – **`[Serializable]`**. Кроме этого, все поля такого типа также должны иметь этот атрибут. Данное замечание актуально в том случае, если поле имеет пользовательский тип, так как встроенные типы и большинство стандартных классов уже помечены как **`[Serializable]`**. В нашем случае мы должны добавить атрибут сериализации к классу `Group` и к классу `Student`. Сериализация некоторых полей может не иметь смысла (например, эти поля вычисляются при работе с объектом или хранят конфиденциальные данные). Для таких полей можно применить атрибут **`[NonSerialized]`**. Изменим код нашего примера с учетом вышесказанного:

```
    [Serializable]
    class Student {
    . . .
    }

    [Serializable]
    class Group {
        public ArrayList GL = new ArrayList();

        [NonSerialized]  // Не надо сохранять – просто посчитаем
        public double MSG;
    . . .
    }
```

Теперь все готово для выполнения сериализации. Метод `Serialize()` класса `BinaryFormatter` получает два параметра: поток, в который требуется

выполнить сериализацию, и объект, который требуется сериализовать. Вот фрагмент кода, сериализующего объект класса Group, а затем выполняющего десериализацию:

```
// Нам понадобятся следующие пространства имен
using System;
using System.Collections;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
. . .
// Создаем группу и добавляем в нее несколько студентов
Group g = new Group();
g.GL.Add(new Student("Vova", 20, 4.5));
g.GL.Add(new Student("Ira", 20, 5));
g.GL.Add(new Student("Peter", 19, 4));

// Выводим некоторую информацию о группе
Console.WriteLine(g.CalcMSG());
Console.WriteLine(g.FindTheBest().Name);

// Создаем поток – это будет файл
Stream fs = new FileStream("data.dat", FileMode.Create);

// Создаем объект для сериализации в двоичном формате
BinaryFormatter fmt = new BinaryFormatter();

// Сериализуем и затем закрываем поток
fmt.Serialize(fs, g);
fs.Close();

// Теперь десериализация. Создаем поток
fs = new FileStream("data.dat", FileMode.Open);

// Десериализация. Обратите внимание на приведение типов
Group d = (Group)fmt.Deserialize(fs);

// Выводим информацию о группе
Console.WriteLine(d.CalcMSG());
Console.WriteLine(d.FindTheBest().Name);
```

Метод Deserialize() получает в качестве параметра поток, из которого десериализуется объект и возвращает объект типа object. В одном потоке можно сериализовать несколько различных объектов – главное, чтобы последовательность десериализации соответствовала сериализации.

Десериализацию удобно представлять как своеобразный вызов конструктора, так как результатом десериализации является ссылка на существующий объект. Однако если некоторые поля класса были помечены как **[NonSerialized]**, то возможно после десериализации потребуется просчитать значения данных полей. Допустимое решение – реализовать в классе интерфейс IDeserializationCallback из пространства имен System.Runtime.Serialization. Данный интерфейс содержит единственный метод – OnDeserialization, кото-

рый вызывается исполняемой средой автоматически после десериализации объекта. Используем интерфейс для класса Group:

```
[Serializable]
class Group : IDeserializationCallback {
    public ArrayList GL = new ArrayList();

    // Не будем сохранять средний балл и лучшего студента
    [NonSerialized]
    public double MSG;
    [NonSerialized]
    public Student BestStudent;

    public double CalcMSG() { . . . }
    public Student FindTheBest() { . . . }

    // После десериализации просчитаем средний балл и
    // найдем лучшего студента. Работа с параметром метода
    // исполняемой средой на данный момент не поддерживается!
    public void OnDeserialization(object o) {
        CalcMSG();
        FindTheBest();
    }
}
```

Рассмотрим несколько примеров сериализации в различных форматах. Класс SoapFormatter из пространства имен System.Runtime.Serialization.Formatters.Soap обеспечивает сериализацию объекта в формате протокола SOAP (для использования данного пространства имен требуется подключить библиотеку system.runtime.serialization.formatters.soap.dll). Изменения в коде примера минимальны:

```
using System.Runtime.Serialization.Formatters.Soap;
. . .
// Создаем объект для сериализации в формате SOAP
SoapFormatter fmt = new SoapFormatter();
. . .
```

Файл в формате SOAP – это xml-файл с дополнительной информацией протокола SOAP. В .NET Framework можно выполнить сериализацию объектов в формате XML. Данный функционал обеспечивает класс XmlSerializer из пространства имен System.Xml.Serialization (файл System.Xml.dll). При XML-сериализации сохраняются данные объекта, доступные через public-свойства и поля. Кроме этого, сериализуемый класс должен иметь конструктор без параметров и являться public-классом.

Гибкая настройка XML-сериализации может быть выполнена при помощи атрибутов. Рассмотрим некоторые из атрибутов. Атрибут XmlRootAttribute применяется к классу и помогает определить корневой элемент в XML-файле. Поля и свойства, которые не должны сохраняться, помечаются атрибутом XmlIgnoreAttribute. Если класс агрегирует объекты других классов, то эти

классы должны быть указаны при помощи атрибута `XmlIncludeAttribute`. Атрибут `XmlAttributeAttribute` используется, если элемент класса требуется сохранить в виде атрибута XML-тэга. Если класс агрегирует массив объектов, то настройка вида сохранения этого массива выполняется при помощи атрибутов `XmlArrayAttribute` и `XmlArrayItemAttribute`.

Рассмотрим пример XML-сериализации. Будем сохранять в формате XML данные классов `Student` и `Group`. При описании этих классов воспользуемся некоторыми атрибутами. Ниже представлен код программы, выполняющей создание объектов, сериализацию и десериализацию.

```
using System;
using System.Collections;
using System.IO;
using System.Xml.Serialization;

public class Student {
    public string Name;
    [XmlAttribute("Age")]
    public int Age;
    public double MeanScore;
    public Student() { }
    public Student(string Name, int Age, double MeanScore) {
        this.Name = Name;
        this.Age = Age;
        this.MeanScore = MeanScore;
    }
}

[XmlRoot("StudentGroup")]
[XmlInclude(typeof(Student))]
public class Group {
    [XmlArray("GroupList")]
    [XmlArrayItem("Student")]
    public ArrayList GL = new ArrayList();
    [XmlIgnore]
    public double MSG;
    public Group() { }
    public double CalcMSG() {
        double sum = 0;
        foreach (Student s in GL)
            sum += s.MeanScore;
        MSG = sum / GL.Count;
        return MSG;
    }
}

class Program {
    static void Main(string[] args) {
        Group g = new Group();
        g.GL.Add(new Student("Ivanov", 20, 6.0));
        g.GL.Add(new Student("Petrov", 21, 7.0));
```

```
            Stream fs= new FileStream("data.dat",FileMode.Create);
            // Указываем тип того объекта, который сериализуем
            XmlSerializer x = new XmlSerializer(typeof(Group));
            // Сериализуем
            x.Serialize(fs, g);
            fs.Close();
            // Восстанавливаем
            fs = new FileStream("data.dat", FileMode.Open);
            Group d = (Group)x.Deserialize(fs);
            fs.Close();
        }
}
```

XML-файл с сохраненной информацией выглядит следующим образом:

```
<?xml version="1.0"?>
<StudentGroup
            xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <GroupList>
    <Student xsi:type="Student" Age="20">
      <Name>Ivanov</Name>
      <MeanScore>6</MeanScore>
    </Student>
    <Student xsi:type="Student" Age="21">
      <Name>Petrov</Name>
      <MeanScore>7</MeanScore>
    </Student>
  </GroupList>
</StudentGroup>
```

2.7. СЕРИАЛИЗАЦИЯ ОБЪЕКТОВ В НЕСТАНДАРТНОМ ФОРМАТЕ

Если программиста не устраивают существующие форматы сериализации или способ организации потока сериализованных данных, он может осуществить сериализацию в собственном формате. Классы, помеченные атрибутом `[Serializable]`, могут дополнительно реализовывать интерфейс `ISerializable`. Это позволяет «вклиниться» в процесс сериализации и выполнить любые действия, связанные с формированием данных для сохранения.

Интерфейс `ISerializable` устроен просто:

```
public interface ISerializable {
    void GetObjectData(SerializationInfo info,
                       StreamingContext context);
}
```

Метод `GetObjectData()` вызывается форматером автоматически при выполнении сериализации. Реализация данного метода подразумевает заполнение объекта `SerializationInfo` набором данных вида «ключ-значение», которые (обычно) соответствуют полям сохраняемого объекта. Класс `SerializationInfo` содержит несколько перегруженных версий метода

AddValue(), а также свойства для указания имени типа, имени сборки и т.п. Фрагмент определения класса SerializationInfo приведен ниже:

```
public sealed class SerializationInfo {
    public SerializationInfo(Type type,
                             IFormatterConverter converter);
    public string AssemblyName { get; set; }
    public string FullTypeName { get; set; }
    public int MemberCount { get; }
    public void AddValue(string name, short value);
    public void AddValue(string name, UInt16 value);
    public void AddValue(string name, int value);
    . . .
}
```

Если тип реализовывает интерфейс ISerializable, то он должен также содержать специальный конструктор:

```
[Serializable]
class SomeClass : ISerializable {
    // Конструктор с такой сигнатурой необходим,
    // чтобы CLR смогла восстановить состояние объекта
    private SomeClass(SerializationInfo si,
                      StreamingContext ctx) { . . . }
    . . .
}
```

Обратите внимание: конструктор объявлен с модификатором private. Первый параметр конструктора – это объект класса SerializationInfo. Второй параметр имеет тип StreamingContext. Наиболее часто используемым элементом данного класса является свойство State из перечисления StreamingContextStates.

Для иллюстрации использования интерфейса ISerializable предположим, что у нас определен класс с двумя текстовыми полями. Далее, пусть требуется сохранять эти поля в верхнем регистре, а считывать в нижнем регистре. Вот код, который выполняет требуемую сериализацию (не забудьте подключить пространство имен System.Runtime.Serialization):

```
[Serializable]
class MyStringData : ISerializable {
    public string dataItemOne, dataItemTwo;
    public MyStringData(){}
    private MyStringData(SerializationInfo si,
                         StreamingContext ctx) {
        // Получаем значения из потока и преобразовываем их
        dataItemOne = si.GetString("First_Item").ToLower();
        dataItemTwo = si.GetString("dataItemTwo").ToLower();
    }

    void ISerializable.GetObjectData(SerializationInfo info,
                                     StreamingContext ctx) {
        // Заполняем объект SerializationInfo
        info.AddValue("First_Item", dataItemOne.ToUpper());
```

```
            info.AddValue("dataItemTwo", dataItemTwo.ToUpper());
        }
    }
```

Как видим, в конструкторе типа сериализованные значения извлекаются при помощи метода `GetString()`. В классе `SerializationInfo` существуют аналогичные методы для других типов данных.

Во второй версии .NET Framework для поддержки собственных форматов сериализации существует ряд атрибутов: `[OnSerializing]`, `[OnSerialized]`, `[OnDeserializing]`, `[OnDeserialized]`. Этими атрибутами помечаются методы класса. Механизм сериализации будет автоматически вызывать помеченный метод в требуемый момент. Метод, который помечается одним из вышеуказанных атрибутов, должен принимать в качестве параметра объект класса `StreamingContext` и не возвращать значений.

Для иллюстрации пользовательской сериализации с применением атрибутов рассмотрим класс `MoreData`, который при сериализации ведет себя аналогично классу `MyStringData`.

```
[Serializable]
class MoreData {
    public string dataItemOne, dataItemTwo;
    // Если метод помечен атрибутом [OnSerializing], то он
    // вызывается перед записью данных в поток
    [OnSerializing]
    internal void OnSerializing(StreamingContext context) {
        dataItemOne = dataItemOne.ToUpper();
        dataItemTwo = dataItemTwo.ToUpper();
    }

    // Если метод помечен атрибутом [OnDeserialized], то он
    // вызывается после чтения данных из потока
    [OnDeserialized]
    internal void OnDeserialized(StreamingContext context) {
        dataItemOne = dataItemOne.ToLower();
        dataItemTwo = dataItemTwo.ToLower();
    }
}
```

2.8. ВВЕДЕНИЕ В XML

XML – это способ записи структурированных данных. Под «структурированными данными» обычно подразумевают такие объекты, как электронные таблицы, адресные книги, конфигурационные параметры. *XML-данные* содержатся в документе, в роли которого может выступать файл, поток или другое хранилище информации, способное поддерживать текстовый формат.

Любой XML-документ строится по определенным правилам. XML-данные состоят из набора *элементов*. Каждый элемент определяется при помощи *имени* и *открывающего* и *закрывающего тэга*. Открывающий тэг элемента записывается в форме `<имя_элемента>`, закрывающий тэг – в форме `</имя_элемента>`.

Между открывающим и закрывающим тэгами размещается *содержимое элемента*. Приведем пример простейшего элемента:

```
<name>This is a text</name>
```

Элемент может содержать вложенные элементы с другими именами, но в XML-документе всегда должен быть единственный элемент, называемый *корневым*, никакая часть которого не входит в содержимое любого другого элемента.

```
<Group>
  <student>
    <name>
      Ivanov
    </name>
  </student>
  <student>
    <name>
      Petrov
    </name>
    <ball>
      4.0
    </ball>
  </student>
</Group>
```

В приведенном примере XML-документ содержит корневой элемент с именем `Group`, который, в свою очередь, содержит два элемента с именем `student`. Обратите внимание на разное количество вложенных элементов у первого и второго «студентов» (это допустимо).

Имена тегов элементов чувствительны к регистру! Элементы, задаваемые открывающим и закрывающим тегами, должны быть правильно вложены друг в друга. Следующий пример **не является** правильным:

```
<text>
  <bold><italic>XML</bold></italic>
</text>
```

Элемент без содержимого может быть записан в специальной форме: `<name/>`.

Имена элементов могут содержать буквы, цифры, дефисы (`-`), символы подчеркивания (`_`), двоеточия (`:`) и точки (`.`), однако начинаться они могут только с буквы или символа подчеркивания. Двоеточие может быть использовано только в специальных случаях, когда оно разделяет так называемое *пространство имен*. Имена элементов, начинающиеся с `xml`, вне зависимости от комбинации регистров букв в этом выражении зарезервированы для нужд стандарта.

Элемент может иметь ни одного, один или несколько *атрибутов*. Правила на имена атрибутов накладываются такие же, как и на имена элементов. Имена атрибутов отделяются от их значений знаком =. Значение атрибута заключается в апострофы или в двойные кавычки. Если апостроф или двойные кавычки при-

сутствуют в значении атрибута, то используются те из них, которые не встречаются в этом значении. Приведем пример элементов с атрибутами:

```
<elements-with-attributes>
  <el _ok = "yes"/>
  <one attr = "a value"/>
  <several first = "1" second = "2" third = "333"/>
  <apos_quote case1 = "John's"
              case2 = 'He said: "Hello, world!" '/>
</elements-with-attributes>
```

Символы < и & не могут быть использованы в тексте, так как они используются в разметке XML-документа. Если эти символы необходимы, следует использовать `<` вместо < и `&` вместо &. Символы >, ", и ' также *могут* быть заменены `>`, `&quot;` и `&apos;`, соответственно.

XML-документ может содержать *комментарии*, записываемые в обрамлении `<!--` и `-->`. В тексте комментариев не должна содержаться последовательность из двух знаков дефиса:

```
<!-- Group 252001 list -->
<Group>
  <student>
    <name>
      Ivanov
    </name>
  </student>
</Group>
```

Документ может содержать *инструкции по обработке* (PI), несущие информацию для приложений. Инструкции по обработке записываются в обрамлении `<?` и `?>`.

```
<example>
  <?perl lower-to-upper-case  ?>
  <?web-server add-header = "university" ?>
  <text>Some text</text>
</example>
```

Секция `CDATA` используется для того, чтобы обозначить части документа, которые не должны восприниматься как разметка. Секция `CDATA` начинается со строки `<![CDATA[` и заканчивается строкой `]]>`. Внутри самой секции не должна присутствовать строка `]]>`.

```
<example>
  <![CDATA[ <aaa>bb&cc<<<]]>
</example>
```

XML-документ может, но не обязан начинаться с *XML-декларации*, определяющей используемую версию XML:

```
<?xml version="1.0"?>
```

Декларация может содержать необязательный атрибут, указывающий кодировку XML-документа:

```
<?xml version="1.0" encoding="UTF-16"?>
```

Если XML-документ оформлен по описанным выше правилам, то он называется *хорошо оформленным XML-документом*. Если хорошо оформленный документ удовлетворяет некой схеме, задающей структуру и содержание, то документ называется *правильно оформленным*. Существует несколько способов для описания приемлемых схем документов. Один из способов – использование *блоков определения типа документа* (*document type definitions*, *DTD*). Современным способом считается использование языка XML Schema Definitions (XSD), который сам основан на XML.

Описание XSD может потребовать нескольких лекций, поэтому просто приведем пример файла `students.xsd` на этом языке:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">
  <xs:element name="Students">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="student">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name" type="xs:string"/>
              <xs:element name="year" type="xs:gYear"/>
              <xs:element name="ball" type="xs:double"
                          minOccurs="0"/>
            </xs:sequence>
            <xs:attribute name="Department" type="xs:string"
                          use="optional"/>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:attribute name="Department"/>
</xs:schema>
```

Данная схема описывает *правильные документы*, которые должны содержать корневой элемент `Students`, элемент `Students` должен содержать один или несколько элементов `student`, а элемент `student` *должен* содержать один элемент `name`, один элемент `year` и *может* содержать элемент `ball`. Также определен порядок и тип элементов, и наличие необязательного атрибута `Department` у элемента `student`.

Существует большое количество разнообразного программного обеспечения для работы с XML-схемами, в частности, для создания схем и проверки соответствия XML-документа некоторой схеме. Упомянем такие программы как XMLSPY и утилиту Xsd.exe, которая входит в состав .NET SDK.

2.9. РАБОТА С XML-ДОКУМЕНТАМИ В .NET FRAMEWORK

Рассмотрим вопросы, связанные с чтением и записью XML-документов. Для этих целей предназначен набор классов из пространства имен `System.Xml`. Упомянем такие классы, как `XmlDocument`, `XmlTextReader`, `XmlTextWriter`, `XmlValidatingReader`.

Класс `XmlReader` – это абстрактный класс, предназначенный для чтения xml-данных из потока. В .NET Framework существуют три класса, построенных на основе `XmlReader`: `XmlTextReader` (разбор xml-данных на основе текстового потока), `XmlNodeReader` (разбор XML из набора объектов `XmlNode`) и `XmlValidatingReader` (при чтении производится проверка документа с использованием DTD и/или Shema).

Обычно для простого чтения XML достаточно класса `XmlTextReader`. Конструктор класса может быть вызван в различных формах. В следующем примере предполагается читать данные из файла `Students.xml`:

```
XmlTextReader tr = new XmlTextReader("Students.xml");
```

Объект класса `XmlTextReader` может быть построен на основе потоков:

```
XmlTextReader tr = new XmlTextReader(
                       new TextReader(
                        new FileStream("Students.xml",
                                       FileMode.Open)));
```

Как видим, `XmlTextReader` может извлекать данные из любых потоков, включая сетевые потоки и потоки, связанные с базами данных. XML-данные можно извлекать и из потоков, сконструированных на основе строк:

```
string xmlContent =
    "<book>" +
    "  <title>Чук и Гек</title>" +
    "  <author>Аркадий Гайдар</author>" +
    "</book>";
XmlTextReader tr = new XmlTextReader(
                       new StringReader(xmlContent));
```

После создания объект `XmlTextReader` извлекает xml-конструкции из потока при помощи метода `Read()`. Тип текущей конструкции (элемент, атрибут) можно узнать, используя свойство `NodeType`, значениями которого являются элементы перечисления `XmlNodeType`. С конструкцией можно работать, используя различные свойства, такие как `Name` (возвращает имя элемента или атрибута), `Value` (возвращает данные элемента) и так далее.

В таблице 16 приведены все возможные значения перечисления `XmlNodeType`.

Таблица 16

Значения перечисления `XmlNodeType`

Значение	Пример
`Attribute`	`Department="Informatics"`
`CDATA`	`<![CDATA["This is character data"]]>`
`Comment`	`<!-- This is a comment -->`

Document	<Students>
DocumentType	<!DOCTYPE Students SYSTEM " Students.dtd">
Element	<student>
EndElement	</student>
Entity	<!ENTITY filename "Strats.xml">
EntityReference	<
Notation	<!NOTATION GIF89a SYSTEM "gif">
ProcessingInstruction	<?perl lower-to-upper-case ?>
Text	**Petrov**
Whitespace	<Make/>\r\n<Model/>
XmlDeclaration	<?xml version="1.0"?>

Дадим некоторые комментарии. Значения DocumentType, Entity, EntityReference и Notation связаны с *блоком определения типа документа* (document type definition, DTD). Напомним, что при помощи подобного блока, который может являться частью XML-документа, задается приемлемая схема документа. Значение Whitespace представляет пустое (вернее, не обрабатываемое) пространство между тэгами.

Следующий пример демонстрирует разбор xml-файла и вывод разобранных конструкций на экран:

```
using System;
using System.Xml;

class XmlFun {
    static void Main() {
        XmlTextReader rdr = new XmlTextReader("Students.xml");
        // Разбираем файл и выводим на консоль каждый элемент
        while(rdr.Read()) {
            // Реагируем в зависимости от NodeType
            // Не все значения NodeType отслеживаются!!!
            switch(rdr.NodeType) {
            case XmlNodeType.Element:
                Console.Write("<{0}>", rdr.Name);
                break;
            case XmlNodeType.Text:
                Console.Write(rdr.Value);
                break;
            case XmlNodeType.CDATA:
                Console.Write("<![CDATA[{0}]]>", rdr.Value);
                break;
            case XmlNodeType.Comment:
                Console.Write("<!--{0}-->", rdr.Value);
                break;
            case XmlNodeType.XmlDeclaration:
                Console.Write("<?xml version='1.0'?>");
                break;
            case XmlNodeType.EndElement:
                Console.Write("</{0}>", rdr.Name);
                break;
            }
```

```
            }
        }
    }
```

Набор методов класса `XmlTextReader` вида `MoveXXX()`, таких как `MoveToNextElement()`, может использоваться для извлечения соответствующей конструкции из потока. Методы используются для перехода к следующей конструкции, вернуться к просмотренным конструкциям нельзя.

Подобно тому, как класс `XmlReader` является абстрактным классом для чтения xml-данных из потока, класс `XmlWriter` – это абстрактный класс для создания xml-данных. Подчеркнем, что xml-данные всегда могут быть созданы при помощи простой строки и затем записаны в любой поток:

```
string xmlContent = "<greeting>" + message + "</greeting>";
```

Однако такой подход не лишен недостатков: возрастает вероятность неправильного формирования структуры XML-документа из-за элементарных ошибок программиста. Класс `XmlWriter` предоставляет более «помехоустойчивый» способ генерации XML-документа. В следующем примере создается документ, содержание которого соответствует `xmlContent`:

```
using System;
using System.Xml;

class XmlFun {
    static void Main() {
        string message = "Hello, dude!";
        XmlTextWriter xw = new XmlTextWriter("greetings.xml",
                                    System.Text.Encoding.UTF8);
        xw.Formatting = Formatting.Indented;
        xw.Indentation = 2;
        xw.WriteStartDocument();
        xw.WriteStartElement("greeting");
        xw.WriteString(message);
        xw.WriteEndElement();
        xw.WriteEndDocument();
        xw.Flush();
    }
}
```

Класс `XmlDocument` представляет XML-документ в виде дерева узлов-элементов. Каждый узел является экземпляром класса `XmlNode`, который содержит методы и свойства для навигации по дереву, чтения и записи информации узла и другие.

Пусть имеется следующий файл `Students.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<Students>
  <student Department="Informatics">
    <name>Ivanov</name>
    <year>1980</year>
  </student>
  <student>
```

```
    <name>Petrov</name>
    <year>1980</year>
    <ball>5.0</ball>
  </student>
  <student Department="Informatics">
    <name>Sidorova</name>
    <year>1981</year>
    <ball>4.5</ball>
  </student>
</Students>
```

Следующий код создает объект класса `XmlDocument`, и этот объект загружается информацией из файла `Students.xml`:

```
XmlDocument doc = new XmlDocument();
doc.Load("Students.xml");
```

Метод `Load()` считывает документ и разбирает его в памяти на элементы. Если документ не является правильно форматированным, генерируется исключение `XmlException`.

За успешным вызовом метода `Load()` обычно следует обращение к свойству `DocumentElement` объекта, представляющего документ. Данное свойство возвращает ссылку на объект класса `XmlNode`. Этот объект позволяет обнаружить у элемента дочерние элементы (свойство `HasChildNodes`) и получить к ним доступ, используя свойство `ChildNodes`, являющееся коллекцией типа `XmlNodeList`. Комбинация свойств `HasChildNodes` и `ChildNodes` позволяет использовать при обработке XML-документов рекурсивный подход:

```
using System;
using System.Xml;

class MainClass {
    public static void Main() {
        XmlDocument doc = new XmlDocument();
        doc.Load("Students.xml");
        OutputNode(doc.DocumentElement);
    }

    public static void OutputNode(XmlNode node) {
        Console.WriteLine("Type={0} \t Name={1} \t Value={2}",
                          node.NodeType, node.Name, node.Value);
        // Если есть дочерние элементы,
        // рекурсивно обрабатываем их
        if(node.HasChildNodes) {
            XmlNodeList children = node.ChildNodes;
            foreach(XmlNode child in children)
                OutputNode(child);
        }
    }
}
```

Приведенный код не выводит значения атрибутов элемента. Свойство `ChildNodes` не включает атрибуты. Все атрибуты элемента хранятся в свойстве

Attributes и представлены типом XmlAttribute. Следующий фрагмент кода позволяет обработать атрибуты элемента:

```
void OutputNode(XmlNode node) {
    . . .
    if(node.Attributes != null){
        foreach (XmlAttribute attr in node.Attributes)
            Console.WriteLine("Type={0}\tName={1}\tValue={2}",
                        attr.NodeType, attr.Name, attr.Value);
    }
    . . .
}
```

У объекта класса XmlNode имеются свойства NodeType, Name и Value. Следует иметь в виду, что обращаться к Name и Value нужно в зависимости от значения NodeType. У элементов (XmlNodeType.Element) нет Value, но определено Name. У текста (XmlNodeType.Text) определено Name, но нет Value. У атрибутов определены и Name, и Value.

Для того чтобы получить определенный узел или набор узлов, нет необходимости итеративно просматривать весь XML-документ. Для этих целей можно использовать методы класса XmlDocument такие как GetElementsByTagName(), SelectNodes(), SelectSingleNode().

При помощи класса XmlDocument можно не только читать, но и создавать XML-документы. В следующем примере загружается файл Students.xml, удаляется его первый элемент (первый студент), добавляется еще один элемент, описывающий студента, и результат сохраняется в файл Students_new.xml.

```
using System;
using System.Xml;
class XmlWork {
    static void Main() {
        XmlDocument doc = new XmlDocument();
        // Загружаем информацию из файла
        doc.Load("Students.xml");

        // Создаем необходимые элементы
        XmlNode student = doc.CreateElement("student");
        XmlNode name = doc.CreateElement("name");
        XmlNode year = doc.CreateElement("year");
        XmlNode ball = doc.CreateElement("ball");

        // Создаем текстовое наполнение элементов
        XmlNode text1 = doc.CreateTextNode("Mr. Zorg");
        XmlNode text2 = doc.CreateTextNode("1990");
        XmlNode text3 = doc.CreateTextNode("4.7");

        // Связываем элементы и текстовое наполнение
        name.AppendChild(text1);
        year.AppendChild(text2);
        ball.AppendChild(text3);
```

```
            // Связываем элементы в одну структуру
            student.AppendChild(name);
            student.AppendChild(year);
            student.AppendChild(ball);

            // Получаем корневой элемент документа
            XmlNode root = doc.DocumentElement;

            // Удаляем первый дочерний элемент
            root.RemoveChild(root.FirstChild);

            // Добавляем сконструированный нами элемент
            root.AppendChild(student);

            // Сохраняем все в файл
            doc.Save("Students_new.xml");
        }
    }
```

Другие методы класса `XmlDocument`, которые могут быть полезны при модификации XML-документа, это методы `PrependChild()`, `InsertBefore()`, `InsertAfter()`, `RemoveAll()`, `ReplaceChild()`. Для конструирования элемента можно использовать свойство `XmlNode.InnerText`, которое должно содержать подходящую строку, представляющую содержимое элемента:

```
XmlNode student = doc.CreateElement("student");
student.InnerText = "<name>Mr. Zorg</name>" +
                    "<year>1990</year>" +
                    "<ball>4.7</ball>";
```

Класс `XmlValidatingReader` позволяет производить проверку допустимости XML-документов заданной XSD-схеме или описанию DTD. Объект класса `XmlValidatingReader` присоединяется к объекту `XmlTextReader` (или `XmlNodeReader`, если нужно проверить лишь часть документа). При чтении документа и возникновении ошибки допустимости генерируется событие `ValidationEventHandler`, с передачей информации о позиции ошибки в документе. Далее рассмотрен пример использования класса `XmlValidatingReader`.

```
// Создаем ридер для некого XML-документа
XmlTextReader xtr = new XmlTextReader("Students.xml");

// Объект XmlValidatingReader порождаем на основе ридера
XmlValidatingReader xvr = new XmlValidatingReader(xtr);

// Задаем тип проверки допустимости.
// В данном случае --- XSD-схема
xvr.ValidationType = ValidationType.Schema;

// Так как эта схема не храниться в самом XML-документе,
// необходимо присоединить ее к XmlValidatingReader.
// Создем коллекцию схем (правда, из одного элемента)
XmlSchemaCollection xsc = new XmlSchemaCollection();
```

```
xsc.Add("","data.xsd");

// Добавляем коллекцию к XmlValidatingReader
xvr.Schemas.Add(xsc);

// Назначаем обработчик события на ошибки
xvr.ValidationEventHandler +=
                    new ValidationEventHandler(ValidationError);
```

Хотя класс `XmlValidatingReader` предназначен для работы с `XmlTextReader`, следующий код демонстрирует, как использовать его с `XmlDocument`.

```
XmlDocument doc = new XmlDocument();
XmlTextReader tr = new XmlTextReader("Sample.xml");
XmlValidatingReader reader = new XmlValidatingReader(tr);
doc.Load(reader);
```

В пространстве имен `System.Xml.XPath` определен класс `XPathNavigator`. Этот класс используется для перемещения по XML-документу или для запроса содержимого документа с помощью выражения XPath. Объект класса `XPathNavigator` создается вызовом метода `CreateNavigator()` у объектов классов `XmlDocument`, `XPathDocument`, `XmlDataDocument`. Методы класса `XPathNavigator` позволяют перемещаться по XML-документу, либо делая выборки узлов (группа методов `Select()`) либо получая очередной элемент, атрибут и т.д. (группа методов `MoveToXXX()`). Если производится выборка узлов, то они доступны через итератор `XPathNodeIterator`.

```
// Считываем XML-файл при помощи объекта XPathDocument
XPathDocument document = new XPathDocument("books.xml");
// Создаем навигатор
XPathNavigator navigator = document.CreateNavigator();
// Выбираем определенные узлы документа
XPathNodeIterator nodes = navigator.Select("/bookstore/book");
while(nodes.MoveNext()) {
    Console.WriteLine(nodes.Current.Name);
}
```

2.10. МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ

Классы, предназначенные для поддержки многопоточного программирования, сосредоточены в пространстве имен `System.Threading`. В среде .NET каждый поток представлен объектом класса `System.Threading.Thread`. Для организации собственного потока необходимо создать объект указанного класса. Класс `Thread` имеет единственный конструктор:

```
public Thread(ThreadStart start);
```

В качестве параметра конструктор принимает делегат типа `ThreadStart`, который должен ссылаться на пользовательский метод, выполняемый в потоке.

```
public delegate void ThreadStart();
```

Следует учесть, что создание потока не подразумевает его автоматического запуска. Для запуска потока требуется вызвать у объекта метод `Start()`.

Продемонстрируем создание и запуск потоков на простейшем примере. Программа содержит два потока, каждый из которых в бесконечном цикле выводит данные на консоль:

```
using System;
using System.Threading; //Необходимо для работы с потоками

class MainClass {
    // Эта функция будет выполняться в отдельном потоке
    public static void DoSomeWork() {
        while (true) {
            Console.WriteLine("The second thread");
            // Создаем видимость работы
            for (int i = 0; i < 1000000; i++){ i++; }
        }
    }

    public static void Main() {
        // Создали объект потока
        Thread th = new Thread(new ThreadStart(DoSomeWork));
        // Запустили поток
        th.Start();
        // Создаем видимость работы
        while (true) {
            Console.WriteLine("The first thread");
            for (int i = 0; i < 1000000; i++){ i++; }
        }
    }
}
```

Рассмотрим работу с членами класса `Thread` подробнее. Любой поток характеризуется приоритетом выполнения, который влияет на количество квантов процессорного времени, выделяемых потоку. Для работы с приоритетами класс `Thread` содержит свойство `Priority`, доступное для чтения и записи. Значением свойства являются элементы перечисления `ThreadPriority`: `Lowest`, `BelowNormal`, `Normal`, `AboveNormal`, `Highest`:

```
// Создали объект потока
Thread th = new Thread(new ThreadStart(DoSomeWork));
// Назначим потоку низкий приоритет
th.Priority = ThreadPriority.Lowest;
// Запустим поток
th.Start();
```

Среда исполнения платформы .NET разделяет все потоки на *фоновые* и *основные*. Главное приложение не может завершиться, пока не завершены все его основные потоки. Если работа приложения завершается, а некоторые фоновые потоки еще работают, то их работа автоматически прекращается. Таким образом, к основным следует относить такие потоки, которые выполняют критические для приложения действия. Для установки типа потока следует использовать свойство `IsBackground` булевого типа. Следующий пример показывает применение свойства:

```
using System;
using System.Threading;

class MainClass {
    public static void DoSomeWork() {
        while (true) {
            Console.WriteLine("The second thread");
            // Этот метод приостанавливает поток на 400 мсек
            Thread.Sleep(400);
        }
    }

    public static void Main() {
        Thread th = new Thread(new ThreadStart(DoSomeWork));
        th.IsBackground = true; // Поток будет фоновым
        th.Start();
        // "Усыпили" основной поток на 2 секунды
         Thread.Sleep(2000);
         Console.WriteLine("Quit from main thread");
    }
}
```

По умолчанию любой поток создается как основной, поэтому, если закомментировать строку `th.IsBackground = true` и запустить приложение, то самостоятельно закончить работу оно не сможет.

Класс `Thread` предоставляет встроенные механизмы управления потоками. Метод `Suspend()` вызывает приостановку потока, метод `Resume()` возобновляет работу потока. Статический метод `Sleep()` приостанавливает выполнение того потока, в котором вызывается, на указанное количество миллисекунд:

```
// Пусть объект th связан с некоторым потоком
th.Suspend();           // Приостановили поток
th.Resume();            // Запустили снова
Thread.Sleep(2000);     // "Усыпили" поток на 2 секунды
```

Для приостановки работы произвольного потока на заданное время может использоваться такой код (штатных методов для этого не существует):

```
th.Suspend();
Thread.Sleep(2000);
th.Resume();
```

Если вызвать метод `Sleep()` с параметром `-1`, то поток будет остановлен навсегда.

Метод `Join()` позволяет дождаться завершения работы того потока, у которого вызывается. Модификация данного метода блокирует выполнение текущего потока на указанное количество миллисекунд:

```
Thread th = new Thread(new ThreadStart(DoSomeWork));
th.Start();     // Создали и запустили поток
th.Join();      // Ждем, пока этот поток отработает
. . .
Thread th = new Thread(new ThreadStart(DoSomeWork));
```

```
th.Start();     // Создали и запустили поток

// Будем ждать 1 секунду. Если за это время поток th
// завершиться, то значение res будет true
bool res = th.Join(1000);
```

Для завершения работы выбранного потока используется метод `Abort()`. Данный метод генерирует специальное исключение `ThreadAbortException`. Особенность этого исключения состоит в том, что его невозможно подавить при помощи `catch`-блока. Исключение может быть отслежено (в частности, тем потоком, который кто-то собирается уничтожить), а при помощи статического метода `ResetAbort()` запрос на уничтожение потока можно отклонить.

```
using System;
using System.Threading;

class MainClass {
    public static void ThreadProc() {
        while(true)
            try {
                Console.WriteLine("Some work...");
                Thread.Sleep(1000);
            } catch (ThreadAbortException e) {
              // Отлавливаем попытку уничтожения и отменяем ее
              Console.WriteLine("Somebody tries to kill me!");
                Thread.ResetAbort();
            }
    }

    public static void Main() {
        // Создаем и запускаем поток
        Thread th = new Thread(new ThreadStart(ThreadProc));
        th.Start();
        // Подождем 10 секунд
        Thread.Sleep(10000);
        // Пытаемся прервать работу потока
        th.Abort();
        // Дождемся завершения потока. Вернее, никогда мы его
        // не дождемся, так как поток сам себя "воскресил"
        th.Join();
    }
}
```

Информацию о текущем состоянии потока можно получить посредством свойства `ThreadState`, значением которого являются элементы перечисления `ThreadState`. Свойство `IsAlive` позволяет определить, исполняется ли в данный момент поток. Статическое свойство `CurrentThread` возвращает объект, представляющий текущий поток. Свойство `Name` служит для установки или чтения строки с именем потока.

Рассмотрим один вспомогательный класс из пространства System.Threading – класс Timer. При помощи этого класса можно организовать вызов определенного метода через указанный промежуток времени.

```
using System;
using System.Threading;
class MyApp {
    static bool TickNext = true;

    static void Main() {
        Console.WriteLine("Press Enter to terminate...");
        TimerCallback callback = new TimerCallback(TickTock);
        Timer timer = new Timer(callback, null, 1000, 2000);
        Console.ReadLine();
    }

    static void TickTock(object state) {
        Console.WriteLine(TickNext ? "Tick" : "Tock");
        TickNext = ! TickNext;
    }
}
```

В приведенном примере через 1 секунду после создания (третий параметр в конструкторе Timer) с периодичностью 2 секунды (четвертый параметр конструктора) вызывается метод, заданный делегатом callback (первый параметр конструктора и инициализация делегата). Более подробное описание класса Timer можно найти в соответствующем разделе документации SDK.

2.11. СИНХРОНИЗАЦИЯ ПОТОКОВ

В предыдущем параграфе рассматривались простейшие ситуации, когда несколько потоков работали с независимыми данными. Однако в реальной программе потоки обычно логически связаны друг с другом по данным или ресурсам. Для того чтобы скоординировать их совместную работу, требуются специальные *механизмы синхронизации*.

Критические секции являются простейшим сервисом синхронизации кода. Они позволяют предотвратить одновременное исполнение защищенных участков кода из различных потоков. Рассмотрим следующий пример. Пусть два потока пытаются выводить данные на консоль порциями по 10 символов:

```
using System;
using System.Threading;
class MyApp {
    static void PrintText(string text) {
        for(int i = 0; i < 10; i++) {
            Console.Write(text);
            Thread.Sleep(100);
        }
    }
    static void FirstPrinter() {
        while(true) PrintText("x");
    }
```

```
        static void SecondPrinter() {
            while(true) PrintText("o");
        }

        static void Main() {
          Thread th1 = new Thread(new ThreadStart(FirstPrinter));
          Thread th2 = new Thread(new ThreadStart(SecondPrinter));
          th1.Start();
          th2.Start();
        }
    }
```

Работа данной программы отличается от ожидаемой: на консоль выводятся символы "x" и "o" в случайном порядке, так как в цикл вывод символов на консоль может «вклиниться» другой поток. В данном примере консоль выступает в качестве такого ресурса, доступ к которому требуется *заблокировать*, чтобы с этим ресурсом мог работать только один поток. Вывод последовательности из десяти символов является критической секцией программы.

Язык C# содержит специальный оператор `lock`, задающий критическую секцию. Формат данного оператора следующий:

```
lock(<выражение>) { <блок критической секции> }
```

`<Выражение>` является идентификатором критической секции. В качестве выражения выступает переменная ссылочного типа. Для `lock`-секций, размещенных в экземплярных методах класса, выражение обычно равно `this`, для критических секций в статических методах в качестве выражения используется `typeof(<имя класса>)`.

Изменим предыдущий пример следующим образом:

```
using System;
using System.Threading;
class MyApp {
    static void PrintText(string text) {
        // Задаем критическую секцию
        lock(typeof(MyApp)) {
            for(int i = 0; i < 10; i++) {
                Console.Write(text);
                Thread.Sleep(100);
            }
        }
    }
. . .
}
```

После подобного изменения данные на консоль выводятся правильно – порциями по 10 символов.

Рассмотрим еще один пример, в котором необходимо использование критической секции. Пусть имеется класс с целочисленным массивом и методами, работающими с данным массивом:

```
using System;
using System.Threading;
```

```
class MyApp {
    // В buffer хранятся данные, с которыми работают потоки
    static int[] buffer = new int[100];
    static Thread writer;

    static void Main() {
        // Инициализируем the buffer
        for(int i=0; i<100; i++)
            buffer[i] = i + 1;
        // Запустим поток для перезаписи данных
        writer = new Thread(new ThreadStart(WriterFunc));
        writer.Start();
        // запустим 10 потоков для чтения данных
        Thread[] readers = new Thread[10];
        for(int i=0; i<10; i++) {
            readers[i] =
                      new Thread(new ThreadStart(ReaderFunc));
            readers[i].Start();
        }
    }

    static void ReaderFunc() {
        // Работаем, пока работает поток записи
        while(writer.IsAlive) {
            int sum = 0;
            // Считаем сумму элементов из buffer
            for(int k=0; k<100; k++) sum += buffer[k];
            // Если сумма неправильная, сигнализируем
            if(sum != 5050) {
                Console.WriteLine("Error in sum!");
                return;
            }
        }
    }

    static void WriterFunc() {
        Random rnd = new Random();
        // Цикл на 10 секунд
        DateTime start = DateTime.Now;
        while((DateTime.Now - start).Seconds < 10) {
            int j = rnd.Next(0, 100);
            int k = rnd.Next(0, 100);
            int tmp = buffer[j];
            buffer[j] = buffer[k];
            buffer[k] = tmp;
        }
    }
}
```

При работе данного приложения периодически возникают сообщения о неправильно посчитанной сумме. Причина этого заключается в том, что метод `WriterFunc()` может изменить данные в массиве `buffer` во время подсчета

суммы. Решение проблемы: объявим критическую секцию, содержащую код, работающий с массивом `buffer`.

```
static void ReaderFunc() {
    while(writer.IsAlive) {
        int sum = 0;
        lock(buffer) {
            for(int k=0; k<100; k++) sum += buffer[k];
        }
        // Далее по тексту
        . . .
    }
}

static void WriterFunc() {
    Random rnd = new Random();
    DateTime start = DateTime.Now;
    while((DateTime.Now - start).Seconds < 10) {
        int j = rnd.Next(0, 100);
        int k = rnd.Next(0, 100);
        lock(buffer) {
            int tmp = buffer[j];
            buffer[j] = buffer[k];
            buffer[k] = tmp;
        }
    }
}
```

Обратите внимание на использование одинаковых идентификаторов при указании критической секции (в разных частях программы).

Команда `lock` языка C# – это всего лишь скрытый способ работы со специальным классом `System.Threading.Monitor`. А именно, объявление вида

```
lock(buffer){ . . . }
```

эквивалентно следующему:

```
Monitor.Enter(buffer);
try {
. . .
}
finally {
    Monitor.Exit(buffer);
}
```

Статический метод `Monitor.Enter()` определяет вход в критическую секцию, статический метод `Monitor.Exit()` – выход из секции. Параметрами данных методов является объект – идентификатор критической секции.

Коротко опишем базовые принципы внутренней организации критической секции. Любой объект имеет скрытое поле `syncnum`, которое хранит указатель на элемент таблицы блокировок. Если некоторый поток пытается войти в критическую секцию, выполняется проверка значения `syncnum`. Если данное значение равно `null`, то код критической секции «свободен» и его можно выпол-

нять. В противном случае поток ставиться в системную очередь, из которой извлекается для выполнения тогда, когда критическая секция освободиться.

Вернемся к предыдущему примеру. Требование наличия критической секции в методе `WriterFunc()` очевидно: иначе подсчет суммы может вклиниться между инструкциями `buffer[j] = buffer[k]` и `buffer[k] = tmp` и получить неверное значение. Когда мы считаем сумму в методе `ReaderFunc()`, то очевидно, что мы не должны менять значение массива. Однако и в первом и во втором случае требуется блокировать потоки на одном ресурсе. Соответственно, речь идет об одной критической секции, но как бы «размазанной» по двум методам. Не важно, что мы используем `buffer` в качестве идентификатора критической секции. Это может быть любой инициализированный объект. Таким образом, следующий код также обеспечивает правильную работу:

```
class MyApp {
. . .
    static object someObj = new Random(); // Какой-то объект
. . .
    static void ReaderFunc() {
        while(. . .) {
            . . .
            lock(someObj) {
                . . .
            }
        . . .
        }
    }
    static void WriterFunc() {
        . . .
        while(. . .) {
            . . .
            lock(someObj) {
                . . .
            }
        }
    }
}
```

Если требуется простая синхронизация потоковых действий с целочисленной переменной, то для этих целей можно использовать класс `System.Threading.Interlocked`. Данный класс располагает следующими четырьмя статическими методами:

- `Increment()` – Увеличивает на единицу переменную типа `int` или `long`;
- `Decrement()` – Уменьшает на единицу переменную типа `int` или `long`;
- `Exchange()` – Обменивает значения двух переменных типа `int`, `long` или любых двух объектов;
- `CompareExchange()` – Сравнивает значения первых двух параметров, в случае совпадения заменяет этим значением значение третьего параметра. Тип параметров: `int`, `float`, `object`.

Платформа .NET предоставляет простой способ синхронизации доступа к методам на основе атрибутов. В пространстве имен `System.Runtime.CompilerServices` описан атрибут `MethodImplAttribute`, который может применяться к конструкторам и методам и указывает для компилятора особенности реализации метода. Аргументом атрибута являются элементы перечисления `MethodImplOptions`. В контексте рассматриваемой темы представляет интерес элемент `MethodImplOptions.Synchronized`. Для того чтобы запретить одновременное выполнение некоторого метода в разных потоках, достаточно объявить метод следующим образом:

```
[MethodImpl(MethodImplOptions.Synchronized)]
void TransformData(byte[] buffer) { . . . }
```

При таком объявлении метода можно считать, что любой его вызов будет неявно заключен в критическую секцию.

В заключение рассмотрим класс `System.Threading.ThreadPool`. Данный класс предназначен для поддержки *пула потоков*. Пул потоков автоматически запускает указанные методы в различных потоках. Одновременно пул поддерживает 25 запущенных потоков, другие потоки ожидают своей очереди в пуле.

Для регистрации методов в пуле потока служит статический метод `QueueUserWorkItem()`. Его параметр – это делегат типа `WaitCallback`:

```
public delegate void WaitCallback(object state);
```

При помощи объекта `state` в метод потока передаются параметры.

Рассмотрим пример приложения, использующего `ThreadPool`. В приложении в пул помещается 5 одинаковых методов, выводящих значение счетчика на экран:

```
using System;
using System.Threading;

class MyApp {
    static int count = 0; // счетчик

    static void Main() {
        WaitCallback callback =
                              new WaitCallback(ProcessRequest);
        ThreadPool.QueueUserWorkItem(callback);
        ThreadPool.QueueUserWorkItem(callback);
        ThreadPool.QueueUserWorkItem(callback);
        ThreadPool.QueueUserWorkItem(callback);
        ThreadPool.QueueUserWorkItem(callback);
        // Приостанавливаемся, чтобы выполнились методы
        Thread.Sleep(5000);
    }

    static void ProcessRequest(object state) {
        int n = Interlocked.Increment(ref count);
        Console.WriteLine(n);
    }
}
```

Перегруженная версия метода ThreadPool.QueueUserWorkItem() имеет два параметра: первый – это делегат, второй – объект, при помощи которого делегату можно передать информацию:

```
int[] vals = new int[5]{1, 2, 3, 4, 5};
ThreadPool.QueueUserWorkItem(callback, vals);
// Объявление и реализация ProcessRequest()
static void ProcessRequest(object state) {
    int[] vals = (int[])state;
    . . .
}
```

Поток из пула никогда не должен уничтожаться «вручную». Автоматический менеджер пула потоков берет на себя работу по созданию потока в пуле, он же будет уничтожать потоки. Для того чтобы определить вид потока, можно использовать свойство IsThreadPoolThread класса Thread. В следующем примере поток уничтожает себя только в том случае, если он не запущен из пула:

```
if (!Thread.CurrentThread.IsThreadPoolThread)
    Thread.CurrentThread.Abort();
```

2.12. АСИНХРОННЫЙ ВЫЗОВ МЕТОДОВ

Платформа .NET содержит средства для поддержки асинхронного вызова методов. При *асинхронном вызове* поток выполнения разделяется на две части: в одной выполняется метод, а в другой – нормальный процесс программы. Асинхронный вызов может служить (в некоторых случаях) альтернативой использованию многопоточности явным образом.

Асинхронный вызов метода всегда выполняется посредством объекта некоторого делегата. Любой такой объект содержит два специальных метода для асинхронных вызовов – BeginInvoke() и EndInvoke(). Данные методы генерируются во время выполнения программы (как и метод Invoke()), так как их сигнатура зависит от делегата.

Метод BeginInvoke() обеспечивает асинхронный запуск. Данный метод имеет два дополнительных параметра по сравнению с описанными в делегате. Назначение первого дополнительного параметра – передать делегат, указывающий на функцию обратного вызова, выполняемую после работы асинхронного метода (*функция завершения*). Второй дополнительный параметр – это объект, при помощи которого функции завершения может быть передана некоторая информация. Метод BeginInvoke() возвращает объект, реализующий интерфейс IAsyncResult, при помощи этого объекта становится возможным различать асинхронные вызовы одного и того же метода.

Приведем описание интерфейса IAsyncResult:

```
interface IAsyncResult {
    object AsyncState{ get; }
    WaitHandle AsyncWaitHandle{ get; }
    bool CompletedSynchronously{ get; }
    bool IsCompleted{ get; }
}
```

Поле IsCompleted позволяет узнать, завершилась ли работа асинхронного метода. В поле AsyncWaitHandle храниться объект типа WaitHandle. Программист может вызывать методы класса WaitHandle, такие как WaitOne(), WaitAny(), WaitAll(), для контроля над потоком выполнения асинхронного делегата. Объект AsyncState хранит последний параметр, указанный при вызове BeginInvoke().

Делегат для функции завершения описан следующим образом:

```
public delegate void AsyncCallback(IAsyncResult ar);
```

Как видим, функции завершения передается единственный параметр: объект, реализующий интерфейс IAsyncResult.

Рассмотрим пример, иллюстрирующий описанные возможности.

```
using System;
using System.Threading;  // Нужно для "усыпления" потоков

// Делегат для асинхронного метода
public delegate void Func(int x);

class MainClass {
    // Этот метод делает необходимую работу
    public static void Fib(int n) {
        int a = 1, b = 1, res = 1;
        for(int i = 3; i <= n; i++) {
            res = a + b;
            a = b;
            b = res;
            Thread.Sleep(10); // Намерено замедлим!
        }
        Console.WriteLine("Fib calculated: " + res);
    }

    public static void Main() {
        Func A = new Func(Fib);
        // Асинхронный вызов "выстрелил и забыл"
        // У BeginInvoke() три параметра, два не используем
        A.BeginInvoke(6, null, null);
        // Изображаем работу
        for (int i = 1; i < 10; i++) {
            Thread.Sleep(20);
            Console.Write(i);
        }
    }
}
```

Вывод программы:

```
12Fib calculated: 8
3456789
```

В данном приложении имеется функция для подсчета n-ного числа Фибоначчи. Чтобы эмулировать продолжительные действия, функция намеренно за-

медлена. После подсчета число выводится на экран. Ни функции завершения, ни возвращаемое `BeginInvoke()` значение не используется. Подобный метод работы с асинхронными методами называется *«выстрелил и забыл»* (*fire and forget*).

Модифицируем предыдущее приложение. Будем использовать при вызове `BeginInvoke()` функцию завершения, выводящую строку текста:

```
using System;
using System.Threading;

public delegate void Func(int x);

class MainClass {
    public static void Fib(int n) { . . . }

    // Это будет функция завершения
    public static void Callback(IAsyncResult ar) {
        // Достаем параметр
        string s = (string) ar.AsyncState;
        Console.WriteLine("AsyncCall is finished with " + s);
    }

    public static void Main() {
        Func A = new Func(Fib);
        // Два асинхронных вызова
        A.BeginInvoke(6, new AsyncCallback(Callback), "The end");
        A.BeginInvoke(8, new AsyncCallback(Callback), "Second call");
        // Изображаем работу
        for (int i = 1; i < 10; i++) {
            Thread.Sleep(20);
            Console.Write(i);
        }
    }
}
```

Вывод программы:

```
12Fib calculated: 8
Async Call is finished with The end
345Fib calculated: 21
Async Call is finished with Second call
6789
```

В рассмотренных примерах использовались такие асинхронные методы, которые не возвращают значения. В приложениях может возникнуть необходимость работать с асинхронными методами-функциями. Для этой цели предназначен метод делегата `EndInvoke()`. Сигнатура метода `EndInvoke()` определяется на основе сигнатуры метода, инкапсулированного делегатом. Во-первых, метод является функцией, тип возвращаемого значения – такой как у делегата. Во-вторых, метод `EndInvoke()` содержит все `out`- и `ref`- параметры делегата, а его последний параметр имеет тип `IAsyncResult`. При вызове метода `EndIn-`

voke() основной поток выполнения приостанавливается до завершения работы соответствующего асинхронного метода.

Изменим метод Fib() из примера. Пусть он имеет следующую реализацию:

```
public static int Fib(int n, ref bool overflow) {
    int a = 1, b = 1, res = 1;
    overflow = false;
    for (int i = 3; i <= n; i++) {
        res = a + b;
        // Устанавливаем флаг переполнения
        if (res < 0) overflow = true;
        a = b;
        b = res;
    }
    return res;
}
```

В следующем примере запускаются два асинхронных метода, затем приложение дожидается их выполнения и выводит результаты на экран.

```
using System;
using System.Threading;

// Вот такой у нас теперь делегат
public delegate int Func(int n, ref bool overflow);

class MainClass {

    // Функция считает числа Фибоначчи, следя за переполнением
    public static int Fib(int n, ref bool overflow) {
        int a = 1, b = 1, res = 1;
        overflow = false;
        for(int i = 3; i <= n; i++) {
            res = a + b;
            // Устанавливаем флаг переполнения
            if(res < 0) overflow = true;
            a = b;
            b = res;
        }
        return res;
    }

    public static void Main() {
        bool over = false;
        Func A = new Func(Fib);
        // Так как отслеживаем окончание работы методов,
        // сохраняем результат работы BeginInvoke()
        IAsyncResult ar1 = A.BeginInvoke(10, ref over, null, null);
        IAsyncResult ar2 = A.BeginInvoke(50, ref over, null, null);

        // Имитируем бурную деятельность
        for (int i = 1; i < 10; i++) {
```

```
            Thread.Sleep(20);
            Console.Write(i);
        }

        // Вспомнили про методы. Остановились, ждем результат
        int res = A.EndInvoke(ref over, ar2);
        Console.WriteLine("Result is {0}, overflowed = {1}",
                          res, over);
        // Теперь второй метод
        res = A.EndInvoke(ref over, ar1);
        Console.WriteLine("Result is {0}, overflowed = {1}",
                          res, over);
    }
}
```

Вывод программы:

```
123456789Result is -298632863, overflowed = True
Result is 55, overflowed = False
```

Подведем небольшой итог. Асинхронный вызов является альтернативой использования многопоточности, так как реализует ее неявно, при помощи среды исполнения. Широкий спектр настроек позволяет решать при помощи асинхронных вызовов большой круг практических задач программирования.

2.13. СОСТАВ И ВЗАИМОДЕЙСТВИЕ СБОРОК

Сборка (assembly) – это единица развертывания и контроля версий в .NET Framework. Заметим, что сборка задает границы видимости для типов (модификатор `internal` в C#). Сборка состоит из одного или нескольких *программных модулей* и *файлов ресурсов*. Эти компоненты могут размещаться в отдельных файлах, либо содержаться в одном файле. В любом случае, сборка содержит в некотором из своих файлов *манифест*, описывающий состав сборки. Будем называть сборку *однофайловой*, если она состоит из одного файла. В противном случае сборку будем называть *многофайловой*. Тот файл, который содержит манифест сборки, будем называть *главным файлом сборки*.

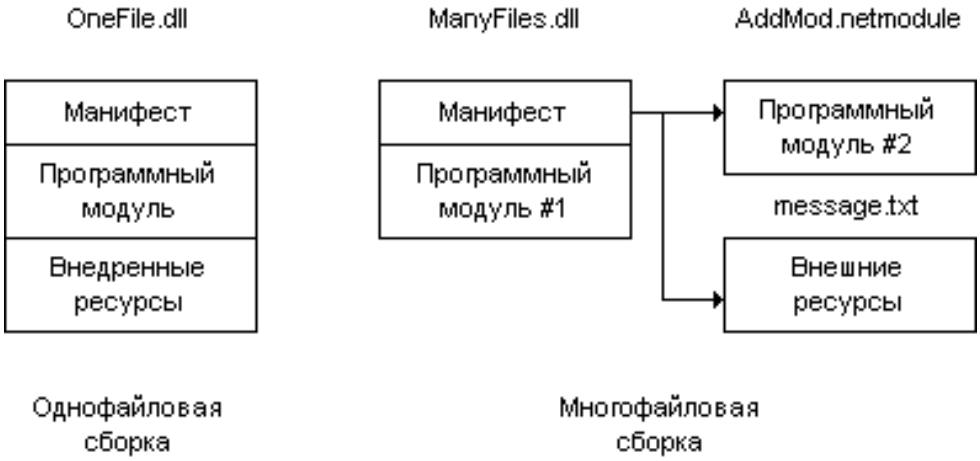

Рис. 5. Однофайловая и многофайловая сборки

Простые приложения обычно представлены однофайловыми сборками. При разработке сложных приложений переход к многофайловым сборкам дает следующие преимущества:

1. Ресурсы приложения (текстовые строки, изображения и т.д.) можно хранить вне кода приложения, что позволяет при необходимости изменять их без перекомпиляции приложения.
2. Если исполняемый код приложения разделен на несколько модулей, то модули загружаются в память только по мере необходимости. Кроме этого, скомпилированный модуль может использоваться в нескольких сборках.

Рассмотрим пример создания и использования многофайловых сборок. К сожалению, IDE Visual Studio не позволяет работать с многофайловыми сборками, поэтому все файлы примера придется компилировать, используя *компилятор командной строки* `csc.exe`.

Пусть требуется создать консольное приложение, в котором функция `Main()` печатает на экране строку. Содержимое строки оформим в виде текстового *файла-ресурса*. Код, который читает данный ресурс, будет оформлен в виде отдельного модуля (файла с расширением `netmodule`).

Ниже представлен класс, выполняющий чтение строки из подключенного к сборке ресурса. Код, выполняющий требуемые операции, стандартен и взят из примеров MSDN:

```
using System;
using System.Reflection;
using System.IO;

public class TextClass {
    public static string GetText() {
        // получаем ссылку на выполняемую сборку
        Assembly a = Assembly.GetExecutingAssembly();
        // получаем доступ к связанному ресурсу как к потоку
        Stream s = a.GetManifestResourceStream("message.txt");
        // "оборачиваем" в текстовый поток (специфика ресурса)
        StreamReader sr = new StreamReader(s);
        return sr.ReadToEnd();
    }
}
```

Файл `TextClass.cs` с исходным текстом класса `TextClass` скомпилируем в виде модуля (обратите внимание на ключ компилятора):

```
C:\Temp\Test>csc /t:module TextClass.cs
```

После компиляции получим файл-модуль `TextClass.netmodule`. Далее, создадим главный файл нашего приложения (`main.cs`):

```
using System;
class MainClass {
    static void Main() {
        Console.WriteLine("Text from resource");
        Console.WriteLine(TextClass.GetText());
```

```
            Console.ReadLine();
        }
    }
```

Также создадим собственно текстовый файл-ресурс с именем `message.txt` (внимание: при работе с текстовыми ресурсами предпочтительнее использовать кодировку Unicode).

Теперь соберем нашу многофайловую сборку:

```
c:\TEMP>csc /linkresource:message.txt
            /addmodule:textclass.netmodule main.cs
```

Обратите внимание на ключи компилятора. Ключ `/addmodule` позволяет добавить к сборке ссылку на внешний файл-модуль, ключ `/linkresource` позволяет связать со сборкой внешний файл-ресурс. Ключи могут использоваться произвольное количество раз.

В итоге, мы получили многофайловую сборку `main.exe`, состоящую из трех файлов: главного файла `main.exe`, дополнительного файла-модуля `textclass.netmodule` и файла-ресурса `message.txt`. Еще раз подчеркнем возможные преимущества многофайловых сборок. Во-первых, мы можем менять содержимое файла `message.txt`, не перекомпилируя сборку. Во-вторых, мы можем создать новую сборку, в которой используется код из модуля `textclass.netmodule`, то есть сделать этот модуль *разделяемым* между несколькими сборками. Важное замечание: предполагается, что все три файла, составляющие нашу многофайловую сборку, размещены в одном каталоге.

Следующий вопрос, рассматриваемый в данном параграфе, это *взаимодействие сборок*. Как правило, большие проекты состоят из нескольких сборок, ссылающихся друг на друга. Среди этих сборок имеется некая основная (обычно оформленная как исполняемый файл `*.exe`), а другие сборки играют роль подключаемых библиотек с кодом необходимых классов (обычно такие сборки – это файлы с расширением `*.dll`). Платформа .NET разделяет сборки на локальные (или *сборки со слабыми именами*) и глобальные (или *сборки с сильными именами*).

Представим пример, который будем использовать в дальнейшем. Пусть имеется следующий класс (в файле `UL.cs`), содержащий «полезную» функцию:

```
using System;
namespace UsefulLibrary {
    public class UsefulClass {
        public void Print() {
            Console.WriteLine("Useful function");
        }
    }
}
```

Скомпилируем данный класс как DLL:

```
c:\TEMP\Test>csc /t:library UL.cs
```

Пусть основное приложение (файл `main.cs`) собирается использовать код из сборки `UL.dll`:

```
using System;
// подключаем требуемое пространство имен
using UsefulLibrary;

class MainClass {
    static void Main() {
        // используем класс
        UsefulClass a = new UsefulClass();
        a.Print();
        Console.ReadLine();
    }
}
```

Если `UL.dll` рассматривается как *локальная сборка*, то она **должна находиться в том же каталоге**[1], что и основное приложение `main.exe` как при компиляции, так и при выполнении приложения. Скомпилируем основное приложение `main.cs`:

```
c:\TEMP\Test>csc /r:UL.dll main.cs
```

Ключ компилятора `/r` (или `/reference`) позволяет установить ссылку на требуемую сборку.

Применением локальных сборок достигается простота развертывания приложения (все его компоненты сосредоточены в одном месте) и изолированность компонентов приложения. Имя локальной сборки – *слабое имя* – это имя файла сборки без расширения.

Хотя использование локальных сборок имеет свои преимущества, иногда необходимо сделать сборку общедоступной. До появления .NET Framework основным был подход, при котором код общих библиотек помещался в системный каталог простым копированием фалов при установке программ. Такой подход привел к проблеме, известной как «ад DLL» (DLL Hell). Новое установленное приложение могло заменить требуемую библиотеку новой версией, причем сделать это так, что приложения, ориентированные на старую версию библиотеки, переставали работать. Для решения проблемы DLL Hell в .NET Framework используется специальное *защищенное хранилище сборок* (*Global Assembly Cache*, *GAC*).

Сборки, помещаемые в GAC, должны удовлетворять определенным условиям. Во-первых, такие *глобальные сборки* должны иметь цифровую подпись. Это исключает подмену сборок злоумышленниками. Во-вторых, для глобальных сборок отслеживаются версии, и вполне допустимой является ситуация, когда в GAC находятся разные версии одной и той же сборки, используемые разными приложениями.

Сборка, помещенная в GAC, получает *сильное имя*. Именно использование сильного имени является тем признаком, по которому среда исполнения понимает, что речь идет не о локальной сборке, а о сборке из GAC. Сильное имя имеет следующую структуру:

```
Name, Version=1.2.0.0, Culture=neutral, PublicKeyToken=1234567812345678
```

[1] Данное правило можно обойти при помощи файла конфигурации сборки.

Сильное имя включает собственно имя главного файла сборки без расширения, версию сборки, указание о региональной принадлежности сборки и маркер открытого ключа сборки.

Рассмотрим процесс создания строго именованной сборки на примере сборки `UL.dll`. Первое: необходимо создать пару криптографических ключей для цифровой подписи сборки. Для этих целей служит утилита `sn.exe`, входящая в состав Microsoft .NET Framework SDK.

```
sn -k keys.snk
```

Параметр `-k` указывает на создание ключей, `keys.snk` – это файл с ключами. Просмотреть полученные ключи можно, используя команду `sn -tp`.

Далее необходимо *подписать* сборку полученными ключами. Для этого используется специальный атрибут уровня сборки `AssemblyKeyFile`:

```
using System;
using System.Reflection;

[assembly: AssemblyKeyFile("keys.snk")]

namespace UsefulLibrary { . . . }
```

Обратите внимание: для использования атрибута необходимо подключить пространство имен `System.Reflection`; в качестве параметра атрибута указывается полное имя файла с ключами; атрибут должен располагаться вне любого пространства имен.

После подписывания сборку можно поместить в GAC. Простейший вариант сделать это – использовать утилиту `gacutil.exe`, входящую в состав .NET Framework SDK. При использовании ключа `/i` сборка помещается в GAC, а ключ `/u` удаляет сборку из GAC:

```
gacutil /i ul.dll
```

Теперь сборка `UL.dll` помещена в GAC. Ее сильное имя (для ссылки в программах) имеет вид:

```
UL, Version=0.0.0.0, Culture=neutral, PublicKeyToken= ff824814c57facfe
```

Компонентом сильного имени является *версия сборки*. Если программист желает указать версию, то для этого используется атрибут `AssemblyVersion`. Номер версии имеет формат `Major.Minor.Build.Revision`. Если номер версии задается, то часть `Major` является обязательной. Любая другая часть может быть опущена (в этом случае она полагается равной нулю). Часть `Revision` можно задать как `*`, тогда компилятор генерирует ее как количество секунд, прошедших с полуночи, деленное на два. Часть `Build` также можно задать как `*`. Для нее будет подставлено количество дней, прошедших с 1 февраля 2000 года. Пример использования атрибута `AssemblyVersion`:

```
using System;
using System.Reflection;

[assembly: AssemblyVersion("1.2.3.*")]
. . .
```

2.14. КОНФИГУРИРОВАНИЕ СБОРОК

Необходимость конфигурирования сборок обычно возникает при развертывании приложений. Платформа .NET предлагает стандартизированный подход к конфигурированию, основанный на использовании *конфигурационных XML-файлов*. Специальное пространство имен `System.Configuration` отвечает за работу с файлами конфигураций.

Рассмотрим общую схему файла конфигурации. Корневым элементом файла всегда является элемент `configuration`. Некоторые подчиненные элементы описаны далее:

- `startup` – совокупность параметров запуска приложения;
- `runtime` – параметры времени выполнения. Они регулируют способ загрузки сборок и сборку мусора;
- `system.runtime.remoting` – параметры конфигурирования, необходимые для настройки механизма .NET Remoting;
- `system.diagnostics` – совокупность диагностических параметров, которые задает способ отладки приложений, перенаправляют сообщения отладки и т. д.;
- `system.web` – параметры конфигурации приложений ASP.NET.

Платформа .NET Framework имеет один файл конфигурации компьютера с параметрами, относящимися к системе в целом. Этот файл называется `machine.config`. Любая сборка может иметь собственный конфигурационный файл. Он должен носить имя файла сборки (с расширением) с добавленным окончанием `.config` и располагаться в одном каталоге со сборкой. Таким образом, файл конфигурации для `main.exe` должен называться `main.exe.config`. В случае web-приложений файл конфигурации всегда называется `web.config`.

Разберем на примерах некоторые возможности конфигурирования. Рассмотрим подробнее структуру секции `runtime`:

```
<configuration>
   <runtime>
      <developmenMode>
      <assemblyBinding>
         <probing>
         <publisherPolicy>
         <qualifyAssembly>
         <dependentAssembly>
            <assemblyIdentity>
            <bindingRedirect>
            <codeBase>
            <publisherPolicy>
      <gcConcurrent>
```

`<developmenMode>`

Прототип тэга:

```
<developmenMode developerInstallation = "true" | "false"/>
```

Если параметр тэга равен "true", то среда исполнения помимо обычных каталогов пытается искать сборки в каталогах, указанных в переменной среды окружения DEVPATH.

<assemblyBinding>
Тэг служит для объединения группы вложенных элементов

<probing>
Прототип тэга:

```
<probing privatePath = "paths"/>
```

Тэг используется для указания подкаталогов, в которых производится поиск локальных сборок приложения. Например, пусть основное приложение AppMain.exe размещено в каталоге C:\Test, и оно использует сборку Add1.dll из каталога C:\Test\bin и сборку Add2.dll из каталога C:\Test\bin\bin2. Тогда для корректного запуска основного приложения файл AppMain.exe.config должен иметь следующий вид:

```
<configuration>
   <runtime>
      <assemblyBinding>
         <probing privatePath = "bin;bin\bin2"/>
      </assemblyBinding>
   </runtime>
</configuration>
```

<publisherPolicy>
Прототип тэга:

```
<publisherPolicy apply = "yes" | "no"/>
```

Пусть приложение использует некую стороннюю библиотеку. Если в тэге <publisherPolicy> определен параметр "yes" (по умолчанию), то при установке новой версии библиотеки, приложение автоматически будет использовать эту новую версию. В случае параметра "no" приложение откажется работать с новой версией.

<qualifyAssembly>
Прототип тэга:

```
<qualifyAssembly partialName="PartName" fullName="FullName"/>
```

Этот тэг позволяет ассоциировать сокращенное имя сборки с полным строгим именем для удобства использования в программе. Рассмотрим следующий пример:

```
<configuration>
   <runtime>
      <assemblyBinding>
         <qualifyAssembly
            partialName = "Strong"
            fullName = "Strong, version=1.2.0.0,
```

```
                        culture=neutral,
                        publicKeyToken=1234567812345678"/>
      </assemblyBinding>
   </runtime>
</configuration>
```

Конфигурационный файл данного примера позволит везде в программе вместо длинного строго имени использовать короткий псевдоним `Strong`.

<dependentAssembly>

Этот тэг позволяет настроить загрузку персонально для каждой сборки, связанной с основным приложением. Он сам является пустым и объединяет элементы `<assemblyIdentity>`, `<bindingRedirect>`, `<codeBase>`, `<publisherPolicy>`. Элемент должен быть указан для каждой сборки, политику загрузки которой мы собираемся настраивать.

<assemblyIdentity>

Прототип тэга:

```
<assemblyIdentity
   name = "assembly Name" publicKeyToken = "public key token"
   culture = "assembly culture"/>
```

Тэг идентифицирует сборку

<bindingRedirect>

Прототип тэга:

```
<bindingRedirect
   old version = "old assembly version"
   new version = "new assembly version"/>
```

Тэг позволяет подменить версию сборки, которая будет загружаться приложением. Рассмотрим пример:

```
<configuration>
   <runtime>
      <assemblyBinding>
         <dependentAssembly>
            <assemblyIdentity name = "myAssembly"
                              publicKeyToken="32ab4ba45e0a69a1"
                              culture = "neutral"/>
            <bindingRedirect oldVersion = "1.0.0.0"
                              newVersion = "2.0.0.0"/>
         </dependentAssembly>
      </assemblyBinding>
   </runtime>
</configuration>
```

Теперь, если основное приложение запросит сборку `myAssembly` версии `1.0.0.0`, то, ничего не зная об этом, получит сборку версии `2.0.0.0`.

<codeBase>

Прототип тэга:

```
<codeBase version="assembly version" href="URL of assembly"/>
```

Этот тэг позволяет задать произвольное местоположение зависимой сборки. Причем, в качестве параметра `href` может быть указан сетевой адрес. Продемонстрируем это на примере:

```
<configuration>
  <runtime>
    <assemblyBinding>
      <dependentAssembly>
        <assemblyIdentity name = "myAssembly"
                          publicKeyToken = "32ab4ba45e0a69a1"
                          culture = "neutral"/>
        <codeBase version = "1.0.0.0"
                  href = "http://localhost/one/Strong.dll"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Этот конфигурационный файл предписывает загружать среде исполнения сборку `Strong` из Интернета по адресу `http://localhost/one/Strong.dll`.

<gcConcurrent>

Прототип тэга:

```
<gcConcurrent enabled = "true" | "false"/>
```

Если параметр `enabled` установлен в `true`, то «сборка мусора» происходит в параллельном потоке по отношению к основному приложению. Это поведение по умолчанию.

Рассмотрим задачу размещения собственных данных в конфигурационном файле сборки. В простейшем варианте для этого используется секция `appSettings`. Данная секция может содержать следующие элементы:

- `<add key = "name" value = "the value"/>` – добавляет новый ключ и значение в коллекцию конфигурационных элементов;
- `<remove key = "name"/>` – удаляет существующий ключ и значение из коллекции конфигурационных элементов;
- `<clear/>` – очищает коллекцию конфигурационных элементов.

Для работы с данными секции `appSettings` используется класс `ConfigurationSettings` из пространства имен `Sistem.Configuration`. Статическое свойство `AppSettings` представляет собой коллекцию, позволяющую получить строковое значение конфигурационного элемента по ключу.

Пусть имеется следующее консольное приложение `main.exe`:

```
using System;
using System.Configuration;

class MainClass {
    static void Main() {
        string v1 = ConfigurationSettings.AppSettings["key1"];
        string v2 = ConfigurationSettings.AppSettings["key2"];
```

```
            Console.WriteLine(v1);
            Console.WriteLine(v2);
            Console.ReadLine();
        }
    }
```

Создадим для этого приложения конфигурационный файл `main.exe.config`:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <appSettings>
        <add key="key1" value="Alex" />
        <add key="key2" value="Volosevich" />
    </appSettings>
</configuration>
```

Результат выполнения `main.exe`:

```
Alex
Volosevich
```

Все секции конфигурационного файла разбираются специальными *классами-обработчиками* (*section handlers*). Такой класс реализует интерфейс `System.Configuration.IConfigurationSectionHandler`. В составе .NET Framework имеется несколько готовых классов-обработчиков: `NameValueSectionHandler`, `IgnoreSectionHandler`, `DictionarySectionHandler`, `SingleTagSectionHandler`.

Чтобы описать в конфигурационном файле собственную секцию, следует использовать тэг `configSections`. Вложенные элементы `section` данного тэга имеют атрибут `name` – имя секции, и атрибут `type` – полное имя класса-обработчика. Рассмотрим следующий пример конфигурационного файла:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name = "mySection"
             type = "System.Configuration.NameValueSectionHandler" />
  </configSections>

  <mySection>
    <add key="key1" value="Alex" />
    <add key="key2" value="Volosevich" />
  </mySection>
</configuration>
```

Этот файл задает пользовательскую секцию с именем `mySection` и указывает, что для ее обработки следует использовать `NameValueSectionHandler`, то есть, содержимое секции будет доступно как `NameValueCollection`.

Для чтения информации из пользовательской секции применяется метод `ConfigurationSettings.GetConfig()`. В качестве параметра методу передается имя секции. Метод возвращает значение типа `object`, которое следует при-

вести к типу класса-обработчика секции. Следующее приложение читает информацию из своего (описанного выше) конфигурационного файла:

```
using System;
using System.Configuration;
using System.Collections.Specialized;

class CustomConfig {
  static void Main() {
    NameValueCollection nvc =
      (NameValueCollection)ConfigurationSettings.GetConfig("mySection");
    Console.WriteLine(nvc["key1"]);
    Console.WriteLine(nvc["key2"]);
  }
}
```

Следует иметь в виду, что файл конфигурации читается и разбирается исполняемой средой один раз, при загрузке приложения. Поэтому все изменения, которые внесены в этот файл, не буду иметь эффекта до перезагрузки приложения. Если такое поведение является неприемлемым, то можно воспользоваться классом `System.IO.FileSystemWatcher`. Как следует из его названия, класс позволяет отследить изменения, происходящие с файлом (или несколькими файлами). Подробности работы с классом `FileSystemWatcher` можно найти в .NET Framework SDK.

3. ТЕХНОЛОГИЯ .NET REMOTING

3.1. ДОМЕНЫ ПРИЛОЖЕНИЙ

Любому запущенному приложению в операционной системе соответствует некий *процесс*. Процесс образует границы приложения, и для взаимодействия процессов требуется применять специальные средства. В .NET Framework процессы дополнительно подразделяются на *домены приложений*. Один домен может содержать несколько сборок. Различным доменам могут соответствовать различные политики безопасности, домены могут создаваться и уничтожаться в ходе работы в рамках одного приложения.

Рассмотрим работу с доменами на примерах. Пусть имеется следующая сборка, размещенная в динамической библиотеке `Students.dll`:

```
using System;

public class Student {
    public string Name;
    public int Age;
    public double MeanScore;
    public Student(string Name, int Age, double MeanScore) {
        this.Name = Name;
        this.Age = Age;
        this.MeanScore = MeanScore;
        log.print("New Student object created");
    }
    public void sayName() {
        log.print("sayName() is called");
        Console.WriteLine("My name is {0}", Name);
    }
}

public class log {
    public static void print(string s) {
        Console.WriteLine("[{0}]: {1}",
                    AppDomain.CurrentDomain.FriendlyName,s);
    }
}
```

Домены приложений инкапсулированы в объектах класса `System.AppDomain`. Любое приложение при запуске создает *домен по умолчанию*, в который загружается главная сборка приложения. Для создания нового домена используется статический метод `AppDomain.CreateDomain()`. Существует несколько перегруженных версий этого метода, которые обеспечивают применение к создаваемому домену определенных *политик безопасности*. В простейшем случае методу передается в качестве параметра только строка с *дружественным именем* домена, которое используется как идентификатор домена:

```
AppDomain ad2 = AppDomain.CreateDomain("New Domain");
```

После создания домена в него можно загрузить сборки, используя метод `Load()`[1] (указывается слабое или сильное имя сборки):

```
ad2.Load("Students");
```

Отдельную сборку выгрузить из домена нельзя, но можно выгрузить домен целиком, используя его метод `Unload()`:

```
AppDomain.Unload(ad2);
```

В любом домене можно создавать объекты различных типов. Данные действия выполняются при помощи методов `CreateInstance()`, `CreateInstanceFrom()`, `CreateInstanceAndUnwrap()`, `CreateInstanceFromAndUnwrap()`. Рассмотрим подробнее метод `CreateInstance()`. Существует несколько перегруженных версий этого метода. В простейшем случае в качестве параметров метода указывается имя сборки и полное имя типа. Самый полный вариант `CreateInstance()` имеет следующую сигнатуру:

```
public virtual ObjectHandle CreateInstance(
                              string assemblyName,
                              string typeName,
                              bool ignoreCase,
                              BindingFlags bindingAttr,
                              Binder binder,
                              object[] args,
                              CultureInfo culture,
                              object[] activationAttributes,
                              Evidence securityAttributes);
```

Опишем параметры метода `CreateInstance()`. Первый параметр указывает имя сборки, содержащей тип создаваемого объекта. Сборка может быть загружена в домен, хотя это и не обязательно. Второй параметр – строка с полным именем создаваемого типа. При помощи третьего параметра можно сделать поиск типа в сборке регистронезависимым. Четвертый параметр позволяет задать область поиска конструктора типа. Если данный параметр равен нулю, выполняется регистрозависимый поиск `public`-конструктора типа[2]. Параметр `binder` позволяет указать особый способ связывания и подстановки вызовов методов и полей. Обычно используется способ по умолчанию (значение параметра – `null`). Следующим указывается массив, содержащий параметры конструктора. Параметр `culture` указывает культуру создаваемого объекта (актуально для приложений, настраиваемый на локализацию). Предпоследний параметр позволяет задать массив объектов-атрибутов, участвующих в активации создаваемого объекта. Последний параметр управляет атрибутами безопасности создаваемого объекта.

Метод `CreateInstanceFrom()` практически идентичен `CreateInstance()`, но сборка загружается из указанного файла. В следующем фрагменте кода происходит создание нового домена приложения и создание объекта типа `Student` в этом домене:

---

[1] Метод `Load()` имеет несколько перегруженных версий.

[2] Описание типа `BindingFlags` смотрите в параграфе, посвященном механизму отражения.

```
AppDomain ad2 = AppDomain.CreateDomain("New Domain");
ad2.CreateInstanceFrom(@"C:\Students.dll", // Имя файла сборки
                       "Student",   // Имя класса
                       false,       // Регистрозависимый поиск
                       // Указываем, что ищем конструктор
                       // Хотя с таким же успехом можно написать 0
                       BindingFlags.CreateInstance,
                       null, // Особой привязки не используем
                       // Параметры конструктора
                       new object[]{"Ivanov", 20, 4.5},
                       null, // Остальные параметры игнорируем
                       null, // или принимаем по умолчанию
                       null);
```

Приведенный фрагмент кода демонстрирует создание объекта в другом домене, однако дальнейшая работа с объектом невозможна – мы просто не имеем никакой ссылки на созданный объект. Технология передачи объектов через границы доменов (процессов) называется *маршалинг*. Пусть имеются два домена – D1 и D2. Предположим, что в домене D1 требуется производить работу с объектом Obj из домена D2. При маршалинге возможна передача объектов как по ссылке, так и по значению. В случае передачи по значению в домене D1 будет создана локальная копия объекта Obj из D2. Естественно, все изменения, внесенные в Obj из D1 не отразятся на Obj из D2.

В случае передачи объекта по ссылке среда исполнения использует специальный *объект-прокси* (посредник) Pr, который в домене D1 выглядит как объект Obj, но фактически перенаправляет все вызовы своих методов к объекту Obj в домене D2.

Простейший способ осуществить маршалинг объектов по значению – пометить тип как сериализуемый (атрибутом `[Serializable]`). Пусть таким образом помечен тип `Student`. В предыдущем примере значение, возвращаемое `CreateInstance()`, игнорировалось. В следующем фрагменте это значение приводится к типу `Student`, и демонстрируется работа с элементами созданного объекта:

```
using System;
using System.Reflection;
using System.Runtime.Remoting;

class MainClass {
    public static void Main() {
        AppDomain ad2 = AppDomain.CreateDomain("New Domain");
        ObjectHandle oh = ad2.CreateInstanceFrom(
                          @"C:\Temp\Students.dll", "Student",
                          false, BindingFlags.CreateInstance,
                          null,
                          new object[]{"Ivanov", 20, 4.5},
                          null, null, null);
        Student s = (Student)oh.Unwrap();
        s.sayName();
        s.Name = "Petrov";
```

```
            s.sayName();
            Student s_new = (Student)oh.Unwrap();
            s_new.sayName();
        }
    }
```

Вот что выводит данная программа на консоль:

```
[New Domain]: New Student object created
[AppDom.exe]: sayName() is called
My name is Ivanov
[AppDom.exe]: sayName() is called
My name is Petrov
[AppDom.exe]: sayName() is called
My name is Ivanov
```

Как видим, объект был создан в одном домене, но затем копия объекта была перемещена в другой домен, где с ней и происходила работа.

Обратите внимание: метод `CreateInstanceFrom()` возвращает объект специального класса `ObjectHandle`. Данный класс реализует «обертки» над реальными объектами, что позволяет не передавать через границы доменов метаданные. Для получения реального объекта используется экземплярный метод `ObjectHandle.Unwrap()`, который возвращает либо объект, либо прокси. Метод `CreateInstanceFromAndUnwrap()` выполняет «разворачивание» созданного объекта автоматически.

Чтобы передача объектов некоторого типа между границами доменов происходила по ссылке, требуется, чтобы тип был унаследован от типа `MarshalByRefObject`. Если допустить, что тип `Student` унаследован от этого типа, то предыдущий пример выведет на консоль следующее:

```
[New Domain]: New Student object created
[New Domain]: sayName() is called
My name is Ivanov
[New Domain]: sayName() is called
My name is Petrov
[New Domain]: sayName() is called
My name is Petrov
```

В завершение приведем таблицу, содержащую некоторые методы и свойства класса `System.AppDomain`.

Таблица 17

Некоторые элементы класса `System.AppDomain`

Имя элемента	Описание
`CurrentDomain`	Статическое свойство, возвращающее текущий домен текущего потока
`CreateDomain()`	Статический метод для создания домена
`GetCurrentThreadID()`	Статическое свойство, возвращает идентификатор текущего потока
`Unload()`	Статический метод который удаляет указанный домен
`FriendlyName`	Строка с именем домена
`DefineDynamicAssembly()`	Метод позволяет динамически создавать сборки

`ExecuteAssembly()`	Метод запускает на выполнение указанную сборку
`GetData()`	Метод позволяет получить данные, сохраненные в домене под неким именем-ключом
`Load()`	Метод загружает сборки в домен
`SetAppDomainPolicy()`	Метод для установки политики безопасности домена
`SetData()`	Метод позволяет сохранить данные в домене под неким именем-ключом

3.2. АРХИТЕКТУРА .NET REMOTING

В настоящее время все большее распространение получают распределенные приложения. В *распределенном приложении* (*distributed application*) отдельные компоненты выполняются на различных компьютерах, которые связаны сетью передачи данных. Как правило, компонент распределенного приложения реализует некий *сервис*, иначе говоря, предоставляет определенные услуги, доступные путем вызова методов компонента. Сам компонент является объектом некоторого класса. Таким образом, создание распределенного приложения подразумевает возможность вызова на одном компьютере методов объекта, размещенного на другом компьютере.

Введем некоторые термины, которые используются в дальнейшем. Рассматривая распределенное приложение, будем выделять клиент и сервер. *Сервер* содержит удаленные компоненты, *клиент* пользуется данными компонентами. Компонентам соответствуют классы. Эти классы будем называть *удаленными классами* (по отношению к клиенту), а объекты удаленных классов – *удаленными объектами*.

Для вызова методов удаленных объектов и клиент и сервер могут использовать традиционные подходы сетевого программирования. А именно:

1. Клиент должен обеспечить соединение с сервером и передачу серверу в закодированном виде имени класса, имени метода и параметров метода.
2. Сервер организует прослушивание сообщений от клиентов и их обработку (желательно в отдельных потоках выполнения).
3. Обработка сообщений клиентов подразумевает декодирование имени метода и параметров, фактический вызов соответствующего метода, кодирование результата и отправку его клиенту.
4. Клиент принимает закодированные результаты, декодирует их и возвращает как результат вызова метода.

Предложенное решение обладает рядом существенных недостатков, главный из которых – отсутствие гибкости. Также отметим возможные проблемы, связанные со временем существования удаленных объектов, обработкой исключительных ситуаций и т. п.

Технологии, подобные .NET Remoting[1] (далее для краткости – Remoting), служат универсальным средством для организации работы с удаленными объектами. Remoting тесно интегрирована с исполняющей средой .NET Framework, а также предоставляет множество средств для тонкой настройки своей инфраструктуры.

[1] В Java имеется сходная технология – *Remote Method Invocation* (RMI).

Рассмотрим основные элементы Remoting, показанные на рисунке 6.

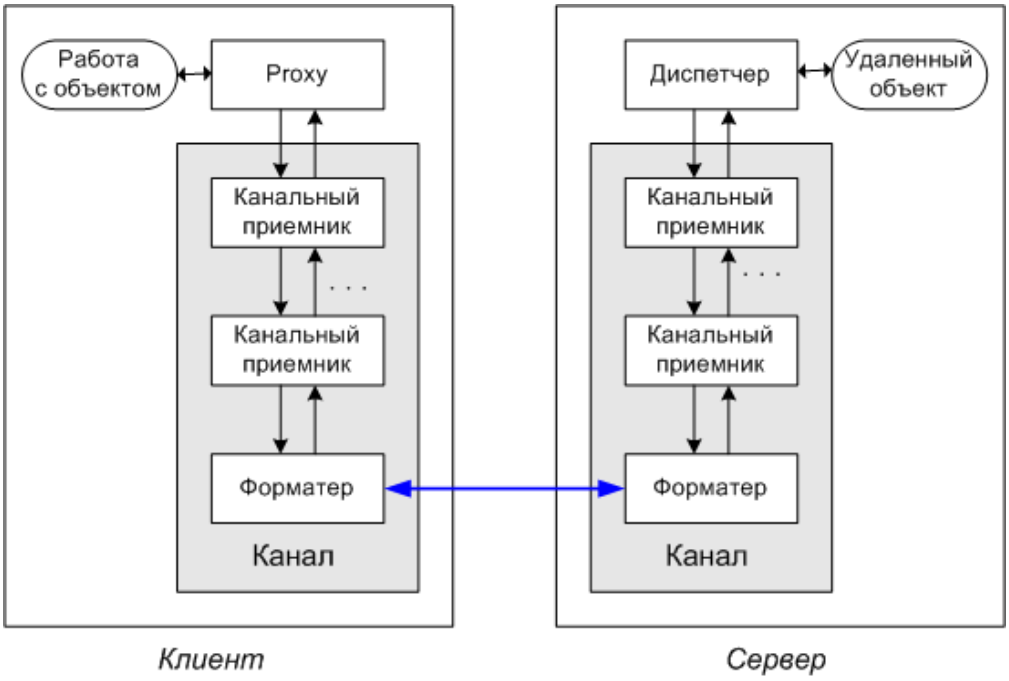

Рис. 6. Основные элементы архитектуры Remoting

Первый элемент архитектуры Remoting – это *объект-заместитель* или *прокси-объект* (*proxy object*). Прокси-объект выполняет следующие задачи. Во-первых, для клиентского кода он выглядит так же, как и любой объект локального класса, что упрощает клиентский код. Во-вторых, все вызовы методов прокси-объект превращает в специальные объекты-сообщения. *Сообщение* служит для описания метода. Оно, в частности, содержит имя метода, коллекцию входных и выходных параметров. Для того чтобы исполняющая среда (CLR) могла правильно создать прокси-объект, удаленный класс должен быть наследником (прямым или косвенным) класса `System.MarshalByRefObject`. В дальнейшем подобные классы будем для краткости называть *MBR-классами*. Кроме этого, клиентскому коду требуется для работы метаданные удаленного класса. Обычно для этого в удаленном классе выделяют интерфейс, который разделяют между сервером и клиентом.

Сообщение, сгенерированное прокси-объектом, попадает в *канал* (*channel*). Канал осуществляет коммуникацию между компьютерами по определенному протоколу. Стандартными каналами являются *HTTP-канал* и *TCP-канал* (входят в поставку Remoting). При необходимости можно реализовать собственный канал передачи.

С каналом могут быть связаны *канальные приемники*, при помощи которых осуществляется перехват сообщений. Обязательным элементом канала является *форматер* (*formatter*). Задача форматера – сериализовать сообщение, то есть представить его в виде потока данных. В составе Remoting имеется бинарный форматер и SOAP-форматер[1]. HTTP-канал использует по умолчанию SOAP-форматер, TCP-канал – бинарный форматер. При необходимости можно реали-

[1] Понятие форматера встречалось при рассмотрении сериализации в .NET Framework.

зовать и собственный форматер данных и канальные приемники. Так как форматер выполняет сериализацию объекта-сообщения, то типы, представляющие входные и выходные параметры метода, должны быть сериализуемыми.

На стороне сервера форматер выполняет десериализацию потока данных из канала, а специальный *диспетчер* находит и вызывает требуемый метод с указанными фактическими параметрами. Затем диспетчер формирует сообщение с результатами работы метода и передает его в серверный канал. Далее это сообщение попадает в форматер на клиенте, десериализуется, а прокси-объект преобразует сообщение в выходное значение метода.

Такова в общих чертах архитектура Remoting. Отметим, что данная технология является расширяемой. В частности, как уже было сказано, пользователь при желании может реализовать собственные форматеры, каналы и канальные приемники, а также предоставлять собственные прокси-объекты вместо стандартных.

3.3. АКТИВАЦИЯ УДАЛЕННЫХ ОБЪЕКТОВ И ИХ ВРЕМЯ ЖИЗНИ

Перед доступом к удаленному объекту он должен быть создан и инициализирован. Данный процесс называется *активацией*. В Remoting удаленные объекты поддерживают два вида активации: серверную активацию и клиентскую активацию[1].

При *серверной активации* инфраструктура Remoting регистрирует тип на сервере и назначает ему *универсальный идентификатор ресурсов* (*Uniform Resource Identifier*, *URI*). Так как каждому типу назначен некий известный URI, то такие типы получили название *общеизвестных типов* (*well-known types*). Объекты общеизвестных типов далее будут обозначаться как *SAO – server activated objects*.

В Remoting поддерживаются два режима серверной активации: режим Singleton и режим SingleCall. При использовании *режима Singleton* существует один объект для обслуживания всех вызовов клиентов. Этот объект создается инфраструктурой на стороне сервера при первой необходимости (при вызове клиентом метода объекта). Будучи активированным, Singleton-объект обслуживает вызовы своих методов от различных клиентов в течение некоторого периода времени (в течение своего *времени жизни*). Затем этот объект автоматически уничтожается. Singleton-объект может сохранять свое состояние между отдельными вызовами методов.

Следующий пример кода показывает конфигурирование типа на сервере в режиме Singleton:

```
RemotingConfiguration.RegisterWellKnownServiceType(
                                    typeof(SomeMBRType),
                                    "SomeURI",
                                    WellKnownObjectMode.Singleton);
```

[1] Вид активации определяется настройкой инфраструктуры Remoting, а не типом. Следовательно, объекты одного типа могут иметь различную активацию в разных приложениях.

В коде используется класс `System.Runtime.Remoting.RemotingConfiguration` для регистрации типа с именем `SomeMBRType`. Клиент также должен сконфигурировать тип `SomeMBRType` как общеизвестный в режиме Singleton:

```
RemotingConfiguration.RegisterWellKnownClientType(
                        typeof(SomeMBRType),
                        "http://SomeWellKnownURL:Port/SomeURI");
```

Режим серверной активации *SingleCall* предназначен для реализации концепции объектов без сохранения состояния. Если некий тип сконфигурирован в режиме SingleCall, инфраструктура Remoting создает объект типа для каждого вызова метода типа. После вызова метода объект уничтожается. Следующий пример демонстрирует конфигурирование типа в режиме SingleCall:

```
RemotingConfiguration.RegisterWellKnownServiceType(
                                typeof(SomeMBRType),
                                "SomeURI",
                                WellKnownObjectMode.SingleCall);
```

С точностью до последнего параметра этот фрагмент кода идентичен коду для случая режима Singleton. Клиентский код регистрации полностью совпадает для двух режимов.

В некоторых программных сценариях требуется, чтобы каждый клиент работал со своей копией удаленного объекта. В этом случае следует использовать *клиентскую активацию*. Объекты типов с клиентской активацией (далее – *CAO*, *client activated objects*) сохраняют свое состояние между вызовами методов. Такие объекты имеют определенное время жизни, после которого автоматически уничтожаются.

Приведем пример кода, конфигурирующего тип на сервере как тип для CAO:

```
RemotingConfiguration.RegisterActivatedServiceType(
                                                typeof(SomeMBRType));
```

Соответствующий конфигурационный код на клиенте выглядит следующим образом:

```
RemotingConfiguration.RegisterActivatedClientType(
                        typeof(SomeMBRType), "http://SomeURL");
```

Более детально типы с клиентской активацией будут рассмотрены ниже.

Рассмотрим некоторые вопросы, связанные с отслеживанием временим жизни удаленных объектов. Эта проблема актуальна для CAO и SAO Singleton. В Remoting для управления временем жизни таких объектов используется механизм на основе лицензий и спонсоров.

Лицензия (*lease*) – это объект, инкапсулирующий несколько значений типа `TimeSpan`[1]. В Remoting для описания лицензий используется интерфейс `ILease`

[1] Структура `System.TimeSpan` служит для представления временных интервалов. Время хранится с точностью до 100 наносекунд.

(пространство имен System.Runtime.Remoting.Lifetime). Интерфейс ILease определяет несколько свойств, связанных с расчетом времени жизни объекта:

- InitialLeaseTime
- RenewOnCallTime
- SponsorshipTimeout
- CurrentLeaseTime

Свойство только для чтения CurrentLeaseTime содержит время, оставшееся до истечения срока действия лицензии. Свойство InitialLeaseTime указывает на первоначальный срок лицензии. При выдаче лицензии CurrentLeaseTime устанавливается равным InitialLeaseTime. Если InitialLeaseTime равно 0, то срок лицензии никогда не заканчивается. При вызове клиентом метода удаленного объекта инфраструктура Remoting определяет время, оставшееся до истечения лицензии. Если это время меньше, чем RenewOnCallTime, то лицензия продлевается на интервал, равный RenewOnCallTime. Свойство SponsorshipTimeout определяет, как долго инфраструктура Remoting будет ожидать ответа спонсора лицензии. Все перечисленные свойства имеют тип TimeSpan.

Когда создается объект с клиентской активацией или объект с серверной активацией в режиме Singleton, исполняющая среда запрашивает у объекта лицензию, вызвав его метод InitializeLifetimeServices(). Это виртуальный метод класса MarshalByRefObject. Для реализации лицензии с нестандартными параметрами метод можно перекрыть.

В каждом домене приложения имеется специальный *диспетчер лицензий*, который управляет лицензиями экземпляров типов, зарегистрированных в домене[1]. После активации удаленного объекта связанная с ним лицензия передается диспетчеру. Диспетчер содержит таблицу, в которой сопоставлены лицензии и момент окончания их действия. Диспетчер периодически просматривает данную таблицу[2]. Если момент окончания действия лицензии наступил, лицензия уведомляется об этом.

Истекшие лицензии пытаются продлить себя, опросив своих спонсоров. Если у лицензии нет спонсоров или никто из спонсоров не продлил ее, то лицензия уведомляет диспетчер о том, что ее следует удалить из таблицы лицензий, и связанный с лицензией удаленный объект уничтожается.

Спонсор – это объект, который может продлить лицензию. Класс спонсора должен реализовывать System.Runtime.Remoting.Lifetime.ISponsor. Спонсоры могут размещаться как на клиенте, так и на сервере, а значит, класс спонсора должен быть MBR-типом. Созданного спонсора разрешается связать с лицензией, вызвав метод ILease.Register(). У лицензии может быть несколько спонсоров.

---

[1] Диспетчер лицензий находится на сервере.

[2] Диспетчер проверяет таблицу каждые 10 секунд. Однако время опроса можно изменить. В следующем фрагменте кода периодичность опроса устанавливается равной 5 минутам:

```
LifetimeServices.LeaseManagerPollTime = System.TimeSpan.FromMinutes(5);
```

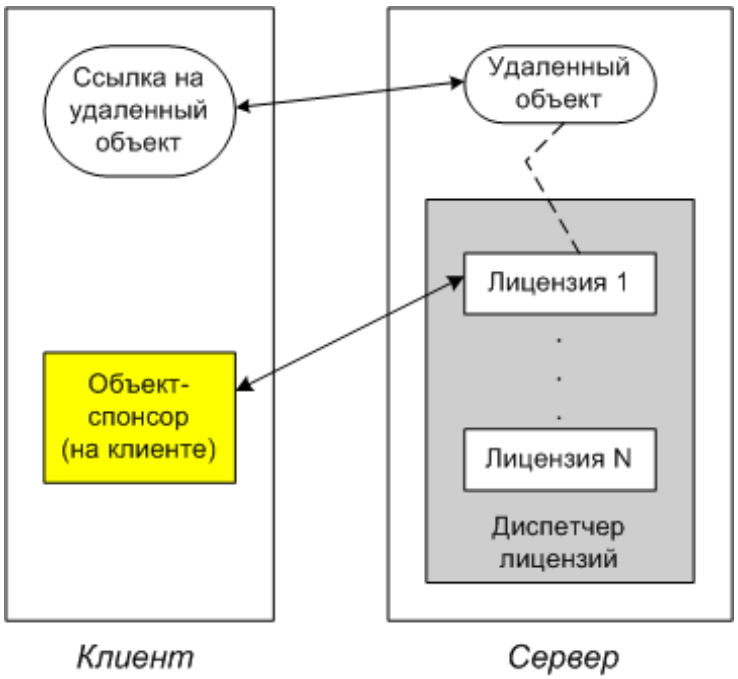

Рис. 7. Спонсоры и лицензии

В пространстве имен `System.Runtime.Remoting.Lifetime` содержится класс `ClientSponsor`. Это MBR-тип и он реализует интерфейс `ISponsor`. Класс `ClientSponsor` позволяет регистрировать ссылки на удаленные объекты, которые предполагается спонсировать. Когда методу `ClientSponsor.Register()` передается ссылка на удаленный объект, этот метод регистрирует экземпляр `ClientSponsor` в качестве спонсора лицензии удаленного объекта и запоминает ссылку на лицензию удаленного объекта во внутренней хэш-таблице. Интервал времени, на который спонсор продлит лицензию, задается свойством `ClientSponsor.RenewalTime`. Ниже показан пример использования `ClientSponsor`:

```
ClientSponsor cp = new ClientSponsor(TimeSpan.FromMinutes(5));
cp.Register(someMBR);
```

3.4. ПРОГРАММНАЯ НАСТРОЙКА REMOTING

Разберем настройку и использование Remoting на следующем примере. Предположим, что планируется реализовать в виде удаленного сервиса калькулятор, который производит четыре арифметических действия. На первом этапе выделим функционал сервиса и оформим его описание в виде интерфейса:

```
namespace BSUIR {
    public interface ICalc {
        double Add(double x, double y);
        double Sub(double x, double y);
        double Mult(double x, double y);
        double Div(double x, double y);
    }
}
```

Скомпилируем интерфейс в динамическую библиотеку `ICalc.dll`.

В качестве непосредственной реализации интерфейса BSUIR.ICalc рассмотрим класс BSUIR.Calculator:

```
using System;
namespace BSUIR {
    public class Calculator: ICalc {
        public Calculator() {
            log("Calculator constructor");
        }
        public double Add(double x, double y) {
            log("Add " +  x + " + " + y);
            return x+y;
        }
        public double Sub(double x, double y) {
            log("Sub " +  x + " - " + y);
            return x-y;
        }
        public double Mult(double x, double y) {
            log("Mult " +  x + " * " + y);
            return x*y;
        }
        public double Div(double x, double y) {
            log("Div " +  x + " / " + y);
            return x/y;
        }
        public static void log(string s) {
            Console.WriteLine("[{0}]: {1}",
                    AppDomain.CurrentDomain.FriendlyName, s);
        }
    }
}
```

Класс Calculator скомпилирован в динамическую библиотеку с именем calc.dll. Для компиляции класса необходимо установить ссылку на сборку ICalc.dll.

Итак, на данном этапе у нас имеются:

1. интерфейс BSUIR.ICalc, размещенный в отдельной динамической библиотеке ICalc.dll;
2. класс BSUIR.Calculator, который реализует интерфейс BSUIR.ICalc и размещается в динамической библиотеке calc.dll.

Что дает подобное разделение на интерфейс и реализующий его класс? Преимущества подхода заключаются в следующем: для клиентов необходим только интерфейс BSUIR.ICalc, без его реализации. Мы можем менять на серверной части нашего распределенного приложения класс BSUIR.Calculator совершенно «прозрачно» для клиентов, пока класс поддерживает интерфейс BSUIR.ICalc. Если бы разделение не выполнялось, то клиенту понадобился бы класс BSUIR.Calculator на локальной машине, хотя и предполагалось бы удаленное использование объектов данного класса.

Построим сервер для нашего распределенного приложения. Прежде всего, нам необходимо внести некоторые изменения в класс BSUIR.Calculator.

```
using System;
namespace BSUIR {
    public class Calculator: MarshalByRefObject, ICalc {
        // далее по коду изменений нет
        . . .
    }
}
```

Напомним, что наследование от `MarshalByRefObject` – это обязательное условие для удаленных типов.

Код настройки сервера будет размещаться в методе `Main()` консольного приложения (собственно, данное приложение и будет сервером[1]). Во-первых, необходимо создать объект, описывающий *канал*. Воспользуемся классом `HttpChannel` (пространство имен `System.Runtime.Remoting.Channels.Http`) для создания стандартного канала на основе протокола HTTP:

```
HttpChannel chan = new HttpChannel(6000);
```

Параметром конструктора является номер порта, с которым связан канал (канал прослушивает данный порт).

Созданный канал должен быть зарегистрирован в инфраструктуре. Для этого используется статический метод `RegisterChannel()` класса `ChannelServices` из пространства имен `System.Runtime.Remoting.Channels`:

```
ChannelServices.RegisterChannel(chan);
```

После создания и регистрации канала требуется зарегистрировать класс, объекты которого предполагается использовать удаленно. Для регистрации класса используются статические методы класса `RemotingConfiguration`. В нашем примере будут использоваться объекты с серверной активацией в режиме SingleCall:

```
RemotingConfiguration.RegisterWellKnownServiceType(
                      typeof(Calculator),
                      "theEndPoint",
                      WellKnownObjectMode.SingleCall);
```

Параметры метода: тип регистрируемого класса; строка, представляющая *концевую точку* для типа; элемент перечисления `WellKnownObjectMode`, указывающий на режим использования серверных объектов (`Singleton` или `SingleCall`).

После регистрации канала и удаленного типа сервер переводится в режим ожидания. Полный текст сервера приведен ниже:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using BSUIR;
```

[1] Использование в качестве сервера консольного приложения – не лучшее решение. В примере таким образом мы поступаем исключительно для простоты.

```
class CalcServer {
    public static void Main() {
        HttpChannel chan = new HttpChannel(6000);
        ChannelServices.RegisterChannel(chan);
        RemotingConfiguration.RegisterWellKnownServiceType(
                              typeof(Calculator),
                              "theEndPoint",
                              WellKnownObjectMode.SingleCall);
        Console.WriteLine("Press [enter] to exit...");
        Console.ReadLine();
    }
}
```

При компиляции сервера необходимо установить ссылку на сборку `calc.dll` и сборку `ICalc.dll`.

Приступим к написанию клиента. Как и сервер, клиент будет консольным приложением, логика работы которого сосредоточена в методе `Main()`.

Клиент должен зарегистрировать канал. При вызове конструктора канала на клиенте можно использовать `0` в качестве параметра или применить конструктор без параметров. В последнем случае клиент не сможет принимать вызовы сервера. Это не всегда приемлемо (например, при работе со спонсорами, асинхронных вызовов методов и т. п.).

```
HttpChannel chan = new HttpChannel(0);
ChannelServices.RegisterChannel(chan);
```

Для соединения с сервером и получения ссылки на удаленный объект можно использовать статический метод `Connect()` класса `RemotingServices`. В качестве параметров методу передается тип запрашиваемого объекта и универсальный идентификатор ресурсов, указывающий на концевую точку, связанную с серверным объектом. Типом запрашиваемого объекта в нашем случае будет интерфейс `ICalc`, так как работа будет вестись через этот интерфейс:

```
object obj = RemotingServices.Connect(typeof(ICalc),
                      "http://localhost:6000/theEndPoint");
```

Получить ссылку на удаленный объект можно при помощи метода `Activator.GetObject()`. Использование данного метода понятно из примера:

```
object obj = Activator.GetObject(typeof(ICalc),
                       "http://localhost:6000/theEndPoint");
```

После запроса полученную ссылку требуется привести к типу `ICalc`, а затем можно вызывать методы удаленного объекта:

```
ICalc calc = (ICalc)obj;
calc.Add(3, 4);
```

Полный листинг клиента приведен ниже:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using BSUIR;
```

```
class CalcClient {
    public static void Main() {
        HttpChannel chan = new HttpChannel(0);
        ChannelServices.RegisterChannel(chan);
        object obj = RemotingServices.Connect(typeof(ICalc),
                     "http://localhost:6000/theEndPoint");
        ICalc calc = (ICalc)obj;
        calc.Add(3, 4);
    }
}
```

Для программной настройки инфраструктуры Remoting применялся класс `RemotingConfiguration`. Подробное описание свойств и методов этого класса приведено в таблице. Все свойств и методы являются статическими.

Таблица 18

Статические элементы класса `RemotingConfiguration`

Имя элемента	Описание
`ApplicationId`	Строка, содержащая GUID для приложения
`ApplicationName`	Строка с именем приложения. Является частью URI удаленного CAO-типа для клиентов
`Configure`	Метод используется в случае, когда настройки Remoting хранятся в конфигурационном файле

`GetRegisteredActivatedClientTypes`
`GetRegisteredActivatedServiceTypes`
`GetRegisteredWellKnownClientTypes`
`GetRegisteredWellKnownServiceTypes`
	Набор методов, позволяющих получить все зарегистрированные типы в виде массива

`IsActivationAllowed`	Метод проверяет на сервере, является ли тип, указанный в качестве параметра, типом с клиентской активацией
`IsRemotelyActivatedClientType`	Метод проверяет на клиенте, является ли тип, указанный в качестве параметра, типом с клиентской активацией
`IsWellKnownClientType`	Метод проверяет на клиенте, является ли тип, указанный в качестве параметра, типом с серверной активацией
`ProcessId`	Строка, содержащая уникальный GUID текущего процесса

`RegisterActivatedClientType`
`RegisterActivatedServiceType`
`RegisterWellKnownClientType`
`RegisterWellKnownServiceType`
	Набор методов для регистрации типов различных видов на клиенте и на сервере

3.5. УДАЛЕННЫЕ ОБЪЕКТЫ С КЛИЕНТСКОЙ АКТИВАЦИЕЙ

Рассмотрим пример распределенного приложения, в котором используются типы с клиентской активацией. Также опишем более подробно работу с лицензиями и спонсорами.

Пусть требуется написать сервер, который предоставляет клиентам услугу по вычислению суммы двух целых чисел. Для получения данной услуги клиент должен пройти процедуру регистрации на сервере. Кроме этого, сервер хранит

для каждого клиента последний рассчитанный результат. В приложении будет задействован механизм лицензий (нестандартное время для параметров лицензии) и спонсоров.

Начнем решение задачи с разработки класса для представления удаленного объекта. Первая «заготовка» класса может выглядеть следующим образом (сборка `RemoteCAO.dll`):

```
using System;
namespace RemoteCAO {
    public class UserCalculator: MarshalByRefObject {
        private bool Registered = false;
        public double CachedValue;
        public bool Register(string Name, string Pass) {
            Registered = (Name == "user") && (Pass == "pass");
            if (Registered)
               Console.WriteLine("User {0} registered", Name);
            else
               Console.WriteLine("Access denied");
            return Registered;
        }
        public double Add(double x, double y) {
            if (Registered) {
                CachedValue = x + y;
                return CachedValue;
            }
            else throw new Exception("Not registered!");
        }
    }
}
```

Обратите внимание на следующие моменты. Класс `UserCalculator` способен генерировать исключительные ситуации[1]. Клиент сможет корректно об-

[1] Класс `System.Exception` реализует интерфейс `ISerializable`. Если создается класс для пользовательской исключительной ситуации, то этот класс должен быть помечен атрибутом `[Serializable]`, а также включать конструктор следующего вида:

```
protected YourApplicationException(SerializationInfo info,
                                   StreamingContext ctx):base(info,ctx) {...}
```

Если класс для исключительной ситуации добавляет к базовому классу `ApplicationException` какие-либо поля, требуется программно записать эти поля в поток сериализации. Для этого переписывается метод `GetObjectData()` и поля добавляются в объект `SerializationInfo` как показано в следующем примере (обратите внимание на пустые перегруженные конструкторы):

```
[Serializable]
public class ExampleEx: ApplicationException {
  public ExampleEx(): base() { }
  public ExampleEx(string message): base(message) { }
  public ExampleEx(string message,Exception inner):base(message,inner){}
  protected ExampleEx(SerializationInfo info, StreamingContext context):
                                                 base(info, context) {
    m_strMachineName = info.GetString("m_strMachineName");
  }
```

работать исключительную ситуацию, которая возникла в удаленном объекте. В процессе работы с удаленным объектом мы будем совершенно свободно манипулировать его полем (не используя методы). В реальном приложении процесс авторизации обычно подразумевает работу с базой данных, в нашем случае применен упрощенный подход.

Класс `UserCalculator`, как и любой удаленный тип, является наследником класса `MarshalByRefObject`. В таблице 19 приведено описание методов класса `MarshalByRefObject`.

Таблица 19

Методы класса `MarshalByRefObject`

Имя метода	Описание
`CreateObjRef()`	Виртуальный метод, возвращающий объект класса `System.Runtime.Remoting.ObjRef`. Такой объект необходим клиентскому приложению, чтобы настроить прокси-объект. Метод можно переписать, если требуется собственная реализация `ObjRef`.
`GetLifetimeService()`	Функция возвращает объект, реализующий интерфейс `ILease`, то есть лицензию, связанную с объектом.
`InitializeLifetimeService()`	Данный виртуальный метод вызывается инфраструктурой Remoting при создании объекта и должен возвращать лицензию объекта. При необходимости метод можно переписать.

В нашем примере мы собираемся использовать нестандартные времена для лицензии объектов класса `UserCalculator`. Следовательно, нам требуется переписать виртуальный метод `InitializeLifetimeService()`:

```
namespace RemoteCAO {
  public class UserCalculator: MarshalByRefObject {
    . . .
    public override object InitializeLifetimeService() {
      ILease lease = (ILease)base.InitializeLifetimeService();
      if (lease.CurrentState == LeaseState.Initial) {
        lease.InitialLeaseTime = TimeSpan.FromMinutes(4);
        lease.SponsorshipTimeout = TimeSpan.FromMinutes(1);
        lease.RenewOnCallTime    = TimeSpan.FromMinutes(3);
```

```
  public override void GetObjectData(SerializationInfo info,
                                     StreamingContext context) {
    info.AddValue("m_strMachineName", m_strMachineName, typeof(String));
    base.GetObjectData(info,context);
  }
  private string m_strMachineName = Environment.MachineName;
  public string MachineName {
    get { return m_strMachineName; }
    set { m_strMachineName = value; }
  }
}
```

Записанные поля доступны в конструкторе исключительной ситуации (параметр `info`). Методы класса `SerializationInfo`, такие как `GetValue()` или, например, `GetString()` позволяют прочесть их.

```
            }
            return lease;
        }
    }
}
```

В начале работы метода `InitializeLifetimeService()` вызывается метод базового класса, который возвращает стандартную лицензию. Параметры можно устанавливать у лицензии, которая не была помещена в диспетчер лицензий. В методе выполняется такая проверка (проверяется *текущее состояние* лицензии). В случае успеха устанавливаются требуемые значения свойств лицензии. Если метод `InitializeLifetimeService()` переписан так, что возвращает значение `null`, то лицензия объекта никогда не заканчивается.

Перейдем к реализации сервера. Как и в примере из предыдущего параграфа, будем использовать в качестве сервера консольное приложение:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using RemoteCAO;
class CalcServer {
    public static void Main() {
        HttpChannel chan = new HttpChannel(6000);
        ChannelServices.RegisterChannel(chan);
        RemotingConfiguration.ApplicationName = "MyServer";
        RemotingConfiguration.RegisterActivatedServiceType(
                                        typeof(UserCalculator));
        Console.WriteLine("Press [enter] to exit...");
        Console.ReadLine();
    }
}
```

В сервере регистрируется канал и удаленный тип с клиентской активацией. Задается имя приложения (`"MyServer"`). Оно будет являться частью URL при доступе к удаленному объекту[1].

Приступим к написанию приложения-клиента:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using RemoteCAO;
class CalcClient {
    public static void Main() {
        HttpChannel chan = new HttpChannel(0);
```

[1] Инфраструктура Remoting создает на сервера для каждого типа с клиентской активацией SAO-объект, публикуемый по адресу `http://<hostname>:<port>/<AppName>/RemoteActivationService.rem`. Этот SAO-объект принимает запросы клиентов, создает CAO-объекты на сервере и передает ссылки на созданные объекты клиентам.

```
            ChannelServices.RegisterChannel(chan);
            RemotingConfiguration.RegisterActivatedClientType(
                                typeof(RemoteCAO.UserCalculator),
                                "http://localhost:60000/MyServer");
            UserCalculator calc = new UserCalculator();
            Console.Write("Input name: ");
            string name = Console.ReadLine();
            Console.Write("Input password: ");
            string pass = Console.ReadLine();
            if (calc.Register(name, pass))
                Console.WriteLine("Successful logon");
            else Console.WriteLine("Failed....");
            try {
                double res = calc.Add(2, 3);
                Console.WriteLine(res);
            }
            catch(Exception e) {
                Console.WriteLine(e.Message);
            }
            Console.WriteLine("Cached value {0}",
                              calc.CachedValue);
            Console.ReadLine();
        }
    }
```

В клиенте требуется обратить внимание на следующие особенности. При создании канала конструктору передан параметр `0`, так как планируется принимать обратные вызовы от сервера. При регистрации типа указано (как часть URL) имя приложения-клиента. После того как на клиенте зарегистрирован удаленный тип с клиентской активацией, для создания удаленных объектов используется оператор `new`. В клиенте мы можем обрабатывать исключительные ситуации, которые возникли на сервере и работать с полем удаленного объекта.

Реализуем спонсора, который сможет продлить время жизни удаленного объекта. Будем использовать клиентский спонсор, то есть объект-спонсор разместим на клиенте. Напомним, что класс для спонсоров должен реализовывать интерфейс `ISponsor`, а также быть MBR-типом:

```
using System.Runtime.Remoting.Lifetime;
. . .
public class FatSponsor: MarshalByRefObject, ISponsor {
    public TimeSpan Renewal(ILease lease) {
        Console.WriteLine("Renewal called....");
        return TimeSpan.FromMinutes(1);
    }
}

class CalcClient {. . .}
```

Наш спонсор прост. Он реализует метод `ISponsor.Renewal()`, который возвращает (независимо от лицензии) время, на которое продлевается лицензия объекта (1 минута).

Объект-спонсор требуется создать и зарегистрировать. Для этого у удаленного объекта запрашивается лицензия (методом `GetLifetimeService()`), а затем при помощи метода `ILease.Register()` регистрируется спонсор:

```
// Код в теле метода Main()
// Создаем спонсора
ISponsor s = new FatSponsor();
// Запрашиваем у удаленного объекта лицензию
ILease currLease = (ILease)calc.GetLifetimeService();
Console.WriteLine(currLease.SponsorshipTimeout);
// Регистрируем спонсора
currLease.Register(s);
```

Метод `ILease.Unregister()` используется для отмены регистрации определенного спонсора.

Приведенный код имеет следующую особенность. Он работает только с Microsoft Framework 1.0. В версии 1.1 и выше данный код работать не будет. В целях безопасности в этих версиях форматерам запрещено передавать вызовы от сервера клиенту без дополнительных настроек. Если проводится настройка Remoting с помощью конфигурационных файлов, то в параметрах форматера (и на клиенте, и на сервере) требуется поместить такой текст (выделен жирным шрифтом):

```
<channels>
  <channel ref="http" port="0" >
    <serverProviders>
      <formatter ref="soap" typeFilterLevel="Full" />
      <formatter ref="binary" typeFilterLevel="Full" />
    </serverProviders>
  </channel>
</channels>
```

3.6. НАСТРОЙКА REMOTING ПРИ ПОМОЩИ КОНФИГУРАЦИОННЫХ ФАЙЛОВ

В предыдущих примерах с использованием Remoting применялось программное конфигурирование инфраструктуры. Такой подход не всегда удобен, особенно если настройки требуется часто изменять. Наряду с программным конфигурированием Remoting допускает настройку с использованием стандартных конфигурационных файлов. Вместо написания такого кода на сервере

```
HttpChannel chnl = new HttpChannel(1234);
ChannelServices.RegisterChannel(chnl);
RemotingConfiguration.RegisterWellKnownServiceType(
                                typeof(CustomerManager),
                                "CustomerManager.soap",
                                WellKnownObjectMode.Singleton);
```

допускается использование следующего конфигурационного файла, обеспечивающего аналогичные настройки:

```
<configuration>
  <system.runtime.remoting>
```

```
    <application>
      <channels>
        <channel ref="http" port="1234" />
      </channels>
      <service>
        <wellknown mode="Singleton"
                   type="Server.CustomerManager, Server"
                   objectUri="CustomerManager.soap" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

Для применения настроек конфигурационного файла в вашем приложении достаточно вызвать метод `RemotingConfiguration.Configure()` и передать ему в качестве параметра имя config-файла:

```
String filename = "server.exe.config";
RemotingConfiguration.Configure(filename);
```

Ниже рассмотрены правила создания конфигурационных файлов. Файл с конфигурацией Remoting имеет следующую структуру:

```
<configuration>
   <system.runtime.remoting>
      <application>
        <lifetime />
        <channels />
        <service />
        <client />
      </application>
   </system.runtime.remoting>
</configuration>
```

`<lifetime />`

Данный тэг используется для изменения параметров лицензии, используемых инфраструктурой по умолчанию. Тэг может иметь следующие атрибуты:

- `leaseTime` – начальное *время жизни* (*time to live*, *TTL*) объекта (по умолчанию – 5 минут);
- `sponsorshipTimeout` – время ожидания ответа от спонсора (по умолчанию – 2 минуты);
- `renewOnCallTime` – время, добавляемое к TTL объекта, когда вызывается его метод (по умолчанию – 2 минуты);
- `leaseManagerPollTime` – интервал проверки TTL диспетчером лицензий (по умолчанию – 10 секунд).

Все атрибуты являются необязательными и могут быть заданы в различных временных единицах. При этом используется обозначение `D` для дней, `H` для часов, `M` для минут, `S` для секунд и `MS` для миллисекунд. Комбинации вида `1H5M` не поддерживаются.

Пример секции `<lifetime>`:

```
<lifetime
  leaseTime="90MS"
  renewOnCallTime="90MS"
  leaseManagerPollTime="100MS"
/>
```

`<channels />`

Этот тэг служит для группировки тэгов, описывающих отдельные каналы. Сам он не имеет каких-либо атрибутов.

`<channel />`

Тэг позволяет указать номер порта для канала на стороне сервера, сослаться на нестандартный пользовательский канал, а также провести дополнительную настройку канала. Если планируется использовать стандартные TCP или HTTP каналы, этот тэг не требуется указывать на клиенте, так как стандартные каналы регистрируются .NET Framework автоматически. На стороне сервера требуется указать, по крайней мере, номер порта для канала.

Для ссылки на канал можно использовать два способа: указать короткую именную ссылку для предварительно зарегистрированных каналов или использовать точное имя типа (пространство имен, имя класса, имя сборки), реализующего канал.

Тэг `<channel>` может иметь следующие атрибуты:

- `ref` – ссылка на стандартный канал (`"tcp"` или `"http"`) или ссылка на канал, предварительно описанный в конфигурационном файле;
- `displayName` – используется .NET Framework Configuration Tool;
- `type` – главный атрибут, если не задан атрибут `ref`. Содержит точное имя типа канала, с указанием пространства имен и сборки. Для глобальных сборок требуется указывать сильное имя. В качестве примера использования рассмотрите описание HTTP-канала в файле `machine.config`;
- `port` – Номер порта канала на сервере. Если клиент планирует получать от сервера сообщения обратного вызова, то в качестве номера порта на клиенте требуется указать `0`.

В дополнение к описанным атрибутам, HTTP-канал поддерживает следующие дополнительные атрибуты (для некоторых атрибутов указано, где их требуется задавать – на клиенте (К) или на сервере (С)):

- `name` – имя канала (по умолчанию `"http"`). При регистрации нескольких каналов должно быть уникальным, или требуется указать пустую строку (`""`). Значение данного атрибута можно использовать при вызове функции `ChannelServices.GetChannel()`;
- `priority` – индикатор вероятности того, что исполняющая среда выберет данный канал для пересылки данных (по умолчанию параметр равен 1). Чем больше указанное целое число, тем выше вероятность;
- `clientConnectionLimit` – число соединений, которые клиент может одновременно открыть к заданному серверу (по умолчанию – 2) (К);

- `proxyName` – имя (адрес) прокси-сервера (К);
- `proxyPort` – номер порта прокси-сервера (К);
- `suppressChannelData` – атрибут указывает, будут ли данные о канале присутствовать в структуре `ChannelData`, используемой при создании объекта `ObjRef` (по умолчанию – `"false"`) (С);
- `useIpAddress` – значение `"true"` (по умолчанию) указывает на использование в URL IP-адреса (а не символьного имени сервера) (С);
- `listen` – значение `"true"` (по умолчанию) позволяет использовать перехватчики, прослушивающие канал (С);
- `bindTo` – IP-адрес обслуживаемого сервером сетевого адаптера (если их несколько) (С);
- `machineName` – имя компьютера, используемого каналом. Указание имени отменяет атрибут `useIpAddress` (С).

TCP-канал (`<channel ref="tcp">`) поддерживает атрибуты HTTP-канала и один дополнительный атрибут:

- `rejectRemoteRequests` – если данное значение установлено в `"true"`, то сервер принимает запросы Remoting только от приложений с локальной машины (С).

Рассмотрим примеры. На стороне сервера следующую конфигурацию можно использовать для указания канала HTTP, прослушивающего порт 1234:

```
<channels>
  <channel ref="http" port="1234">
</channels>
```

На стороне клиента при помощи такой настройки можно увеличить число одновременных запросов от клиента к серверу:

```
<channels>
  <channel ref="http" port="0" clientConnectionLimit="100">
</channels>
```

В рамках каждого канала можно указать и настроить нестандартные канальные приемники и форматеры. Как было сказано выше, Remoting основана на передаче объектов-сообщений. Используя секции конфигурационных файлов `<clientProviders>` и `<serverProviders>`, можно задать цепочку канальных приемников, через которые проходит сообщение, и форматер сериализации сообщения.

Структура секции `<serverProviders>` на стороне сервера выглядит следующим образом:

```
<channels>
  <channel ref="http" port="1234">
    <serverProviders>
      <formatter />
      <provider />
    </serverProviders>
  </channel>
</channels>
```

Свойство `<formatter>` может присутствовать в одном экземпляре, свойств `<provider>` допускается несколько. Требуется учесть, что порядок свойств `<provider>` имеет значение. Подробно об атрибутах, специфичных для канальных приемников, будет рассказано ниже.

Следующие атрибуты являются общими, как для приемника, так и для форматера:

- `ref` – ссылка на стандартный канальный приемник или форматер (`"soap"`, `"binary"`, `"wsdl"`) или ссылка на канальный приемник или форматер, предварительно описанный в конфигурационном файле;
- `type` – главный атрибут, если не задан атрибут `ref`. Содержит точное имя типа канального приемника, с указанием пространства имен и сборки. Для глобальных сборок требуется указывать сильное имя.

В качестве примера использования обсуждаемых атрибутов и секций рассмотрим следующий. Как известно, по умолчанию канал HTTP использует SOAP-форматер для кодирования сообщений. Следующие фрагменты конфигурационного файла настроят канал на использование двоичного форматера.

На стороне сервера требуется записать в конфигурационном файле[1]:

```
<channels>
  <channel ref="http" port="1234">
    <serverProviders>
      <formatter ref="binary" />
    </serverProviders>
  </channel>
</channels>
```

На стороне клиента используется такой фрагмент:

```
<channels>
  <channel ref="http>
    <clientProviders>
      <formatter ref="binary" />
    </clientProviders>
  </channel>
</channels>
```

`<service />`

Тэг позволяет зарегистрировать и настроить удаленные классы с серверной и клиентской активацией, которые публикует данная сборка как сервер. Секция `<service>` содержит требуемое количество подсекций `<wellknown>` и `<activated>`.

`<wellknown>`

В этой подсекции на стороне сервера описываются публикуемые удаленные типы с серверной активацией. Тэг поддерживает атрибуты, которые аналогичны параметрам вызова метода `RegisterWellKnownServiceType()`:

[1] Настройка на стороне сервера не является обязательной, так как HTTP-канал на сервере автоматически использует два форматера и выбирает подходящий в зависимости от настроек клиента.

- `type` – информация о типе публикуемого класса в форме `"<namespace>.<classname>, <assembly>"`. Если используется сборка из GAC, требуется указать версию, культуру и public key;
- `mode` – индикатор, указывающий вид активации: `"Singleton"` или `"SingleCall"`;
- `objectUri` – концевая точка, URI для обращений к объектам удаленного класса. Если используется IIS-хостинг для классов, URI должен оканчиваться на `.soap` или `.rem` для корректной обработки. Эти расширения отображаются на инфраструктуру Remoting в метабазе IIS;
- `displayName` – необязательный атрибут, используемый .NET Framework Configuration Tool.

Используя следующий файл, сервер позволяет осуществить доступ к объектам класса `CustomerManager` по URI `http://<host>:1234/CustomerManager.soap`:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="1234" />
      </channels>
      <service>
        <wellknown mode="Singleton"
                   type="Server.CustomerManager, Server"
                   objectUri="CustomerManager.soap" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

`<activated>`

В подсекции `<activated>` описываются публикуемые сервером удаленные типы с клиентской активацией. Так как полное имя концевой точки в случае клиентской активации определяется именем приложения сервера, единственный атрибут, который указывается в секции `<activated>` – это тип публикуемого класса:

- `type` – информация о типе публикуемого класса в форме `"<namespace>.<classname>, <assembly>"`. Если используется сборка из GAC, требуется указать версию, культуру и public key.

Следующий пример позволит клиентам создавать объекты класса `MyClass` по адресу `http://<hostname>:1234/`

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="1234" />
      </channels>
```

```
        <service>
          <activated type="MyClass, MyAssembly"/>
        </service>
      </application>
    </system.runtime.remoting>
</configuration>
```

<client />

На клиентской машине тэг <client> является аналогом тэга <service> на сервере. Структура этих тэгов совпадает:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown />
        <activated />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

При использовании объектов с клиентской активацией тэг <client> должен при помощи атрибута url указать адрес сервера, содержащего типы, указанные в секции <activated>[1]:

- url – URL сервера. Обязательно указывать, если используются объекты с клиентской активацией;
- displayName – необязательный атрибут, используемый .NET Framework Configuration Tool.

Секция **<wellknown>** используется, чтобы зарегистрировать типы с серверной активацией на клиенте и позволяет использовать оператор new для инициализации ссылок на удаленные объекты. Секция <wellknown> на клиентской стороне имеет такие же атрибуты, как и параметры метода Activator.GetObject():

- url – полный URL к классу, зарегистрированному на сервере;
- type – информация о типе публикуемого класса в форме "<namespace>.<classname>, <assembly>". Если используется сборка из GAC, требуется указать версию, культуру и public key;
- displayName – необязательный атрибут, используемый .NET Framework Configuration Tool.

Если клиент указал некий тип как удаленный, исполняющая среда изменяет поведение оператора new. CLR отслеживает вызовы new, и в том случае, когда речь идет об удаленном типе, возвращается ссылка на серверный объект, а не создается локальный экземпляр типа. В случае следующего конфигурацион-

[1] Если клиент использует объекты с клиентской активации более чем с одного сервера, требуется указать в конфигурационном файле несколько секций <client>.

ного файла для получения удаленной ссылки достаточно написать `CustomerManager x = new CustomerManager()`.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown type="Server.CustomerManager, Client"
           url="http://localhost:1234/CustomerManager.soap" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

Секция **<activated>** используется, чтобы указать типы с клиентской активацией на клиенте. Так как URL сервера уже был указан как атрибут секции `<client>`, то единственный атрибут секции `<activated>` позволяет специфицировать тип:

- `type` – информация о типе публикуемого класса в форме `"<namespace>.<classname>, <assembly>"`. Если используется сборка из GAC, требуется указать версию, культуру и public key.

Данные из секции **<activated>** также используются для переопределения поведения оператора `new`. При использовании конфигурационного файла, приведенного ниже, для получения удаленной ссылки достаточно написать `MyRemote x = new MyRemote()`.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client url="http://localhost:1234">
        <activated type="Server.MyRemote, Client" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

Клиентское приложение может иметь локальный доступ к библиотеке с удаленными классами (в том случае, если не используется разделение удаленного класса на интерфейс и реализацию). В этом случае следует иметь в виду, что загрузка конфигурационных параметров инфраструктуры Remoting должна быть выполнена до первого вызова конструктора удаленного типа. В противном случае тип будет рассматриваться как локальный и никаких обращений к серверу при работе с объектами производиться не будет.

3.7. ХОСТИНГ РАСПРЕДЕЛЕННЫХ ПРИЛОЖЕНИЙ

В данном параграфе обсуждаются различные виды размещения (*хостинга*) серверных компонент распределенных приложений. В качестве примера удаленного класса на протяжении параграфа будет использоваться простой класс `BSUIR.Calculator`, который скомпилирован в сборку с именем `calc.dll`.

```
using System;
namespace BSUIR {
    public class Calculator: MarshalByRefObject {
        public Calculator() {
            log("Calculator constructor");
        }
        public double Add(double x, double y) {
            log("Add " +  x + " + " + y);
            return x + y;
        }
        public static void log(string s) {
            Console.WriteLine("[{0}]: {1}",
                    AppDomain.CurrentDomain.FriendlyName, s);
        }
    }
}
```

Обратите внимание, что в компоненте не выделен интерфейс. Для предоставления клиенту метаданных компонента будем просто передавать сборку с компонентом клиенту.

Клиент, используемый в примерах, достаточно стандартен:

```
using System;
using System.Runtime.Remoting;
using BSUIR;

class CalcClient {
    public static void Main() {
        string filename = "client.exe.config";
        RemotingConfiguration.Configure(filename);
        Calculator c = new Calculator();
        Console.WriteLine(c.Add(3, 4));
    }
}
```

Настройка Remoting на стороне клиента выполняется при помощи файлов конфигурации, которые будут описаны ниже.

В Remoting серверные компоненты распределенных приложений могут использовать хостинг на основе консольных приложений или приложений Windows Forms, хостинг Windows-сервиса или хостинг при помощи веб-сервера Internet Information Server (IIS).

Хостинг на основе консольных приложений или приложений Windows Forms прост в реализации. Однако такой вид хостинга имеет ряд недостатков. В частности, необходим ручной запуск приложения-сервера, затруднено решение проблем аутентификации и логирования.

Рассмотрим хостинг компонент с использованием Windows-сервиса. В .NET Framework пользовательский Windows-сервис – это класс, производный от `ServiceProcess.ServiceBase`. Обычно в классе требуется переписать виртуальный метод `onStart()` для выполнения некоторых полезных действий. В следующем листинге приведена «заготовка» пользовательского сервиса:

```
using System;
using System.ServiceProcess;

namespace WindowsService {
    public class DummyService : ServiceBase {
        public static string SVC_NAME = "Some dummy service";
        public DummyService() {
            // в конструкторе установим свойство - имя сервиса
            this.ServiceName = SVC_NAME;
        }
        static void Main() {
            //стартуем сервис
            ServiceBase.Run(new DummyService());
        }
        protected override void OnStart(string[] args) {
            // полезный код сервиса
        }
        protected override void OnStop() {
             // действия сервиса перед остановкой
       }
    }
}
```

Для установки пользовательского сервиса в систему требуется создать специальный класс-инсталлятор. Этот класс будет обрабатываться утилитой установки `installutil.exe`, которая входит в состав .NET Framework. В следующем листинге показан простой класс-инсталятор, настраивающий сервис на автоматический старт при запуске системы:

```
using System.Configuration.Install;
using System.ServiceProcess;
using System.ComponentModel;

using WindowsService;

[RunInstallerAttribute(true)]
public class MyProjectInstaller: Installer {
    private ServiceInstaller serviceInstaller;
    private ServiceProcessInstaller processInstaller;
    public MyProjectInstaller() {
        processInstaller = new ServiceProcessInstaller();
        serviceInstaller = new ServiceInstaller();
        processInstaller.Account = ServiceAccount.LocalSystem;
        serviceInstaller.StartType =
                                    ServiceStartMode.Automatic;
        serviceInstaller.ServiceName = DummyService.SVC_NAME;
        Installers.Add(serviceInstaller);
        Installers.Add(processInstaller);
   }
}
```

Два файла – файл с кодом сервиса и файл с классом-инсталлятором – требуется скомпилировать в одну сборку (например, `CustomWinServ.exe`). Для установки сервиса выполняется следующая команда:

```
installutil CustomWinServ.exe
```

В случае успешной установки сервис ведет себя как любой стандартный сервис, он допускает управление (старт-стоп) при помощи *оснастки администрирования* (MMC).

Разрабатываемый сервис имеет возможность записывать информацию в *Журнал событий* (Event Log) системы. Для этого достаточно объявить в классе-сервисе статическую переменную типа `System.Diagnostics.EventLog` и использовать ее метод `WriteEntry()`.

В листинге представлен класс-сервис для `BSUIR.Calculator`:

```
using System;
using System.Diagnostics;
using System.ServiceProcess;
using System.Runtime.Remoting;

namespace WindowsService {
  public class RemotingService : ServiceBase {
    private static EventLog evt = new EventLog("Application");
    public static string SVC_NAME = "Remoting Sample Service";
    public RemotingService() {
      this.ServiceName = SVC_NAME;
    }
    static void Main() {
      evt.Source = SVC_NAME;
      evt.WriteEntry("Remoting Service intializing");
      ServiceBase.Run(new RemotingService());
    }
    protected override void OnStart(string[] args) {
      evt.WriteEntry("Remoting Service started");
      String filename = "WinServ.exe.config";
      RemotingConfiguration.Configure(filename);
    }
    protected override void OnStop() {
      evt.WriteEntry("Remoting Service stopped");
    }
  }
}
```

Класс-инсталятор остается практически без изменений (используется имя класса-сервиса `RemotingService`). Два файла компилируются в сборку `CustomWinServ.exe`, к которой подключается сборка `calc.dll`.

Конфигурационный файл `CustomWinServ.exe.config` имеет такой вид:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="1234" />
```

```
        </channels>
        <service>
          <wellknown mode="Singleton"
                     type="BSUIR.Calculator, Calc"
                     objectUri="Calculator.soap" />
        </service>
      </application>
    </system.runtime.remoting>
</configuration>
```

Соответствующий конфигурационный файл для клиента:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="0" />
      </channels>
      <client>
        <wellknown type="BSUIR.Calculator, Calc"
               url="http://localhost:1234/Calculator.soap" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

Хостинг с использованием IIS обладает рядом преимуществ по сравнению с другими видами хостинга. В частности, IIS может быть сконфигурирован для обеспечения аутентификации пользователей удаленного компонента, а также для шифрования серверного трафика. Необходимо понимать, что IIS-хостинг накладывает ряд ограничений на конфигурацию удаленного класса. Допустимо использование только HTTP-канала (возможно, с нестандартным форматером). По сравнению с другими видами хостинга, использование IIS является менее производительным решением.

Общая схема IIS-хостинга проста:

1. Удаленные типы размещаются в отдельной сборке (библиотеке классов).
2. При помощи *оснастки IISAdmin* создается виртуальная директория, соответствующая серверной части распределенного приложения.
3. В поддиректорию `bin` помещается библиотека классов.
4. В виртуальной директории размещается конфигурационный файл, который называется `web.config`.

Создадим при помощи оснастки IISAdmin виртуальный каталог. *Виртуальный каталог* является частью URL и соответствует некому физическому каталогу на сервере. Например, в URL вида `http://host_name/directory_name` виртуальный каталог – это `directory_name`. Ему может соответствовать физический каталог `c:\somedirectory`. Стандартному URL `http://host_name/` по умолчанию соответствует каталог `c:\inetpub\wwwroot`. После запуска оснастки IISAdmin (*Пуск* → *Программы* → *Администрирование* → *Internet Information Services*) требуется выбрать пункт *Веб-узел по умолчанию*, в контекстном меню пункта выбрать *Создать* → *Виртуальный каталог*. Запустится

Мастер создания виртуального каталога. Требуется указать имя для каталога (укажем `remote`) и соответствующую каталогу директорию на диске (`c:\rem`). В сформированной виртуальной директории необходимо создать подкаталог `bin` и поместить в него сборку с удаленным типом. Альтернативное решение заключается в размещении сборки в GAC, но помните, что для доступа к сборке будет необходимо использовать сильное имя.

Конфигурационный файл `web.config`, размещенный в каталоге `c:\rem`, выглядит следующим образом:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton"
                   type="BSUIR.Calculator, Calc"
                   objectUri="Calculator.soap" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

В конфигурационном файле отсутствует описание канала (ISS-хостинг подразумевает HHTP и стандартный порт). При необходимости изменить форматер по умолчанию для HTTP-канала описание канала должно присутствовать. Также при указании концевой точки обязательно должен быть записан суффикс `soap`.

Конфигурационный файл клиента выглядит следующим образом:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown type="BSUIR.Calculator, Calc"
              url="http://localhost/remote/Calculator.soap" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

3.8. ОБЪЕКТЫ-СООБЩЕНИЯ

При удаленном взаимодействии исполняющая среда манипулирует не ассемблерным кодом вызова методов, а специальными объектами-сообщениями. Для поддержки сообщений предоставлено несколько стандартных интерфейсов. Базовым является интерфейс `IMessage` (пространство имен `System.Runtime.Remoting.Messaging`). Любое *сообщение* – это объект, реализующий данный интерфейс. Интерфейс `IMessage` устроен просто. Он содержит единственное свойство `Properties` типа `IDictionary` для помещения в сообщение любых данных и идентифицирующих их ключей. Для удобства в .NET

Framework описаны еще несколько интерфейсов – наследников `IMessage`. Схема наследования представлена на рисунке 8.

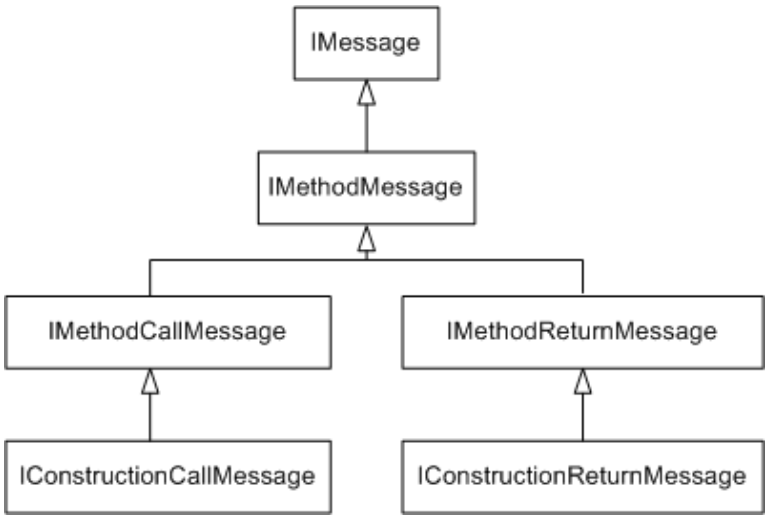

Рис. 8. Схема наследования интерфейсов

В таблице 20 указано назначение интерфейсов и перечислены некоторые их элементы.

Таблица 20

Интерфейсы для сообщений и их элементы

Интерфейс или класс	**Имя элемента**	**Описание**
`IMessage`		Реализуется любым сообщением
`IMethodMessage`		Сообщения, описывающие работу с методами
	`Args`	Массив объектов, соответствующих аргументам метода
	`MethodName`	Имя метода (строка)
	`Uri`	Универсальный идентификатор (URI) объекта, которому направляется сообщение
	`TypeName`	Строка с полным именем типа объекта, к которому направляется сообщение
`IMethodReturnMessage`		Сообщение, описывающее результат метода
	`Exception`	Объект исключительной ситуации, если она сгенерирована удаленным объектом
	`OutArgs`	Массив объектов, представляющих выходные параметры метода
	`ReturnValue`	Объект, содержащий значение работы метода
`IMethodCallMessage`		Сообщение, соответствующее запуску метода
	`InArgs`	Массив объектов, представляющих входные параметры метода
`IConstructionCall-Message`		Сообщение является первым направляемым к удаленному объекту, и определяет свойства для создания объекта
	`ActivationType`	Тип удаленного объекта
	`Activator`	Свойство для работы с активаторами объекта
`IConstructionReturnMessage`		Сообщение, реализующее интерфейс, посылается как ответ на `IConstructionCallMessage`

3.9. ПОЛЬЗОВАТЕЛЬСКИЕ КАНАЛЬНЫЕ ПРИЕМНИКИ

Одна из возможностей расширения .NET Remoting – написание собственного канального приемника. *Канальный приемник* (*channel message sink*) выступает в роли перехватчика сообщений или потока данных на стороне клиента или сервера. Укажем некоторые сценарии, требующие использования канальных приемников:

- **Шифрование данных.** Необходимы канальные приемники для обработки потока данных на стороне клиента и сервера. На стороне клиента производится шифровка отсылаемого потока, на стороне сервера – расшифровка.
- **Авторизация доступа.** На стороне клиента канальный приемник добавляет в сообщение дополнительную информацию, идентифицирующую клиента (имя/пароль). На стороне сервера приемник извлекает данную информацию и в зависимости от ее содержания обрабатывает или отклоняет сообщение.
- **Протоколирование сообщений.** На стороне сервера канальный приемник заносит информацию об обработанных сообщениях в специальный файл или базу данных.

В .NET Framework канальные приемники описываются как классы, реализующие интерфейс `IClientChannelSink` (для клиента) или `IServerChannelSink` (для сервера). Канальные приемники соединены в *цепочку* (*message sink chain*). Каждый приемник хранит информацию о следующем элементе цепочки. Канальный приемник производит предварительную обработку сообщения, передает сообщение следующему приемнику в цепочке, после этого производится пост-обработка исходящего сообщения.

Установкой и настройкой канальных приемников занимаются специальные классы – *провайдеры приемников*. Это классы, реализующие интерфейс `IClientChannelSinkProvider` (для клиента) или `IServerChannelSinkProvider` (для сервера).

Созданные провайдеры подключаются к каналу при помощи конфигурационного файла[1]. Например, пусть имеется следующий конфигурационный файл (фрагмент):

```
<channels>
  <channel ref="http">
    <clientProviders>
      <provider type="MySinks.MessageSinkProvider, Client" />
      <formatter ref="soap" />
      <provider
           type="MySinks.ClientChannelSinkProvider, Client" />
    </clientProviders>
  </channel>
</channels>
```

[1] Возможно программное подключение провайдеров, однако этот способ в лекциях не рассматривается.

В этом случае в структуре HTTP-канала будут находиться такие провайдеры (и соответствующие им приемники):

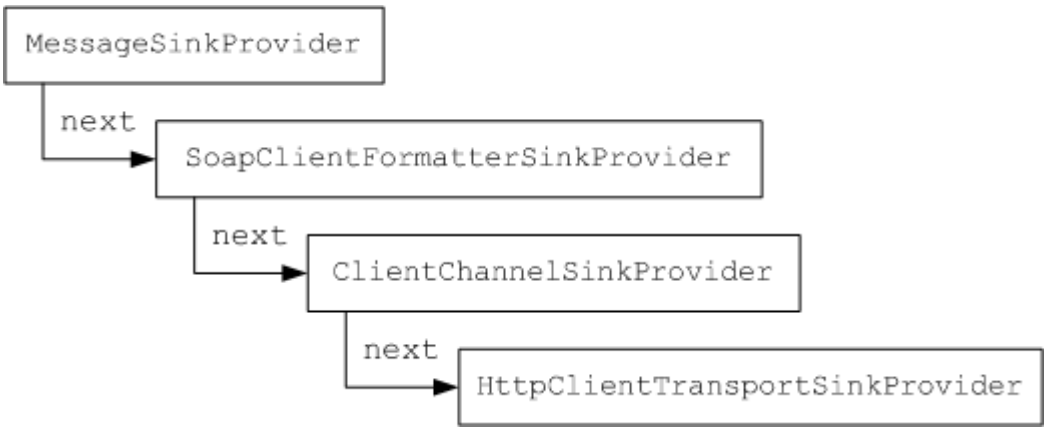

Рис. 9. Провайдеры в структуре HTTP-канала

Обратите внимание: среди провайдеров имеется стандартный, отвечающий за SOAP-форматирование, а последний провайдер – это элемент HTTP-канала. Подчеркнем, что порядок объявления провайдеров имеет значение.

Работу с канальными приемниками рассмотрим на нескольких примерах. В первом примере создадим серверный приемник для протоколирования сообщений. Начнем с рассмотрения интерфейса `System.Runtime.Remoting.Channels.IServerChannelSink`.

```
public interface IServerChannelSink {
    IServerChannelSink NextChannelSink { get; }
    ServerProcessing ProcessMessage(
                        IServerChannelSinkStack sinkStack,
                        IMessage requestMsg,
                        ITransportHeaders requestHeaders,
                        Stream requestStream,
                        out IMessage responseMsg,
                        out ITransportHeaders responseHeaders,
                        out Stream responseStream);
    void AsyncProcessResponse(
                    IServerResponseChannelSinkStack sinkStack,
                    object state,
                    IMessage msg,
                    ITransportHeaders headers,
                    Stream stream);
    Stream GetResponseStream(
                    IServerResponseChannelSinkStack sinkStack,
                    object state,
                    IMessage msg,
                    ITransportHeaders headers);
}
```

Свойство `NextChannelSink` содержит ссылку на следующий канальный приемник в цепочке. Оно устанавливается конструктором класса-приемника.

Метод `ProcessMessage()` – основной метод при синхронной обработке сообщений и потоков. Его параметры:

- `sinkStack` – стек серверных канальных приемников, предшествующих вызываемому;
- `requestMsg` – сообщение-запрос;
- `requestHeaders` – транспортные заголовки сообщения-запроса. Это таблица пар «ключ-значение». При помощи заголовка к сообщению можно присоединять дополнительную описательную информацию;
- `requestStream` – поток, который следует обработать и направить в десериализатор;
- `responseMsg` – после выполнения метода удаленного объекта этот параметр содержит возвращаемый `IMessage`;
- `responseHeaders` – после выполнения метода удаленного объекта этот параметр содержит транспортные заголовки возвращаемого сообщения (если они есть);
- `responseStream` – параметр содержит поток, направляемый клиенту после выполнения метода удаленного объекта.

При реализации тела метода `ProcessMessage()` нужно действовать по такой схеме. Вначале производится предварительная обработка сообщений или потоков[1]. После этого обработчик помещается в стек (`sinkStack`). Стек используется для асинхронной обработки. Затем следует вызвать метод обработки у следующего приемника в цепочке. Далее производится пост-обработка сообщений или потока.

Метод `ProcessMessage()` возвращает значение из перечисления `ServerProcessing`:

- `Async` – вызов обработан асинхронно, и приемник должен сохранить возвращаемые данные в стеке для дальнейшей обработки;
- `Complete` – сервер обработал сообщение в нормальном, синхронном режиме;
- `OneWay` – сообщение было обработано, ответ высылаться не будет.

Метод `AsyncProcessResponse()` возвращает ответ сервера при обработке асинхронного сообщения. Параметры метода:

- `sinkStack` – стек серверных канальных приемников, предшествующих серверному транспортному приемнику;
- `state` – дополнительная информация, помещенная в стек вместе с объектом-обработчиком;
- `msg` – сообщение-ответ;
- `headers` – транспортные заголовки возвращаемого сообщения;
- `stream` – поток, направляемый в транспортный приемник.

Метод `GetResponseStream()` создает объект `Stream`, содержащий объект `IMessage`, а также нужные пары «ключ-значение» из объекта `ITransportHeaders`.

[1] Что обрабатывается – сообщение или поток – зависит от места канального приемника в цепочке: до форматера или после него.

Вернемся к нашему примеру. Представим код серверного канального приемника. Для краткости в листинге опущены параметры методов:

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Messaging;
using System.Collections;
using System.IO;

namespace MySink {
    public class LogSink : BaseChannelSinkWithProperties,
                           IServerChannelSink {
        private IServerChannelSink nextSink;
        public LogSink(IServerChannelSink next) {
            nextSink = next;
        }
        public IServerChannelSink NextChannelSink {
            get { return nextSink; }
        }
        public void AsyncProcessResponse(. . .) {
            IMethodReturnMessage msg_ret =
                                    (IMethodReturnMessage)msg;
            Console.WriteLine("[{0}]: {1} - return {2}",
                              DateTime.Now.ToString(),
                              msg_ret.MethodName,
                              msg_ret.ReturnValue);
           sinkStack.AsyncProcessResponse(msg,headers,stream);
        }
        public Stream GetResponseStream(...) { return null; }
        public ServerProcessing ProcessMessage(. . .) {
            IMethodCallMessage msg =
                              (IMethodCallMessage) requestMsg;
            Console.WriteLine("[{0}]: {1}",
                              DateTime.Now.ToString(),
                              msg.MethodName);
            foreach (DictionaryEntry de in requestHeaders) {
                Console.WriteLine("KEY: {0} VALUE: {1}",
                                  de.Key, de.Value);
            }
            sinkStack.Push(this,null);
            ServerProcessing srvProc =
                  nextSink.ProcessMessage(sinkStack,
                                          requestMsg,
                                          requestHeaders,
                                          requestStream,
                                          out responseMsg,
                                          out responseHeaders,
                                          out responseStream);
            IMethodReturnMessage msg_ret =
                             (IMethodReturnMessage) responseMsg;
            Console.WriteLine("[{0}]: {1} - return {2}",
                              DateTime.Now.ToString(),
```

```
                                        msg_ret.MethodName,
                                        msg_ret.ReturnValue);
                return srvProc;
            }
        }
    }
```

Дадим комментарии по коду класса. Наш класс `LogSink` является наследникам класса `BaseChannelSinkWithProperties`. Дело в том, что интерфейс `IServerChannelSink` (как и `IClientChannelSink`) наследуется от интерфейса `IChannelSinkBase`, имеющего единственное свойство – словарь `Properties`. Абстрактный класс `BaseChannelSinkWithProperties` предоставляет реализацию данного свойства.

Конструктор класса `LogSink` не выполняет никаких особых действий. Он просто устанавливает указатель на следующий канальный приемник. Вызывать данный конструктор будет провайдер нашего приемника, который и передаст требуемое значение параметра.

Основная логика работы класса сосредоточена в методе `ProcessMessage()`. Получив сообщений, метод производит вывод на консоль информации о нем. Кроме этого, выводятся данные транспортных заголовков сообщения. Затем объект-приемник помещается в стек. После обработки сообщение передается в следующий канальный приемник. Если этот канальный приемник завершил свою работу, то это значит, что можно производит обработку возвращаемого клиенту сообщения. Об этом сообщении также выводится некоторая информация на консоль.

Метод `AsyncProcessResponse()` по сути содержит логику пост-обработки сообщения, а затем передает сообщение дальше по цепочке. Так как наш приемник не создает особого потока по сообщению, метод `GetResponseStream()` просто возвращает `null`.

Для установки серверного канального приемника необходимо написать класс, реализующий `IServerChannelSinkProvider`:

```
public interface IServerChannelSinkProvider {
    IServerChannelSinkProvider Next { get; set; }
    IServerChannelSink CreateSink(IChannelReceiver channel);
    void GetChannelData(IChannelDataStore channelData);
}
```

Свойство `Next` хранит ссылку на следующий провайдер в цепочке. В методе `CreateSink()` создается объект, соответствующий пользовательскому канальному приемнику. При этом в начале вызывается метод для создания стека приемников, а затем пользовательский приемник помещается на вершину стека. При помощи метода `GetChannelData()` можно получить различные характеристики того канала, с которым ассоциирован приемник.

Создадим провайдер приемника для нашего примера. Поместим его в ту же сборку, что и класс-приемник. Код провайдера приведен ниже:

```
namespace MySink {
    public class LogSinkProvider: IServerChannelSinkProvider {
```

```
        private IServerChannelSinkProvider nextProvider;
        public LogSinkProvider(IDictionary properties,
                               ICollection providerData) { }
        public IServerChannelSinkProvider Next {
            get { return nextProvider; }
            set { nextProvider = value; }
        }
        public IServerChannelSink CreateSink(
                                    IChannelReceiver channel) {
           IServerChannelSink next =
                              nextProvider.CreateSink(channel);
           return new LogSink(next);
        }
        public void GetChannelData(
                             IChannelDataStore channelData) { }
    }
}
```

В классе `LogSinkProvider` задан пустой конструктор. Наличие у провайдера конструктора с подобной сигнатурой является обязательным требованием. При помощи таких конструкторов можно извлечь данные для настройки провайдера, хранящиеся, например, в конфигурационном файле. Реализация остальных методов типична для провайдера.

Использование указанного канального приемника происходит при помощи следующего конфигурационного файла сервера. Для провайдера указывается его тип (включая имя пространства имен) и имя сборки.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="1234">
          <serverProviders>
            <formatter ref="soap" />
            <provider type="MySink.LogSinkProvider, MySink" />
          </serverProviders>
        </channel>
      </channels>
      <service>
        <wellknown mode="SingleCall"
                   type="BSUIR.Calculator, Calc"
                   objectUri="Calculator.soap" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

Сервер и клиент для данного примера можно написать по аналогии с рассматриваемыми в параграфе 4. Вот что выводит на консоль сервер:

```
Press [enter] to stop server...
[02.10.2005 16:33:04]: Add
KEY: __ConnectionId VALUE: 1
```

```
KEY: __IPAddress VALUE: 127.0.0.1
KEY: __RequestUri VALUE: /Calculator.soap
KEY: Content-Type VALUE: text/xml; charset="utf-8"
KEY: __RequestVerb VALUE: POST
KEY: __HttpVersion VALUE: HTTP/1.1
KEY: User-Agent VALUE: Mozilla/4.0+(compatible; MSIE 6.0;
                Windows 5.1.2600.0; MS .NET Remoting;
                MS .NET CLR 1.1.4322.573 )
KEY: SOAPAction VALUE:
    "http://schemas.microsoft.com/clr/nsassem/BSUIR.Calculator/calc#Add"
KEY: Host VALUE: localhost:1234
KEY: __CustomErrorsEnabled VALUE: False
[02.10.2005 16:33:04]: Add - return 7
```

Рассмотрим другой пример. Пусть требуется производить шифрование потока, передаваемого от сервера к клиенту и наоборот. Для этого требуется реализовать канальные приемники, работающие с потоком, на стороне клиента и на стороне сервера. В нашем учебном примере «шифрование» будет выполняться как применение к каждому байту потока побитовой операции XOR с некой константой.

Начнем с реализации клиентского канального приемника. Рассмотрим интерфейс `IClientChannelSink`.

```
public interface IClientChannelSink {
    IClientChannelSink NextChannelSink { get; }
    void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
                             IMessage msg,
                             ITransportHeaders headers,
                             Stream stream);
    void AsyncProcessResponse(
                    IClientResponseChannelSinkStack sinkStack,
                    object state, ITransportHeaders headers,
                    Stream stream);
    Stream GetRequestStream(IMessage msg,
                            ITransportHeaders headers);
    void ProcessMessage(IMessage msg,
                        ITransportHeaders requestHeaders,
                        Stream requestStream,
                        out ITransportHeaders responseHeaders,
                        out Stream responseStream);
}
```

Основное отличие этого интерфейса от интерфейса `IServerChannelSink` заключается в наличие отдельных методов для осуществления асинхронного запроса и получения асинхронного ответа. Кроме этого, метод `ProcessMessage()` не работает со стеком объектов-приемников. В остальном (включая схему использования) интерфейс практически аналогичен `IServerChannelSink`.

Код клиентского канального приемника `EncClientSink` представлен ниже (для краткости опущены формальные параметры методов):

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Messaging;
using System.IO;
using System.Collections;

namespace Encryption {
    public class EncClientSink: BaseChannelSinkWithProperties,
                                IClientChannelSink {
        private IClientChannelSink nextSink;
        public EncClientSink(IClientChannelSink next) {
            nextSink = next;
        }
        public IClientChannelSink NextChannelSink {
            get { return nextSink; }
        }
        public void AsyncProcessRequest(. . .) {
           stream =
              EncriptionHelper.GetEncryptedStreamCopy(stream);
           sinkStack.Push(this,null);
           nextSink.AsyncProcessRequest(sinkStack,msg,headers,stream);
        }
        public void AsyncProcessResponse(. . .) {
            stream =
              EncriptionHelper.GetDecryptedStreamCopy(stream);
            sinkStack.AsyncProcessResponse(headers,stream);
        }
        public Stream GetRequestStream(. . .) {
            return nextSink.GetRequestStream(msg, headers);
        }
        public void ProcessMessage(. . .) {
            requestStream =
               EncriptionHelper.GetEncryptedStreamCopy(requestStream);
            nextSink.ProcessMessage(msg, requestHeaders,
                                    requestStream,
                                    out responseHeaders,
                                    out responseStream);
            responseStream =
              EncriptionHelper.GetDecryptedStreamCopy(responseStream);
        }
    }
}
```

Для осуществления шифрования и дешифровки клиентский канальный приемник (равно как и серверный) используют методы вспомогательного класса `EncriptionHelper`.

```
namespace Encryption {
  public class EncriptionHelper {
    public static Stream GetEncryptedStreamCopy(Stream inStream) {
      Stream outStream = new MemoryStream();
      byte[] buf = new byte[1000];
      int cnt = inStream.Read(buf,0,1000);
```

```
        while (cnt > 0) {
          for (int i = 0; i < cnt; i++) { buf[i] ^= 123; }
          outStream.Write(buf,0,cnt);
          cnt = inStream.Read(buf,0,1000);
        }
        outStream.Position = 0;
        return outStream;
      }
      public static Stream GetDecryptedStreamCopy(Stream inStream) {
        // В нашем случае дешифровка и шифровка симметричны
        return GetEncryptedStreamCopy(inStream);
      }
    }
}
```

Нам необходим клиентский канальный провайдер. Это должен быть класс, реализующий интерфейс `IClientChannelSinkProvider`:

```
public interface IClientChannelSinkProvider {
    IClientChannelSinkProvider Next { get; set; }
    IClientChannelSink CreateSink(IChannelSender channel,
                                  string url,
                                  object remoteChannelData);
}
```

Код класса для клиентского провайдера тривиален:

```
public class EncClientSinkProvider:
                                  IClientChannelSinkProvider {
    private IClientChannelSinkProvider nextProvider;
    public EncClientSinkProvider(IDictionary properties,
                                 ICollection providerData) { }
    public IClientChannelSinkProvider Next {
        get { return nextProvider; }
        set { nextProvider = value; }
    }
    public IClientChannelSink CreateSink(IChannelSender channel,
                                         string url,
                                         object remoteChannelData)
    {
        IClientChannelSink next =
                nextProvider.CreateSink(channel,url,remoteChannelData);
        return new EncClientSink(next);
    }
}
```

Далее представлен листинг серверного приемника канала и провайдера (размещенных в пространстве имен `Encryption`):

```
public class EncServerSink : BaseChannelSinkWithProperties,
                             IServerChannelSink {
    private IServerChannelSink nextSink;
    public EncServerSink(IServerChannelSink next) {
        nextSink = next;
    }
```

```
        public IServerChannelSink NextChannelSink {
            get { return nextSink; }
        }
        public void AsyncProcessResponse(. . .) {
            stream =
                  EncriptionHelper.GetEncryptedStreamCopy(stream);
            sinkStack.AsyncProcessResponse(msg,headers,stream);
        }
        public Stream GetResponseStream(. . .) { return null; }
        public ServerProcessing ProcessMessage(. . .) {
            requestStream =
                   EncriptionHelper.GetDecryptedStreamCopy(requestStream);
            sinkStack.Push(this,null);
            ServerProcessing srvProc =
                      nextSink.ProcessMessage(sinkStack,
                                              requestMsg,
                                              requestHeaders,
                                              requestStream,
                                              out responseMsg,
                                              out responseHeaders,
                                              out responseStream);
             responseStream =
                  EncriptionHelper.GetEncryptedStreamCopy(responseStream);
             return srvProc;
        }
    }

    public class EncServerSinkProvider : IServerChannelSinkProvider {
        private IServerChannelSinkProvider nextProvider;
        public EncServerSinkProvider(IDictionary properties,
                                     ICollection providerData) { }
        public IServerChannelSinkProvider Next {
            get { return nextProvider; }
            set { nextProvider = value; }
        }
        public IServerChannelSink CreateSink(IChannelReceiver channel) {
            IServerChannelSink next =
                                nextProvider.CreateSink(channel);
            return new EncServerSink(next);
        }
        public void GetChannelData(IChannelDataStore channelData) { }
    }
```

Выполним подключение разработанных канальных приемников при помощи конфигурационных файлов. Для сервера файл имеет следующий вид:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="1234">
          <serverProviders>
            <provider type =
```

```
                "Encryption.EncServerSinkProvider, EncSink" />
              <formatter ref="soap" />
              <provider type="MySink.LogSinkProvider, MySink" />
            </serverProviders>
          </channel>
        </channels>
        <service>
          <wellknown mode="SingleCall"
                     type="BSUIR.Calculator, Calc"
                     objectUri="Calculator.soap" />
        </service>
      </application>
    </system.runtime.remoting>
</configuration>
```

Клиентский конфигурационный файл:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="0">
          <clientProviders>
            <formatter ref="soap" />
            <provider type =
                 "Encryption.EncClientSinkProvider, EncSink"/>
          </clientProviders>
        </channel>
      </channels>
      <client>
        <wellknown type="BSUIR.Calculator, Calc"
                url="http://localhost:1234/Calculator.soap" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

Обратите внимание на порядок, в котором провайдеры размещены на клиенте и сервере.

Рассмотрим некоторые возможности расширения предыдущего примера. Представим ситуацию, согласно которой не все клиенты реализуют шифрование. Как на стороне сервера различать клиентов? Решение может быть следующим. Клиент с шифрованием внедряет в заголовок транспортного протокола некоторую служебную информацию, а сервер анализирует ее и либо выполняет дешифровку, либо нет.

Работа с заголовком транспортного протокола выполняется как с таблицей «ключ-значение». По соглашению имена пользовательских ключей в заголовке принято начинаться с `"X"`. Вот как могут выглядеть измененные методы на клиенте:

```
public void AsyncProcessRequest(. . .) {
    headers["X-Encrypted"] = "yes";
```

```
        stream = EncriptionHelper.GetEncryptedStreamCopy(stream);
        sinkStack.Push(this,null);
        nextSink.AsyncProcessRequest(sinkStack,msg,headers,stream);
    }
    public void AsyncProcessResponse(. . .) {
        string xencrypted = (string)headers["X-Encrypted"];
        if(xencrypted != null && xencrypted == "yes") {
            stream =
                  EncriptionHelper.GetDecryptedStreamCopy(stream);
        }
        sinkStack.AsyncProcessResponse(headers,stream);
    }
    public void ProcessMessage(. . .) {
        requestStream =
           EncriptionHelper.GetEncryptedStreamCopy(requestStream);
        requestHeaders["X-Encrypted"] = "yes";
        nextSink.ProcessMessage(. . .);
        string xencrypted = (string)responseHeaders["X-Encrypted"];
        if(xencrypted != null && xencrypted == "yes") {
            responseStream =
                  EncriptionHelper.GetDecryptedStreamCopy(responseStream);
        }
    }
```

В коде серверного приемника метод `ProcessMessage()` должен поместить информацию о том, является ли поток зашифрованным в стек обработчиков, чтобы она могла быть извлечена при асинхронной обработке:

```
    public ServerProcessing ProcessMessage(. . .) {
        bool isEncrypted = false;
        string xencrypted = (string)requestHeaders["X-Encrypted"];
        if(xencrypted != null && xencrypted == "yes") {
            requestStream =
                   EncriptionHelper.GetDecryptedStreamCopy(requestStream);
            isEncrypted = true;
        }
        sinkStack.Push(this, isEncrypted);
        ServerProcessing srvProc = nextSink.ProcessMessage(. . .);
        if(srvProc == ServerProcessing.Complete ) {
            if(isEncrypted) {
                responseStream =
                  EncriptionHelper.GetEncryptedStreamCopy(responseStream);
                responseHeaders["X-Encrypted"] = "yes";
            }
        }
             return srvProc;
    }
```

В методе `AsyncProcessResponse()` на сервере анализируется значение параметра `state`. Если этот параметр указывает на то, что принятое сообщение было зашифровано, ответ шифруется и в транспортный заголовок помещается необходимая информация:

```
public void AsyncProcessResponse(. . .) {
    bool hasBeenEncrypted = (bool) state;
    if(hasBeenEncrypted) {
      stream = EncriptionHelper.GetEncryptedStreamCopy(stream);
      headers["X-Encrypted"] = "yes";
    }
    sinkStack.AsyncProcessResponse(msg,headers,stream);
}
```

4. ADO.NET

4.1. АРХИТЕКТУРА ADO.NET

Практически в любом серьезном приложении возникает необходимость работы с базами данных (БД). Обычно для этого используются некие стандартные интерфейсы и библиотеки, вместе составляющие *технологию работы с БД*. Примерами таких технологий являются ODBC, JDBC, OLE DB, ADO. При создании платформы .NET фирма Microsoft представила новую технологию доступа к данным – ADO.NET. Особенностями ADO.NET являются интеграция с XML и ориентированность на *рассоединенные данные*. Последнее означает, что для обработки данных в приложении не требуется постоянное соединение с БД. Данные перемещаются в специальную структуру для хранения и соединение с базой разрывается. Данные обрабатываются в спецструктуре, далее устанавливается соединение, и обновленные данные «закачиваются» обратно в базу.

Рассмотрим общую схему архитектуры ADO.NET, показанную на рис. 10.

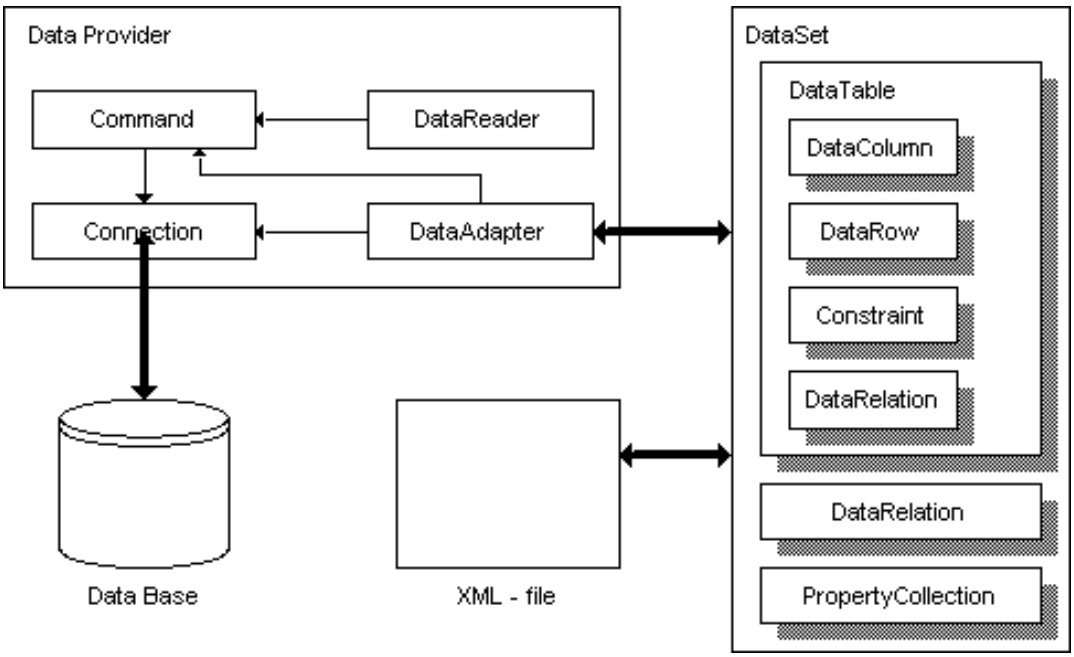

Рис. 10. Общая схема архитектуры ADO.NET

Одним из основных элементов архитектуры является *поставщик* (*провайдер*) *данных* (*data provider*). Поставщик данных – это совокупность классов, предназначенных для непосредственного взаимодействия с базой. Поставщики данных не универсальны, они специфичны для каждой СУБД: Oracle, MS SQL Server, MySQL и т. д. Унификация поставщиков достигается благодаря тому, что классы любого поставщика реализуют некие стандартные интерфейсы. С платформой .NET 1.0 инсталлируются два поставщика: поставщик OLE DB и поставщик для Microsoft SQL Server. Классы данных поставщиков находятся в пространствах имен `System.Data.OleDb` и `System.Data.SqlClient` соответственно. Кроме этого, пространство имен `System.Data.SqlTypes` содержит структуры для описания типов, используемых в СУБД SQL Server.

Любой поставщик данных содержит четыре основных класса[1]: Connection, Command, DataReader и DataAdapter. Назначение класса Connection – установка и поддержка соединения с базой данных. Класс Command служит для выполнения запросов и команд. Можно выполнить как команды, не возвращающие данных (например, создание таблицы в базе), так и запросы, возвращающие скалярное значение или набор данных (SELECT). В последнем случае для чтения данных, полученных командой, используется объект класса DataReader – *ридер*. Отличительной чертой ридера является то, что он представляет собой однонаправленный курсор данных в режиме «только-для-чтения». Класс DataAdapter служит своеобразным «мостом» между поставщиком данных и рассоединенным набором данных. Этот класс содержит четыре команды для выборки, обновления, вставки и удаления данных.

Вторым важным элементом ADO.NET является набор классов, представляющих *рассоединенный набор данных* из базы. Главным компонентом данного набора является класс DataSet, агрегирующий объекты остальных классов. Класс DataTable служит для описания таблиц базы. Класс DataRelation описывает связи между таблицами. Класс PropertyCollection представляет произвольный набор пользовательских свойств. Элементами класса DataTable являются объекты классов DataColumn (колонки таблицы), DataRow (строки таблицы) и Constraint (ограничения на значения элементов таблицы).

4.2. УЧЕБНАЯ БАЗА CD RENT

Опишем простую базу данных, которая будет использоваться в примерах. Назначение базы CD_Rent – хранить информацию о компакт-дисках и о том, кому и когда был предоставлен диск. База состоит из четырех таблиц:

1. **Artists**. Хранит информацию об исполнителях. Информация о колонках таблицы:
   - id – первичный ключ, тип данных: int[2];
   - name – имя исполнителя, тип данных: varchar(50).

2. **Disks**. Таблица содержит информацию о дисках:
   - id – первичный ключ, тип данных: int;
   - title – название альбома, тип данных: varchar(50);
   - artist_id – идентификатор исполнителя, тип данных: int;
   - release_year – год выпуска альбома, тип данных: char(4), допустимы пустые значения.

3. **Users**. Таблица с описанием пользователей (тех, кто берет диски):
   - id – первичный ключ, тип данных: int;
   - user_name – имя пользователя, тип данных: varchar(50);

[1] Имена приведенных классов не точны. Точное имя специфично для поставщика. Так, поставщик данных для SQL Server содержит классы SqlConnection, SqlCommand, SqlDataReader, SqlDataAdapter.

[2] Указаны типы данных, применяемые в Microsoft SQL Server 2000.

- `user_address` – адрес пользователя, тип данных: `varchar(50)`, допустимы пустые значения.

4. `Rent`. Данная рабочая таблица содержит информацию о том, какой диск кем был взят и когда был возвращен:

- `id` – первичный ключ, тип данных: `int`;
- `user_id` – идентификатор пользователя, тип данных: `int`;
- `disk_id` – идентификатор диска, тип данных: `int`;
- `rent_date` – дата, когда взяли диск, тип данных: `datetime(8)`;
- `return_date` – дата, когда вернули диск, тип данных: `datetime(8)`, допустимы пустые значения;
- `disk_rate` – рейтинг популярности диска, тип данных: `float(8)`, допустимы пустые значения;

Схема связей между таблицами в базе представлена на диаграмме 11.

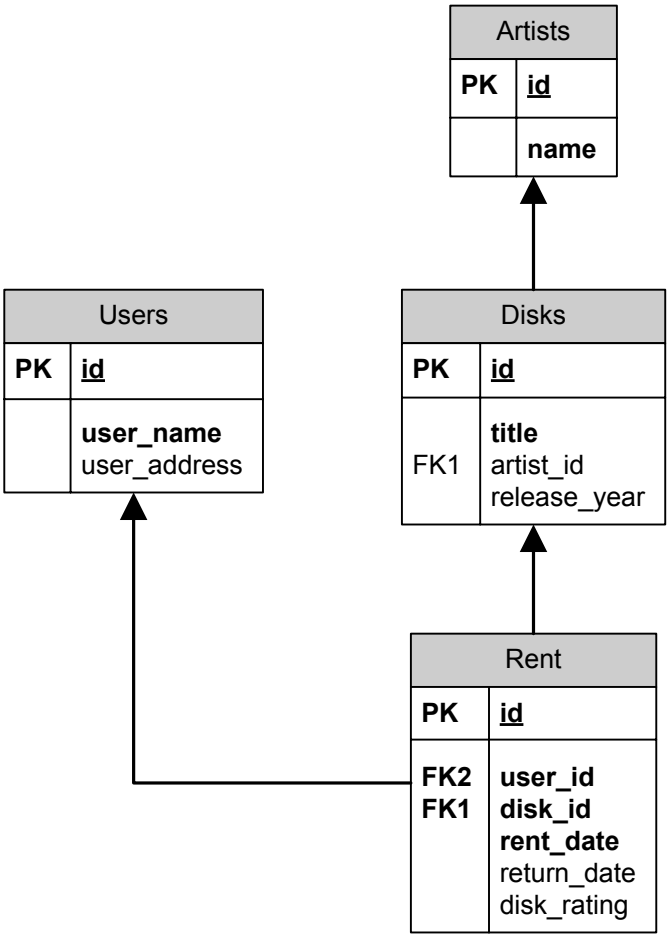

Рис. 11. Связи между таблицами в базе `CD_Rent`

4.3. СОЕДИНЕНИЕ С БАЗОЙ ДАННЫХ

Любое действие с базой данных начинается с подключения к базе. Аналогией может служить разговор по телефону. Прежде чем получить некую информацию, необходимо набрать номер, установить соединение, представиться,

а лишь только затем обращаться с запросом. В ADO.NET поставщики данных предоставляют собственные классы для описания соединений с базой данных:

- Класс `System.Data.SqlClient.SqlConnection` используется для соединения с базами данных SQL Server (версии 7.0 или более поздней);
- Класс `System.Data.OleDb.OleDbConnection` позволяет подключиться к базам данных через интерфейс OLE DB.

Любой класс соединения реализует интерфейс `System.Data.IDbConnection`. Свойства и методы данного интерфейса перечислены в таблице 21. Наиболее важными являются свойство `ConnectionString` и методы `Open()` и `Close()`. Все свойства интерфейса `IDbConnection` – это свойства только для чтения, за исключением `ConnectionString`.

Таблица 21

Элементы интерфейса `IDbConnection`

Свойство или метод `IDbConnection`	**Описание**
`ConnectionString`	Строка, описывающая параметры подключения к базе данных
`ConnectionTimeout`	Время ожидания открытия подключения (в секундах) перед тем, как возникнет исключение «Невозможно подключиться к базе». По умолчанию – 15 секунд, ноль соответствует бесконечному ожиданию. Значение устанавливается строкой подключения
`Database`	Имя базы данных, к которой подключаемся. Значение устанавливается строкой подключения, но может быть изменено вызовом метода `ChangeDatabase()`. Не все поставщики поддерживают это свойство
`State`	Элемент перечисления `ConnectionState`. Поддерживаются `ConnectionState.Open` и `ConnectionState.Closed`
`BeginTransaction()`	Метод начинает транзакцию в базе данных
`ChangeDatabase()`	Устанавливает новую базу данных для использования. Является аналогом команды `USE` в SQL Server. Поставщики для СУБД Oracle не поддерживают этот метод
`CreateCommand()`	Возвращает объект, реализующий интерфейс `IDbCommand` (команду), но специфичный для конкретного поставщика данных
`Open()` и `Close()`	Попытка соединиться и разъединиться с источником данных

Конкретные поставщики данных могут добавлять дополнительные свойства и методы в класс `Connection`. Например, класс `SqlConnection` имеет свойство `ServerVersion` (строка с информацией о версии СУБД), свойство `WorkstationId` (строка, идентифицирующая подключившегося клиента), свойство `PacketSize` (размер пакета обмена с сервером в байтах). Также этот класс (как впрочем, и класс `OleDbConnection`) поддерживает событие `StateChange`, которое генерируется при открытии или закрытии соединения, и событие `InfoMessage`, возникающее, если сервер БД послал строку с предупреждением или ошибкой.

Строка подключения служит для указания параметров подключения к базе данных. В строке через точку с запятой перечислены пары вида «имя параметра=значение параметра». В таблице 22 перечислены некоторые возможные па-

раметры подключения. Не все из них являются обязательными для указания, некоторые специфичны для определенных поставщиков данных. Через наклонную черту указаны возможные альтернативные названия параметров.

Таблица 22

Параметры строки подключения

Имя параметра	Описание параметра
`AttachDBFilename / Initial File Name`	Используется при подключении к базе данных, представленной файлом (например, файл .mdf). Обычно вместо этого параметра используется параметр `Initial Catalog`
`Connect Timeout / Connection Timeout`	Время ожидания подключения в секундах. Если подключение не осуществлено по истечении этого времени, генерируется исключение. Значение по умолчанию – 15 секунд
`Data Source / Server/ Address / Addr / Network Address`	Имя или сетевой адрес сервера. Для локальных серверов используется значение `localhost`
`Initial Catalog / Database`	Имя базы данных
`Integrated Security/ Trusted_Connection`	По умолчанию равно `false`. Если установлено в `true` или `SSPI`, поставщик данных пытается подключиться к серверу, используя имя и пароль пользователя в системе Windows
`Persist Security Info`	Если установлено в `false` (по умолчанию), критическая в плане безопасности информация (например, пароль) удаляется из свойства `ConnectionString` сразу после осуществления подключения
`User ID`	Идентификатор пользователя базы данных
`Password/Pwd`	Пароль пользователя базы данных

Строка подключения к SQL Server может иметь несколько дополнительных параметров, перечисленных в таблице 23.

Таблица 23

Дополнительные параметры строки подключения к MS SQL Server

Имя параметра	Описание параметра
`Current Language`	Язык, используемый SQL Server (при сортировке и т. п.)
`Network Library / Net`	Сетевая библиотека, которая используется для подключения к SQL Server. Поддерживаемые значения: `dbnmpntw` (именованные каналы), `dbmsrpcn` (мультипротокол), `dbmsadsn` (Apple Talk), `dbmsgnet` (VIA), `dbmsipcn` (Shared Memory), `dbmsspxn` (IPX/SPX) и `dbmssocn` (TCP/IP) (используется по умолчанию)
`Packet Size`	Размер в байтах сетевого пакета для обмена с сервером (по умолчанию – 8192 байт)
`Workstation ID`	Имя рабочей станции, подключающейся к серверу (по умолчанию – это имя компьютера клиента).

Параметры, специфичные для строк подключения других поставщиков данных, в данном курсе лекций не рассматриваются.

Как задать строку подключения? Это можно сделать, либо указав строку как параметр конструктора объекта соединения, либо при помощи свойства

ConnectionString. Естественно, строка подключения задается до вызова у соединения метода Open().

Рассмотрим примеры кода, в которых создается соединение с БД. В случае поставщика данных SQL Server подключение может быть выполнено следующим образом:

```
SqlConnection con = new SqlConnection("Data Source=(local);" +
                        "Initial Catalog=Northwind;" +
                        "user id=userid;" +
                        "password=password");
con.Open();
```

Для SQL Server обязательным является задание в строке подключения источника данных (Data Source), имени базы (Initial Catalog) и способа аутентификации. Можно применять два способа аутентификации. Если база данных использует аутентификацию SQL Server (как в примере), то передается идентификатор пользователя и пароль. Сервер БД может использовать встроенную Windows-аутентификацию. Тогда в строке подключения указывается "Integrated Security=SSPI".

Еще два примера показывают подключение к базе при помощи провайдера OLE DB. Данный провайдер можно использовать для работы с SQL Server. В этом случае строка подключения выглядит так:

```
OleDbConnection con = new OleDbConnection(
                        "Data Source=(local);" +
                        "Initial Catalog=Northwind; " +
                        "user id=sa;password=secret;" +
                        "Provider=SQLOLEDB");
```

Как правило, провайдер OLE DB используют для работы с менее мощными СУБД. Следующий пример демонстрирует соединение с базой данных Access через поставщика Jet:

```
OleDbConnection con = new OleDbConnection(
                        "Data Source=localhost;" +
                        "Initial Catalog=c:\Nortwdind.mdb;"+
                        "Provider=Microsoft.Jet.OLEDB.4.0");
```

Приведем два совета, касающихся подключения к базам. Во-первых, строка подключения обычно не прописывается в коде программы, а берется из конфигурационного файла. Во-вторых, не забывайте закрывать подключения к базам, так как некоторые СУБД имеют лимит на количество одновременных подключений клиентов.

Для увеличения производительности приложений поставщики данных могут поддерживать *пул соединений* (*connection pool*). Сущность пула заключается в следующем. При вызове метода Close() соединение с базой не разрывается, а помещается в буфер. Если приложение захочет открыть соединение, аналогичное существующему в буфере, то система возвращает открытое подключение

из пула[1]. Какие подключения считаются «аналогичным», зависит от поставщика данных. Например, провайдер для MS SQL Server требует буквального совпадения строк подключения с точностью до символа.

Настройка пула соединений выполняется при помощи параметров в строке подключения. Для поставщика SQL Server можно использовать параметры, перечисленные в таблице 24.

Таблица 24

Параметры строки подключения, управляющие пулом соединений

Имя параметра	Описание параметра
`Pooling`	Булево выражение, определяет необходимость использования пула. Значение по умолчанию – `true`
`Min Pool Size`	Минимальное число соединений в пуле в любой момент времени (по умолчанию – 0)
`Max Pool Size`	Максимальное число соединений в пуле (по умолчанию – 100). Если достигнут лимит соединений, клиент ждет до тех пор, пока не появиться свободное соединение или пока не истечет таймаут установки соединения
`Connection Lifetime`	Время жизни открытого соединения в пуле (в секундах). Значение по умолчанию – 0, это означает, что `Connection Lifetime = Connect Timeout`

В заключение данного параграфа рассмотрим вопросы, связанные с обработкой ошибок при работе с базами данных. Управляемые поставщики обычно содержат специальные классы, описывающие исключения. В стандартных поставщиках SQL Server и OLE DB имеются классы `SqlException` и `OleDbException`. В каждом из этих классов есть свойство `Errors`, которое является набором объектов типа `SqlError` или `OleDbError`. В данных объектах содержится дополнительная информация об определенном исключении, полученная от базы данных. Использование обсуждаемых классов демонстрирует следующий код:

```
try {
    . . .
}
catch (SqlException ex) {
    string error = "";
    error += ex.Message;
    foreach (SqlError err in ex.Errors) {
        error += "Message: " + err.Message + "\n" +
                 "LineNumber: " + err.LineNumber + "\n" +
                 "Source: " + err.Source + "\n" +
                 "Procedure: " + err.Procedure + "\n";
    }
}
```

[1] Следует учитывать, что пул существует в рамках одного домена. Разные домены приложения имеют разные пулы.

4.4. ВЫПОЛНЕНИЕ КОМАНД И ЗАПРОСОВ К БАЗЕ ДАННЫХ

Работа с базой данных основана на выполнении запросов. Запросы принято классифицировать по следующей схеме:

1. *Запросы, не возвращающие данных*. Такие запросы часто называют *командным* (*action query*). Основных видов командных запросов два:

- *DML-запросы* (Data Manipulation Language, язык управления данными). Они изменяют содержимое базы. Вот несколько примеров DML-запросов:

```
UPDATE Customers SET CompanyName = 'NewCompanyName'
WHERE CustomerID = 'ALF'

INSERT INTO Customers (CustomerID, CompanyName)
VALUES ('NewID', 'NewCustomer')

DELETE FROM Customers WHERE CustomerID = 'ALF'
```

- *DDL-запросы* (Data Definition Language, язык определения данных). Эти запросы изменяют структуру БД:

```
CREATE TABLE Disks ([id] [int] NOT NULL,
                    [title] [varchar] (50) NOT NULL,
                    [artist_id] [int] NULL)

DROP PROCEDURE StoredProc
```

2. *Запросы, возвращающие данные* (*select query*). Эти запросы производят выборку данных из базы либо путем выполнения SQL-оператора `SELECT`, либо при помощи вызова хранимой процедуры, возвращающей некий результат (возможно, набор данных).

Для выполнения запросов любых типов в ADO.NET используются объекты класса `Command` (конкретное имя класса специфично для поставщика данных). Класс команды реализует интерфейс `IDbCommand`.

Для создания объекта `Command` существуют два основных способа. Первый – использование конструктора. Класс `SqlCommand` содержит три перегруженных конструктора: конструктор без параметров; конструктор, которому передается как параметр текст команды; конструктор, принимающий как параметры текст команды и объект-соединение:

```
SqlCommand cmd = new SqlCommand();

SqlCommand cmd1 = new SqlCommand("SELECT * FROM Artists");

SqlConnection con = new SqlConnection("Data Source=(local);" +
                                      "Initial Catalog=CD_Rent;" +
                                      "Integrated Security=SSPI");
SqlCommand cmd = new SqlCommand("SELECT * FROM Artists", con);
```

Второй способ – это создание команды на основе объекта `Connection`:

```
SqlConnection c = new SqlConnection();
SqlCommand cmd3 = c.CreateCommand();
```

В последнем случае команда автоматически связывается с соединением. Связь с соединением должна быть установлена для любой команды перед выполнением. В общем случае для этого используется свойство `Connection`:

```
SqlConnection c = new SqlConnection();
SqlCommand cmd = new SqlCommand();
cmd.Connection = c;
```

Свойство `CommandText` содержит текст команды, а свойство `CommandType` определяет, как следует понимать этот текст. Это свойство может принимать следующие значения:

- `CommandType.Text`. Текст команды – это SQL-инструкции. Обычно такие команды передаются в базу без предварительной обработки (за исключением случаев передачи параметров). Этот тип команды устанавливается по умолчанию.
- `CommandType.StoredProcedure`. Текст команды – это имя хранимой процедуры, которая находится в базе данных.
- `CommandType.TableDirect`. Команда предназначена для извлечения из БД полной таблицы. Имя таблицы указывается в `CommandText`. Данный тип команд поддерживает только поставщик OLE DB.

Еще одну возможность для настройки команд предоставляет свойство `CommandTimeout`. Это время в секундах, в течение которого ожидается начало выполнения команды (по умолчанию – 30 секунд). Следует учитывать, что после начала выполнения команды данное свойство никакой роли не играет. Выполнение команды не прервется, даже если она будет получать данные из базы на протяжении, к примеру, одной минуты.

```
// Пример настройки команды
SqlConnection con = new SqlConnection();
con.ConnectionString = "Server=(local);Database=CD_Rent;" +
                       "Integrated Security=SSPI";
SqlCommand cmd = new SqlCommand();
cmd.Connection = con;
cmd.CommandText = "SELECT * FROM Artists";
// Следующую строку можно не писать
cmd.CommandType = CommandType.Text;
cmd.CommandTimeout = 3;
```

За подготовкой команды следует ее выполнение. Естественно, до выполнения команда должна быть связана с соединением, и это соединение должно быть открыто. В ADO.NET существует несколько способов выполнения команд, которые отличаются лишь информацией, возвращаемой из БД. Ниже перечислены методы выполнения команд, поддерживаемые всеми поставщиками.

- `ExecuteNonQuery()`. Этот метод применяется для запросов, не возвращающих данные. Метод возвращает суммарное число строк, добавленных, измененных или удаленных в результате выполнения команды. Если выполнялся DDL-запрос, метод возвращает значение `-1`.
- `ExecuteScalar()`. Метод выполняет команду и возвращает первый столбец первой строки первого результирующего набора данных. Метод

может быть полезен при выполнении запросов или хранимых процедур, возвращающих единственных результат.

- `ExecuteReader()`. Этот метод выполняет команду и возвращает объект `DataReader`. Тип возвращаемого ридера соответствует поставщику. Метод `ExecuteReader()` используется, когда требуется получить набор данных из базы (например, как результат выполнения команды `SELECT`).

В качестве примера использования метода `ExecuteScalar()` приведем код, при помощи которого подсчитывается число записей в таблице `Artists`:

```
SqlConnection con = new SqlConnection();
con.ConnectionString = "Server=(local);Database=CD_Rent;" +
                       "Integrated Security=SSPI";
SqlCommand cmd =
          new SqlCommand("SELECT COUNT(*) FROM Artists", con);
con.Open();

// Приведение типов необходимо, так как метод ExecuteScalar()
// возвращает значение типа object
int count = (int) cmd.ExecuteScalar();

con.Close();
Console.WriteLine(count);
```

Изменим в предыдущем примере единственную строку:

```
SqlCommand cmd = new SqlCommand("SELECT * FROM Artists", con);
```

Как видим, команда настроена на получение набора данных. Однако в результате выполнения кода примера получим на консоли число 1. Это первый элемент в первой колонке `id`.

Поставщик SQL Server поддерживает метод, возвращающий объект класса, производного от `XmlReader`. Это метод `ExecuteXmlReader()`. Метод поддерживается для SQL Server 2000 и более поздних версий и требует, чтобы возвращаемые запросом или хранимой процедурой данные были в формате XML:

```
SqlConnection con = new SqlConnection();
con.ConnectionString = . . .;
SqlCommand cmd = new SqlCommand();
cmd.Connection = con;
cmd.CommandText = "SELECT * FROM Artists FOR XML AUTO";
// Открыли соединение
con.Open();
// Получили XmlReader
XmlReader xml = cmd.ExecuteXmlReader();
// Здесь как-то читаем и обрабатываем XML-данные
. . .
// Закрываем соединение
con.Close();
```

4.5. ЧТЕНИЕ ДАННЫХ И ОБЪЕКТ DATAREADER

Чтение данных из базы – одна из наиболее часто выполняемых операций. В том случае, когда необходимо прочитать набор данных, используется метод `ExecuteReader()`, возвращающий объект `DataReader` (далее для краткости – просто «ридер»). Каждый поставщик имеет собственный класс для ридера, однако любой такой класс реализует интерфейс `IDataReader`.

Использование ридеров имеет следующие особенности. Во-первых, ридеры не создаются при помощи вызова конструктора. Единственный способ создать ридер – это выполнить метод `ExecuteReader()`. Во-вторых, ридеры позволяют перемещаться по данным набора строго последовательно и в одном направлении – от начала к концу. Большинство СУБД выполняют подобную «навигацию» максимально быстро. В-третьих, данные, полученные при помощи ридера, доступны только для чтения. И, наконец, на время чтения данных соответствующее соединение с базой блокируется, то есть соединение не может быть использовано другими командами, пока чтение данных не завершено.

Рассмотрим работу с ридерами на примере класса `SqlDataReader`. Основным методом ридера является метод `Read()`, который перемещает указатель на следующую запись в наборе данных и возвращает `false`, если записей в наборе больше нет. После прочтения всех записей у ридера необходимо вызвать метод `Close()`. Этот метод освобождает соединение, которое было занято ридером.

Типичный код использования ридера выглядит следующим образом:

```
// Стандартные подготовительные действия
SqlConnection con = new SqlConnection();
con.ConnectionString = . . .
SqlCommand cmd = new SqlCommand("SELECT * FROM Artists", con);

// Открываем соединение и получаем ридер, выполняя команду
con.Open();
SqlDataReader r = cmd.ExecuteReader();

// В цикле читаем данные (пока кода в цикле нет!)
while(r.Read()) { . . . }

// Закрываем ридер; если необходимо, закрываем соединение
r.Close();
con.Close();
```

Как прочитать данные, получаемые ридером? Для этого существует несколько возможностей. Первая заключается в использовании *индексатора ридера*, в качестве индекса выступает строка с именем столбца. Возвращаемое значение имеет тип `object`, поэтому, как правило, необходимо выполнять приведение типов:

```
// Изменяем фрагмент кода из предыдущего примера
while(r.Read()) {
    int id = (int) r["id"];
    Console.WriteLine(id);
}
```

Поиск столбца по имени регистронезависим[1]. Если соответствующего столбца в наборе данных нет, то генерируется исключение `IndexOutOfRangeException`.

Выполнение поиска столбца по имени требует сравнения строк и происходит медленно. Альтернативой является использование в качестве индекса номера столбца. В этом случае производительность повышается:

```
while(r.Read()) {
    int id = (int) r[0];
    Console.WriteLine(id);
}
```

Теоретически, использование числовых индексов вместо имен столбцов означает снижение гибкости приложения. Однако порядок столбцов в наборе данных меняется, только если меняется строка запроса или структура объекта БД (таблицы, хранимой процедуры). В большинстве приложений можно без каких-либо проблем жестко задать порядковые номера всех столбцов. Тем не менее, в некоторых ситуациях известно только имя столбца, но не его порядковый номер. Метод ридера `GetOrdinal()` принимает строку, представляющую имя столбца, и возвращает целое значение, соответствующее порядковому номеру столбца[2]. Это позволяет достичь компромисса между гибкостью и производительностью:

```
// Получаем ридер
SqlDataReader r = cmd.ExecuteReader();
// Один раз находим номер столбца по имени (а не в цикле!)
int id_index = r.GetOrdinal("id");
int name_index = r.GetOrdinal("name");
// В цикле доступ будет быстрее
while(r.Read()) {
    Console.WriteLine(r[id_index] + "\t" + r[name_index]);
}
```

Класс `SqlDataReader` (как и класс `OleDBDataReader`) имеет ряд методов вида `Get<тип данных>` (например, `GetInt32()`). Эти методы получают в качестве параметра индекс столбца, а возвращают значение в столбце, приведенное к соответствующему типу:

```
SqlDataReader r = cmd.ExecuteReader();
int name_index = r.GetOrdinal("name");
while(r.Read()) {
    Console.WriteLine(r.GetString(name_index));
}
```

Отдельные поля записей набора данных могут иметь пустые (null) значения, то есть быть незаполненными. При попытке извлечь значения из null-поля

[1] Более точно, вначале производится поиск соответствия с учетом регистра, если соответствие не найдено – повторный поиск без учета регистра.

[2] С помощью метода ридера `GetName()` можно узнать имя конкретного столбца, указав его номер. Метод `GetDataTypeName()` принимает целое число, соответствующее порядковому номеру столбца, и возвращает в строковом представлении тип данных этого столбца.

(а точнее, при попытке преобразования null-поля в требуемый тип) будет сгенерировано исключение. Ридер имеет метод `IsDBNull()`, предназначенный для индикации пустых полей:

```
if (!r.IsDBNull(id_index))
    Console.WriteLine(r.GetInt32(id_index);
```

Некоторые СУБД и соответствующие поставщики данных позволяют выполнить запрос к базе, возвращающий несколько наборов данных. Ридер такого поставщика реализует метод `NextResult()`, который выполняет переход к следующему набору или возвращает `false`, если такого набора нет. Рассмотрим пример для `SqlDataReader` (обратите внимание на место вызова метода `NextResult()`):

```
// Отсутствует код создания соединения и команды

// Запрос возвращает два набора данных
cmd.CommandText = "SELECT * FROM Disks;" +
                  "SELECT * FROM Artists";
con.Open();
SqlDataReader r = cmd.ExecuteReader();

// Вложенные циклы, внешний - по наборам данных
do {
    while(r.Read())
        Console.WriteLine(r[1]);
} while(r.NextResult());

r.Close();
con.Close();
```

Упомянем некоторые свойства и методы класса `SqlDataReader`. Свойство `FieldCount` возвращает целое число, соответствующее числу столбцов в наборе результатов. Свойство `IsClosed` возвращает логическое значение, указывающее, закрыт ли ридер. Свойство `RecordsAffected` позволяет определить число записей, измененных запросом[1].

Метод ридера `GetValues()` позволяет поместить содержимое записи набора данных в массив. Если нужно максимально быстро получить содержимое каждого поля, использование метода `GetValues()` обеспечит более высокую производительность, чем проверка значений отдельных полей.

```
SqlDataReader r = cmd.ExecuteReader();
object[] data = new object[r.FieldCount];
while(r.Read()) {
    // на самом деле GetValues() - функция,
    // которая возвращает количество полей прочитанной записи
    r.GetValues(data);
```

[1] Это имеет смысл в том случае, если выполняется команда, которая комбинирует несколько SQL-операторов. Например, команда с таким текстом:

```
SELECT * FROM Disks;
DELETE FROM Disks WHERE id = '1'
```

```
        Console.WriteLine(data[1].ToString());
    }
```

Для исследования структуры возвращаемого набора данных можно применить метод `GetSchemaTable()`. Этот метод создает объект `DataTable`, строки которого описывают столбцы полученного набора данных. Колонки таблицы соответствуют атрибутам этих столбцов[1]. Следующий пример кода выводит для каждого столбца его имя и тип:

```
SqlDataReader r = cmd.ExecuteReader();
DataTable tbl = r.GetSchemaTable();
foreach (DataRow row in tbl.Rows)
    Console.WriteLine(row["ColumnName"] + " - " +
                ((SqlDbType)row["ProviderType"]).ToString());
```

Для таблицы `Artists` код выводит на консоль следующее:

```
id - Int
name - VarChar
```

Вернемся к методу `ExecuteReader()`. Этот метод перегружен и может принимать значения из перечисления `CommandBehavior` , которые перечислены в таблице 25 (допустимо использовать побитовую комбинацию значений).

Таблица 25

Значения перечисления `CommandBehavior`

Имя	Значение	Описание
`CloseConnection`	32	При закрытии ридера закрывается и соединение
`KeyInfo`	4	Ридер получает сведения первичного ключа для столбцов, входящих в набор результатов
`SchemaOnly`	2	Ридер содержит только информацию о столбцах, запрос фактически не выполняется
`SequentialAccess`	16	Значения столбцов доступны только в последовательном порядке
`SingleResult`	1	Ридер содержит результаты только первого запроса, возвращающего записи
`SingleRow`	8	Ридер содержит только первую запись, возвращенную запросом

Если при вызове метода `ExecuteReader()` передать ему константу `CloseConnection`, то при вызове метода `Close()` ридера последний вызовет метод `Close()` связанного с ним объекта `Connection`.

При использовании в качестве параметра метода `ExecuteReader()` константы `SequentialAccess` строки считываются последовательно на уровне столбцов. Например, просмотрев содержимое третьего столбца, просмотреть содержимое первого и второго столбцов уже нельзя. Данное поведение оправ-

[1] Для поставщика MS SQL Server таблица с описанием содержит следующие столбцы: `ColumnName`, `ColumnOrdinal`, `ColumnSize`, `NumericPrecision`, `NumericScale`, `IsUnique`, `IsKey`, `BaseServerName`, `BaseCatalogName`, `BaseColumnName`, `BaseSchemaName`, `BaseTableName`, `DataType`, `AllowDBNull`, `ProviderType`, `IsAliased`, `IsExpression`, `IsIdentity`, `IsAutoIncrement`, `IsRowVersion`, `IsHidden`, `IsLong`, `IsReadOnly`. За более подробной информацией обратитесь к справке по .NET Framework.

дано при наличии в строках длинных двоичных данных. Если не применять константу SequentialAccess, то последние будут считаны в ридер вне зависимости от того, будут они использованы или нет. Таким образом, параметр SequentialAccess помогает более эффективно работать с запросами большого количества двоичных данных, когда реально работа ведется только с их частью.

Выше было описано, как с помощью метода ридера GetSchemaTable() можно получить метаданные о столбцах набора данных. Вызвав ExecuteReader() и указав в качестве параметра константу SchemaOnly, мы фактически получим информацию схемы о столбцах, не выполняя запроса. Если указать в параметре константу KeyInfo, ридер выберет из источника данных дополнительную информацию для схемы, чтобы показать, являются ли столбцы набора ключевыми столбцами таблиц источника данных. При использовании константы SchemaOnly дополнительно указывать константу KeyInfo не требуется.

Указав в качестве параметра константу KeyInfo, мы требуем выполнить чтение данных запроса без блокировки записей в таблице базы. Если не блокировать строки при работе с ридером, то любые из них (еще не извлеченные) могут быть изменены. Чтение с блокировкой происходит по умолчанию, чтение без блокировки способно выполняться быстрее, но применять его следует с осторожностью.

Если нужно просмотреть только первую запись или первый набор результатов, возвращаемый запросом, передайте при вызове метода ExecuteReader() константу SingieRow или SingleResult соответственно. При указании константы SingieRow создается ридер, содержащий не более одной записи данных. Все прочие записи отбрасываются. При использовании SingleResult аналогичным образом отбрасываются наборы результатов.

4.6. ПАРАМЕТРИЗИРОВАННЫЕ ЗАПРОСЫ

Запросы к базе данных могут содержать параметры. Рассмотрим запрос со следующим текстом:

```
SELECT name FROM Artists WHERE id = @id
```

В данном примере параметром является @id, а запрос означает следующее: получить из таблицы Artists все значения колонки name таких записей, у которых колонка id равна параметру @id. Вот пример предыдущего запроса для поставщика OLE DB (маркером параметра является символ «?»):

```
SELECT name FROM Artists WHERE id = ?
```

Для работы с параметрами поставщики данных определяют особые классы (например, SqlParameter). Некоторые свойства параметров перечислены ниже (не каждый параметр требует определения всех свойств):

- ParameterName. Имя параметра. У поставщика данных SQL Server имя любого параметра предваряется символом @. Другие поставщики могут не использовать специальных имен вовсе, а определять параметры по позиции.

- `DbType`. Тип хранящихся в параметре данных. Перечисление `DbType` содержит элементы, которые можно использовать как значения данного свойства. Кроме этого, каждый поставщик имеет перечисления либо классы, более точно отражающие реальный тип данных СУБД. Например, поставщик SQL Server содержит перечисление `SqlDbType`.
- `Size`. Свойство зависит от типа данных параметра и обычно используется для указания его максимальной длины. Например, для строковых типов (`VarChar`) свойство `Size` представляет максимальный размер строки. Значение по умолчанию определяется по свойству `DbType`. В случае числовых типов изменять это значение не требуется.
- `Direction`. Данное свойство определяет способ передачи параметра хранимой процедуры. Его возможные значения – `Input`, `Output`, `InputOutput` и `ReturnValue` – представлены перечислением `ParameterDirection`. По умолчанию используется значение `Input`.
- `IsNullable`. Это свойство определяет, может ли параметр принимать пустые значения. По умолчанию свойство установлено в `false`.
- `Value`. Значение параметра. Для параметров типа `Input` и `InputOutput` это свойство должно быть установлено до выполнения команды, для параметров типа `Output`, `InputOutput` и `ReturnValue` значение свойства устанавливается в результате выполнения команды. Чтобы передать пустой входной параметр, нужно либо не устанавливать значение свойства `Value`, либо установить его равным `DBNull`.
- `Precision`. Определяет число знаков после запятой, использующихся для представления значений параметра. По умолчанию имеет значение 0.
- `Scale`. Определяет общее число десятичных разрядов для представления параметра.
- `SourceColumn` и `SourceVersion`. Данные свойства определяют способ использования параметров с объектом `DataAdapter` и подробнее будут рассмотрены ниже.

Любой объект команды содержит свойство `Parameters`, представляющее коллекцию параметров. Для доступа к параметрам в коллекции используется строковый индекс (имя параметра) или целочисленный индекс (позиция параметра).

Чтобы создать параметр можно применить один из конструкторов типа, описывающего параметр. Имеется шесть перегруженных версий конструктора, позволяющих задать некоторые свойства параметра. После создания параметр помещается в коллекцию определенной команды:

```
SqlCommand cmd = new SqlCommand();
cmd.CommandText = "SELECT name FROM Artists WHERE id = @id";
. . .
SqlParameter p = new SqlParameter();
p.ParameterName = "@id";
p.Direction = ParameterDirection.Input;
p.DbType = DbType.Int32;
cmd.Parameters.Add(p);
```

Для создания и добавления параметра часто достаточно воспользоваться одной из перегруженных версий метода `Add()` коллекции параметров:

```
cmd.Parameters.Add("@id", DbType.Int32);
```

Следующий пример показывает выборку данных из таблицы с использованием параметризированного запроса:

```
SqlConnection con = new SqlConnection();
con.ConnectionString = "Server=(local);Database=CD_Rent;" +
                       "Integrated Security=SSPI";
string s = "SELECT name FROM Artists WHERE id = @id";
SqlCommand cmd = new SqlCommand(s, con);
cmd.Parameters.Add("@id", DbType.Int32);
while (true) {
    Console.Write("Input id: ");
    cmd.Parameters["@id"].Value =
                                Int32.Parse(Console.ReadLine());
    con.Open();
    string name = (string)cmd.ExecuteScalar();
    Console.WriteLine(name);
    con.Close();
}
```

Заметим, что параметризированные запросы зачастую удобнее сформировать, воспользовавшись функциями для работы со строками (и такой метод более быстрый, чем работа с объектами-параметрами). Однако без параметров не обойтись, если речь заходит о вызове *хранимой процедуры* на сервере БД.

Пусть на сервере MS SQL Server описана хранимая процедура `get_id`:

```
CREATE PROCEDURE get_id @sp_name varchar(50) AS
    RETURN (SELECT id FROM Artists WHERE name = @sp_name)
GO
```

Данная процедура получает в качестве параметра строку с именем исполнителя и возвращает идентификатор данной строки в таблице `Artists`[1].

Следующий код работает с хранимой процедурой при помощи параметров:

```
// Создаем и настраиваем соединение
SqlConnection con = new SqlConnection();
con.ConnectionString = "Server=(local);Database=CD_Rent;" +
                       "Integrated Security=SSPI";

// Создаем команду, текст команды - имя хранимой процедуры
SqlCommand cmd = new SqlCommand("get_id", con);

// Требуется изменить тип команды
cmd.CommandType = CommandType.StoredProcedure;

// Создаем и настраиваем первый (входной) параметр
```

[1] В MS SQL Server при помощи конструкции `RETURN` можно вернуть из хранимой процедуры только значения целого типа. Для возврата значений других типов следует использовать output-параметры.

```
SqlParameter p1 = new SqlParameter();
p1.ParameterName = "@sp_name";
p1.DbType = DbType.String;
cmd.Parameters.Add(p1);

// Настраиваем параметр, соответствующий выходному значению
SqlParameter p2 = new SqlParameter();
// У такого параметра особое имя!
p2.ParameterName = "@RetVal";
p2.DbType = DbType.Int32;
p2.Direction = ParameterDirection.ReturnValue;
cmd.Parameters.Add(p2);

// Запрашиваем у пользователя информацию...
Console.Write("Input name to find the id: ");
cmd.Parameters["@sp_name"].Value = Console.ReadLine();

// ...и возвращаем ему результат
con.Open();
cmd.ExecuteNonQuery();
Console.WriteLine(cmd.Parameters["@RetVal"].Value);
con.Close();
```

Заметим, что для удобной работы с хранимыми процедурами в клиентском приложении часто описываются специальные методы-«оболочки». Создание подобного метода оставляем читателям в качестве упражнения.

4.7. РАССОЕДИНЕННЫЙ НАБОР ДАННЫХ

ADO.NET предоставляет возможность работы с рассоединенным набором данных. Такой набор данных реализуется объектом класса `DataSet` (далее для краткости – просто `DataSet`). `DataSet` не зависит от поставщика данных, он универсален. Это реляционная структура, которая хранится в памяти. `DataSet` содержит набор таблиц (объектов класса `DataTable`) и связей между таблицами (объекты класса `DataRelation`). В свою очередь, отдельная таблица содержит набор столбцов (объекты класса `DataColumn`), строк (объекты класса `DataRow`) и ограничений (объекты наследников класса `Constraint`). Столбцы и ограничения описывают структуру отдельной таблицы, а строки хранят данные таблицы.

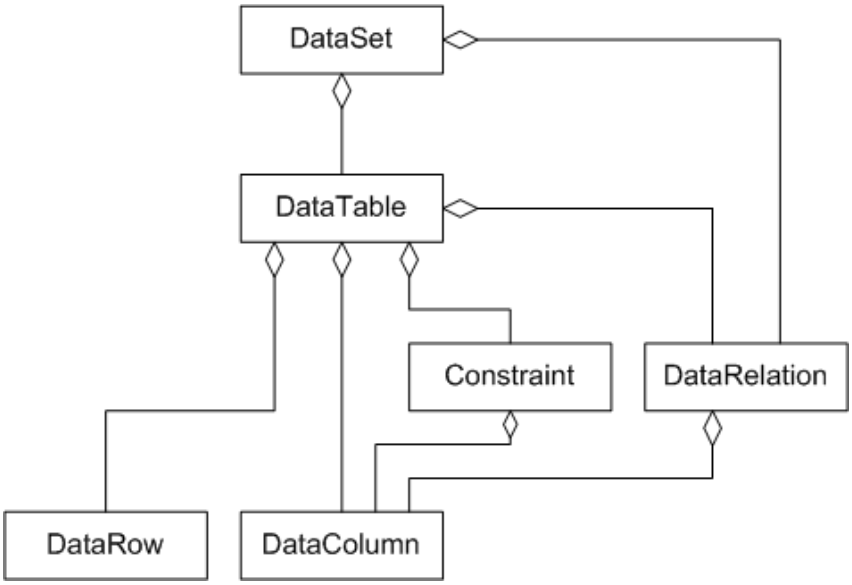

Рис. 12. Связи между классами набора данных

Технически, отдельные компоненты `DataSet` хранятся в специализированных коллекциях. Например, `DataSet` содержит коллекции `Tables` и `Relations`. Таблица имеет коллекции `Columns` (для колонок), `Rows` (для строк), `Constraints` (для ограничений), `ParentRelations` и `ChildRelations` (для связей таблицы). *Любая* подобная коллекция обладает набором сходных свойств и методов. Коллекции имеют перегруженные индексаторы для обращения к элементу по номеру и по имени, методы добавления, поиска и удаления элементов. Методы добавления перегружены и обеспечивают как добавление существующего объекта, так и автоматическое создание соответствующего объекта перед помещением в коллекцию.

Для набора данных `DataSet` введем понятие схемы данных. Под *схемой* будем понимать совокупность следующих элементов:

- Имена таблиц;
- Тип и имя отдельных столбцов таблицы;
- Ограничения на столбцы таблицы такие как уникальность, отсутствие пустых значений, первичные и внешние ключи;
- Связи между таблицами;
- События набора данных и таблицы, которые происходят при работе со строками (аналоги *триггеров* в базах данных).

Схема данных может быть задана определена способами:

- Вручную, путем создания и настройки свойств столбцов, таблиц, связей;
- Автоматически, при загрузке данных в набор из базы;
- Загрузкой схемы, которая была создана и сохранена ранее в XSD-файле.

Правильно созданная схема обеспечивает контроль целостности данных в приложении перед их загрузкой в базу. К сожалению, при загрузке данных из базы в пустой набор генерируется только часть схемы данных (в схеме будут отсутствовать связи между таблицами и события). Рекомендуется подход, при

котором в пустом наборе программно создается полная схема, и только затем в этот набор производится считывание данных.

4.8. ЗАПОЛНЕНИЕ РАССОЕДИНЕННОГО НАБОРА ДАННЫХ

Каждый поставщик данных содержит класс, описывающий *адаптер данных* (`DataAdapter`). В частности, поставщик для SQL Server имеет класс `SqlDataAdapter`. Адаптер данных является своеобразным мостом между базой данных и `DataSet`. Он позволяет записывать данные из базы в набор и производит обратную операцию. В принципе, подобные действия вполне осуществимы при помощи команд и ридеров. Использование адаптера данных – более унифицированный подход.

Основными свойствами адаптера являются `SelectCommand`, `InsertCommand`, `DeleteCommand` и `UpdateCommand`. Это объекты класса `Command` для выборки данных и обновления базы. При помощи метода адаптера `Fill()` происходит запись данных из базы в `DataSet` или таблицу, метод `Update()` выполняет перенос данных в базу.

В начале работы с адаптером его нужно создать и инициализировать свойства-команды[1]. Адаптер содержит несколько перегруженных конструкторов. Варианты вызова конструктора адаптера показаны в примере:

```
// 1. Обычный конструктор без параметров.
// Необходимо заполнить команды вручную
SqlDataAdapter da_1 = new SqlDataAdapter();

// 2. В качестве параметра конструктора – объект-команда
SqlCommand cmd = new SqlCommand("SELECT * FROM Disks");
SqlDataAdapter da_2 = new SqlDataAdapter(cmd);

// 3. Параметры: текст запроса для выборки и объект-соединение
SqlConnection con = new SqlConnection("Server=(local);" +
                  "Database=CD_Rent;Integrated Security=SSPI");
SqlDataAdapter da_3 = new SqlDataAdapter(
                           "SELECT * FROM Disks", con);

// 4. Параметры – строка запроса и строка соединения
string s = "SELECT * FROM Disks";
string c = "Server=(local);Database=CD_Rent;Integrated Security=SSPI";
SqlDataAdapter da_4 = new SqlDataAdapter(s, c);
```

Любой адаптер должен иметь ссылку на соединение с базой данных. Адаптер использует то соединение, которое задано в его объектах-коммандах.

Итак, адаптер создан. Теперь можно использовать его метод `Fill()` для заполнения некоторого набора данных:

```
DataSet ds = new DataSet();
// Строго говоря, метод Fill() - функция, возвращающая
// число строк (записей), добавленных в DataSet
da.Fill(ds);
```

[1] В данном параграфе нас будет интересовать только команда для выборки данных.

Заметим, что вызов метода Fill() не нарушает состояние соединения с БД. Если соединение было открыто до вызова Fill(), то оно останется открытым и после вызова. Если соединение было не установлено, метод Fill() откроет соединение, произведет выборку данных и закроет соединение. Так же ведут себя и все остальные методы адаптера, работающие с базой.

Поведение адаптера при заполнении DataSet зависит от настроек адаптера и от наличия схемы в объекте DataSet. Пусть при помощи адаптера заполняется пустой DataSet. В этом случае адаптер создаст в DataSet *минимальную схему*, используя имена и тип столбцов из базы и стандартные имена для таблиц. В результате выполнения следующего кода в ds будет создана одна таблица с именем Table[1].

```
string con_str = ". . .";
string cmd_text = "SELECT * FROM Disks";
SqlDataAdapter da = new SqlDataAdapter(cmd_text, con_str);
DataSet ds = new DataSet();
da.Fill(ds);
```

Команда выборки данных может быть настроена на получение нескольких таблиц. В следующем примере в пустой DataSet помещаются две таблицы:

```
string con_str = ". . .";
string cmd_text = "SELECT * FROM Disks;" +
                  "SELECT * FROM Artists";
SqlDataAdapter da = new SqlDataAdapter(cmd_text, con_str);
DataSet ds = new DataSet();
da.Fill(ds);
```

В наборе данных ds окажутся две таблицы с именами Table и Table1. Адаптер имеет свойство-коллекцию TableMappings, которое позволяет сопоставить имена таблиц базы и таблиц DataSet:

```
string cmd_text = "SELECT * FROM Disks;SELECT * FROM Artists";
SqlDataAdapter da = new SqlDataAdapter(cmd_text, con_str);
// Первая таблица будет в наборе данных называться Disks
da.TableMappings.Add("Table", "Disks");
// Вторая таблица будет называться Performers
da.TableMappings.Add("Table1", "Performers");
```

Любой элемент коллекции TableMappings содержит свойство ColumnMappings, которое осуществляет отображение имен столбцов:

```
DataTableMapping dtm;
dtm = da.TableMappings.Add("Table1", "Performers");
dtm.ColumnMappings.Add("id", "Performer_id");
dtm.ColumnMappings.Add("name", "Performer_name");
```

[1] Существует перегруженная версия метода Fill() для заполнения одной таблицы (вне набора):

```
DataTable dt = new DataTable();
da.Fill(dt);
```

Предположим, что заполняемый набор данных уже обладает некой схемой. Адаптер содержит свойство `MissingSchemaAction`, значениями которого являются элементы одноименного перечисления. По умолчанию значение свойства – `Add`. Это означает добавление в схему новых столбцов, если они в ней не описаны. Возможными значениями являются также `Ignore` (игнорирование столбцов, не известных схеме) и `Error` (если столбцы не описаны в схеме, генерируется исключение).

Использование свойства `MissingSchemaAction` демонстрирует следующий код:

```
SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Disks",
                                       con);
// Пустой набор данных без схемы
DataSet ds = new DataSet();
// После заполнения в наборе будет схема,
// полученная по таблице базы данных
da.Fill(ds);

// сейчас будем "запихивать" в непустой набор новую таблицу
da = new SqlDataAdapter("SELECT * FROM Artists", con_str);

// 1 вариант. В таблице Table будет 13 записей
// (8 из Disks, 5 из Artists) и колонки: id, title, artist_id,
// release_year (из Disks), name (из Artists)
da.MissingSchemaAction = MissingSchemaAction.Add;
da.Fill(ds);

// 2 вариант. Получим таблицу из 13 записей, но с 4 столбцами
// (без столбца name)
da.MissingSchemaAction = MissingSchemaAction.Ignore;
da.Fill(ds);

// 3 вариант. Попытка заполнения вызовет исключение!
da.MissingSchemaAction = MissingSchemaAction.Error;
da.Fill(ds);
```

Если многократно выполненить метод `Fill()` с идентичной командой и набором данных, то в случае отсутствия в таблице `DataSet` первичного ключа, в таблицу заносится дублирующая информация. Если в таблице задан первичный ключ, произойдет обновление данных таблицы:

```
SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Disks", con);
DataSet ds = new DataSet();
da.Fill(ds);
da.Fill(ds);
```

В результате выполнения этого кода в `DataSet` будет создана **одна таблица**, которая содержит **два одинаковых множества записей**. Важно понимать этот факт, так как таблица в наборе данных может содержать ограничения (первичный ключ, уникальные значения), которые будут нарушены.

Обсудим дополнительные возможности адаптера, связанные с заполнением DataSet. Существует перегруженный вариант метода Fill(), который возвращает диапазон записей:

```
// Первый параметр – целевой набор DataSet,
// второй – номер стартовой записи (нумерация с нуля),
// третий – количество записей,
// четвертый – имя таблицы в целевом наборе DataSet
da.Fill(ds, 3, 10, "Disks");
```

Адаптер имеет метод FillSchema(), который переносит схему таблиц запроса в DataSet. Метод FillSchema() получает из базы имена и типы всех задействованных в запросе столбцов. Кроме этого, данный метод получает сведения о допустимости для столбца значений Null и задает значение свойства AllowDBNull создаваемых им объектов DataColumn. Метод FillSchema() также пытается определить на объекте DataTable первичный ключ.

Метод FillSchema() принимает как параметр объект DataSet, или DataSet и имя таблицы, или объект DataTable. Однако у FillSchema() имеется дополнительный параметр. Он позволяет указать, нужно ли применить к информации схемы параметры набора TableMappings. Можно указать любое значение из перечисления SchemaType – Source или Mapped. При значении Mapped адаптер обратится к содержимому набора TableMappings точно так же, как сопоставляет столбцы при вызове метода Fill(). Вот пример вызова FillSchema():

```
SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Disks", con);
DataSet ds = new DataSet();
da.FillSchema(ds, SchemaType.Source, "Disks");
```

Адаптер данных имеет три события:

- FillError – событие наступает, если при заполнении DataSet или DataTable адаптер столкнулся с какой-либо ошибкой;
- RowUpdating – событие наступает перед передачей измененной строки в базу данных;
- RowUpdated – событие наступает после передачи измененной записи в базу данных.

4.9. ОБЪЕКТ КЛАССА DATACOLUMN – КОЛОНКА ТАБЛИЦЫ

Структура любой таблицы описывается свойствами ее столбцов. Столбец таблицы представлен объектом класса DataColumn. Данный класс содержит следующий набор свойств, перечисленный в таблице 26.

Таблица 26

Свойства класса DataColumn

Имя свойства	Тип	Описание
AllowDBNull	boolean	Определяет, допустимы ли в столбце пустые значения
AutoIncrement	boolean	Генерируется ли для столбца новое значение автоинкремента

AutoIncrementSeed	int	Начальное значение автоинкремента
AutoIncrementStep	int	Шаг автоинкремента
Caption	string	Заголовок столбца, отображаемый в элементах управления
ColumnMapping	MappingType	Определяет, как будет записано содержимое столбца в XML-документ
ColumnName	string	Имя столбца в таблице
DataType	Type	Тип данных столбца
DefaultValue	object	Значение по умолчанию в столбце
Expression	string	Выражение для *вычисляемых столбцов*
ExtendedProperties	PropertyCollection	Набор динамических свойств столбца и их значений
MaxLength	int	Максимально допустимая длина строки данных для столбца
Namespace	string	Имя пространства имен XML, используемого при загрузке и чтении столбца из XML-файла
Ordinal	int	Порядковый номер столбца в таблице
Prefix	string	Префикс пространства имен XML; используется при загрузке и чтении столбца из XML-файла
ReadOnly	boolean	Указывает, что содержимое столбца доступно только для чтения
Table	DataTable	Таблица, в состав которой входит столбец
Unique	Boolean	Должно ли быть значение в столбце уникальным в пределах таблицы

«Ручное» создание объектов-столбцов используется при самостоятельном формировании схемы DataSet. Минимально допустимая настройка столбца заключается в указании его имени и типа данных[1]. После этого столбец может быть добавлен в таблицу (при этом заполнятся свойства Ordinal и Table).

```
DataColumn dc = new DataColumn();
dc.ColumnName = "New Column";
dc.DataType = typeof(int);

DataTable dt = new DataTable();
dt.Columns.Add(dc);
```

Класс DataColumn имеет несколько перегруженных конструкторов, один из которых позволяет указать имя и тип столбца как параметры:

```
DataColumn dc = new DataColumn("New Column", typeof(int));
```

Для строк таблицы можно указать значение по умолчанию в столбце – свойство столбца DefaultValue. Если свойство AllowDBNull установлено в

[1] При занесении информации в таблицу при помощи адаптера автоматически настраиваются следующие свойства: AllowDBNull, Caption, ColumnName, DataType, Ordinal, Table, Unique.

true, то допустимы пустые значения столбца (столбец может содержать объект класса DBNull[1]).

Свойства AutoIncrement, AutoIncrementSeed и AutoIncrementStep используются для организации автоматического приращения значений столбца (по умолчанию автоприращение не активно). Тип свойства с автоприращением должен быть целочисленным. Автоприращение может быть полезно при организации первичного ключа таблицы[2].

При работе с набором данных существует возможность сохранить набор в виде XML-документа. Свойство ColumnMapping настраивает представление столбца при сохранении. Значениями свойства выступают элементы перечисления MappingType. Следующий код обеспечивает сохранение значений столбца id в виде атрибута:

```
id.ColumnMapping = MappingType.Attribute;
```

К свойствам для поддержки работы с XML относятся также свойства Namespace и Prefix.

В любой таблице существует возможность создать *вычисляемый столбец*. Для этого у столбца устанавливается свойство Expression. Это строка, содержащая специальное выражение, используемое для вычисления значений столбца. Подробнее о работе с вычисляемыми столбцами смотрите в документации MSDN.

Столбец имеет свойство ExtendedProperties, которое можно рассматривать как хэш-таблицу, связанную со столбцом. Ключами такой хэш-таблицы должны является строки. Вот пример использования свойства ExtendedProperties:

```
// Помещаем в ExtendedProperties некоторую информацию
id.ExtendedProperties.Add("Comment", "Primary key");
id.ExtendedProperties.Add("Author", "Alex");

// В произвольный момент можем извлечь ее
Console.WriteLine(id.ExtendedProperties["Comment"]);
```

Для хранения столбцов класс DataTable использует свойство Columns типа DataColumnCollection. Добавлять столбцы можно по одному (метод DataColumnCollection.Add()) или целым массивом (метод DataColumnCollection.AddRange()). Кроме этого, метод Add() имеет удобную перегруженную версию, которая позволяет неявно создать столбец, указав его имя и тип:

```
// Создали таблицу
DataTable Artists = new DataTable("Artists");
// Добавили столбцы, которые создали и настроили до этого
Artists.Columns.Add(id);
Artists.Columns.Add(name);
```

---

[1] Класс System.DBNull соответствует понятию NULL в реляционных БД и имеет единственное статическое поле только для чтения Value.

[2] Для столбцов с автоприращением можно принудительно указать произвольное целочисленное значение при занесении информации в строку таблицы.

```
DataTable Users = new DataTable("Users");
// Столбцы можно создавать при добавлении в таблицу...
Users.Columns.Add("id", typeof(int));
Users.Columns.Add("user_name", typeof(string));
Users.Columns.Add("user_address", typeof(string));

// ...а потом их вот так настраивать
Users.Columns["user_name"].MaxLength = 30;
```

4.10. ОБЪЕКТЫ КЛАССА DATAROW – СТРОКИ ТАБЛИЦЫ

Строки содержат данные таблицы. Отдельная строка представлена объектом класса `DataRow`. Таблица содержит коллекцию `Rows` типа `DataRowCollection` для хранения своих строк. Каждая строка обладает свойством `Table`, в котором хранится ссылка на таблицу, владеющую строкой.

Для создания строки применяется метод таблицы `NewRow()`. Метод генерирует пустую строку согласно структуре столбцов таблицы, но не добавляет эту строку в таблицу. Для добавления строки необходимо заполнить ее данными, а затем воспользоваться методом `DataTable.Rows.Add()`. Существует перегруженный вариант этого метода, который принимает в качестве параметра массив объектов, являющихся значениями полей строки:

```
// Создали пустую строку с требуемой структурой
DataRow r = Artists.NewRow();
// Заполняем ее содержимым
r["name"] = "Depeche Mode";
// Добавляем в таблицу
Artists.Rows.Add(r);
// Вариант покороче, в котором совмещены сразу три действия
Artists.Rows.Add(new object[2]{null, "Nirvana"});
```

Рассмотрим вопросы, связанные с редактированием строки таблицы. Прежде всего, требуется получить из таблицы строку для редактирования. В простейшем случае это можно сделать по номеру строки. Если номер строки неизвестен, то можно использовать метод `Find()` коллекции `Rows` (подробнее о поиске строк в таблице будет рассказано ниже).

```
// Знаем номер строки, в данном случае – получаем вторую
DataRow row = Users.Rows[1];
// Можно найти строку по значению первичного ключа
DataRow row_2 = Users.Rows.Find(1);
```

Любая строка имеет несколько перегруженных индексаторов (свойство `Item`) для доступа к своим полям. В качестве индекса может использоваться имя столбца (как в предыдущем фрагменте кода), номер столбца или объект `DataColumn`, представляющий столбец:

```
// меняем содержимое определенных полей строки
row["user_name"] = "Alex Volosevich";
row["user_address"] = "Mars";
```

Второй способ редактирования строки аналогичен первому, за исключением того, что добавляются вызовы методов BeginEdit() и EndEdit() класса DataRow:

```
row.BeginEdit();
row["user_name"] = "Alex Volosevich";
row["user_address"] = "Mars";
row.EndEdit();
```

Методы BeginEdit() и EndEdit() позволяют *буферизировать* изменения строки. При вызове EndEdit() коррективы сохраняются в строке. Если вы решите отменить их, вызовите метод CancelEdit(), и строка вернется в состояние на момент вызова BeginEdit().

Есть еще одно отличие между этими двумя способами редактирования строки. Объект DataTable предоставляет события RowChanging, RowChanged, ColumnChanging и ColumnChanged, с помощью которых удается отслеживать изменения строки или поля. Порядок наступления этих событий зависит от того, как редактируется строка – с вызовом методов BeginEdit() и EndEdit() или без них. Если вызван метод BeginEdit(), наступление событий откладывается до вызова EndEdit() (если вызывается CancelEdit() никакие события не наступают).

Третий способ изменения строки – воспользоваться свойством строки ItemArray, Как и свойство Item, ItemArray позволяет просматривать и редактировать содержимое строки. Различие свойств в том, что Item рассчитано на работу с одним полем, а ItemArray принимает и возвращает массив, элементы которого соответствуют полям.

```
object[] data = new object[] {null, "A. Volosevich", "Mars"};
row.ItemArray = data;
```

Если необходимо с помощью свойства ItemArray отредактировать содержимое лишь отдельных полей строки, воспользуйтесь ключевым словом null. В предыдущем примере первое поле строки после редактирования осталось без изменений.

Поле строки может содержать пустое значение. Проверить это помогает метод IsNull(). Как и индексатор Item, IsNull() принимает в качестве параметра имя поля, его порядковый номер или объект DataColumn. Чтобы сделать значение поля пустым, требуется использовать специальный класс DBNull:

```
if (!row.IsNull("user_address"))
    row["user_address"] = DBNull.Value;
```

Чтобы удалить строку, достаточно вызвать метод Delete() объекта DataRow. После вызова метода Delete() строка помечается как удаленная, из базы она будет удалена после «закачки» содержимого DataSet в базу. Можно удалить строку из коллекции Rows таблицы, воспользовавшись методами коллекции Remove() или RemoveAt(). Первый метод принимает как параметр объект DataRow, второй – порядковый номер строки. Если строка удалена подобным образом, то при синхронизации изменений с БД, строка из базы удалена не будет.

Пусть имеется некий рассоединенный набор данных, в который помещена информация из БД. Допустим, что информация была изменена (в таблице редактировались, добавлялись или удалялись строки), и необходимо переместить содержимое набора обратно в базу. Было бы не продуктивно «перекачивать» в базу набор целиком. Более эффективный вариант – *отслеживание изменений*, и внесение в базу только корректирующих поправок.

Для поддержки корректирующих изменений базы каждая строка имеет *состояние* и *версию*. Состояние строки хранится в свойстве `RowState` и принимает следующие значения из перечисления `DataRowState`:

- `Unchanged` – строка не менялась (совпадает со строкой в базе);
- `Detached` – строка не относится к объекту `DataTable`;
- `Added` – строка добавлена в объект `DataTable`, но не существует в БД;
- `Modified` – строка была изменена по сравнению со строкой из базы;
- `Deleted` – строка ожидает удаления из базы.

В таблице 27 показано, как может изменяться состояние отдельной строки.

Таблица 27

Изменение состояния строки `DataRow`

Действие	**Код**	**Значение `RowState`**
Создание строки, не относящейся к объекту `DataTable`	`row = tbl.NewRow();` `row["id"] = 100;`	`Detached`
Добавление новой строки в `DataTable`	`tbl.Rows.Add(row);`	`Added`
Получение существующей строки	`row = tbl.Rows[0];`	`Unchanged`

Редактирование строки	`row.BeginEdit();`
	`row["id"] = 10000;`
	`row.EndEdit();`
		`Modified`

Удаление строки	`row.Delete();`	`Deleted`

С помощью свойства `RowState` можно найти в таблице измененные строки. Кроме этого, для любой строки существует возможность просмотреть, каким было значение ее полей *до изменения*. Свойство строки `Item` имеет перегруженный вариант, принимающий значение из перечисления `DataRowVersion`:

- `Current` – текущее значение поля;
- `Original` – оригинальное значение поля;
- `Proposed` – предполагаемое значение поля (действительно только при редактировании записи с использованием `BeginEdit()`).

Следующий фрагмент кода изменяет содержимое поля `name` первой строки таблицы `Artists`, а затем выводит оригинальное и текущее содержимое поля:

```
// Заполняем таблицу из базы, чтобы была "оригинальная" версия
string con = "Server=. . .";
SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Artists", con);
DataTable Artists = new DataTable("Artists");
da.Fill(Artists);
// Меняем содержимое первой строки
DataRow row = Artists.Rows[0];
row["name"] = "Alex";
```

```
Console.WriteLine(row["name", DataRowVersion.Current]);
Console.WriteLine(row["name", DataRowVersion.Original]);
```

При редактировании строки с использованием методов `BeginEdit()` и `EndEdit()` доступна версия с «предполагаемым» содержимым строки. После вызова `EndEdit()` все изменения сохраняются в текущей версии строки. Тем не менее, до этого изменения считаются отложенными, поскольку их удается отменить вызовом метода `CancelEdit()`. Чтобы при редактировании строки просмотреть предполагаемое значение поля, воспользуйтесь свойством `Item` со вторым элементом `DataRowVersion.Proposed`:

```
row.BeginEdit();
row["name"] = "Alex";
// Выводит состояние записи до вызова BeginEdit()
Console.WriteLine(row["name", DataRowVersion.Current]);
// Естественно, выводит то значение, которое прочитали из базы
Console.WriteLine(row["name", DataRowVersion.Original]);
// Выводит Alex – предполагаемое значение
Console.WriteLine(row["name", DataRowVersion.Proposed]);
row.EndEdit();
// Выводит Alex – мы "закрепили" значение, вызвав EndEdit()
Console.WriteLine(row["name", DataRowVersion.Current]);
```

Таблица 28 показывает возможные значения, возвращаемые свойством `Item` в зависимости от указанной версии ([Искл.] обозначает генерацию исключительной ситуации при попытке получить определенную версию).

Таблица 28

Значения свойства `Item` в зависимости от версии строки

Пример	`Current`	`Original`	`Proposed`

Только что созданная строка, не связанная с таблицей
`row = tbl.NewRow();`
`row["id"] = 10;`
	[Искл.]	[Искл.]	10

В таблицу добавлена новая строка `tbl.Rows.Add(row);`	10	[Искл.]	[Искл.]
Данные загружены из базы, из таблицы получена существующая строка `row = tbl.Rows[0];`	1[35]	1	[Искл.]

Первое изменение существующего поля
`row.BeginEdit();`
`row["id"] = 100;`
	1	1	100

После первого изменения `row.EndEdit();`	100	1	[Искл.]

После второго изменения содержимого поля
`row.BeginEdit();`
`row["id"] = 300;`
`row.EndEdit();`
	300	1	[Искл.]

После отмены изменений
`row.BeginEdit();`
`row["id"] = 500;`
`row.CancelEdit()`
	300	1	[Искл.]

[35] Это значение получено из базы.

После удаления записи
`DataRow row = Artists.Rows[0];`
`row.Delete();`
	[Искл.]	1	[Искл.]

Для контроля существования версии можно использовать метод `HasVersion()`:

```
if(row.HasVersion(DataRowVersion.Current))
    Console.WriteLine("Current version exists");
```

Любая строка предоставляет методы `AcceptChanges()` и `RejectChanges()`. Вызов первого метода закрепляет в строке все отложенные изменения, а вызов второго – отбрасывает отложенные изменения. Иными словами, вызов `AcceptChanges()` приводит к замене значений `Original`-версии строки значениями из `Current`-версии и установке у строки `RowState = Unchanged`. Вызов `RejectChanges()` устанавливает `RowState= Unchanged`, но значения `Current`-версии строки меняются на значения `Original`-версии. При загрузке изменений `DataSet` в базу у каждой строки неявно вызывается метод `AcceptChanges()`. Внимание: явное использование указанных методов может породить проблемы при синхронизации набора данных и базы.

Отдельная строка таблицы, а также отдельное поле строки позволяют задать текстовую метку при наличии ошибочных значений. Метка для поля устанавливается методом `SetColumnError()`, а читается методом `GetColumnError()`. Можно пометить всю строку, используя свойство строки `RowError`. Если метка была указана, то свойство строки `HasErrors = true`. Поля с метками ошибок могут быть получены при помощи вызова метода строки `GetColumnsInErrors()`. Метод `ClearErrors()` удаляет все метки ошибок в полях строки и очищает свойство `RowError`. Работу с описанными методами демонстрирует следующий пример:

```
// Получили строку таблицы
DataRow row = Artists.Rows[0];

// После продолжительного анализа решили, что с полем name
// что-то не так; поставим метку на это поле
// (можно было использовать номер поля)
row.SetColumnError("name", "Wrong Name");
// Заодно пометим всю строку как "бракованную"
row.RowError = "Row with Errors";

// Где-то в коде: проверим-ка мы строку на предмет ошибок...
if (row.HasErrors) {
    // Выведем описание ошибки для всей строки
    Console.WriteLine(row.RowError);
    // Получим массив колонок (полей) с ошибками
    DataColumn[] err_Columns = row.GetColumnsInError();
    // Пройдемся по массиву и выведем информацию об ошибках
    foreach (DataColumn dc in err_Columns) {
        Console.WriteLine(dc.ColumnName);
        Console.WriteLine(row.GetColumnError(dc));
    }
```

```
        // Считаем, что с ошибками разобрались; очистим сообщения
        row.ClearErrors();
    }
```

4.11. РАБОТА С ОБЪЕКТОМ КЛАССА DATATABLE

Любой рассоединенный набор данных содержит одну или несколько таблиц. В данном параграфе рассматривается настройка отдельной таблицы, ее свойства и методы.

Наиболее важные свойства класса `DataTable` приведены в таблице 29.

Таблица 29

Свойства класса `DataTable`

Имя свойства	Описание
`CaseSensitive`	Определяет, учитывается ли регистр при поиске строк в таблице (по умолчанию `false` – не учитывается)
`ChildRelations`	Возвращает коллекцию подчиненных связей для таблицы
`Columns`	Набор столбцов таблицы
`Constraints`	Набор ограничений, заданных для таблицы
`DataSet`	Рассоединенный набор данных, включающий таблицу
`DefaultView`	Указывает на представление по умолчанию (view) для таблицы
`ExtendedProperties`	Коллекция пользовательских свойств таблицы
`HasErrors`	Указывает, содержит ли таблица ошибки
`Locale`	Свойство имеет тип `CultureInfo` и определяет региональные параметры, используемые таблицей при сравнении строк
`MinimumCapacity`	Служит для получения или установки исходного количества строк таблицы (по умолчанию – 25 строк)
`ParentRelations`	Коллекция родительских отношений для таблицы
`PrimaryKey`	Массив столбцов, формирующих первичный ключ таблицы
`Rows`	Набор строк таблицы
`TableName`	Строка с именем таблицы

Если требуется вручную создать таблицу, то можно воспользоваться конструктором класса `DataTable`. Конструктор имеет перегруженную версию, которая позволяет установить имя таблицы – свойство `TableName`:

```
DataTable dt = new DataTable();
dt.TableName = "Main Table";
// аналогичный результат:
DataTable dt = new DataTable("Main Table");
```

Работа со свойствами `Columns` и `Rows` обсуждалась ранее. Свойства `ChildRelations`, `ParentRelations`, `Constraints` и `PrimaryKey` будут рассмотрены в следующем параграфе. При помощи свойства `DataSet` таблица связывается с определенным набором данных. Работа со свойством `ExtendedProperties` аналогична примерам, приводимым для класса `DataColumn`. Свойство таблицы `HasErrors` устанавливается в `true`, если хотя бы у одной из строк таблицы свойство `HasErrors = true`.

Кроме набора свойств, класс `DataTable` обладает следующими методами, перечисленными в таблице 30.

Таблица 30

Методы класса DataTable

Имя метода	Описание
AcceptChanges()	Метод фиксирует все изменения данных в строках таблицы, которые были проделаны с момента предыдущего вызова AcceptChanges()
BeginLoadData()	Отключает все ограничения при загрузке данных
Clear()	Уничтожаются все строки таблицы
Clone()	Метод клонирует структуру таблицы и возвращает таблицу без строк
Compute()	Метод применяет строку-выражение, заданную в качестве параметра, к диапазону строк таблицы
Copy()	Метод клонирует и структуру, и данные таблицы
EndLoadData()	Активирует ограничения после загрузки данных
GetChanges()	Метод возвращает таблицу с идентичной схемой, содержащую изменения, которые еще не зафиксированы методом AcceptChanges()
GetErrors()	Возвращает массив объектов DataRow, которые нарушают ограничения таблицы
ImportRow()	В таблицу вставляется строка, указанная в качестве параметра метода
LoadDataRow()	Добавляет или обновляет строку таблицы, основываясь на содержимом массива-параметра
NewRow()	Создается пустая строка по схеме столбцов таблицы
RejectChanges()	Метод отменяет изменения, которые еще не зафиксированы вызовом AcceptChanges()
Reset()	Восстанавливает оригинальное состояние объекта DataTable, в котором он находился до инициализации
Select()	Возвращает массив строк таблицы на основании заданного критерия поиска

Рассмотрим некоторые методы таблицы подробнее. Если в таблицу добавляется группа строк (объектов DataRow), то для повышения производительности кода следует применить методы BeginLoadData() и EndLoadData(). При вызове метода BeginLoadData() отключаются определенные на таблице ограничения. Активировать их можно, вызвав метод EndLoadData(). Если в таблице есть строки, нарушающие какие-либо ограничения, при вызове метода EndLoadData() генерируется исключение ConstraintException. Чтобы определить, какие именно строки нарушают ограничения, следует просмотреть массив, возвращаемый методом GetErrors().

Метод Clear() позволяет удалить из таблицы все строки. Вызвать его быстрее, чем освободить оригинальный и создать новый объект DataTable с идентичной структурой.

Метод Compute() позволяет выполнять агрегатные запросы к отдельным столбцам таблицы на основе заданных критериев поиска[36]. Метод принимает

[36] Метод Compute() не позволяет вычислять агрегатные значения на основе нескольких столбцов.

два строковых параметра. Первый содержит *выражение для вычисления*[37], второй является *фильтром*, отбирающим некий диапазон строк таблицы. Следующий пример кода загружает из базы CD_Rent таблицу Rent и вычисляет средний рейтинг диска с disk_id = 1:

```
SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Rent",
            "Server=(local);Database=CD_Rent;Integrated Security=SSPI");
DataTable Rent = new DataTable("Rent");
da.Fill(Rent);

double r = (double) Rent.Compute("Avg(disk_rating)", "disk_id = 1");
Console.WriteLine(r);
```

Группа методов используется для добавления в таблицу новых строк. Метод таблицы ImportRow() принимает объект-строку и добавляет его в таблицу. Метод LoadDataRow() принимает в качестве первого аргумента массив, элементы которого соответствуют элементам коллекции Columns таблицы. Метод пытается найти в таблице строку, соответствующую своему первому аргументу и обновить ее. Если строка не найдена, она создается. Второй аргумент метода LoadDataRow() – это логическое значение, управляющее свойством RowState нового объекта DataRow. Чтобы задать свойству RowState значение Added, передайте в качестве второго аргумента false; если необходимо задать значение Unmodified, передайте true. Метод LoadDataRow() возвращает ссылку на созданную или найденную строку. Метод NewRow() возвращает для таблицы новую строку, но не добавляет его в коллекцию Rows таблицы. Добавление следует осуществить вручную, предварительно заполнив необходимые поля строки. Рекомендуется следующее:

- Для импорта строки из сторонней таблицы используйте метод ImportRow().
- Для одновременного импорта нескольких строк, например на основе содержимого файла, применяйте метод LoadDataRow(). Это позволит вам писать меньше кода, чем при работе с методом NewRow().
- Во всех остальных случаях вызывайте метод NewRow().

Любая таблица поддерживает следующий набор событий:

- ColumnChanged – происходит после изменения содержимого поля некой строки;
- ColumnChanging – перед изменением содержимого поля строки;
- RowChanged – происходит после изменения содержимого строки;
- RowChanging – генерируется перед изменением содержимого строки;
- RowDeleted – происходит после удаления строки;
- RowDeleting – происходит перед удалением строки.

События ColumnChanged и ColumnChanging наступают каждый раз, когда изменяется содержимое одного из полей строки. Они позволяют осуществлять проверку данных, активировать и деактивировать элементы управления, и т.д.

[37] Как и в случае с использованием свойства Expression класса DataColumn, для подробного ознакомления с синтаксисом построения выражений отсылаем заинтересованных читателей к справке MSDN.

У данных событий есть аргумент типа DataColumnChangeEventArgs, обладающий свойствами Row и Column, которые позволяют определить, какие именно поле и строка изменены. Помните: при использовании данных событий для редактирования содержимого строки иногда возникает замкнутый цикл.

События RowChanged и RowChanging наступают каждый раз, когда изменяется содержимое строки или значение свойства RowState строки. Чтобы определить, почему наступило событие, достаточно просмотреть значение свойства Action аргумента DataColumnChangeEventArgs этого события. Свойство Row указанного аргумента позволяет обращаться к изменяемой записи.

События RowDeleted и RowDeleting предоставляют такие же аргументы и свойства, как события RowChanged и RowChanging. Единственное отличие в том, что данные события наступают при удалении строки из таблицы.

4.12. DATASET И СХЕМА РАССОЕДИНЕННОГО НАБОРА ДАННЫХ

Описав в предыдущих параграфах основные компоненты набора данных DataSet, рассмотрим этот класс более подробно. Начнем с таблицы свойств класса DataSet.

Таблица 31

Свойства класса DataSet

Имя свойства	Описание
CaseSensitive	Определяет, учитывается ли регистр при поиске строк
DataSetName	Строка с именем набора данных
EnforceConstraints	Определяет, обеспечивает ли DataSet выполнение определенных на нем ограничений
ExtendedProperties	Коллекция пользовательских свойств набора данных
HasErrors	Указывает, содержит ли набор данных ошибки
Locale	Свойство имеет тип CultureInfo и определяет региональные параметры, используемые набором данных при сравнении строк
Relations	Коллекция отношений, связывающих таблицы из набора данных
Tables	Возвращает коллекцию таблиц набора данных

Большинство свойств в особых пояснениях не нуждается, ибо, по сути, является аналогом свойств таблицы, но для всего набора данных. Свойство Relations содержит коллекцию связей между таблицами и является часть описания схемы данных. Если значение свойства EnforceConstraints установить в false, то при загрузке информации в DataSet данные не будут проверяться на соответствие ограничениям. Это повышает производительность.

В таблице 32 перечислены методы набора данных:

Таблица 32

Методы класса DataSet

Имя метода	Описание
AcceptChanges()	Метод фиксирует все изменения данных, которые были проделаны с момента предыдущего вызова AcceptChanges()
Clear()	Уничтожаются все строки всех таблиц набора данных

`Clone()`	Метод клонирует структуру набора и возвращает пустой набор
`Copy()`	Метод клонирует и структуру, и данные набора
`GetChanges()`	Возвращает новый `DataSet` с идентичной схемой, содержащий измененные строки и таблицы оригинального объекта `DataSet`
`GetXml()`	Возвращает содержимое объекта `DataSet` в виде XML-строки
`GetXmlSchema()`	Возвращает схему объекта `DataSet` в виде XML-строки
`HasChanges()`	Возвращает логическое значение, указывающее, содержат ли строки из состава `DataSet` отложенные изменения
`Merge()`	Осуществляет слияние данных из другого объекта `DataSet`, `DataTable` или массива объектов `DataRow` и данных текущего объекта `DataSet`
`ReadXml()`	Читает содержимое `DataSet` в XML-формате из файла, `Stream`, `TextReader` или `XmlReader`
`ReadXmlSchema()`	Работает как `ReadXml()`, но читает только схему `DataSet`
`RejectChanges()`	Метод отменяет изменения, которые еще не зафиксированы вызовом `AcceptChanges()`
`Reset()`	Восстанавливает оригинальное состояние `DataSet`
`WriteXml()`	Записывает содержимое `DataSet` в XML-формате в файл, `Stream`, `TextWriter` или `XmlWriter`
`WriteXmlSchema()`	Работает как `WriteXml()`, но записывает только схему `DataSet`

Рассмотрим, как вручную задать схему набора данных. Напомним, что правильная схема включает тип и имя отдельных столбцов таблицы, ограничения на столбцы и связи между таблицами. Будем создавать схему для таблиц `Artists` и `Disks` из базы `CD_Rent`.

Вначале создадим объекты, соответствующие столбцам и таблицам, и поместим столбцы в таблицы:

```
DataColumn id_artists = new DataColumn("id", typeof(int));
DataColumn name = new DataColumn("name", typeof(string));

DataTable Artists = new DataTable("Artists");
Artists.Columns.Add(id_artists);
Artists.Columns.Add(name);

DataColumn id_disks = new DataColumn("id", typeof(int));
DataColumn title = new DataColumn("title", typeof(string));
DataColumn artist_id = new DataColumn("artist_id",
                                      typeof(int));
DataColumn release_year=new DataColumn("release_year",
                                       typeof(string));

DataTable Disks = new DataTable("Disks");
Disks.Columns.Add(id_disks);
Disks.Columns.Add(title);
Disks.Columns.Add(artist_id);
Disks.Columns.Add(release_year);
```

Теперь можно выполнить дополнительную настройку отдельных столбцов – задать максимальную длину, отсутствие нулевых значений:

```
id_artists.AllowDBNull = false;
name.AllowDBNull = false;
```

```
id_disks.AllowDBNull = false;
title.AllowDBNull = false;
name.MaxLength = 50;
title.MaxLength = 50;
release_year.MaxLength = 4;
```

Перейдем к работе с таблицами. Как и в реляционных база данных, один или несколько столбцов таблицы `DataTable` могут исполнять роль *первичного ключа*. Первичный ключ должен быть уникальным в пределах таблицы. Свойство таблицы `PrimaryKey` служит для получения или установки массива столбцов, формирующих первичный ключ. Если столбец является частью первичного ключа, его свойство `AllowDBNull` автоматически устанавливается в `false`. Установим первичные ключи наших таблиц:

```
DataColumn[] Artists_PK = new DataColumn[1] {id_artists};
Artists.PrimaryKey = Artists_PK;

DataColumn[] Disks_PK = new DataColumn[1] {id_disks};
Disks.PrimaryKey = Disks_PK;
```

Таблица поддерживает свойство `Constraints` – набор ограничений таблицы. Значением данного свойства является коллекция объектов класса `Constraint`. Класс `Constraint` – абстрактный базовый класс, имеющий два производных класса: `ForeignKeyConstraint` и `UniqueConstraint`.

Класс `UniqueConstraint` – это класс, при помощи объектов которого реализуется концепция уникальности значений полей строки. Основными свойствами этого класса является массив столбцов `Columns`, имя ограничения `ConstraintName` и булево свойство `IsPrimaryKey`, которое показывает, представляет ли данное ограничение первичный ключ таблицы. Ограничение вида `UniqueConstraint` автоматически добавляется в таблицу при создании первичного ключа. Это же ограничение появляется, если в таблицу добавляется столбец с установленным свойством `Unique`.

Так как в нашем примере мы уже создали первичные ключи таблиц, то их коллекции `Constraints` не пусты. Изменим имя ограничения в одной из таблиц и добавим ограничение для столбца `name` таблицы `Artists`, полагая значения в этом столбце уникальными:

```
Artists.Constraints[0].ConstraintName = "Primary Key";
UniqueConstraint uc = new UniqueConstraint("Unique Name",
                                           new DataColumn[]{name});
Artists.Constraints.Add(uc);
```

Класс `ForeignKeyConstraint` служит для описания внешних ключей таблицы. Его основные свойства перечислены в таблице 33.

Таблица 33

Основные свойства класса `ForeignKeyConstraint`

Имя свойства	Описание
`AcceptRejectRule`	Определяет, каскадируются ли результаты вызова методов `AcceptChanges()` и `RejectChanges()` родительского объекта `DataRow` в дочерние строки

Columns	Возвращает столбцы дочерней таблицы, составляющие ограничение
ConstraintName	Имя ограничения
DeleteRule	Определяет, каскадируются ли удаление родительского объекта DataRow в дочерние строки
ExtendedProperties	Набор динамических свойств
RelatedColumns	Столбцы родительской таблицы, составляющие ограничение
RelatedTable	Родительская таблица ограничения
Table	Дочерняя таблица ограничения
UpdateRule	Управляет каскадированием изменений родительской строки в дочерние строки

Свойства AcceptRejectRule, DeleteRule и UpdateRule управляют порядком каскадирования изменений родительской строки в дочерние строки. Свойство AcceptRejectRule принимает значение из одноименного перечисления. Значение этого свойства по умолчанию – None: вызов метода AcceptChanges() или RejectChanges() объекта DataRow не сказывается на дочерних строках последнего. Если задать свойству AcceptRejectRule значение Cascade, изменения каскадируются в дочерние строки, определенные объектом ForeignKeyConstraint.

Свойства DeleteRule и UpdateRule функционируют аналогичным образом, но принимают значения из перечисления Rule. Значение этих свойств по умолчанию – Cascade, т. е. изменения родительской строки каскадируются в дочерние строки. Например, при вызове метода Delete() родительского объекта DataRow вы неявно вызываете и метод Delete() его дочерних строк. Точно так же, редактируя значение поля ключа родительского объекта DataRow, вы неявно изменяете содержимое соответствующих полей дочерних строк. Если каскадировать изменения не требуется, задайте свойствам DeleteRule и UpdateRule значение None. Можно также задать им значение SetNull или SetDefault. В первом случае при изменении или удалении содержимого родительской строки соответствующим полям дочерних строк задаются значения NULL, а во втором – их значения по умолчанию.

Обычно нет необходимости создавать объекты класса ForeignKeyConstraint вручную. Они генерируются автоматически при добавлении связи между таблицами.

Связь между таблицами представлена объектом класса DataRelation. Большинство свойств DataRelation доступно только для чтения. Задать их значение можно средствами конструкторов объекта DataRelation. В таблице 34 перечислены наиболее часто используемые свойства.

Таблица 34

Свойства класса DataRelation

Имя свойства	Описание
ChildColumns	Свойство возвращает массив, содержащий объекты DataColumn из дочернего объекта DataTable
ChildKeyConstraint	Указывает ограничение FOREIGN KEY в дочерней таблице
ChildTable	Указывает дочернюю таблицу связи
DataSet	Набор данных, в котором находится объект DataRelation

ExtendedProperties	Набор динамических свойств
Nested	Логическое значение. Указывает, нужно ли преобразовывать дочерние строки в дочерние элементы при записи содержимого DataSet в XML-файл
ParentColumns	Родительские столбцы, определяющие отношение
ParentKeyConstraint	Указывает ограничение UNIQUE в родительской таблице
ParentTable	Указывает родительскую таблицу
RelationName	Строка с именем отношения

При создании объекта DataRelation следует указать его имя, чтобы объект удалось найти в наборе; кроме этого, необходимо указать родительский и дочерний столбцы, на которых будет основано отношение. Чтобы упростить создание связей, класс DataRelation предоставляет отдельные конструкторы, принимающие как объекты DataColumn, так и массивы таких объектов.

Вернемся к нашему примеру. Создадим рассоединенный набор данных и поместим в него таблицы:

```
DataSet CD_Rent = new DataSet("CD_Rent_Part");
CD_Rent.Tables.Add(Artists);
CD_Rent.Tables.Add(Disks);
```

Между таблицами Artists и Disks существует связь по внешнему ключу. А именно, таблица Disks является дочерней для таблицы Artists, так как значения поля artist_id – это значения первичного ключа таблицы Artists. Создадим объект DataRelation, описывающий эту связь:

```
// Используем самый простой конструктор, который устанавливает
// имя отношения, родительский и дочерний столбцы отношения
DataRelation Artists_to_Disks = new DataRelation(
                                        "Artists_to_Disks",
                                        id_artists, artist_id);
// Добавим отношение в набор данных
CD_Rent.Relations.Add(Artists_to_Disks);
```

После выполнения данного кода коллекция Constraints объекта Artists будет содержать два отношения с именами Primary Key и Unique Artist Name (оба – класса UniqueConstraint). Коллекция Constraints объекта Disks будет также содержать два отношения. Одно – с именем Constraint1 типа UniqueConstraint, второе – с именем Artists_to_Disks типа ForeignKeyConstraint. Создание схемы данных можно считать завершенным. При желании можно добавить обработчики событий в наши таблицы.

При помощи метода адаптера WriteXmlSchema() созданную схему можно сохранить в XML-файле (правильным расширением файла является *.xsd, так как схема сохраняется в виде XSD-описания).

4.13. ТИПИЗИРОВАННЫЕ DATASET

В обычных объектах DataSet для доступа к отдельным элементам используются индексированные коллекции. Недостатком такого подхода является отсутствие контроля правильности имен таблиц и столбцов во время компиляции.

Платформа .NET поддерживает типизированные наборы данных. *Типизированный набор данных* – это класс, который является наследником класса

DataSet. В отличие от DataSet, в типизированном наборе для доступа к отдельным компонентам служат свойства. Имя свойства – это имя элемента DataSet (например, имя таблицы), тип свойства – класс-наследник стандартного класса из DataSet.

Создать типизированный DataSet можно несколькими способами. В Visual Studio для этого используется специальный *мастер* (*wizard*). В состав .NET Framework SDK входит утилита xsd.exe, формирующая типизированные наборы данных на основании XSD-файла со схемой набора.

Разберем пример использования xsd.exe. Пусть есть набор данных со схемой, построенной в предыдущем параграфе и сохраненной в файле schema.xsd. Выполним команду[38]:

```
C:\TMP>xsd schema.xsd /d
```

В результате получим файл schema.cs, фрагмент которого приведен ниже:

```
using System;
using System.Data;
using System.Xml;
using System.Runtime.Serialization;

public class CD_Rent_Part : DataSet {
    private ArtistsDataTable tableArtists;
    private DisksDataTable tableDisks;
    private DataRelation relationArtists_to_Disks;

    public CD_Rent_Part() {. . .}

    public ArtistsDataTable Artists {
        get {
            return this.tableArtists;
        }
    }
    public DisksDataTable Disks {
        get {
            return this.tableDisks;
        }
    }
    public override DataSet Clone() {
        CD_Rent_Part cln = ((CD_Rent_Part)(base.Clone()));
        cln.InitVars();
        return cln;
    }
    public class ArtistsDataTable: DataTable,IEnumerable {...}
    public class ArtistsRow : DataRow {. . .}
    public class DisksDataTable : DataTable, IEnumerable {...}
    public class DisksRow : DataRow {. . .}
}
```

[38] Подробный состав параметров утилиты xsd.exe изучите самостоятельно.

Используя файл schema.cs и класс CD_Rent_Part, работать с набором данных можно следующим образом. Вместо кода, подобного следующему,

```
string s = (string) CD_Rent.Tables["Artists"].Rows[0]["name"];
```

можно писать такой код:

```
CD_Rent_Part cd = new CD_Rent_Part();
string s1 = cd.Artists[0].name;
```

Обратите внимание:

1. Таблица представлена свойством Artists (типа ArtistsDataTable).
2. Для доступа к определенной строке таблицы используется индексатор, примененный к свойству Artists.
3. Тип строки – ArtistsRow. Это позволяет обратиться к полю строки как к свойству и не выполнять приведение типов.

Подытожим: типизированные DataSet позволяют установить контроль правильности кода во время компиляции. Это способно упростить разработку приложений и сделать использование типов более безопасным.

4.14. ПОИСК И ФИЛЬТРАЦИЯ ДАННЫХ В DATASET

В примерах данного параграфа предполагается, что имеется DataSet с заданной схемой (см. параграф 12), и в этот набор данных загружены две таблицы из базы CD_Rent:

```
SqlDataAdapter da = new SqlDataAdapter(
                        "SELECT * FROM Artists",
                        "Server=(local);Database=CD_Rent;" +
                        "Integrated Security=SSPI");
da.MissingSchemaAction = MissingSchemaAction.Error;
da.Fill(CD_Rent, "Artists");
da.SelectCommand.CommandText = "SELECT * FROM Disks";
da.Fill(CD_Rent, "Disks");
```

Наличие в схеме данных связей между таблицами позволяет осуществлять *навигацию* по набору данных. Для этого используются методы GetChildRows(), GetParentRow() и GetParentRows() класса DataRow.

Метод GetChildRows() возвращает массив дочерних строк той строки, у которой вызывается. Метод принимает в качестве параметра имя связи между таблицами либо объект, описывающий связь. В следующем фрагменте кода для каждого исполнителя выводится список его дисков:

```
foreach (DataRow row in Artists.Rows) {
    Console.WriteLine(row["name"]);
    DataRow[] childRows =
                          row.GetChildRows("Artists_to_Disks");
    foreach (DataRow r in childRows)
        Console.WriteLine("\t" + r["title"]);
}
```

Метод GetParentRow() позволяет получить родительскую строку. Как и предыдущий метод, GetParentRow() принимает в качестве параметра имя свя-

зи между таблицами либо объект, описывающий связь. В примере для каждого диска выводится его исполнитель:

```
foreach(DataRow row in Disks.Rows) {
    Console.Write(row["title"]);
    DataRow parentRow = row.GetParentRow("Artists_to_Disks");
    Console.WriteLine("" + parentRow["name"]);
}
```

Метод `GetParentRows()` возвращает все родительские строки определенной строки. Данный метод имеет смысл использовать, если задана такая связь, при которой колонка родительской таблицы не является первичным ключом (не уникальна).

Все три рассмотренных метода имеют перегруженные варианты, которые позволяют получить заданную версию строки. Версия указывается в качестве дополнительного параметра.

Достаточно часто возникает задача поиска информации в таблице или наборе данных. Для этого можно использовать два метода. Первый метод – `Find()` – позволяет искать строки по значениям первичного ключа. Второй – `Select()`, выступает в качестве фильтра, возвращая строки данных, удовлетворяющие критериям поиска.

Метод `Find()`, который предоставляет класс `DataRowCollection` ищет строки таблицы с определенным значением первичного ключа. Метод `Find()` принимает как параметр значение первичного ключа искомой строки. Поскольку значения первичного ключа уникальны, метод вернет не более одного объекта `DataRow`. Следующий фрагмент кода ищет данные в таблице `Artists`:

```
// В Artists определен первичный ключ.
// Find() принимает как параметр объект
// и сам приводит его к нужному типу
DataRow row = Artists.Rows.Find(1);
if (row != null) Console.WriteLine(row["name"]);
```

Метод `Find()` перегружен в расчете на случаи, когда первичный ключ `DataTable` состоит из нескольких объектов `DataColumn`. В этом случае в `Find()` передается массив объектов, представляющих значения первичного ключа.

Метод `Select()` класса `DataTable` позволяет искать ряды по определенному критерию. В простейшем случае методу передается строка, описывающая критерий поиска. Метод возвращает массив найденных строк. В следующем примере из таблицы `Disks` выбираются все диски, записанные в 2006 году:

```
DataRow[] result = Disks.Select("release_year = 2006");
foreach (DataRow row in result)
    Console.WriteLine(row["title"]);
```

Несколько слов о построении строки-критерия. Строка может содержать слово `LIKE`, которое осуществляет поиск по частичному совпадению. В этом случае используется *шаблон*, содержащий метасимволы `*` или `%` для обозначения начальной или конечной части строки. Например, осуществим поиск всех дисков, название которых начинается с `"U"`:

```
DataRow[] result = Disks.Select("title LIKE 'U*'");
```

Иногда требуется заключить в символы-разделители имена столбцов, используемые в критерии поиска. Например, когда имя содержит пробел или другой символ, не относящийся к алфавитно-цифровым, или похоже на зарезервированное слово типа `LIKE` или `SUM`. Так, если имя столбца – `Space In Name` и требуется выбрать все строки, значение поля `Space In Name` которых равно 3, воспользуйтесь следующим критерием поиска:

```
strCriteria = "[Space In Name] = 3"
```

Если имя столбца включает символ-разделитель, поставьте в критерии поиска перед закрывающим символом-разделителем (`]`) управляющий символ (`\`). Например, если имя столбца – `Bad]Column[Name` и требуется выбрать все строки, у которых значение этого поля равно 5, используйте такую строку поиска:

```
strCriteria = @"[Bad\]Colunin[Name] = 5";
```

Метод `Select()` перегружен. Вы можете просто передать строку запроса, а можете включить в нее порядок сортировки, а также параметр, определяющий состояние искомых строк (например, только добавленные строки или только измененные).

Управлять порядком строк, возвращаемых `Select()`, можно посредством одной из сигнатур перегруженного метода. Следующий фрагмент получает массив строк, отсортированных в убывающем порядке по полю `title`:

```
DataRow[] result = Disks.Select("release_year = 2005", "title DESC");
```

Если надо выполнить поиск только в измененных строках таблицы, воспользуйтесь перегруженным методом `Select()` и укажите значение из перечисления `DataViewRowState`. Можно считать, что это значение – фильтр, добавленный в критерий поиска. Предположим, требуется просмотреть только измененные и удаленные записи таблицы. Воспользуемся константами `ModifiedOriginal` и `Deleted` из перечисления `DataViewRowState` и укажем в качестве параметров фильтрации и сортировки пустые строки:

```
DataRow[] result = Disks.Select("", "",
                           DataViewRowState.ModifiedOriginal |
                           DataViewRowState.Deleted);
```

4.15. КЛАСС DATAVIEW

Метод `DataTable.Select()` – очень мощный и гибкий, но не является оптимальным решением для всех ситуаций. У него есть два основных ограничения. Во-первых, метод принимает динамические критерии поиска и поэтому не может быть сверхэффективным. Во-вторых, Windows- и Web-формы не поддерживают *связывание* с возвращаемым значением метода `Select()` – массивом объектов `DataRow`. В ADO.NET предусмотрено решение, обходящее оба этих ограничения – объект класса `DataView`.

Объекты `DataView` позволяют фильтровать, сортировать и вести поиск в содержимом таблиц. Объекты `DataView` не являются SQL-запросами, в отличие от *представлений* (*view*) в базах данных. С помощью `DataView` нельзя объеди-

нить данные двух таблиц, равно как и просмотреть отдельные столбцы таблицы. Объекты DataView поддерживают фильтрацию запросов на основе динамических критериев, но разрешают обращаться только к одному объекту DataTable; кроме того, через DataView всегда доступны все столбцы таблицы. У объекта DataView нет собственной копии данных. При обращении через DataView к данным он возвращает записи, хранящиеся в соответствующем объекте DataTable.

Чтобы просмотреть с помощью объекта DataView данные некоторой таблицы, его следует связать с этой таблицей. Есть два способа сделать это: используя свойство Table объекта DataView или при помощи конструктора DataView. Следующие фрагменты эквивалентны:

```
DataTable dt = CD_Rent.Tables["Disks"];

// Первый вариант
DataView dv = new DataView();
dv.Table = dt;

// Второй вариант
DataView dv = new DataView(dt);
```

У DataView также есть конструктор, сигнатура которого более точно соответствует методу DataTable.Select(). Этот конструктор задает значения свойств Table, RowFilter, Sort и RowStateFilter объекта DataView в одной строке кода. Следующие варианты кода эквивалентны:

```
DataTable dt = CD_Rent.Tables["Disks"];

// Первый вариант
DataView dv = new DataView();
dv.Table = dt;
dv.RowFilter = "release_year = 2005";
dv.Sort = "title DESC";
dv.RowStateFilter = DataViewRowState.ModifiedOriginal;

// Второй вариант
DataView dv = new DataView(dt, "release_year = 2005",
                           "title DESC",
                           DataViewRowState.ModifiedOriginal);
```

Свойство RowStateFilter принимает значения из перечисления DataViewRowState. Это перечисление можно рассматривать как комбинацию свойства RowState объекта DataRow и перечисления DataRowVersion.

- Added – отображаются добавленные строки;
- CurrentRows – отображаются строки, которые не были удалены (значение по умолчанию);
- Deleted – отображаются удаленные строки;
- ModifiedCurrent – отображаются измененные строки с их текущими значениями;

- ModifiedOriginal – отображаются измененные строки с их оригинальными значениями;
- None – строки не отображаются;
- OriginalRows – отображаются удаленные, измененные и не изменявшиеся строки с их оригинальными значениями;
- Unchanged – отображаются строки, которые не изменялись.

Свойство RowStateFilter работает в качестве двойного фильтра. Например, если задать ему значение ModifiedOriginal, через объект DataView окажутся доступны только измененные записи, и вы будете видеть их оригинальные значения.

Объект DataView возвращает данные с помощью собственного специализированного объекта – DataRowView. Функциональность DataRowView в целом аналогична функциональности DataRow. DataRowView обладает свойством Item, позволяющим обращаться к содержимому поля как по имени, так и по порядковому номеру. И хотя свойство Item разрешает просматривать и изменять содержимое поля, через DataRowView доступна только одна версия данных строки – та, которая указана при помощи свойства DataRowVersion. Если объект DataRowView не обеспечивает требуемых возможностей, обратитесь при помощи свойства Row этого объекта к соответствующему объекту DataRow из таблицы.

Доступ к данным объекта DataTable при помощи DataView осуществляется иначе, чем непосредственный доступ к объекту DataTable. Таблица предоставляет свои строки через свойство Rows. У DataView нет похожего, допускающего простое перечисление, набора. DataView имеет свойство Count, возвращающее число строк, и индексатор Item с целым индексом. Используя эти свойства, можно создать простой цикл для просмотра всех строк:

```
DataView dv = new DataView(dt, "release_year = 2005",
                           "title DESC",
                           DataViewRowState.CurrentRows);
for (int i = 0; i < dv.Count; i++) {
    DataRowView drv = dv[i];
    Console.WriteLine(drv["title"]);
}
```

Класс DataView предоставляет методы Find() и FindRows(), позволяющие искать в нем данные. Эти методы аналогичны методу Find() коллекции Rows таблицы. Задав значение свойства Sort объекта DataView, вы получите возможность с помощью метода Find() искать строки по значениям столбцов, перечисленных в свойстве Sort. Как и в случае с методом DataRowCollection.Find(), одноименному методу DataView разрешено передавать одно значение или массив значений. Тем не менее, метод DataView.Find() возвращает не объект DataRow или DataRowView, а значение целого типа, соответствующее порядковому номеру нужной строки в объекте DataView. Если искомая строка не найдена, метод вернет -1:

```
DataTable dt = CD_Rent.Tables["Disks"];
DataView dv = new DataView(dt);
```

```
dv.RowFilter = "release_year = 2005";
dv.Sort = "title";
int findIndex = dv.Find("Mezmerize");
if (findIndex != -1)
    Console.WriteLine(dv[findIndex]["artist_id"]);
```

Метод `DataView.Find()` осуществляет поиск по столбцам, указанным в свойстве `Sort`. У многих строк могут быть одинаковые значения полей, используемых для сортировки данных. Например, при сортировке дисков по полю `release_year` это поле может иметь значение `"2005"` для нескольких строк. Тем не менее, найти посредством метода `Find()` *все* диски, выпущенные в 2005 году нельзя, поскольку он возвращает только целочисленное значение. К счастью, класс `DataView` предоставляет метод `FindRows()`. Его вызывают так же, как и метод `Find()`, но метод `FindRows()` возвращает массив объектов `DataRowView`, содержащих строки, которые удовлетворяют критериям поиска.

```
DataTable dt = CD_Rent.Tables["Disks"];
DataView dv = new DataView(dt);
dv.RowFilter = "release_year = 2005";

dv.Sort = "release_year";
DataRowView[] res = dv.FindRows("2005");
if (res.Length != 0)
    Console.WriteLine("Find {0} rows", res.Length);
```

Строка данных модифицируется с помощью объекта `DataRowView` аналогично изменению содержимого объекта `DataRow`. Объект `DataRowView`, как и `DataRow`, предоставляет методы `BeginEdit()`, `EndEdit()`, `CancelEdit()` и `Delete()`. Создание новой строки данных при помощи объекта `DataRowView` несколько отличается от создания нового объекта `DataRow`. У класса `DataView` есть метод `AddNew()`, возвращающий новый объект `DataRowView`. В действительности же новая строка добавляется в базовый объект `DataTable` только при вызове метода `EndEdit()` объекта `DataRowView`. Ниже показано, как средствами объекта `DataRowView` создать, изменить и удалить строку данных:

```
// Создание новой строки с использованием DataView
DataRowView drv = dv.AddNew();
drv[0] = 10;
drv[1] = "Hipnotize";
drv.EndEdit();

// Получение и редактирование строки
drv = dv[1];
drv.BeginEdit();
drv[1] = "H";
drv.EndEdit();

// Получение и удаление строки
drv = dv[0];
drv.Delete();
```

В таблице 35 приведено краткое описание основных свойств и методов класса DataView.

Таблица 35

Свойства и методы класса DataView

Имя свойства или метода	Описание
AddNew()	Создает новый объект DataRowView
AllowDelete AllowEdit AllowNew	Булевы свойства; указывают, допустимо ли удаление, изменение или добавление записей в объект DataView
ApplyDefaultSort	Если задать свойству значение true, содержимое DataView сортируется по первичному ключу таблицы, связанной с DataView. Кроме того, если изменить значение свойства на true, свойству Sort будут заданы столбцы, составляющие первичный ключ связанной таблицы
BeginInit()	Временно кэширует изменения содержимого DataView
CopyTo()	Позволяет копировать объекты DataRowView, доступные через DataView, в массив
Count	Возвращает число записей в DataView (доступно только для чтения)
Delete()	Метод принимает порядковый номер строки в объекте DataView и удаляет эту строку из базового объекта DataTable
EndInit()	Подтверждает внесение кэшированных изменений в DataView
Find()	Выполняет в DataView поиск отдельной строки данных
FindRows()	Выполняет в DataView поиск нескольких строк данных
GetEnumerator()	Возвращает экземпляр объекта IEnumerator для просмотра строк DataView
Item	Индексатор, возвращает объекты DataRowView, доступные через DataView
RowFilter	Свойство аналогично разделу WHERE SQL-запроса. Через DataView доступны только строки, удовлетворяющие заданному в свойстве критерию. Значение свойства по умолчанию – пустая строка
RowStateFilter	Указывает, какие строки доступны через объект DataView, а также версию этих строк
Sort	Свойство определяет порядок сортировки данных, доступных через DataView, и функционирует аналогично разделу ORDER BY SQL-запроса
Table	Таблица, с которой связан объект DataView

4.16. СИНХРОНИЗАЦИЯ НАБОРА ДАННЫХ И БАЗЫ

Пусть в набор данных заносится информация из БД при помощи адаптера:

```
SqlDataAdapter da = new SqlDataAdapter(. . .);
DataSet CD_Rent = new DataSet("CD_Rent");
da.Fill(CD_Rent, "Artists");
```

Для переноса изменений из набора в базу используется метод адаптера Update(). Данный метод обновляет в базе одну таблицу набора (**всегда!**), кото-

рая, как правило, задается через параметр метода. Однако попытка выполнения следующего кода вызовет исключительную ситуацию:

```
CD_Rent.Tables["Artists"].Rows[3]["name"] = "Alex";
da.Update(CD_Rent, "Artists");

System.InvalidOperationException: Update requires a valid
UpdateCommand when passed DataRow collection with modified rows.
```

Дело в том, что при создании адаптера формируется только `SelectCommand` – команда для выборки данных. Остальные свойства-команды адаптера не инициализированы.

Программист может настроить необходимые команды вручную. Вначале рассмотрим SQL-синтаксис возможных команд для обновления информации в нашем примере:

```
INSERT INTO Artists(id ,name) VALUES (@p1, @p2)

DELETE FROM Artists WHERE (id = @p1) AND (name = @p2)

UPDATE Artists SET id = @p1, name= @p2
               WHERE (id= @p3) AND (name= @p4)
```

В принципе, этот текст можно скопировать в свойство `CommandText` команд, которые будут созданы. Отдельного пояснения требует настройка параметров. Параметр, кроме установки таких свойств как имя и тип, должен быть связан со столбцом таблицы из набора данных, а в случае с командой `UPDATE` – еще и с определенной версией информации в столбце. Для этого используются свойства параметра `SourceColumn` и `SourceVersion`. Приведем полный текст создания и настройки команд:

```
// Создали соединение, которое будут использовать наши команды
SqlConnection con = new SqlConnection(. . .);

// Создали три объекта-команды
SqlCommand ins_cmd = con.CreateCommand();
SqlCommand del_cmd = con.CreateCommand();
SqlCommand upd_cmd = con.CreateCommand();

// Настраиваем текст команд
ins_cmd.CommandText = "INSERT INTO Artists(id,name) VALUES (@p1,@p2)";
del_cmd.CommandText = "DELETE FROM Artists" +
                      "WHERE (id=@p1) AND (name=@p2)";
upd_cmd.CommandText = "UPDATE Artists SET id=@p1, name= @p2" +
                      "WHERE (id= @p3) AND (name= @p4)";

// Займемся параметрами
// Создадим два параметра и поместим их в коллекцию
ins_cmd.Parameters.Add("@p1", DbType.Int32);
ins_cmd.Parameters.Add("@p2", DbType.String);

// Дополнительная настройка – укажем столбец, из которого
// берется значение параметра
```

```
ins_cmd.Parameters[0].SourceColumn = "id";
ins_cmd.Parameters[1].SourceColumn = "name";

// В случае с командой удаления – аналогичные действия
del_cmd.Parameters.Add("@p1", DbType.Int32);
del_cmd.Parameters.Add("@p2", DbType.String);
del_cmd.Parameters[0].SourceColumn = "id";
del_cmd.Parameters[1].SourceColumn = "name";

// Для команды обновления число параметров в два раза больше
upd_cmd.Parameters.Add("@p1", DbType.Int32);
upd_cmd.Parameters.Add("@p2", DbType.String);
upd_cmd.Parameters.Add("@p3", DbType.Int32);
upd_cmd.Parameters.Add("@p4", DbType.String);
upd_cmd.Parameters[0].SourceColumn = "id";
upd_cmd.Parameters[1].SourceColumn = "name";
upd_cmd.Parameters[2].SourceColumn = "id";
upd_cmd.Parameters[3].SourceColumn = "name";

// Требуется настройка – указать версию поля таблицы
upd_cmd.Parameters[2].SourceVersion = DataRowVersion.Original;
upd_cmd.Parameters[3].SourceVersion = DataRowVersion.Original;

// Помещаем наши команды в адаптер
da.InsertCommand = ins_cmd;
da.DeleteCommand = del_cmd;
da.UpdateCommand = upd_cmd;
```

После того, как в адаптере определены все команды, можно свободно изменять данные в рассоединенном наборе, а затем обновить их в базе вызовом `Update()`:

```
DataTable dt = CD_Rent.Tables["Artists"];
DataRow row = dt.NewRow();
row["id"] = 100;
row["name"] = "Uma Thurman";
dt.Rows.Add(row);
row = dt.Rows[1];
row["name"] = "Alex";
da.Update(CD_Rent, "Artists");
```

Как показывает пример, ручное создание команд для адаптера даже в случае простого набора данных выглядит громоздким (хотя это очень гибкое решение). ADO.NET предоставляет класс для автоматической генерации команд адаптера. Это класс `CommandBuilder` (класс зависит от поставщика данных, поэтому приведено его «обобщенное» имя).

Работа с классом `CommandBuilder` происходит следующим образом. Создается объект класса и связывается с определенным адаптером данных, у которого уже задана команда `SelectCommand`. После установки подобной связи `CommandBuilder` отслеживает событие обновления строки данных, которое происходит при вызове метода `Update()`, и генерирует и выполняет необходимые SQL-команды на основе текста команды `SELECT`.

Приведем пример кода, использующего CommandBuilder. Вот как могла бы выглядеть «генерация» команд для адаптера, с которым мы работали:

```
// Создаем объект CommandBuilder и связываем его с адаптером
SqlCommandBuilder cb = new SqlCommandBuilder(da);
// И, собственно, все! Можем работать с методом Update()
. . .
da.Update(CD_Rent, "Artists");
```

Конечно, класс CommandBuilder не «всемогущ». Он генерирует правильные команды обновления, если выполняются **все** следующие условия:

- запрос возвращает данные только из одной таблицы;
- на таблице в базе определен первичный ключ;
- первичный ключ есть в результатах вашего запроса.

Кроме этого, объект CommandBuilder не предоставляет максимальной производительности периода времени выполнения. Вы можете написать и добавить в код собственную логику обновления за время, меньшее, чем объекту CommandBuilder потребуется, чтобы выбрать и обработать необходимые для создания аналогичного кода метаданные таблицы из БД. CommandBuilder не позволяет управлять генерацией логики. Нельзя указать нужный способ оптимистического управления параллелизмом. Нельзя передавать обновления средствами хранимых процедур.

В таблице 36 приведены свойства и методы класса SqlCommandBuilder.

Таблица 36

Свойства и методы класса SqlCommandBuilder

Имя свойства или метода	Описание
DataAdapter	Свойство позволяет просмотреть или изменить объект DataAdapter, сопоставленный с объектом CommandBuilder. Значение этого свойства можно задать в конструкторе объекта CommandBuilder
DeriveParameters()	Статический метод. Получает в качестве параметра объект-команду для вызова хранимой процедуры. На основании информации из БД, заполняет коллекцию Parameters команды-параметра
GetDeleteCommand()	Возвращает объект Command с логикой для свойства DeleteCommand объекта DataAdapter
GetInsertCommand()	Возвращает объект Command с логикой для свойства InsertCommand объекта DataAdapter
GetUpdateCommand()	Возвращает объект Command с логикой для свойства UpdateCommand объекта DataAdapter
QuotePrefix	Содержит префикс, используемый CommandBuilder для имен таблиц и столбцов в генерируемых им запросах
QuoteSuffix	Содержит суффикс, используемый CommandBuilder для имен таблиц и столбцов в генерируемых им запросах
RefreshScbema()	Указывает объекту CommandBuilder создать логику обновления заново

5. ASP.NET

5.1. АРХИТЕКТУРА И ОБЩИЕ КОНЦЕПЦИИ ASP.NET

В данном параграфе рассматриваются базовые принципы функционирования и архитектуры технологии ASP.NET. Вначале рассмотрим схему работы в сети Internet, которую можно назвать «классической», так как эта схема является исторически первой. Основными элементами классической схемы являются *браузер* и *веб-сервер*. При взаимодействии браузер и веб-сервер проходят следующие этапы:

1. Браузер формирует *запрос* к серверу, используя *протокол HTTP*. Как правило, браузер запрашивает *HTML-страницу*, то есть текстовый файл, содержащий HTML-код.
2. Сервер анализирует запрос браузера и извлекает из локального хранилища требуемый файл.
3. Сервер формирует *HTTP-ответ*, включающий требуемую информацию, и отсылает его браузеру по протоколу HTTP.
4. Браузер выполняет отображение страницы.

Классическая схема проста, но обладает существенным недостатком – страницы статичны, и их содержимое не может меняться динамически в зависимости от запросов клиента. В настоящее время подобный подход не соответствует большинству информационных услуг, предоставляемых с помощью сети Internet[39]. Все большее распространение получают технологии, при использовании которых страницы (целиком или частично) генерируются на сервере *непосредственно* перед отправкой клиенту. Эти технологии обычно содержат в своем названии словосочетание *Server Pages* – «серверные страницы» (ASP, ASP.NET). Работают технологии «серверных страниц» по схожим принципам:

- Для представления информации на сайте используются не страницы с HTML-кодом, а серверные страницы специального синтаксиса (который часто является HTML-подобным).
- При запросе серверной страницы веб-сервер запускает отдельный служебный процесс, которому перенаправляется запрос.
- В служебном процессе страница анализируется, по ней генерируется некий объект, соответствующий странице.
- Служебный процесс выполняет методы сгенерированного объекта. Как правило, объект имеет специальный метод, генерирующий выходной поток страницы в виде HTML-кода.
- Сервер перехватывает выходной поток страницы, формирует HTTP-ответ и отсылает его браузеру.

[39] Классическим примером является типичный Интернет-магазин. Если пользователь на странице определяет некое условие фильтрации, то браузер должен отобразить список товаров, имеющихся в данный момент в магазине и удовлетворяющих фильтру. Естественно, подобный процесс подразумевает динамическую генерацию страницы с результатами на стороне сервера.

• Браузер выполняет отображение страницы.

Рассмотрим особенности технологии серверных страниц, применительно к ASP.NET. Служебный процесс ASP.NET основан на управляемом коде и выполняется в Common Language Runtime. Серверной странице в ASP.NET соответствует некий .NET-класс. Это обеспечивает возможности межъязыкового взаимодействия, использования готовых библиотек кода и т. п. Серверная страница может содержать «вкрапления» серверного кода, написанные на нескольких языках – C#, VB.NET, J#.

ASP.NET представляет и поддерживает концепцию *фонового кода* (*Code Behind*). Эта концепция позволяет разделить логику и визуальное представление. Согласно данной концепции класс страницы наследуется от некоторого базового класса, в котором размещаются методы, образующие логику выполнения страницы. Сама страница содержит только описание визуальной части (какие визуальные компоненты и где размещены, как настроены их свойства).

Технология ASP.NET также пытается перенести принципы, используемые при написании приложений Windows Forms, на web-программирование. Речь идет о программировании, основанном на обработке событий. Отдельный компонент страницы ASP.NET, как правило, обладает набором некоторых событий. Хорошим примером является кнопка, у которой есть событие On_Click. Для того чтобы закодировать логику страницы, программист пишет обработчики соответствующий событий. Когда событие происходит, информация о нем пересылается от клиента на сервер, где срабатывает обработчик события. Затем страница (вся или только ее часть) вновь пересылается клиенту.

Чтобы страница сохраняла свое состояние между отдельными циклами приема-передачи, это состояние фиксируется в специальном скрытом поле перед отправкой на сервер, а сервер восстанавливает состояние страницы перед отправкой клиенту. Данный технологический прием называется в ASP.NET *поддержкой View State* (отображаемого состояния).

5.2. ПРИМЕР ASPX-СТРАНИЦЫ. СТРУКТУРА СТРАНИЦЫ

Любая серверная страница ASP.NET представляет собой текстовый файл с расширением `.aspx`. Далее такие страницы будут для краткости называться aspx-страницами. Рассмотрим пример простой aspx-страницы, которую разместим в файле `test.aspx`:

```
<%@ Page Language="C#" %>

<script runat="server">
    string[] Data = {"Hello ", "world ", "from ", "ASP.NET"};
    string getData(int i) {
        return Data[i];
    }
    void Button1_Click(object sender, EventArgs e) {
        Label1.Text = TextBox1.Text;
    }
</script>
```

```
<html>
<body>
  <% for(int i=0; i < 4; i++) Response.Write(getData(i)); %>
  <form runat="server">
    <p>
    <asp:Label ID="Label1" runat="server">No Name</asp:Label>
    </p>
    <p>
    <asp:TextBox ID="TextBox1" runat="server" />
    </p>
    <p>
    <asp:Button ID="Button1" OnClick="Button1_Click"
                runat="server" Text="Send" />
    </p>
  </form>
</body>
</html>
```

Для просмотра данной страницы требуется:

1. На компьютере-сервере должен быть установлен и запущен веб-сервер. Можно использовать *сервер IIS* или, например, свободно распространяемый фирмой Microsoft *сервер Cassini*.

2. На сервере создается виртуальный каталог, в соответствующий физический каталог помещается файл `test.aspx`. В нашем примере будет использоваться физический каталог `C:\Test` и виртуальный каталог `test`.

3. Пользователь набирает в браузере следующий адрес:

```
http://localhost/test/test.aspx
```

Вид страницы в браузере показан на рисунке 13.

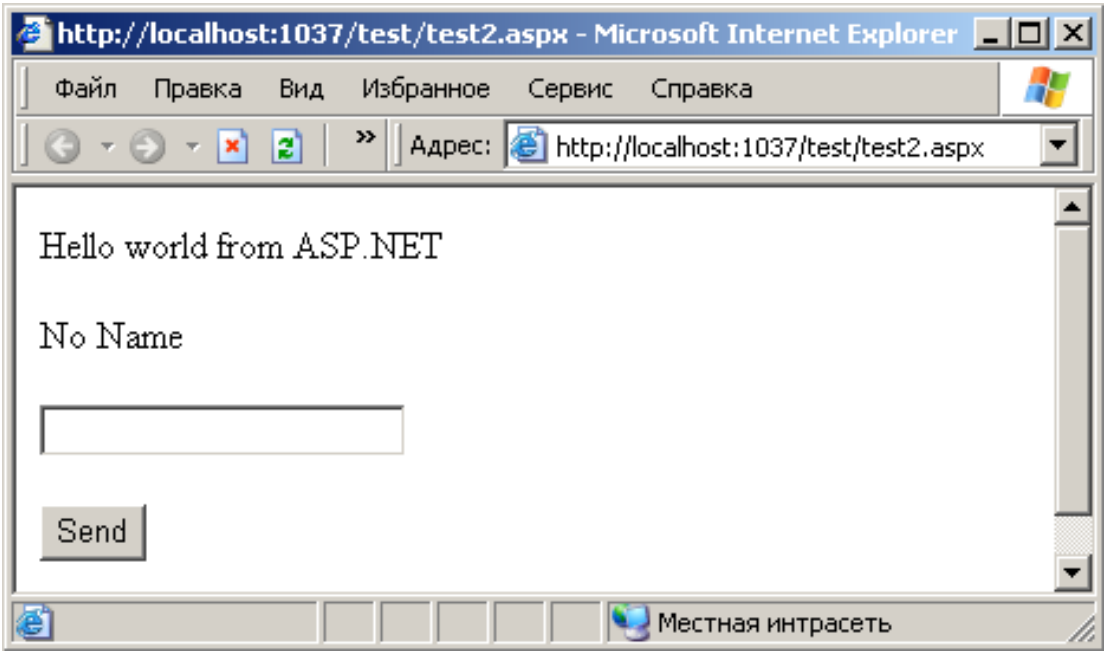

Рис. 13. Страница, отображаемая в браузере

HTML-код отображаемой страницы выглядит следующим образом:

```
<html>
<body>
    Hello world from ASP.NET
    <form name="ctl00" method="post" action="test.aspx"
          id="ctl00">
        <div>
            <input type="hidden" name="__VIEWSTATE"
                   id="__VIEWSTATE"
                   value="/wEPDwULLTEz4aswGyDFp8uH0tpf" />
        </div>
        <p>
            <span id="Label1">No Name</span>
        </p>
        <p>
            <input name="TextBox1" type="text" id="TextBox1"/>
        </p>
        <p>
            <input type="submit" name="Button1" value="Send"
                   id="Button1" />
        </p>
        <div>
            <input type="hidden" name="__EVENTVALIDATION"
                   id="__EVENTVALIDATION"
                   value="/wE54rGBu07o0F5j+sOYUEyYF3Dz5/uj" />
        </div>
    </form>
</body>
</html>
```

Обратите внимание на скрытое поле формы `__VIEWSTATE` (поддержка отображаемого состояния) и на поле `__EVENTVALIDATION` (указатель на источник события).

Рассмотрим подробнее структуру страницы. В любой странице можно выделить несколько логических секций:

1. HTML-код. HTML-код не обрабатывается процессом ASP.NET специальным образом, а сразу пересылается клиенту. Выводом HTML-кода занимается специальный внутренний метод `__Render__control()` того класса, который соответствует странице. В нашем случае фрагмент страницы

```
<html>
<body>
```

будет выведен в методе `__Render__control()` так:

```
private void __Render__control(. . .) {
    Response.Write("<html>\r\n<body>\r\n");
    . . .
}
```

2. Директивы страницы. Директивы используются для установки отдельных параметров страницы, таких как язык программирования для кода страницы или подключение пространства имен. Директива начинается с символа `@`, за которым следует имя директивы и набор пар «атрибут = значение». На-

пример, следующая директива указывает на использование C# в качестве языка программирования на странице:

```
<%@ Page Language="C#" %>
```

Директивы могут размещаться в любом месте страницы, но вне HTML-элементов. Как правило, директивы помещают в начале страницы.

3. Блоки серверного кода. Это блоки, обрамленные тэгом `<script>` с обязательным атрибутом `runat="server"`:

```
<script runat="server">
    string[] Data = {"Hello ", "world ", "from ", "ASP.NET"};
    string getData(int i) {
        return Data[i];
    }
    void Button1_Click(object sender, EventArgs e) {
        Label1.Text = TextBox1.Text;
    }
</script>
```

Блоки серверного кода транслируются в члены класса, соответствующего странице. В приведенном примере класс будет содержать поле-массив `Data`, методы `getData()` и `Button1_Click()`.

Отметим, что наличие блоков серверного кода на странице противоречит концепции Code Behinde. Согласно данной концепции код, связанный со страницей, должен быть помещен в отдельный класс, от которого наследуется класс страницы.

4. Блоки рендерного кода. Блоки рендерного кода используются для генерации потока вывода. В нашем примере блоком рендерного кода является фрагмент

```
<% for (int i = 0; i < 4; i++) Response.Write(getData(i)); %>
```

При обработке на сервере блоки рендерного кода помещаются непосредственно в метод, «заведующий» выводом HTML-кода. Таким образом, метод `__Render__control()` будет содержать следующий код:

```
private void __Render__control(. . .) {
    Response.Write("<html>\r\n<body>\r\n");
    for (int i = 0; i < 4; i++) Response.Write(getData(i));
    . . .
}
```

Если блок рендерного кода записывается в форме `<%=` *выражение* `%>`, то в метод вставляется вывод вычисленного выражения.

5. Серверные элементы управления. В нашем примере страница содержит три серверных элемента управления: текстовую метку, поле ввода, кнопку:

```
<asp:Label id="Label1" runat="server">No Name</asp:Label>
<asp:TextBox id="TextBox1" runat="server"></asp:TextBox>
<asp:Button id="Button1" onclick="Button1_Click"
            runat="server" Text="Send"></asp:Button>
```

Серверные элементы управления описываются с помощью специальных тегов с обязательным атрибутом `runat="server"`. Они соответствуют полям в классе страницы. Так, наша страница будет содержать поле с именем `Label1`, тип которого – `System.Web.UI.WebControls.Label`. Серверный элемент управления обладает набором свойств, установка которых возможна на странице как задание соответствующий атрибутов. Большинство серверных элементов должны быть размещены в пределах *серверной формы* (`<form runat="server">`).

Кроме упомянутых выше элементов, страница ASP.NET может содержать также комментарии, блоки привязки данных (будут подробно рассмотрены позднее), клиентские скрипты.

5.3. ДИРЕКТИВЫ СТРАНИЦЫ

Директивы страницы используются для настройки страницы. Как правило, при помощи директив выполняются такие операции, которым в «традиционном» программировании соответствуют настройка опций компилятора, подключение сборок и пространств имен, установка свойств определенных классов и т. п. Рассмотрим возможные директивы страницы.

`@Page`

При помощи директивы `@Page` устанавливаются самые важные настроечные параметры страницы. Директива имеет множество атрибуты, которые перечислены в таблице 37 (выделены значения атрибутов по умолчанию):

Таблица 37

Атрибуты директивы `@Page`

Имя атрибута	Возможные значения	Описание
`AspCompat`	`True` **`False`**	Устанавливает выполнение страницы в однопоточном апартменте. Позволяет обращаться к COM-компонентам, разработанным в VB (которые могли быть только STA-компонентами)
`AutoEventWireup`	**`True`** `False`	Указывает, выполняется ли автоматическое связывание событий страницы с методами, имеющими специфичные имена (например, `Page_Load`)
`Buffer`	**`True`** `False`	Разрешает буферизацию ответа. При включенной буферизации выходной поток страницы сначала записывается в память на сервере, а отсылается клиенту полностью сформированным
`ClassName`	`Имя класса`	Имя для класса страницы. По умолчанию имя файла страницы, в котором точка между именем и расширением `aspx` заменена символом подчеркивания
`ClientTarget`	`Имя User Agent`	Указывает браузер, на который ориентирована страница.
`CodePage`	`Имя кодовой`	Имя кодовой страницы, в кодировке которой

	страницы	пользователю посылается ответ
CompilerOptions	Допустимые опции компилятора	Опции компилятора; аналоги опций, передаваемых компилятору командной строки
ContentType	Допустимый MIME-тип	Указывает на тип содержимого ответа. Атрибут используется, когда возвращаемое значение отличается от text/html (например, ответ сервера – изображение)
Culture	Идентификатор культуры	Идентификатор культуры устанавливает язык и форматы записи дат, чисел и т. п. Например, идентификатор культуры для американского английского – en-US (используется по умолчанию)
Debug	True **False**	Атрибут указывает, компилировать ли страницу с отладочной информацией
Description	Любой текст	Описание страницы – игнорируется в ASP.NET
EnableSessionState	**True** False ReadOnly	Показывает, имеет ли страница доступ к состоянию сеанса (объекту Session). В случае ReadOnly состояние сеанса доступно только для чтения
EnableViewState	**True** False	Поддерживает ли страница сохранение состояния для серверных элементов управления
ErrorPage	Допустимый URL	Страница для переадресации при возникновении необработанной ошибки
Explicit	**True** False	Используется режим VB.NET Option Explicit
Inherits	Имя класса (возможно, с пространством имен)	Указывает пространство имен (опционно) и имя класса, от которого будет наследоваться класс страницы. Атрибут используется совместно с атрибутом Src
Language	Имя языка .NET	Язык программирования, который применяется для блоков серверного и рендерного кода. Обычно используются значения VB и C#. Атрибут Language может быть записан в серверном тэге <script>. ASP.NET поддерживает только один язык на странице
LCID	Допустимый локальный идентификатор	Локальный идентификатор страницы, если он отличается от локального идентификатора web-сервера. Например, при LCID="1041" функция FormatCurrency возвращает значения, форматированные как японские йены (¥).
ResponseEncoding	Имя системы кодировки символов	Формат кодирования текста, отправляемого в ответ. По умолчанию – Unicode (UTF-8)
Src	Имя исходного файла	Путь и имя файла с исходным кодом класса, наследником которого является страница. Используется совместно с атрибутом Inherits
SmartNavigation	True **False**	Разрешает или запрещает *развитую навигацию*. Развитая навигация поддерживается в IE версии 5.x и выше. Это средство загружает страницу в скрытое поле IFrame, а затем визуализирует только изменившиеся части страницы
Strict	True **False**	Используется режим VB.NET Option Strict

Trace	True **False**	Указывает на необходимость выполнять трассировку страницы. Трассировка заключается в выводе в нижней части страницы различной дополнительной информации и может быть полезна при отладке
TraceMode	**SortByTime** SortByCategory	При включенной трассировке – порядок сортировки сообщений (по времени или по категории)
Transaction	**Disabled** NotSupported Supported Required RequiresNew	Задает установки транзакции для страницы
ValidateRequest	**True** False	Указывает, производится ли проверка запроса. Проверка заключается в контроле всех передаваемых данных на потенциально опасные значения, к которым отнесены код HTML и скриптов. Если опасные значения выявлены, генерируется HttpRequestValidationException
WarningLevel	0, 1, 2, 3, 4	Указывает уровень предупреждений копилятора, при котором компиляция страницы прекращается

@Control

Данная директива обладает функциональностью, схожей с директивой @Page, но используется для пользовательских элементов ASP.NET. Атрибуты, доступные в директиве @Control, образуют подмножество атрибутов директивы @Page: AutoEventWireup, ClassName, CompilerOptions, Debug, Description, EnableViewState, Explicit, Inherits, Language, Strict, Src, WarningLevel.

@Import

Директива используется для импорта пространств имен для страницы. Это делает доступным на странице все классы и интерфейсы из импортированного пространства. Пример использования директивы:

```
<%@Import Namespace="BSUIR" %>
```

Одна директива @Import позволяет импортировать одно пространство имен, поэтому для импортирования нескольких пространств надо соответствующее количество директив. Платформа .NET при работе с ASP.NET автоматически импортирует следующие пространства имен:

```
System                          System.Web
System.Collections              System.Web.Caching
System.Collections.Specialized  System.Web.Security
System.Configuration            System.Web.SessionState
System.IO                       System.Web.UI
System.Text                     System.Web.UI.HtmlControls
System.Text.RegularExpressions  System.Web.UI.WebControls
```

@Implements

Директива @Implements позволяет реализовать на странице интерфейс. При реализации интерфейса вы сообщаете, что страница будет поддерживать определенные свойства, методы и события (аналогично реализации интерфейса в классе). В следующем примере указано, что страница реализует интерфейс IPostBackEventHandler:

```
<%@Implements Interface="System.Web.UI.IPostBackEventHandler" %>
```

@Register

Данная директива используется при добавлении на страницу индивидуального серверного элемента управления для сообщения компилятору информации об этом элементе. Существуют две формы директивы @Register:

```
<%@Register TagPrefix="tagprefix"
            TagName="tagname" Src="pathname" %>

<%@Register TagPrefix="tagprefix" Namespace="namespace"
                                  Assembly="assembly" %>
```

Первая форма используется, когда серверный элемент представлен в виде исходного текста, вторая – для ссылки на пространство имен скомпилированных серверных элементов. Атрибут TagPrefix задает префикс для пользовательских элементов на странице. Например, пусть задана следующая директива @Register:

```
<%@Register TagPrefix="Ecommerce" TagName="Header"
            Src="UserControls\Header.ascx" %>
```

Тогда перед каждым экземпляром элемента Header на странице указывается префикс Ecommerce:

```
<Ecommerce:Header id="Header" runat="server"/>
```

Атрибут TagName задает имя, которое будет употребляться для ссылки на пользовательский элемент на странице. Атрибут Src определяет имя файла с пользовательским элементом. В случае использования второй формы директивы @Register, атрибуты Namespace и Assembly определяют пространство имен и сборку, содержащую пользовательский элемент:

```
<%@Register TagPrefix="Wrox" Namespace="WroxControls"
                             Assembly="RatingMeter" %>
```

Следуя примеру, для указания на странице некоего элемента управления из сборки RatingMeter необходимо будет указать тэг <Wrox:*имя_элемента*>.

@Assembly

Эта директива используется для непосредственной ссылки на сборку, с тем чтобы содержащиеся в ней классы и интерфейсы стали доступны коду на странице. Можно передать имя скомпилированной сборки

```
<%@ Assembly Name="имя сборки" %>
```

или путь к исходному файлу, который будет компилироваться при компиляции страницы:

```
<%@ Assembly Src="путь" %>
```

Этот тэг обычно не требуется, так как любая сборка, находящаяся в каталоге bin приложения ASP.NET, автоматически присоединяется к странице во время компиляции. Директива используется при подключении сборок из GAC или сборок из других каталогов

@OutputCache

Директива применяется для контроля кэширования страницы на сервере. Подробнее работа с директивой @OutputCache будет рассмотрена ниже.

@Reference

Данная директива позволяет динамически скомпилировать пользовательский элемент управления и поместить его в коллекцию Controls страницы или серверного элемента управления. Директива используется в соединении с методом Page.LoadControl().

5.4. КЛАСС SYSTEM.WEB.UI.PAGE. СОБЫТИЯ СТРАНИЦЫ

Любая aspx-страница компилируется в объект определенного класса. Этот класс является прямым или косвенным (если используется концепция Code Behind) наследником класса System.Web.UI.Page.

Рассмотрим класс System.Web.UI.Page подробнее. Свойства данного класса описаны в следующей таблице 38.

Таблица 38

Свойства класса System.Web.UI.Page

Имя свойства	Описание
Application	Объект класса HttpApplicationState, описывающий web-приложение, к которому относится страница. Для отдельного web-приложения существует ровно один объект Application, который используется всеми клиентами
Cache	Объект класса Cache, ссылка на кэш страницы. По существу, класс Cache – это словарь, чье состояние фиксируется с помощью скрытых полей формы или других средств, чтобы данные могли сохраняться от одного запроса страницы к другому
ClientTarget	Свойство позволяет переопределить встроенное в ASP.NET распознавание браузера и задать тот конкретный браузер (в виде строки-описания), которому предназначена страница. Любые элементы управления, которые зависят от браузера, будут использовать запись в свойстве ClientTarget
EnableViewState	Булево значение, управляющее поддержкой View State для страницы. Значение свойства влияет на все элементы управления страницы. По умолчанию – значение true
ErrorPage	URL страницы, которая показывается, если при компиляции или выполнении страницы произошло необработанное исключение или ошибка
IsPostBack	Это булево свойство устанавливается в true, если страница выпол-

	няется в цикле обмена с клиентом. Ложное значение свойства указывает на то, что страница отображается первый раз, и для серверных элементов управления не сохранено View State. В этом случае нужно установить состояние элементов вручную – обычно в обработчике события Page_Load
IsValid	Булево свойство; устанавливается в true, если проверочные элементы управления на странице сообщают, что условия проверки выполнены с положительным результатом. Если хотя бы одно условие проверки не выполнено, значение свойства – false
Request	Ссылка на объект HttpRequest, обеспечивающий доступ к информации о HTTP-запросе
Response	Ссылка на объект HttpResponse, обеспечивающий доступ к информации о HTTP-ответе
Server	Объект класса HttpServerUtility, описывающий параметры web-сервера
Session	Ссылка на объект класса HttpSessionState, хранящий данные текущей сессии пользователя в web-приложении
SmartNavigation	Булево свойство, разрешающее поддержку Smart Navigation
Trace	Объект класса TraceContext. Если на странице разрешена трассировка, то можно пользоваться данным свойством для записи особой информации в журнал трассировки
TraceEnabled	Булево свойство, разрешающее поддержку трассировки
User	Ссылка на объект, реализующий интерфейс IPrincipal и описывающий пользователя. Свойство используется при проведении аутентификации
Validators	Коллекция проверочных элементов, размещенных на странице

Легко заметить, что некоторым из свойств страницы (ClientTarget, EnableViewState, ErrorPage, SmartNavigation, TraceEnabled) соответствуют атрибуты директивы @Page.

Свойство Trace удобно при отладке. Тип свойства – класс TraceContext. При помощи перегруженных методов Write() и Warn() данного класса в отладочную информацию страницы можно записать свою информацию. Булевское свойство TraceContext.IsEnabled позволяет включить или отключить трассировку, а свойство TraceContext.TraceMode управляет порядком сортировки отладочной информации (по категориям или по времени).

Рассмотрим пример страницы test.aspx:

```
<%@ Page Language="C#" Trace="true" %>

<script runat="server">
    void Page_Load(object sender, EventArgs e) {
      Trace.Write("Usual Message");
      Trace.Warn("My Category", "Warn Message - draw in red");
    }
</script>

<html>
<body>
    Some Text
</body>
</html>
```

На странице при помощи директивы `@Page` включена трассировка, а в обработчике события загрузки страницы `Page_Load()` выводится два отладочных сообщения.

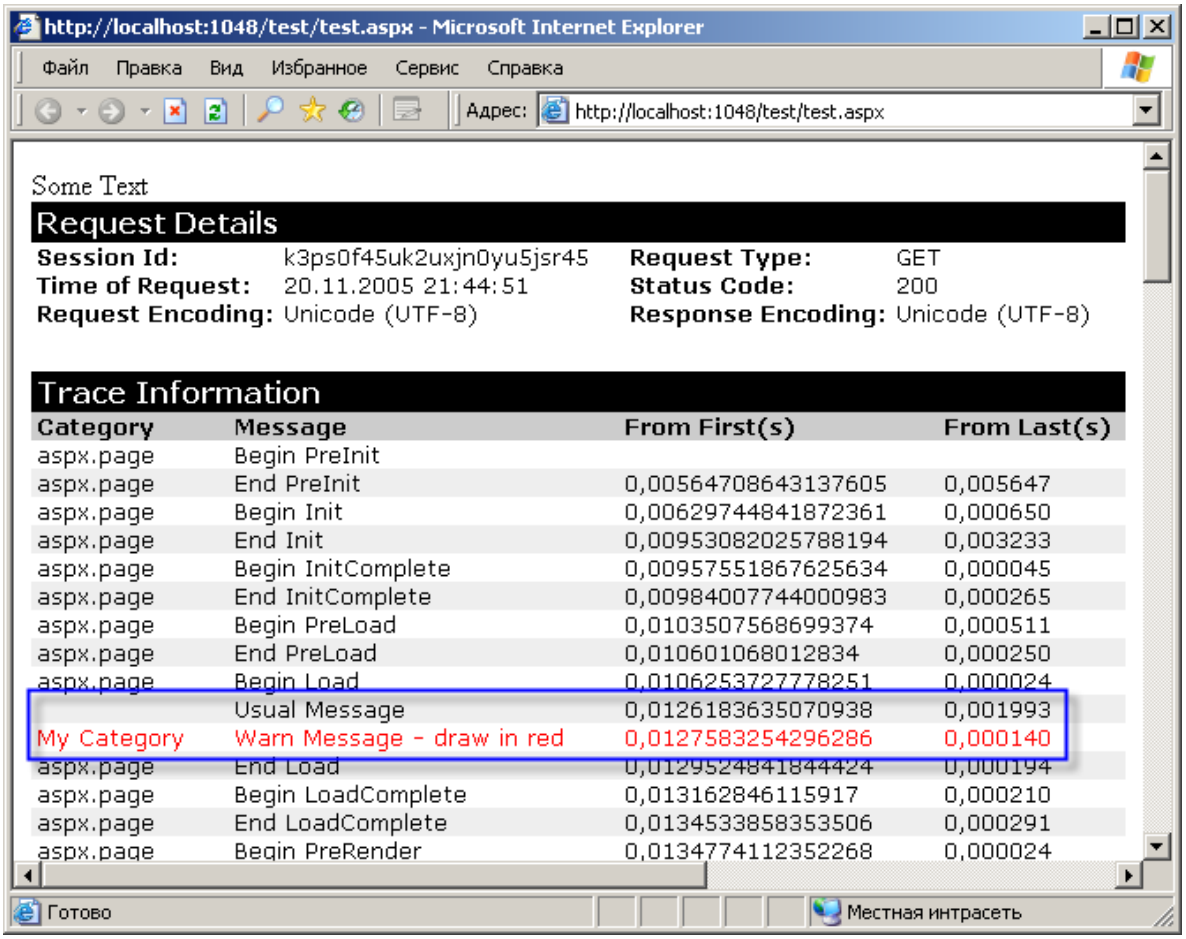

Рис. 14. Страница с отладочной информацией

Работу со свойствами `Application`, `Cache`, `IsValid`, `Request`, `Response`, `Session`, `User`, `Validators` рассмотрим более подробно позднее.

Основные методы класса `System.Web.UI.Page` перечислены в таблице 39.

Таблица 39

Методы класса `System.Web.UI.Page`

Имя метода	Описание
`DataBind()`	Выполняет *связывание данных* для всех элементов управления на странице
`FindControl()`	Метод позволяет найти некий элемент управления на странице
`LoadControl()`	Динамическая загрузка элемента управления из файла `.ascx`
`LoadTemplate()`	Метод выполняет динамическую загрузку шаблона
`MapPath()`	Метод строит физический путь для указанного элемента приложения
`ResolveUrl()`	Преобразует виртуальный URL в абсолютный URL
`Validate()`	Дает команду всем проверочным элементам на странице выполнить проверку

Использование методов MapPath() и ResolveUrl() демонстрирует следующая простая страница:

```
<%@ Page Language="C#" %>

<script runat="server">
    void Page_Load(object sender, EventArgs e) {
        Label1.Text = this.MapPath("test");
        Label1.Text += this.ResolveUrl("test2.aspx");
    }
</script>

<html>
<body>
    <asp:Label ID="Label1" runat="server" />
</body>
</html>
```

Вид страницы в браузере показан на рисунке 15.

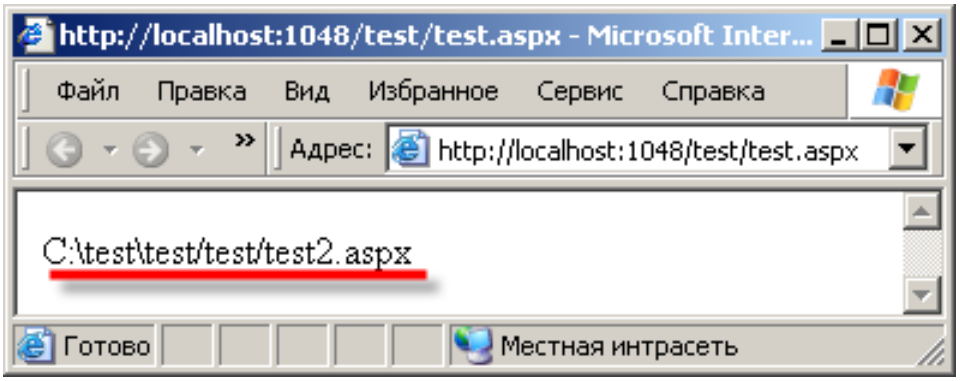

Рис. 15. Страница в браузере

Любая aspx-страница обладает определенным набором событий. Важно знать, что это за события и в какой последовательности они происходят. Это позволит разработчику корректно создавать собственные обработчики событий страницы.

Набор и последовательность событий страницы зависят от следующего фактора: показывается ли страница первый раз или страница отображается в цикле обмена с клиентом[40]. Если страница отображается в первый раз, то происходят такие события (в описанной последовательности):

1. `Init`. Событие происходит после того, как каждый управляющий элемент был создан и помещен во внутреннюю коллекцию элементов страницы.
2. `Load`. Установлены заданные в описании страницы свойства управляющих элементов. Как правило, в обработчике события `Load` помещается код, который выполняет дополнительную настройку свойств элементов – например, читает некоторые свойства из базы данных. Для того чтобы

[40] Если свойство страницы `IsPostBack = false`, то страница отображается в первый раз.

дополнительная настройка проходила только один раз, обычно проверяют значение свойства IsPostBack (if(!IsPostBack) ...).

3. **PreRender**. Каждый элемент управления готов к выводу.
4. **SaveVeiwState**. Информация о состоянии элементов управления сериализуется в закодированную строку и передается клиенту.
5. **Render**. Событие происходит перед выводом страницы клиенту.
6. **Dispose**. Страница и ее элементы управления освобождаются из памяти.

Если страница отображается не в первый раз, а в цикле обмена с клиентом, то дополнительно происходят следующие события:

1. Init.
2. **LoadViewState**. Информация о состоянии загружена в соответствующие свойства управляющих элементов.
3. Load.
4. **RaisePostDataChanged**. В управляющее элементы была помещена информация, которая передана методом POST протокола HTTP.
5. **RaisePostBackEvent**. Обрабатываются события, которые привели к отправке информации на сервер (такие, как нажатие кнопки).
6. PreRender.
7. SaveVeiwState.
8. Render.
9. Dispose.

Класс System.Web.UI.Page позволяет программисту задать обработчики следующих событий: Init, Load, Unload (вся информация страницы послана клиенту), PreRender, Error (на странице возникло необработанное исключение). При этом назначение обработчиков может быть выполнено двумя способами. В режиме автоматического связывания (директива @Page, атрибут AutoEventWireup="true") для назначения обработчика достаточно в блоке серверного кода записать метод со специальным именем в формате Page_*<имя события>*. Если режим автоматического связывания выключен, то методы обработки связываются с событиями в коде инициализации (конструкторе) с использованием соответствующих делегатов.

Рассмотрим пример страницы с обработчиками. Для простоты кода воспользуемся автоматическим связыванием. Разметим на странице поле ввода и кнопку и реализуем некоторые обработчики:

```
<%@ Page Language="C#" %>

<script runat="server">
    void Page_Init(object sender, EventArgs e) {
        TextBox1.Text = "Init...";
    }
    void Page_Load(object sender, EventArgs e) {
        if (!IsPostBack) Response.Write("First Load");
        else Response.Write("Second Load or more...");
    }
</script>
```

```
<html>
<body>
    <form id="Form1" runat="server">
        <asp:TextBox ID="TextBox1" runat="server" />
        <asp:Button ID="But1" runat="server" Text="Button" />
    </form>
</body>
</html>
```

Страница после первой загрузки показана на рисунке 16.

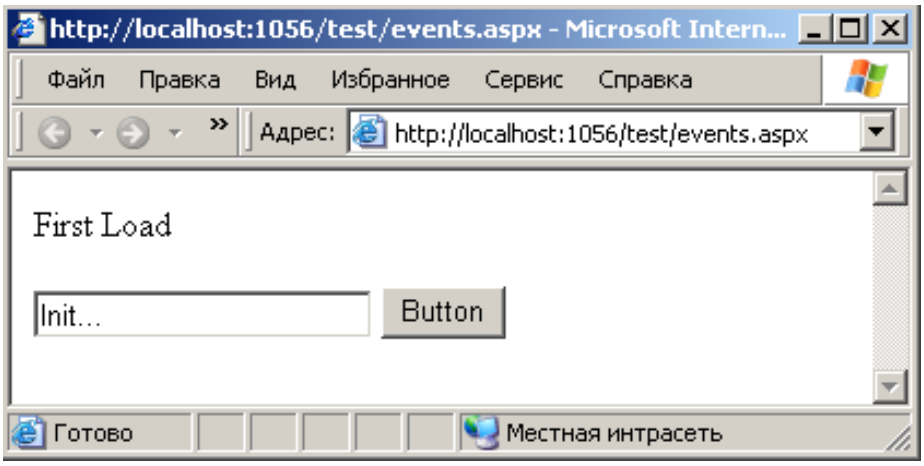

Рис. 16. Страница с обработчиками событий в браузере

5.5. СЕРВЕРНЫЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

Важным элементом технологии ASP.NET являются *серверные элементы управления* (server controls). Серверный элемент управления – это некий класс, объект которого агрегирован в страницу, то есть является полем класса-страницы. При выполнении страницы серверный элемент, как правило, транслируется в HTML-элемент управления.

Библиотека серверных элементов насчитывает порядка сотни классов. Условно все элементы можно разделить на следующие группы:

1. **Серверные HTML-элементы** – базирующиеся на сервере эквиваленты обычных HTML-элементов. Имеют те же атрибуты, что и обычные HTML-элементы.
2. **Элементы управления Web Controls** – набор элементов, равнозначный обычным HTML-элементам формы `<form>`. Имеют расширенный набор свойств и событий, облегчающий создание страницы.
3. **Проверочные элементы управления** – набор специальных элементов для проверки корректности значений, вводимых в другие элементы на странице. Выполняют проверку на стороне клиента, на стороне сервера или в обоих местах.
4. **Списковые элементы управления** – обеспечивают различные способы построения списков. Помимо этого, списковые элементы могут быть связаны с данными, хранящимися как в локальных структурах, так и в базах данных.

5. **Развитые элементы управления ASP.NET** – создают сложные, специфичные для задач выходные данные (например, календарь или меняющийся баннер).
6. **Мобильные элементы управления** – используются при построении приложений ASP.NET для мобильных платформ.

Для всех серверных элементов управления справедливы следующие замечания. Настройку таких элементов можно выполнять программно в методах страницы. Как правило, при работе с визуальными элементами используется IDE, облегчающее настройку. В частности, для элементов и страницы имеется окно свойств и событий. Важно понимать, что значения свойств элементов транслируются в описании страницы в соответствующие атрибуты. Например, следующий фрагмент страницы показывает, что у элемента `<asp:Button>` (кнопка) заданы свойства `ID` (идентификатор элемента, имя поля в классе страницы), `Text` (надпись на кнопке), `OnClick` (имя метода-обработчика события):

```
<asp:Button ID="B1" OnClick="B1_Click" runat="server" Text="Send" />
```

При задании серверного элемента на странице используется соответствующий тэг с обязательным атрибутом `runat="server"`.

В некоторых случаях элемент управления образует в описании страницы секцию, ограниченную открывающими и закрывающим тэгом, а свойство элемента задано как содержимое секции (literal content). Во фрагменте кода так задана надпись на странице:

```
<asp:Label id="Label1" runat="server">
    No Name
</asp:Label>
```

Рассмотрим набор серверных HTML-элементов. Они представлены классами из пространства имен `System.Web.UI.HtmlControls`. Базовым классом для всех серверных HTML-элементов является класс `HtmlControl`. Основные элементы данного класса перечислены в таблице 40.

Таблица 40

Элементы класса `System.Web.UI.HtmlControls`

Имя элемента	Описание
`Attributes`	Возвращает коллекцию пар имя-значение для всех атрибутов в файле `.aspx` для элемента управления. Может использоваться для чтения и установки нестандартных атрибутов
`Controls`	Объект типа `ControlCollection`. Содержит ссылки на элементы управления, являющиеся «дочерними»[41] для данного
`Disabled`	Булево значение; показывает, отключен ли элемент
`EnableViewState`	Булево значение, которое управляет поддержкой сохранения состояния в элементе и его «потомках». По умолчанию – сохранение состояния включено (`true`)
`ID`	Свойство задает строку-идентификатор элемента управления (имя поля в классе-странице)

[41] В данном контексте понятия «дочерний» и «родительский» следует понимать так: родительский элемент является контейнером для своих дочерних элементов.

Page	Ссылка на объект Page, содержащий элемент управления
Parent	Ссылка на родительский элемент в страничной иерархии
Style	Коллекция свойств CSS, которые будут добавлены к тэгу элемента при отображении. Позволяет настраивать внешний вид элемента
TagName	Строка, имя HTML-тега, который соответствует элементу
Visible	Булево значение; показывает, виден ли элемент на странице
DataBind()	Метод выполняет связывание данных для элемента управления и всех его потомков
FindControl()	Метод позволяет найти элемент управления в коллекции элементов-потомков данного
HasControls()	Метод возвращает true, если элемент управления имеет потомков
DataBinding	Данное событие происходит при связывании данных

Класс HtmlContainerControl – это второй базовый класс для серверных HTML-элементов. Он используется как предок теми элементами, которые должны иметь закрывающий тэг (то есть обязательно образуют секцию в описании страницы). Данный класс наследуется от класса HtmlControl и имеет два полезных свойства:

- InnerHtml – строка с HTML-содержимым и текстом, заключенным между открывающим и закрывающим тэгом элемента;
- InnerText – строка только с текстовым содержимым элемента.

Далее приведена таблица, содержащая краткое описание серверных HTML-элементов.

Таблица 41

Серверные HTML-элементы

Имя элемента	Описание	Специфичные свойства и события
HtmlAnchor	Ссылка на страницу или часть страницы	Href, Target, Title, Name, OnServerClick
HtmlImage	Изображение на странице с настраиваемыми параметрами	Align, Alt, Border, Height, Src, Width
HtmlForm	Обрамляющая форма для других элементов управления	Name, Enctype, Method, Target
HtmlButton	Кнопка-контейнер (не поддерживается браузерами Opera и Navigator)	CausesValidation, OnServerClick
HtmlInputButton	Кнопка на форме	Name, Type, Value, CausesValidation, OnServerClick
HtmlInputText	Текстовое поле	MaxLength, Name, Size, Type, Value, OnServerChange
HtmlInputCheckBox	Переключатель-флаг	Checked, Name, Type, Value, OnServerChange
HtmlInputRadioButton	Один из группы взаимно исключающих переключателей	Checked, Name, Type, Value, OnServerChange
HtmlInputImage	Изображение на странице, которое допускает щелчок (как кнопка Submit)	Align, Alt, Border, Name, Src, Type, Value, CausesValidation, OnServerClick
HtmlInputFile	Реализует возможность копирования файлов на сервер	Accept, MaxLength, Name, PostedFile, Size, Type,

		Value
HtmlInputHidden	Невидимое поле на странице (для передачи служебных значений)	Name, Type, Value, OnServerChange
HtmlTextArea	Текст, состоящий из нескольких строк	Cols, Name, Rows, Value, OnServerChange
HtmlSelect	Списковый элемент управления	Multiple, SelectedIndex, Size, Value, DataSource, DataTextField, DataValueField Items (*коллекция*), OnServerChange
HtmlTable	Таблица	Align, BgColor, Border, BorderColor, CellPadding, CellSpacing, Height, NoWrap, Width Rows (*коллекция*)
HtmlTableRow	Строка таблицы	Align, BgColor, Border, BorderColor, Height, VAlign Cells (*коллекция*)
HtmlTableCell	Ячейка таблицы	Align, BgColor, Border, BorderColor, ColSpan, Height, NoWrap, RowSpan, VAlign, Width

Обсудим некоторые свойства и события элементов. Если булево свойство CausesValidation установлено в true, то при щелчке на элементе управления выполняется проверка страницы (о проверочных элементах будет рассказано ниже). Элементы управления HtmlAnchor, HtmlButton, HtmlInputButton, HtmlInputImage генерируют событие OnServerClick, которое происходит при нажатии на элементе. Обработчик данного события выполняется на сервере и получает два параметра. Первый – ссылка на объект, породивший событие, откуда можно получить ID источника события. Второй – объект типа EventArgs с дополнительной информацией о событии. Элементы HtmlInputText, HtmlInputCheckBox, HtmlInputRadioButton, HtmlInputHidden и HtmlSelect генерируют событие OnServerChange. Событие порождается, когда страница передается на сервер, для каждого элемента, чье значение изменилось с момента отправки клиенту страницы.

Рассмотрим пример страницы с некоторыми серверными HTML-элементами. Страница демонстрирует настройку отдельных свойств элементов, а также наличие обработчиков событий.

```
<%@ Page Language="C#" %>
<script runat="server">
    void Page_Load(object sender, EventArgs e) {
        if (!IsPostBack) Radio1.Checked = true;
    }
    void Submit1_Click(object sender, EventArgs e) {
        DIV1.InnerHtml += "Name is " + Text1.Value + "<br />";
    }
    void Radio1_Change(object sender, EventArgs e) {
        if (Radio1.Checked)
            DIV1.InnerHtml += "Have an ID" + "<br />";
        else DIV1.InnerHtml += "Dont have an ID" + "<br />";
```

```
    }
</script>

<html>
<body>
  <form id="form1" runat="server">
    <div>
      Your name:
      <input id="Text1" runat="server" type="text" />
      <br /><br />
      Password:
      <input id="Pass1" runat="server" type="password" />
      <br /><br />
      Have an ID?
      <input id="Radio1" runat="server" type="radio"
                         onserverchange="Radio1_Change" />
      Yes
      <input id="Radio2" runat="server" type="radio"
                         onserverchange="Radio1_Change" />
      No
      <br /><br />
      <input id="Submit1" type="submit" value="Submit!"
             onserverclick="Submit1_Click" runat="server" />
      <br /><br />
    </div>
  </form>
</body>
</html>
```

Вид страницы в браузере после заполнения полей и нажатия кнопки Submit показан на рисунке 17.

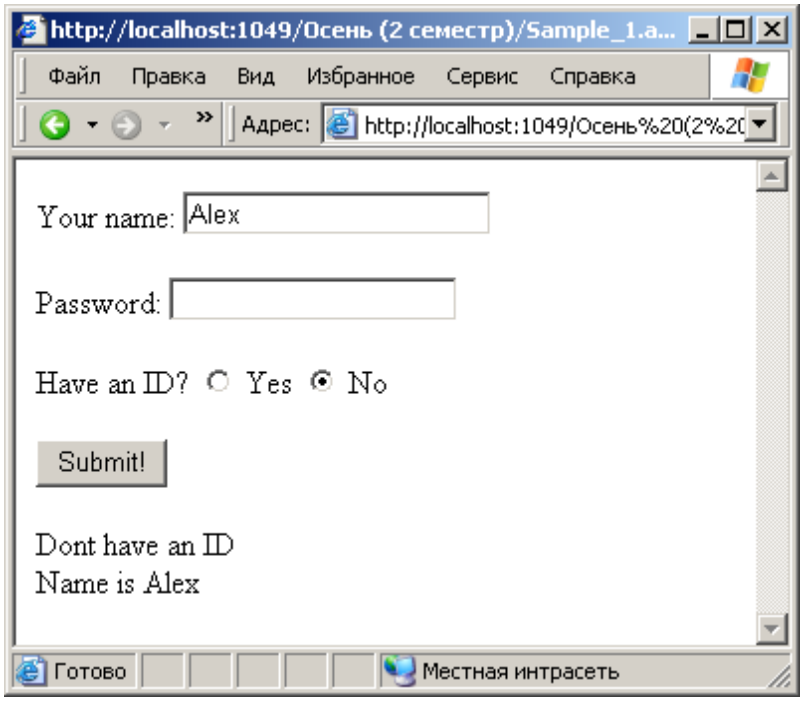

Рис. 17. Страница с серверными HTML-элементами в браузере

5.6. ЭЛЕМЕНТЫ УПРАВЛЕНИЯ WEB CONTROLS

Наряду с серверными HTML-элементами ASP.NET предоставляет альтернативный набор классов для создания web-страниц. Это классы иерархии Web Controls. Классы данной иерархии также соответствуют элементам управления, но предоставляют развитый, унифицированный набор свойств, методов и событий, упрощающий программирование.

Иерархия классов Web Control представлена на схеме 18.

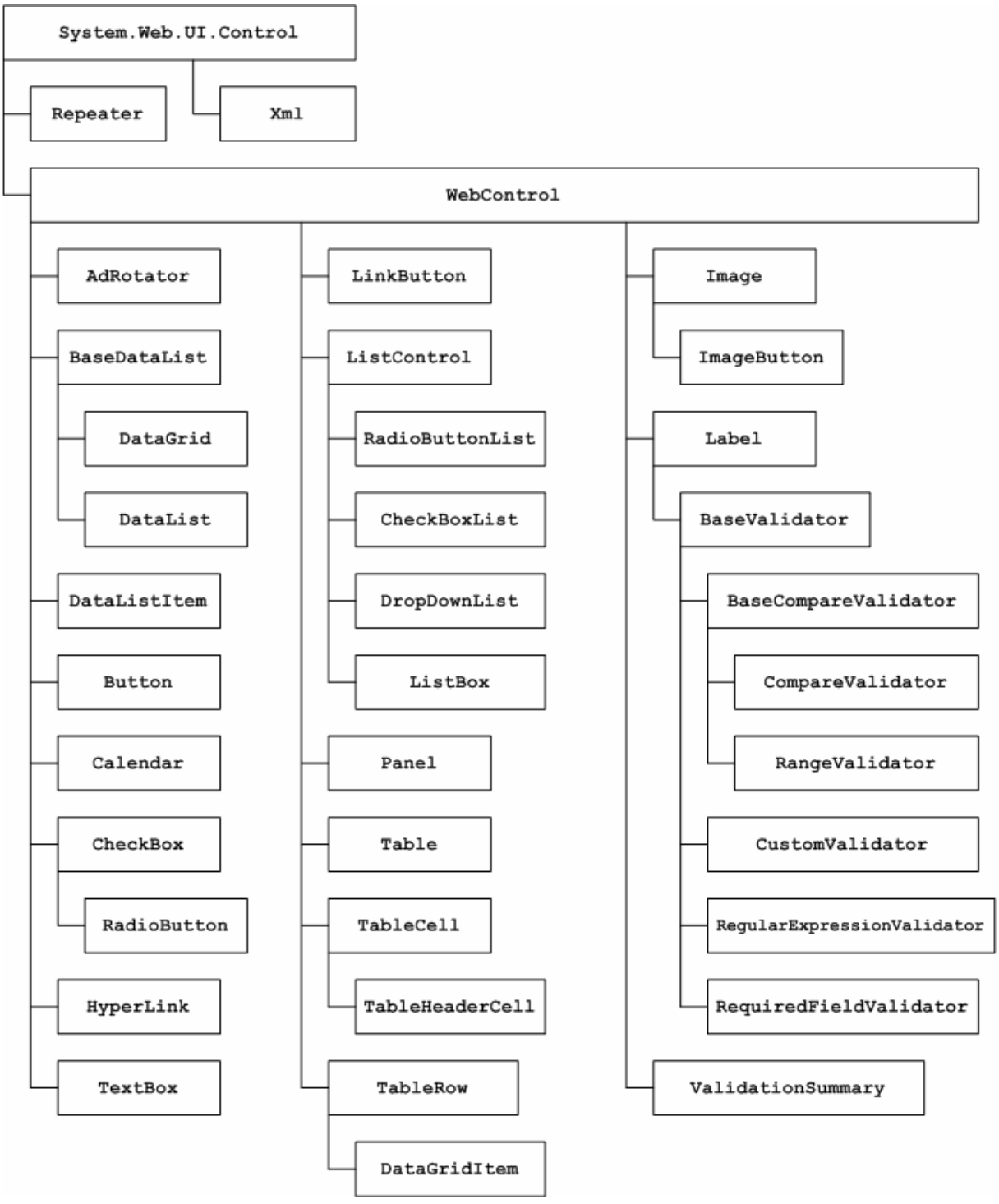

Рис. 18. Иерархия классов Web Control

Базовым классом для всех элементов управления Web Controls является класс `WebControl`. Основные свойства этого класса перечислены в таблице 42.

Таблица 42

Основные свойства класса `WebControl`

Имя свойства	Описание
`Attributes`	Возвращает коллекцию пар имя-значение для всех атрибутов в файле `.aspx` для элемента управления. Может использоваться для чтения и установки нестандартных атрибутов
`AccessKey`	Позволяет установить клавишу быстрого доступа для элемента управления
`BackColor`	Фоновый цвет элемента управления
`BorderColor`	Цвет границы элемента управления
`BorderStyle`	Стиль границы: сплошная, пунктирная, двойная и т. д.
`BorderWidth`	Ширина границы элемента управления
`Controls`	Объект типа `ControlCollection`. Содержит ссылки на элементы управления, являющиеся «дочерними» для данного
`Enabled`	Булево значение; показывает, активен ли элемент
`EnableViewState`	Булево значение, которое управляет поддержкой сохранения состояния в элементе и его «потомках». По умолчанию – сохранение состояния включено (`true`)
`Font`	Свойство для настройки параметров шрифта, которым отображается информация в элементе
`ForeColor`	Цвет «переднего плана» в элементе управления. Обычно это цвет текста
`Height`	Общая высота элемента управления
`ID`	Свойство задает строку-идентификатор элемента управления (имя поля в классе-странице)
`Page`	Ссылка на объект `Page`, содержащий элемент управления
`Parent`	Родительский элемент управления в страничной иерархии
`Style`	Коллекция свойств CSS, которые будут добавлены к тэгу элемента при отображении. Позволяет настраивать внешний вид элемента
`TabIndex`	Позиция элемента управления в порядке переноса фокуса ввода на странице
`ToolTip`	Текст подсказки, появляющийся при наведении мыши на элемент
`Visible`	Булево значение; показывает, виден ли элемент на странице
`Width`	Общая ширина элемента управления

Кроме перечисленных свойств класс `WebControl` предоставляет методы `DataBind()`, `FindControl()`, `HasControls()`, назначение которых аналогично соответствующим методам класса `HtmlControl`.

В данном параграфе будут рассмотрены те элементы управления Web Controls, которые можно назвать *базовыми*. Эти элементы управления, а также их специфические свойства и событие перечислены в таблице 43:

Таблица 43

Базовые элементы управления Web Controls

Элемент	Специфические свойства	Специфические события
`HyperLink`	`ImageUrl, NavigateUrl, Target, Text`	
`LinkButton`	`CommandArgument, CommandName, Text,`	`OnClick(),`

	CausesValidation	OnCommand()
Image	AlternateText, ImageAlign, ImageUrl	
Panel	BackImageUrl, HorizontalAlign, Wrap	
Label	Text	
Button	CommandArgument, CommandName, Text, CausesValidation	OnClick(), OnCommand()
TextBox	AutoPostBack, Columns, MaxLength, ReadOnly, Rows, Text, TextMode, Wrap	OnTextChanged()
CheckBox	AutoPostBack, Checked, Text, TextAlign	OnCheckedChanged()
RadioButton	AutoPostBack, Checked, GroupName, Text, TextAlign	OnCheckedChanged()
ImageButton	CommandArgument,CommandName,CausesValidation	OnClick(), OnCommand()
Table	BackImageUrl, CellPadding, CellSpacing, GridLines, HorizontalAlign, Rows	
TableRow	Cells, HorizontalAlign, VerticalAlign	
TableCell	ColumnSpan, HorizontalAlign, RowSpan, Text, VerticalAlign, Wrap	
Literal	Text	
PlaceHolder		

Объясним назначение некоторых свойств и событий элементов управления. Если булево свойство AutoPostBack установлено в значение true, то изменение состояния элемента автоматически ведет к отправке страницы на сервер. Свойство Text содержит текст или поясняющую надпись на элементе управления. Если булево свойство CausesValidation установлено в true, то при щелчке на элементе управления выполняется проверка страницы. События OnTextChanged() и OnCheckedChanged() срабатывают на сервере при изменении состояния соответствующих элементов управления. Некоторые элементы управления генерируют событие OnClick(), которое происходит при нажатии на элементе.

Три базовых элемента управления – Button, ImageButton и LinkButton – способны генерировать событие OnCommand() в дополнение к событию OnClick(). У этих элементов можно задать два текстовых свойства: CommandName и CommandArgument. Когда кнопка нажимается, значение указанных свойств можно использовать в обработчике события OnCommand().

5.7. ПРОВЕРОЧНЫЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

Кроме визуальных элементов управления, ASP.NET предоставляет набор проверочных элементов. *Проверочные элементы управления* налагают определенные разработчиком ограничения на данные, вводимые пользователем в формы. При настройке проверочный элемент связывается с элементом управления на форме. В качестве проверяемых могут выступать элементы HTMLInputText, HTMLTextArea, HTMLSelect, HTMLInputFile, TextBox, DropDownList, ListBox, RadioButtonList. В случае если проверка закончилась неудачей, проверочный элемент способен отобразить текстовое разъясняющее сообщение около проверяемого элемента.

В web-приложении проверка данных, вводимым пользователем, может выполняться на стороне клиента, на стороне сервера или в обоих местах. Провер-

ка на стороне клиента уменьшает количество обменов между клиентом и сервером, необходимых для успешного завершения формы. Однако клиентская проверка может быть выполнена не всегда. Во-первых, для выполнения проверки браузер должен поддерживать язык сценариев. Во-вторых, клиент часто не обладает достаточной информацией, требуемой для завершения проверки. Поэтому проверки на стороне клиента обычно используются в сочетании с проверками на стороне сервера. Достоинство проверочных элементов ASP.NET заключается в том, что они способны автоматически распознавать поддержку клиентом языка сценариев и в зависимости от этого генерировать клиентский либо серверный код проверки.

Рассмотрим общую архитектуру проверочных элементов. Любой проверочный элемент реализует интерфейс `System.Web.UI.IValidator`, который объявлен следующим образом:

```
interface IValidator {
    string ErrorMessage{set; get; }
    bool IsValid{set; get; }
    void Validate();
}
```

Метод `Validate()` выполняет процедуру проверку, свойство `IsValid` указывает, успешно ли выполнилась проверка, а свойство `ErrorMessage` позволяет определить строку-сообщение в случае провала проверки.

Для всех проверочных элементов базовым является абстрактный класс `BaseValidator` (из пространства имен `System.Web.UI.WebControls`), основные элементы которого перечислены в таблице 44.

Таблица 44

Элементы класса `System.Web.UI.WebControls.BaseValidator`

Имя элемента	Описание
`ControlToValidate`	Строка, идентификатор того элемента управления, который проверяется
`Display`	Свойство определяет, должно ли значение проверочного элемента занять некоторое пространство, если оно не выводится. Значение свойства – элемент перечисления `ValidatorDisplay`
`EnableClientScript`	Булево свойство, управляет использованием клиентского скрипта для проверки
`Enabled`	Булево свойство для включения или выключения проверочного элемента
`ErrorMessage`	Свойство-строка, позволяет установить или прочитать текстовое сообщение, которое отображается в элементе `ValidationSummary` при неуспешной проверке
`ForeColor`	Цвет строки проверочного элемента (по умолчанию – красный)
`IsValid`	Булево свойство, которое показывает, успешно ли выполнилась проверка
`Text`	Строка, которую отображает проверочный элемент при провале проверки
`Validate()`	Метод выполняет процедуру проверку и обновляет значение свойства `IsValid`

Класс Page поддерживает список всех проверочных элементов на странице в коллекции Validators. Класс Page предоставляет метод верхнего уровня Validate(), который применяется для коллективного вызова одноименного метода всех проверочных элементов. Этот метод вызывается на стороне сервера автоматически, после того загружено состояние элементов управления страницы. Метод Page.Validate() устанавливает булево свойство **страницы** IsValid. Как правило, значение данного свойства проверяется в обработчике события Page_Load.

Опишем подробнее конкретные проверочные элементы ASP.NET.

Элемент: RequiredFieldValidator

Назначение: Используется для проверки того, что элемент управления не пуст или значение в нем изменено.

Специфичные свойства:

- InitialValue – проверка считается не пройденной, если при потере фокуса элементом управления значение в нем равно строке InitialValue. По умолчанию значение свойства – пустая строка.

Элемент: CompareValidator

Назначение: Применяется для сравнения двух полей формы или поля и константы. Может использоваться для того, чтобы проверить, соответствует ли значение поля определенному типу.

Специфичные свойства:

- ControlToCompare – строка, идентификатор того элемента управления, с которым сравнивается связанный с CompareValidator элемент.
- ValueToCompare – значение (в виде строки), с которым сравнивается элемент, связанный с CompareValidator[42].
- Operator – операция сравнения. Тип свойства – перечисление ValidationCompareOperator, в которое входят следующие элементы: Equal, NotEqual, GreaterThan, GreaterThanEqual, LessThan, LessThanEqual, DataTypeCheck. Если Operator равен DataTypeCheck, то выполняется **только** проверка того, соответствует ли значение в элементе управления типу, заданному в свойстве Type.
- Type – тип, в который будет преобразовано значение в элементе управления перед проверкой. Свойство принимает значения из перечисления ValidationDataType с элементами String, Integer, Double, Date, Currency.

Элемент: RangeValidator

Назначение: Проверяет, входит ли значение элемента управления в указанный текстовый или числовой диапазон.

[42] Не устанавливайте свойства ControlToCompare и ValueToCompare одновременно. В противном случае, преймущество имеет свойство ControlToCompare.

Специфичные свойства:

- `MaximumValue`, `MinimumValue` – строки, задающие границы диапазона проверки.
- `Type` – тип, в который будет преобразовано значение в элементе управления перед проверкой. Аналог соответствующего свойства из `CompareValidator`.

Элемент: `RegularExpressionValidator`

Назначение: Проверяет, удовлетворяет ли значение элемента управления заданному регулярному выражению.

Специфичные свойства:

- `ValidationExpression` – строка с регулярным выражением.

Элемент: `CustomValidator`

Назначение: Выполняет определенную пользователем проверку при помощи заданной функции (на стороне клиента, на стороне сервера или в обоих местах).

Специфичные свойства и события:

- `OnServerValidate` – обработчик этого события должен быть задан для проведения проверки на стороне сервера. Фрагмент страницы показывает пример обработчика:

```
<script runat="server">
    void ServerValidation(object source,
                          ServerValidateEventArgs arguments) {
        int i = int.Parse(arguments.Value);
        arguments.IsValid = ((i%2) == 0);
    }
</script>
```

- `ClientValidationFunction` – строки с именем клиентской функции, которая будет использоваться для проверки. Так как проверка выполняется на клиенте, то проверочная функция должна быть включена в клиентский скрипт и может быть написана на JavaScript или на VBScript. Следующий фрагмент страницы показывает пример клиентской функции для проверки. Обратите внимание на аргументы функции и их использование:

```
<script language="vbscript">
   Sub ClientValidate(source, arguments)
      If (arguments.Value mod 2) = 0 Then
         arguments.IsValid = true
      Else
         arguments.IsValid = false
      End If
   End Sub
</script>
```

Элемент: `ValidationSummary`

Назначение: Элемент может использоваться для отображения на странице итогов проверок. Если у проверочных элементов определены свойства `ErrorMessage`, то элемент покажет их в виде списка.

Специфичные свойства:

- `DisplayMode` – свойство позволяет выбрать вид суммарного отчета об ошибках. Значения свойства – элемент перечисления `ValidationSummaryDisplayMode`: `BulletList` (по умолчанию), `List`, `SingleParagraph`.
- `HeaderText` – строка с заголовком отчета.
- `ShowMessageBox` – если данное булево свойство установлено в `true`, то отчет отображается в отдельном диалоговом окне.
- `ShowSummary` – если данное булево свойство установлено в `true` (по умолчанию), то отчет отображается на странице. Свойство часто используется совместно с `ShowMessageBox`. Если, например `ShowMessageBox=true`, `ShowSummary=false`, то отчет отображается только в диалоговом окне, но не на странице.

В качестве примера использования проверочных элементов, а также элементов Web Controls, рассмотрим простую aspx-страницу регистрации. Пользователю необходимо ввести свой адрес электронной почты в качестве Login, ввести и повторить пароль входа. Опционально пользователь может указать свой возраст, который должен быть целым числом в пределах от 10 до 80. При нажатии на кнопку Save происходит проверка страницы (автоматически). Если проверка прошла успешна, выполняется код «сохранения» информации.

Исходный код aspx-страницы приведен ниже:

```
<%@ Page Language="C#" %>

<script runat="server">
    void Button1_Click(object sender, EventArgs e) {
        if (IsValid)
            Label2.Text = "Info saved successfully";
    }
</script>

<html>
<body>
  <form runat="server" id="Form1">
    <asp:Label ID="Lbl1" Text="Registration" runat="server"/>
    <br>
    Email as Login
    <asp:TextBox ID="TB1" runat="server" />
    <asp:RequiredFieldValidator ID="RFV1" runat="server"
         ErrorMessage="Email field cannot be empty"
         ControlToValidate="TB1" Text="*" />
    <asp:RegularExpressionValidator ID="REV1" runat="server"
         ErrorMessage="This is not a valid email"
         ControlToValidate="TB1"
```

```
            ValidationExpression="\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*"
                Text="*" />
        <br>
        Password
        <asp:TextBox ID="TB2" runat="server" TextMode="Password"/>
        <asp:RequiredFieldValidator ID="RFV2" runat="server"
            ErrorMessage="Password field cannot be empty"
            ControlToValidate="TB2" Text="*" />
        <asp:CompareValidator ID="CV1" runat="server" Text="*"
            ErrorMessage="Passwords must be the same"
            ControlToValidate="TB2" ControlToCompare="TB3" />
        <br>
        Repeate Password
        <asp:TextBox ID="TB3" runat="server" TextMode="Password"/>
        <br>
        Your age (optional)
        <asp:TextBox ID="TB4" runat="server" />
        <asp:RangeValidator ID="RangeV1" runat="server"
            ErrorMessage="Age must be in 10 to 80"
            ControlToValidate="TB4" MaximumValue="80"
            MinimumValue="10" Type="Integer" Text="*" />
        <br>
        <asp:Button ID="Button1" OnClick="Button1_Click"
                    runat="server" Text="Save" />
        <br>
        <asp:ValidationSummary ID="VS1" runat="server"
            HeaderText="Some info is not correct" />
        <br>
        <asp:Label ID="Label2" runat="server" />
      </form>
    </body>
    </html>
```

Вид страницы в браузере, когда введены некоторые неправильные значения, показан на рисунке 19.

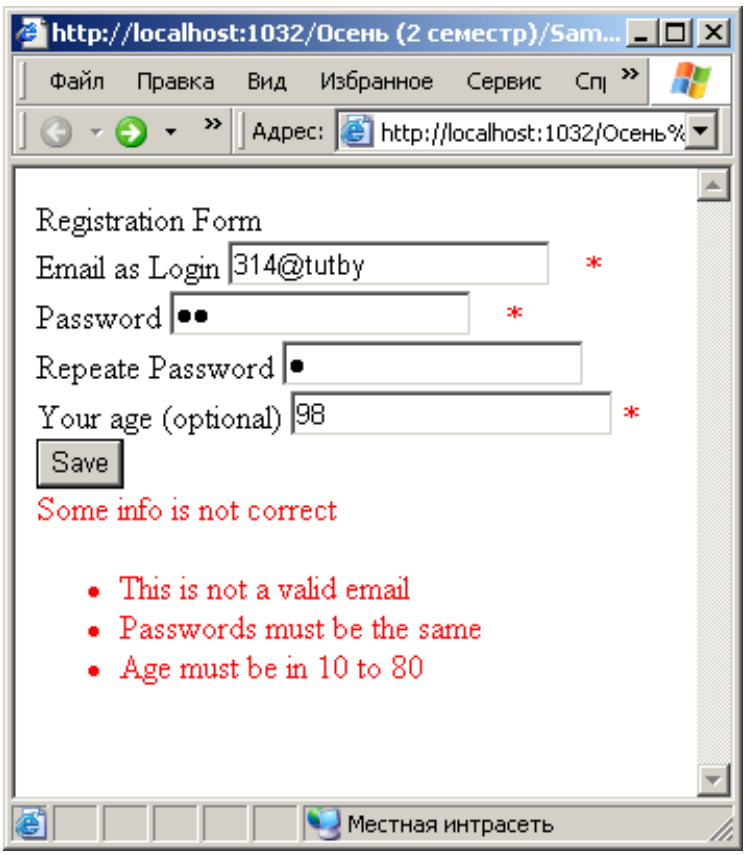

Рис. 19. Страница с проверочными элементами управления

5.8. СПИСКОВЫЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

Списковые элементы управления обеспечивают различные способы представления списков и таблиц на aspx-странице. Название и назначение списковых элементов приведено в таблице 45.

Таблица 45

Списковые элементы управления

Элемент управления	Описание
`DropDownList`	Создает на странице элемент `<select>` с атрибутом `size="1"`, то есть раскрывающийся список с одной видимой строкой. Этот список можно заполнить при помощи элементов управления `ListItem` (тег `<asp:ListItem>`) или посредством связывания данных
`ListBox`	Создает элемент `<select>` с атрибутом `size="n"`, чтобы построить обычное поле списка с единичным или множественным выбором и более чем одной видимой строкой. Этот список можно заполнить при помощи элементов управления `ListItem` или посредством связывания данных
`CheckBoxList`	Создает HTML-элемент `<table>` или простой список, содержащий HTML-флажки. Список можно заполнить при помощи элементов `ListItem` или посредством связывания данных
`RadioButtonList`	Создает HTML-элемент `<table>` или простой список, содержащий HTML-переключатели. Список можно заполнить при помощи элементов `ListItem` или посредством связывания данных

ListItem	Это класс для представления отдельного элемента некоторых списков
Repeater	Повторяет содержимое, указанное один раз, для каждого исходного элемента источника данных, связанного с элементом управления
DataList	Создает HTML-элемент <table>, содержащий строки для каждого элемента источника данных, связанного с элементом управления. Настраиваются шаблоны, которые определяют содержимое и вид каждой строки
DataGrid	Создает HTML-элемент <table>, используемый совместно со связыванием данных на стороне сервера и имеющий встроенные средства поддержки выборки, сортировки и редактирования строк

Элементы DropDownList, ListBox, CheckBoxList и RadioButtonList имеют общего предка – класс ListControl. Полезные свойства данного класса описываются в таблице 46.

Таблица 46

Свойства класса ListControl

Имя свойства	Описание
AutoPostBack	Булево значение; показывает, будет ли страница автоматически отправляться на сервер при изменении пользователем выбора в списке
DataMember	Имя таблицы в DataSource, которая является источником данных для значений списка при заполнении списка путем связывания данных. Свойство используется, если DataSource содержит более одной таблицы (например, если DataSource содержит DataSet)
DataSource	Источник данных для значений списка при заполнении списка путем связывания данных
DataTextField	Имя поля в DataSource, содержимое которого будет отображаемым текстом элементов списка
DataTextFormatString	Строка форматирования для значений из DataTextField (например, {0:C} для денежных сумм)
DataValueField	Имя поля в DataSource, содержимое которого будет значением элементов списка (свойство Value объекта ListItem)
Items	Коллекция элементов ListItem, содержащихся в списке
SelectedIndex	Индекс первого выбранного элемента в списке[43]
SelectedItem	Ссылка на первый выбранный элемент ListItem
SelectedValue	Значение первого выбранного элемента ListItem. Если у элемента задано свойство Value, то возвращается именно оно. Иначе возвращается значение свойства ListItem.Text

Кроме описанных свойств класс ListControl предоставляет событие OnSelectedIndexChanged(). Оно возникает на сервере, когда выбор в списке изменяется и страница пересылается на сервер.

Каждый списковый элемент добавляет к своему базовому классу некоторые специфичные свойства и методы. Они приведены в таблице 47[44].

[43] Для того, чтобы установить или извлечь несколько выбранных элементов, используется булево свойство Selected отдельного объекта ListItem.

[44] Элементы управления Repeater, DataList, DataGrid будут рассмотрены в отдельном параграфе.

Таблица 47

Специфичные свойства списковых элементов управления

Элемент управления или объект	Свойства и методы
DropDownList	
ListBox	Rows
CheckBoxList	CellPadding, CellSpacing, RepeatColumns, RepeatDirection, RepeatLayout, TextAlign
RadioButtonList	CellPadding, CellSpacing, RepeatColumns, RepeatDirection, RepeatLayout, TextAlign
ListItem	Attributes, Selected, Text, Value, метод FromString()

Следующий пример показывает использование элемента управления CheckBoxList. Для элемента управления при помощи редактора свойств были определены значения Text и Value отдельных строк. Настроены свойства ID, CellPadding, CellSpacing, RepeatColumns, RepeatDirection. При нажатии на кнопку Send страница обрабатывается на сервере и выводится информация о выбранных пользователем значениях.

```
<%@ Page Language="C#"%>
<script runat="server">
  protected void Send_Click(object sender, EventArgs e) {
    if(CBL1.SelectedIndex != -1) {
      Label1.Text = "";
      foreach(ListItem LI in CBL1.Items)
        if(LI.Selected)
          Label1.Text += LI.Text + " " + LI.Value + " ";
        }
    else Label1.Text = "No Item Selected";
  }
</script>
<html>
<body>
  <form id="form1" runat="server">
    <asp:CheckBoxList ID="CBL1" runat="server" CellPadding="2"
                      CellSpacing="5" RepeatColumns="2"
                      RepeatDirection="Horizontal">
        <asp:ListItem Value="1">Monday</asp:ListItem>
        <asp:ListItem Value="2">Tuesday</asp:ListItem>
        <asp:ListItem Value="3">Wednesday</asp:ListItem>
        <asp:ListItem Value="4">Thursday</asp:ListItem>
        <asp:ListItem Value="5">Friday</asp:ListItem>
        <asp:ListItem Value="6">Saturday</asp:ListItem>
        <asp:ListItem Value="7">Sunday</asp:ListItem>
    </asp:CheckBoxList>
    <asp:Button ID="Send" runat="server" Text="Send"
                OnClick="Send_Click" /> <br />
    <asp:Label ID="Label1" runat="server"></asp:Label>
  </form>
</body>
</html>
```

Вид страницы в браузере после выбора нескольких элементов и нажатия кнопки Send показан на рисунке 20.

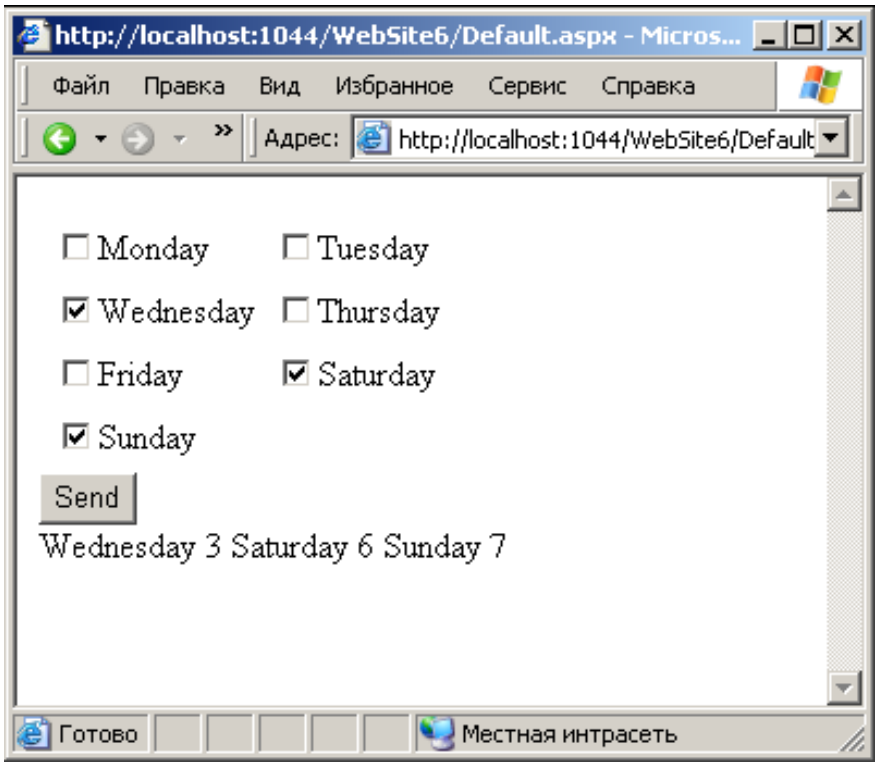

Рис. 20. Страница с элементом управления CheckBoxList

5.9. СВЯЗЫВАНИЕ ДАННЫХ

Под *связыванием данных* (*data binding*) будем понимать помещение данных из некоего источника в элемент управления на странице. При разработке страницы применение связывания подразумевает два этапа:

- указание на странице или в свойстве элемента управления источника данных;
- собственно связывание, то есть перенос данных в элемент управления.

Элементы управления, поддерживающие связывание, имеют в своем составе метод DataBind() для выполнения связывания. Вызов метода DataBind() у родительского элемента управления автоматически ведет к вызову этого метода у дочерних элементов. В частности, вызов DataBind() страницы обеспечивает связывание для всех ее элементов.

Различают два вида связывания данных: одиночное и итеративное. При *одиночном связывании* данных источник данных располагает одним значением для связывания. Для указания источника данных при одиночном связывании используется один из следующих вариантов синтаксиса:

- <%# имя-свойства %>
- <%# вызов-метода(. . .) %>
- <%# выражение %>

Синтаксическая конструкция, указанная выше, располагается в HTML-коде страницы. В частности, таким образом может быть задано значение атрибутов некоторых элементов. Рассмотрим следующий пример страницы:

```
<%@ Page Language="C#" %>
<script runat="server">
    void Page_Load(object sender, EventArgs e) {
        DataBind();
    }
</script>
<html>
<body>
  <form id="Form1" runat="server">
    <asp:TextBox ID="TB1" runat="server" />
    <br>
    <asp:Label ID="Lb1" runat="server">
         <%# TB1.Text%>
    </asp:Label>
    <br>
    <asp:Button ID="B1" runat="server" Text="Button" />
  </form>
</body>
</html>
```

В данной странице при каждой загрузке выполняется метод `DataBind()`. Содержимое метки `Lb1` задано через синтаксис одиночного связывания. Это означает, что при каждой загрузке страницы содержимое метки будет равно содержимому текстового поля `TB1`.

Итеративное связывание подразумевает связывание компонента с источником данных, содержащим некоторую коллекцию данных. Компоненты, поддерживающие итеративное связывание, имеют свойство `DataSource`, которое можно инициализировать любым объектом, поддерживающим интерфейс `IEnumerable`. Схема 21 показывает элементы управления, поддерживающие итеративное связывание, и некоторые наиболее популярные классы, пригодные для подключения к связанному элементу управления:

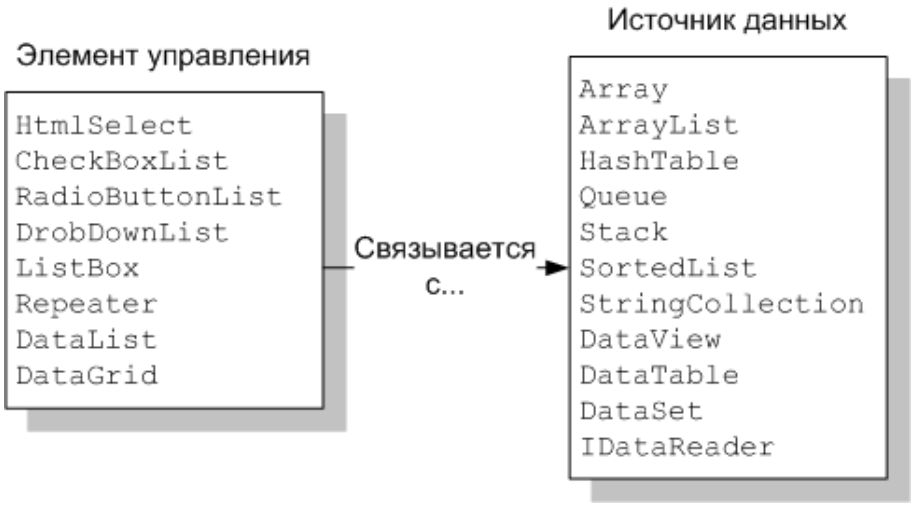

Рис. 21. Схема связывания данных

На рисунке 22 показана страница, в которой два списковых элемента заполнены данными на основе содержимого объекта `ArrayList`:

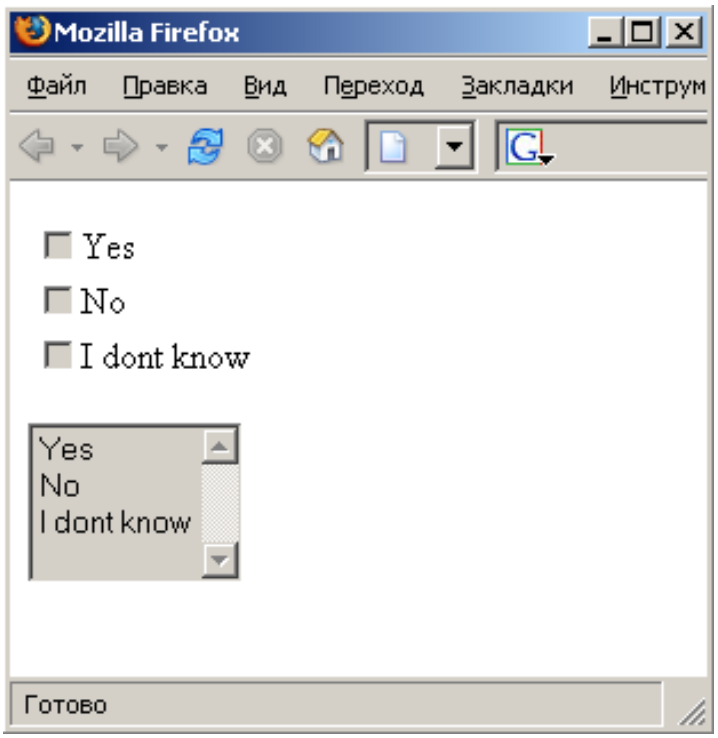

Рис. 22. Связывание с объектом `ArrayList`

Исходный код страницы приведен ниже:

```
<%@ Page Language="C#" %>
<script runat="server">
    void Page_Load(object sender, EventArgs e) {
        if (!Page.IsPostBack) {
            ArrayList vals = new ArrayList();
            vals.Add("Yes");
            vals.Add("No");
            vals.Add("I dont know");
            CBList.DataSource = vals;
            LB.DataSource = vals;
            DataBind();
        }
    }
</script>

<html>
<body>
    <form id="Form1" runat="server">
        <asp:CheckBoxList ID="CBList" runat="server" />
        <br>
        <asp:ListBox ID="LB" runat="server" />
    </form>
</body>
</html>
```

Обратите внимание на следующий момент, характерный для связывания данных в aspx-страницах. Любой элемент управления, допускающий связывание, имеет *локальный кэш данных*, в котором хранит копию связываемых данных. Именно поэтому в методе `Page_Load()` выполняется проверка свойства страницы `IsPostBack`. Нет необходимости создавать массив и выполнять связывание при каждой загрузке; достаточно сделать это один (первый) раз.

Итеративное связывание можно выполнять, используя в качестве источника объект, реализующий `IDataReader`. Пример показывает, как отобразить на странице данные, прочитанные из таблицы базы данных:

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<script runat="server">
  void Page_Load(object sender, EventArgs e) {
    if(!Page.IsPostBack) {
      SqlConnection con = new SqlConnection();
      con.ConnectionString = "Server=(local);" +
                  "Database=CD_Rent;Integrated Security=SSPI";
      SqlCommand cmd = new SqlCommand(
                              "SELECT name FROM Artists", con);
      // Открываем соединение и получаем ридер
      con.Open();
      SqlDataReader r = cmd.ExecuteReader();
      CBList.DataSource = r;
      CBList.DataTextField = "name";
      DataBind();
      con.Close();
    }
  }
</script>

<html>
<body>
    <form id="Form1" runat="server">
        <asp:CheckBoxList ID="CBList" runat="server" />
    </form>
</body>
</html>
```

В состав компонентов ASP.NET входят три *шаблонных элемента управления*: `Repeater`, `DataList` и `DataGrid`. *Шаблон* (template) описывает отдельную строку спискового элемента, позволяя настраивать внешний вид этой строки. При выводе шаблон повторяется требуемое число раз, в зависимости от количества строк в источнике данных.

В таблице 48 приведены допустимые шаблоны и указаны поддерживающие их элементы управления.

Таблица 48

Описание шаблонов

Шаблон	Описание	DataGrid	DataList	Repeater
ItemTemplate	Генерирует внешний вид элемента данных	Да	Да	Да
AlternatingItemTemplate	Если задан, то чередуется с ItemTemplate	Нет	Да	Да
HeaderTemplate	Генерирует вид заголовка столбца	Да	Да	Да
FooterTemplate	Генерирует внешний вид колонтитула столбца	Да	Да	Да
SeparatorTemplate	Генерирует вид разделителя элементов данных	Нет	Да	Да
EditItemTemplate	Генерирует вид редактируемого элемента данных	Да	Да	Нет

Чтобы понять принципы использования шаблонов рассмотрим следующий пример. Пусть при помощи элемента DataList планируется вывести на странице содержимое таблицы Disks из базы CD_Rent. При этом нужно вывести заголовок таблицы, разделить строки, а год выпуска альбома отобразить красным цветом. Вот код aspx-страницы, которая решает поставленную задачу:

```
<%@ Page Language="C#"%>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<script runat="server">
  void Page_Load(object sender, EventArgs e) {
    if(!Page.IsPostBack) {
      SqlConnection con = new SqlConnection();
      con.ConnectionString = "Server=(local);" +
                  "Database=CD_Rent;Integrated Security=SSPI";
      SqlCommand cmd = new SqlCommand(
                              "SELECT name FROM Artists", con);
      con.Open();
      SqlDataReader r = cmd.ExecuteReader();
      DataList1.DataSource = r;
      DataBind();
      con.Close();
    }
  }
</script>

<html>
<body>
  <form id="form1" runat="server">
  <div>
    <asp:DataList ID="DataList1" runat="server">
      <HeaderTemplate>
        Data from Disks
      </HeaderTemplate>
      <ItemTemplate>
```

```
            <asp:TextBox id=_name runat="server"
                 Text = '<%#((IDataRecord)Container.DataItem)["title"]%>'/>
            <asp:Label id=_year ForeColor="Red" runat="server"
            Text='<%#((IDataRecord)Container.DataItem)["release_year"]%>'/>
          </ItemTemplate>
          <SeparatorTemplate><hr /></SeparatorTemplate>
        </asp:DataList>
      </div>
      </form>
    </body>
    </html>
```

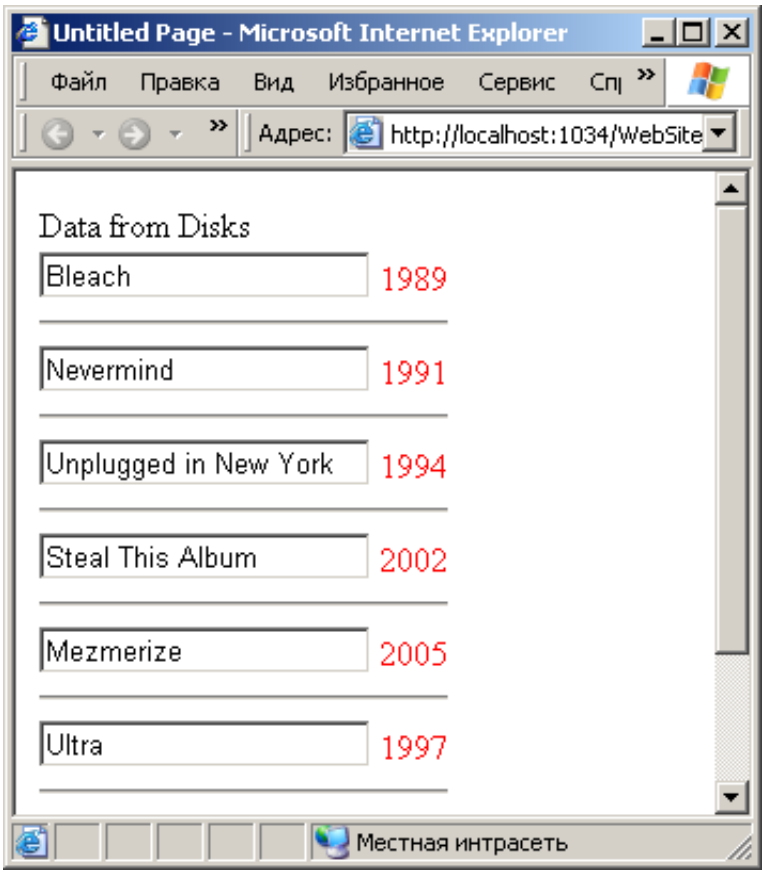

Рис. 23. Вывод данных при помощи шаблонов

Рассмотрим принципы, по которым происходит связывание данных в шаблонных элементах управления. Каждое выражение связывания данных, встретившееся в шаблоне, преобразуется в экземпляр специального класса `DataBoundLiteralControl` и добавляется как дочерний элемент управления в шаблонный элемент управления (в нашем примере – в элемент `DataList`). Кроме этого, синтаксический анализатор страницы генерирует обработчик, подключаемый к событию `DataBinding` класса `DataBoundLiteralControl`:

```
public void @__DataBinding__control4(
                            object sender, System.EventArgs e) {
  TextBox dataBindingExpressionBuilderTarget;
  DataListItem Container;
  dataBindingExpressionBuilderTarget = (TextBox)sender;
```

```
    Container =
        (DataListItem)dataBindingExpressionBuilderTarget.BindingContainer;
    dataBindingExpressionBuilderTarget.Text =
       System.Convert.ToString(((IDataRecord)Container.DataItem)["title"],
              System.Globalization.CultureInfo.CurrentCulture);
}
```

Событие `DataBinding` класса `DataBoundLiteralControl` генерируется один раз для каждой строки источника данных во время вызова метода `DataBind()`. Код, сгенерированный для обработчика события, подготавливает локальную переменную `Container`, присваивая ей значение свойства `BindingContainer` элемента управления, сгенерировавшего событие. Свойство `BindingContainer` возвращает ссылку на текущий шаблон `ItemTemplate`, который для класса `DataListItem` является экземпляром класса `DataListItem`. И, наконец, для доступа к текущей строке источника данных класс `DataListItem` предоставляет свойство `DataItem`. Поскольку элемент управления `DataList` связывается со считывателем `SqlDataReader`, источник данных предоставлет интерфейс `IDataRecord`, который можно использовать для доступа к текущему значению столбца `title`.

В предыдущем примере для извлечения информации из контейнера при связывании данных можно было использовать статический метод `Eval()` класса `DataBinder`. Метод `Eval()` использует механизм отражения, чтобы выяснить тип источника данных и сконструировать требуемый вызов индексатора источника. Метод `Eval()` допускает две перегрузки, одна из которых позволяет указать строку форматирования, применяемую при отображении данных:

```
<ItemTemplate>
  <asp:TextBox id=_name runat="server"
  Text= '<%# DataBinder.Eval(Container.DataItem, "title")%>'/>
  <asp:Label id=_year ForeColor="Red" runat="server"
  Text = '<%# DataBinder.Eval(Container.DataItem, "release_year")%>' />
</ItemTemplate>
```

В заключение параграфа рассмотрим пример отображения данных на странице с возможностью их редактирования. Пусть требуется показать содержимое таблицы `Artists` из базы `CD_Rent`, а также разрешить пользователю редактировать эту таблицу. Вначале опишем несколько вспомогательных методов. Первый метод будет использоваться для того, чтобы выполнить SQL-команду над базой `CD_Rent`:

```
public void ExecuteSQLStatement(string strSQL) {
    SqlConnection con = new SqlConnection(. . .);
    SqlCommand cmd = new SqlCommand(strSQL, con);
    // Выполняем команду
    con.Open();
    cmd.ExecuteNonQuery();
    con.Close();
}
```

Следующий метод будет использоваться, чтобы получить данные из базы и связать их с компонентом DataList:

```
public void BindDataList() {
    // Создаем соединение
    SqlConnection con = new SqlConnection(. . .);
    // Создаем команду для получения всех данных таблицы
    SqlCommand cmd = new SqlCommand("SELECT * FROM Artists", con);
    // Открываем соединение и получаем ридер, выполняя команду
    con.Open();
    SqlDataReader r = cmd.ExecuteReader();
    // Выполняем связывание (DList – идентификатор DataList)
    DList.DataSource = r;
    DList.DataBind();
    con.Close();
}
```

Четыре метода будут обработчиками событий соответствующих команд DataList:

```
public void DoItemEdit(object source,
                       DataListCommandEventArgs e) {
    // Указываем строку, которую переводим в режим
    // редактирования и "пересвязываем" данные
    DList.EditItemIndex = e.Item.ItemIndex;
    BindDataList();
}

void DoItemUpdate(object source, DataListCommandEventArgs e) {
    // Ищем в строке элемент управления
    TextBox tb = (TextBox)e.Item.FindControl("TB");
    // Формируем SQL-команду
    string strSQL = "UPDATE Artists SET name='" + tb.Text +
          "' WHERE id='"+DList.DataKeys[e.Item.ItemIndex]+"'";
    // Выполняем команду
    ExecuteSQLStatement(strSQL);
    // Показываем, что строка больше не редактируется
    DList.EditItemIndex = -1;
    // "Пересвязываем" данные
    BindDataList();
}

public void DoItemCancel(object source,
                         DataListCommandEventArgs e) {
    // Показываем, что строка больше не редактируется
    DList.EditItemIndex = -1;
    // "Пересвязываем" данные
    BindDataList();
}

public void DoItemDelete(object source,
                         DataListCommandEventArgs e) {
    //Формируем и выполняем SQL-команду
```

```
        string strSQL = "DELETE FROM Artists WHERE id='" +
                        DList.DataKeys[e.Item.ItemIndex] + "'";
        ExecuteSQLStatement(strSQL);
        DList.EditItemIndex = -1;
        BindDataList();
    }
```

Отдельно представим шаблоны для заголовка DataList, элемента (с кнопкой Edit), элемента в режиме редактирования (поле ввода, кнопки Update, Delete, Cancel), разделителя:

```
<HeaderTemplate>
  <b>Information about Artists</b>
</HeaderTemplate>

<ItemTemplate>
  <asp:Button ID="B1" CommandName="Edit" runat="server" Text="Edit" />
  ID: <%# DataBinder.Eval(Container.DataItem, "id") %>  
  Name: <%# DataBinder.Eval(Container.DataItem, "name") %>
</ItemTemplate>

<EditItemTemplate>
  ID: <%# DataBinder.Eval(Container.DataItem, "id") %>  
  <asp:Button ID="B2" CommandName="Update" runat="server"
                                           Text="Update" />
  <asp:Button ID="B3" CommandName="Delete" runat="server"
                                           Text="Delete" />
  <asp:Button ID="B4" CommandName="Cancel" runat="server"
                                      Text="Cancel" /> <br />
  Name: <asp:TextBox ID="TB" runat="server"
             Text='<%# DataBinder.Eval(Container.DataItem, "name") %>'/>
</EditItemTemplate>

<SeparatorTemplate> <hr /> </SeparatorTemplate>
```

В свойствах DataList настроим внешний вид, укажем ключевое поле, и укажем обработчики команд (связь между обработчиком и соответствующей кнопкой устанавливается автоматически, если у кнопки задано правильное свойство CommandName):

```
<asp:DataList ID="DList" runat="server" CellSpacing="2"
              OnEditCommand="DoItemEdit"
              OnUpdateCommand="DoItemUpdate"
              OnDeleteCommand="DoItemDelete"
              OnCancelCommand="DoItemCancel" DataKeyField="ID">
```

Рис. 24 демонстрирует страницу в режиме редактирования второй строки.

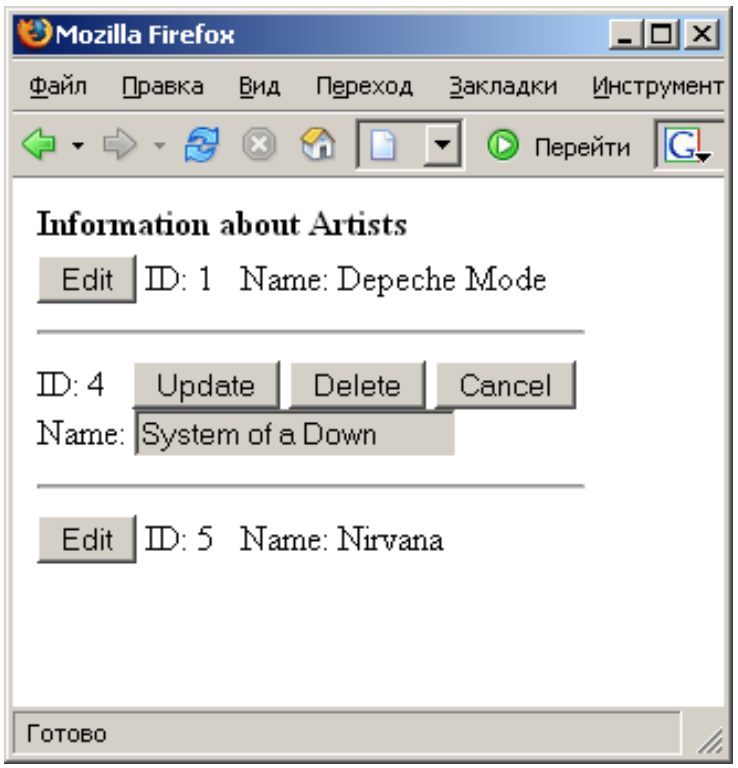

Рис. 24. Страница в режиме редактирования

5.10. WEB-ПРИЛОЖЕНИЕ. ФАЙЛ GLOBAL.ASAX

Технология ASP.NET поддерживает концепцию web-приложений. *Web-приложение* – это совокупность файлов, размещенных в отдельном каталоге, которому сопоставлен виртуальный каталог web-сервера. Рабочий процесс ASP.NET обслуживает каждое web-приложение в отдельном домене, используя специфические настройки приложения. *Границей* приложения является виртуальный каталог, в том смысле, что, перемещаясь по страницам внутри виртуального каталога, пользователь остается в рамках одного web-приложения.

Каждое web-приложение содержит, по крайней мере, один aspx-файл. Кроме этого, в состав приложения могут входить следующие файлы:

- Единственный файл `global.asax`, размещаемый в корневом каталоге приложения.
- Один или несколько файлов конфигурации `web.config`. Если web-приложение содержит подкаталоги, то допускается не более одного файла `web.config` на подкаталог.
- Один или несколько файлов, описывающих пользовательские элементы управления (расширение `*.ascx`).
- Один или несколько файлов классов для поддержки технологии Code-Behind.
- Поддиректорию `/bin`, содержащую сборки, автоматически подключаемые к приложению.
- Файлы любых других типов (`*.htm`, `*.asp`, изображения и т.д.).

ASP.NET поддерживает глобальный файл для каждого web-приложения – файл `global.asax`. Этот файл играет роль пункта реализации глобальных событий, объектов и переменных. Общий формат файла `global.asax` следующий:

```
<%@ директива атрибут = значение %>

<script runat="server">
    [обработчики событий приложения]
</script>
```

Файл `global.asax` поддерживает три директивы: `@Application`, `@Import`, `@Assembly`. Директива `@Application` позволяет определить базовый класс, на основе которого создается класс приложения (атрибут `Inherits`), указать язык программирования для серверного кодаобработчиков событий (атрибут `Language`), а также задать описание приложения (атрибут `Description`). Директива `@Import` позволяет импортировать пространства имен для использования в `global.asax`. Директива `@Assembly` используется для подключения сборок к приложению (сборки из поддиректории `\bin` подключаются автоматически).

Каждое web-приложение описывается объектом класса, производного от класса `HttpApplication`. Свойства данного класса описаны в таблице 49.

Таблица 49

Свойства класса `HttpApplication`

Имя свойства	Описание
`Application`	Объект класса `HttpApplicationState`, описывающий состояние web-приложения
`Context`	Объект класса `HttpContex`, описывающий контекст запроса
`Modules`	Коллекция *модулей* – специальных обработчиков, дополняющих функции работы с запросом пользователя
`Request`	Ссылка на объект `HttpRequest`, обеспечивающий доступ к информации о HTTP-запросе
`Response`	Ссылка на объект `HttpResponse`, обеспечивающий доступ к информации о HTTP-ответе
`Server`	Объект класса `HttpServerUtility`, описывающий параметры web-сервера
`Session`	Ссылка на объект класса `HttpSessionState`, хранящий данные текущей сессии пользователя в web-приложении
`User`	Ссылка на объект, реализующий интерфейс `IPrincipal` и описывающий пользователя. Свойство используется при проведении аутентификации

Для разработчика важной является возможность перехватывать события приложения. В таблице 50 перечислены события приложения, предоставляемые классом `HttpApplication`. Большинство из них генерируются при обработке приложением каждого запроса. Для добавления обработчика любого из событий нужно или явно подключить делегат к событию во время инициализации приложения, или определить метод с именем `Application_`*`событие`*`()`, автоматически подключаемый во время выполнения.

Таблица 50

События web-приложения

Событие	Причина срабатывания	Последовательность
BeginRequest	Получение нового запроса	1
AuthenticateRequest	Завершение аутентификации пользователя	2
AuthorizeRequest	Завершение авторизации пользователя	3
ResolveRequestCache	Генерируется после авторизации, но перед запуском обработчика. Используется модулями кэширования для отмены выполнения обработчиков запроса, если в кэше есть нужная запись	4
AcquireRequestState	Загрузка состояния сеанса	5
PreRequestHandlerExecute	Перед передачей запроса обработчику	6
PostRequestHandlerExecute	Завершение обработчика запроса	7
ReleaseRequestState	После завершения всех обработчиков запроса. Используется модулями состояний для сохранения значений состояния	8
UpdateRequestCache	После завершения обработчика. Используется модулями кэширования для сохранения ответа в кэше	9
EndRequest	После обработки запроса	10
Disposed	Перед закрытием приложения	–
Error	При наступлении необработанной исключительной ситуации	–
PreSendRequestContent	Перед передачей клиенту содержимого ответа	–
PreSendRequestHeaders	Перед передачей клиенту заголовков HTTP	–

Некоторые события можно обработать, используя только обработчики, размещенные в файле `global.asax`. Ни одно из них не генерируется на уровне запроса.

Таблица 51

Особые события web-приложения

Событие	Причина срабатывания
Application_Start	Запуск приложения
Application_End	Завершение приложения
Session_Start	Начало сеанса пользователя
Session_End	Завершение сеанса пользователя

Приведем пример файла `global.asax`, содержащего обработчики событий `BeginRequest` и `EndRequest`:

```
<%@ Application Language="C#" %>
<script RunAt="server">
  void Application_BeginRequest(object sender, EventArgs e) {
    Response.Write("Request starts!" + "<br />");
  }
```

```
    void Application_EndRequest(object sender, EventArgs e) {
      Response.Write("Request ends!" + "<br />");
    }
</script>
```

5.11. УПРАВЛЕНИЕ СОСТОЯНИЯМИ В WEB-ПРИЛОЖЕНИЯХ

Специфика большинства web-приложений подразумевает хранение информации (состояния) при переходе от одной страницы приложения к другой. Типичным примером является Internet-магазин, в котором пользователь просматривает страницы с информацией о товаре, помещает некоторые товары в свою корзину покупок, а затем оформляет окончательный заказ. Протокол HTTP не имеет стандартных средств для поддержки сохранения состояний, поэтому данная задача реализуется внешними по отношению к протоколу средствами. Платформа ASP.NET предоставляет программистам достаточный встроенный «арсенал» средств управления состояниями, что зачастую избавляет от написания лишнего кода.

ASP.NET поддерживает четыре типа состояний: *состояние приложения*, *состояние сеанса*, *cookie* и *состояние отображения* (viewstate). Таблица 52 предназначена для характеристики и сравнения различных типов состояний.

Таблица 52

Характеристика различных типов состояний

Тип	**Область видимости**	**Преимущества**	**Недостатки**
Состояния приложения	Глобальная в приложении	Совместно используются всеми клиентами	Не могут вместе использоваться многими компьютерами Область применения перекрывается средствами кэширования данных
Состояния сеанса	На уровне клиента	Могут конфигурироваться на совместное использование многими компьютерами	Требуется коррекция файлов cookie и адресов URL для управления взаимодействием клиентов
Cookie	На уровне клиента	Не зависят от настроек сервера Сохраняются у клиента Могут существовать после завершения сеанса	Ограниченный объем памяти (около 4 Кбайт) Клиент может отключить поддержку cookie Состояние передается в обоих направлениях с каждым запросом
Состояния отображения	На уровне запросов POST к этой же странице	Не зависят от настроек сервера	Состояние сохраняется, только если запрос POST передается этой же странице. Состояния передаются в обоих направлениях с каждым запросом

К состоянию приложения можно обратиться, используя свойство `Application` класса страницы или приложения. Свойство возвращает объект типа `HttpApplicationState`. Этот класс является коллекцией, хранящей данные любых типов в парах ключ/значение, где ключом является строка или числовой индекс. Пример использования состояния приложения приведен в следующем фрагменте кода. Как только приложение запускается, оно загружает данные из БД. После этого для обращения к данным можно сослаться на кэшированную версию объекта:

```
// Фрагмент файла global.asax:
void Application_Start(object src, EventArgs e) {
    DataSet ds = new DataSet();
    // Заполняем набор данных (не показано)
    // Кэшируем ссылку на набор данных
    Application["FooDataSet"] = ds;
}

// Фрагмент файла страницы:
void Page_Load(object src, EventArgs e) {
    DataSet ds = (DataSet)(Application["FooDataSet"]);
    MyDataGrid.DataSource = ds;
}
```

Состояние приложения разделяется между всеми клиентами приложения. Это значит, что несколько клиентов одновременно могут осуществлять доступ (чтение или запись) к данным в состоянии приложения. Класс `HttpApplicationState` имеет встроенные средства контроля целостности данных. В некоторых ситуациях можно вручную вызвать методы класса `Lock()` и `Unlock()` для установки и снятия блокировки состояния приложения.

В большинстве web-приложениях требуется хранить некоторую информацию персонально для каждого клиента (например, для отслеживания заказанных товаров при посещении интернет-магазина). ASP.NET предполагает использование для этих целей состояний сеансов (или *сессий*). Когда с приложением начинает взаимодействовать новый клиент, новый *идентификатор сеанса* автоматически генерируется и ассоциируется с каждым последующим запросом этого же клиента (или с помощью cookies, или путем коррекции URL).

Состояния сеанса поддерживаются в экземпляре класса `HttpSessionState` и доступны через свойство `Session` страницы или объекта `HttpContext`. Таблица 53 показывает основные элементы класса `HttpSessionState`.

Таблица 53

Элементы класса `HttpSessionState`

Свойство/Метод	Описание
`Count`	Количество элементов в коллекции
`IsCookieless`	Булево свойство; указывает, используются ли cookie для сохранения идентификатора сессии
`IsNewSession`	Булево свойство; `true`, если сессия создана текущим запросом
`IsReadOnly`	Булево свойство; `true`, если элементы сессии доступны только для чтения

Mode	Режим сохранения объекта сессии, одно из значений перечисления SessionStateMode: InProc – объект сохраняется в рабочем процессе ASP.NET, Off – отсутствует поддержка сессий, StateServer – за сохранение объекта сессии отвечает внешняя служба, SQLServer – объект сессии сохраняется при помощи SQL Server
SessionID	Строка с уникальным идентификатором сессии
Timeout	Время жизни сессии в минутах
Add()	Метод для добавления элемента в коллекцию сессии
Abandon()	Метод вызывает уничтожение сессии
Clear()	Удаляет все элементы коллекции, аналог RemoveAll()
Remove()	Удаляет элемент с указанным именем (ключом)
RemoveAll()	Удаляет все элементы коллекции
RemoveAt()	Удаляет элемент в указанной позиции

Для доступа к отдельным элементам, хранящимся в сессии, можно использовать строковый или целочисленный индексатор.

В качестве примера использования состояний сеансов рассмотрим реализацию классического *списка покупок*. Web-приложение будет содержать две страницы: на первой выбираются товары, на второй показывается список выбранных товаров и их цена. Для описания информации об отдельном товаре используем вспомогательный класс Item:

```
public class Item {
    public string name;
    public int cost;
    public Item(string name, int cost) {
        this.name = name;
        this.cost = cost;
    }
}
```

Код первой (default.aspx) и второй (default2.aspx) страниц представлен ниже:

```
<!-- Файл: default.aspx -->
<%@ Page Language="C#" %>
<script runat="server">
    void Button1_Click(object sender, EventArgs e) {
        ArrayList cart = (ArrayList)Session["cart"];
        if(CB1.Checked) cart.Add(new Item("Pencil", 10));
        if(CB2.Checked) cart.Add(new Item("Notebook", 15));
        Response.Redirect("default2.aspx");
    }
</script>

<html>
<body>
  <form id="form1" runat="server">
  <div>
    Чудо-товары:<br />
    Карандаш - 10 у.е.
    <asp:CheckBox ID="CB1" runat="server" Text="Заказ!" />
    <br />
```

```
      Блокнот    - 15 у.е.
      <asp:CheckBox ID="CB2" runat="server" Text="Заказ!" />
      <br />
      <asp:Button ID="Button1" runat="server" Text="Оплатить!"
                  OnClick="Button1_Click" />
    </div>
    </form>
  </body>
  </html>

  <!-- Файл: default2.aspx -->
  <%@ Page Language="C#" %>
  <script runat="server">
    void Page_Load(object sender, EventArgs e) {
      ArrayList cart = (ArrayList)Session["cart"];
      int total = 0;
      foreach (Item item in cart) {
        total += item.cost;
        Response.Output.Write("<p>Товар: {0}, Цена: ${1}</p>",
                              item.name, item.cost);
      }
      Response.Write("<hr/>");
      Response.Output.Write("<p>Стоимость: {0}</p>", total);
    }
  </script>

  <html>
  <body>
    <form id="form1" runat="server">
    <div>
      <a href="Default.aspx">Продолжить выбор!</a>
    </div>
    </form>
  </body>
  </html>
```

Для выполнения инициализации данных в состоянии сеанса используется событие `Session_Start` объекта `Application`. Далее представлен файл `global.asax` нашего web-приложения:

```
<%@ Application Language="C#" %>
<script runat="server">
    void Session_Start(object sender, EventArgs e) {
        Session["cart"] = new ArrayList();
    }
</script>
```

Как было сказано выше, чтобы ассоциировать состояние сеанса с определенным клиентом, используется *идентификатор сеанса*, уникальный для клиента. По умолчанию идентификатор сохраняется в файле cookie. В качестве альтернативного решения ASP.NET поддерживает методику отслеживания

идентификаторов с помощью коррекции URL. Управление идентификаторами сеанса настраивается с помощью конфигурационного файла:

```
<configuration>
    <system.web>
        <sessionState cookieless="true" />
    </system.web>
</configuration>
```

Сделаем небольшое отступление и рассмотрим подробнее возможные атрибуты конфигурационного элемента `sessionState`.

Таблица 54

Атрибуты конфигурационного элемента `sessionState`

Атрибут	Значение	Описание
`cookieless`	`True`, `False`	Если `True`, то идентификатор сеанса передается путем коррекции URL, если `False` – то с помощью cookie
`mode`	`Off`, `InProc`, `SQLServer`, `StateServer`	Место хранения состояния сеанса (если `mode` не равно `Off`)
`stateConnectionString`	Пример: `'192.168.1.100:42424'`	При `mode=StateServer` – имя сервера и порт
`sqlConnectionString`	Пример: `'server=192.168.1.100; uid=sa;pwd='`	При `mode=SQLServer` – строка соединения с базой данных. Предполагается, что это база `tempdb`
`timeout`	Пример: `40`	Время жизни состояния сеанса в минутах (по умолчанию – 20 минут)

Платформа ASP.NET предоставляет возможность хранить сессии вне рабочего процесса. Если атрибут `mode` элемента `sessionState` файла `web.config` установлен в `StateServer` или в `SqlServer`, то ASP.NET хранит сессии в другом процессе (выполняется как служба) или в базе данных SQL Server. Если используется сохранение сессии вне рабочего процесса, то любой тип, хранящийся в сессии, должен быть сериализуемым.

Пусть атрибут `mode` установлен в `StateServer`. В этом случае на компьютере, который будет ответственным за хранение сессий (это не обязательно должен быть компьютер, на котором выполняется рабочий процесс ASP.NET) должна быть запущена служба *ASP.NET state service*. Именно в адресном пространстве этой службы будут храниться данные. По умолчанию служба прослушивает порт 42424, но этот номер можно изменить при помощи ключа `HKLM\SYSTEM\CurrentControlSet\Services\aspnet_state\Parameters\Port`.

Еще один способ хранения состояний сеансов вне процесса – их размещение в базе данных SQL Server. Прежде чем задать этот режим, нужно выполнить сценарий `InstallSqlState.sql`[45]. Главная задача сценария – создать таб-

[45] Данный сценарий находится в главном каталоге .NET.

лицу в базе tempdb, способную хранить сессии отдельных клиентов и индексированную идентификаторами сессий. В конфигурационном файле указывается сервер БД, пользователь и пароль.

По умолчанию ASP.NET предполагает, что каждая страница просит загрузить состояния сеанса во время инициализации страницы и сохранить состояние после вывода страницы. Это означает два обращения к серверу состояний или базе данных при хранении сессий вне рабочего процесса. Атрибут EnableSessionState директивы страницы Page позволяет более тонко настроить работу с сессиями. Если атрибут установлен в значение ReadOnly, то сессии загружаются, но не сохраняются после вывода страницы. Если же установить атрибут в false, то работа с сессиями на странице отключается.

Для управления данными cookie платформа ASP.NET предоставляет класс HttpCookie. Классы HttpResponse и HttpRequest содержат свойство-коллекцию Cookies типа HttpCookieCollection для работы с установленными cookie клиента. Определение класса HttpCookie представлено ниже:

```
public sealed class HttpCookie {
    // Открытые конструкторы
    public HttpCookie(string name);
    public HttpCookie(string name, string value);

    // Открытые свойства
    public string Domain { set; get; }
    public DateTime Expires { set; get; }
    public bool HasKeys { get; }
    public string Name { set; get; }
    public string Path { set; get; }
    public bool Secure { set; get; }
    public string this[string key] { set; get; }
    public string Value { set; get; }
    public NameValueCollection Values { get; }
}
```

Каждый файл cookie может хранить много пар имя/значение, доступных посредством коллекции Values класса HttpCookie или посредством индексатора по умолчанию этого класса. Важно отметить следующее:

1. Объем информации, хранящейся в cookie, ограничен 4 Кбайтами.
2. И ключ, и значение сохраняются в cookie в виде строк.
3. Некоторые браузеры могут отключить поддержку cookie.

Чтобы вынудить клиента установить cookie, добавьте новый экземпляр класса HttpCookie в коллекцию cookie перед выводом страницы. При этом можно проверить, поддерживает ли клиентский браузер cookie, при помощи булевого свойства Request.Browser.Cookies. Для доступа к файлу cookie, переданном клиентом в запросе, нужно обратиться к коллекции Cookies, являющейся свойством объекта запроса.

В следующем листинге представлена страница, устанавливающая и использующая cookie с именем name:

```
<%@ Page Language="C#" %>
```

```
<script runat="server">
  void Page_Load(object sender, EventArgs e) {
    if(Request.Cookies["name"] != null)
      Label1.Text = "Your name is " + Request.Cookies["name"].Value;
  }

  // Нажатие на кнопку устанавливает cookie
  void Button1_Click(object sender, EventArgs e) {
    HttpCookie ac = new HttpCookie("name");
    // Устанавливаем ненулевое время жизни (3 дня),
    // чтобы информация сохранилась после закрытия браузера
    ac.Expires = DateTime.Now.AddDays(3);
    ac.Value = nameTextBox.Text;
    Response.Cookies.Add(ac);
  }
</script>

<html>
<body>
  <form id="form1" runat="server">
    <asp:Label ID="Label1" runat="server" /> <br /> <br />
    <asp:TextBox ID="nameTextBox" runat="server" /> <br />
    <asp:Button ID="Button1" runat="server" Text="Set cookie"
                OnClick="Button1_Click" />
  </form>
</body>
</html>
```

Кроме состояний сеансов и cookie, ASP.NET предоставляет средства хранения состояний отдельных клиентов с помощью механизма, называемого *состоянием отображения*. Состояние отображения храниться в скрытом поле `__VIEWSTATE` каждой страницы ASP.NET. При каждом запросе страницы самой себя содержимое поля `__VIEWSTATE` передается в составе запроса. Состояния отображения предназначены главным образом для сохранения состояний элементов управления в последовательности ответных запросов, однако их можно использовать и как механизм сохранения состояний отдельных клиентов между ответными запросами к этой же странице.

Состояние отображения доступно из любого элемента управления как свойство `ViewState`. Оно представляется как объект типа `StateBag`, поддерживающий хранение любых сериализуемых типов.

5.12. КЭШИРОВАНИЕ

Платформа ASP.NET поддерживает кэширование страницы, фрагмента страницы и данных. Кэширование динамически генерируемой страницы (*кэширование вывода*) повышает производительность благодаря тому, что страница при этом генерируется только при первом обращении. При любом следующем обращении она будет возвращена из кэша, что экономит время на ее генерацию.

Время жизни кэшированной страницы можно установить явно. По истечении времени жизни страница будет удалена из кэша.

Кэширование вывода управляется директивой `OutputCache` страницы. Атрибуты данной директивы и их описание представлены в таблице 55.

Таблица 55

Атрибуты директивы `OutputCache`

Атрибут	Значение	Описание
`Duration`	Число	Время существования страницы или пользовательского элемента управления в кэше (в секундах)

`Location`	`Any`
	`Client`
	`Downstream`
	`Server`
	`ServerAndClient`
	`None`
		Атрибут управляет передаваемыми клиенту заголовком и метадескрипторами, указывающими, где может быть кэширована страница. При значении `Client` страница кэшируется только в браузере, при значении `Server` – только на Web-сервере, при значении `Downstream` – на прокси-сервере и у клиента. При `ServerAndClient` кэш располагается на клиенте или на Web-сервере, прокси-сервера для кэша не используются. `None` отключает кэширование страницы

`VaryByCustom`	`browser` Спец. строка	Страница кэшируется по-разному в зависимости от имени и версии браузера клиента или от заданной строки, обрабатываемой переопределенным методом `GetBaryByCustomString()`
`VaryByHeader`	* Имена заголовков	Список строк, разделенных точкой с запятой и представляющих передаваемые клиентам заголовки
`VaryByParam`	`none` * Имена параметров	Список строк, разделенных точкой с запятой и представляющих значения строки запроса `GET` или переменные запроса `POST`
`VaryByControl`	Список полностью квалифицированных имен свойств	Список строк, представляющих свойства пользовательского элемента управления, используемые для настройки кэшей вывода (атрибут применяется только с пользовательскими элементами управления)

В директиве `OutputCache` атрибуты `Duration` и `VaryByParam` является обязательным. Атрибут `VaryByParam="none"` означает, что для каждого типа запроса (`GET`, `HEAD` или `POST`) будет кэшировано по одной копии страницы.

В листинге показан пример использования директивы `OutputCache`, задающей существование страницы в кэше на протяжении одного часа после первого обращения. В странице выводится дата ее генерации, поэтому во всех выводах после первого вы увидите одно и то же время, пока не истечет срок жизни страницы.

```
<%@ Page Language="C#" %>
<%@ OutputCache Duration="3600" VaryByParam="none"%>
<script runat="server">
    void Page_Load(object sender, EventArgs e) {
        msg.Text = DateTime.Now.ToString();
```

```
    }
</script>

<html>
<body>
    <h3>Вывод кэшированной страницы</h3>
    <p>В последний раз страница сгенерирована:
    <asp:Label ID="msg" runat="server" /></p>
</body>
</html>
```

Запрос страницы может выполняться несколькими способами: при помощи глагола GET без параметров, при помощи глагола HEAD, при помощи GET с параметрами, или с использованием POST с телом, содержащим пары имя/значение. Кэшировать страницу, к которой обращаются с разными строками GET или переменными POST сложно, так как требуется кэшировать разные версии страницы для различных комбинаций параметров. Токую настройку кэширования в этом случае можно провести при помощи атрибута VaryByParam директивы OutputCache. Если задать VaryByParam="none", то кэш сохраняет одну версию страницы для каждого типа запроса. Получив запрос GET со строкой переменных, кэш вывода игнорирует строку и возвращает экземпляр страницы, кэшированный для GET. Если указать VaryByParam="*", будет кэширована отдельная версия страницы для каждой уникальной строки GET и каждого уникального набора параметров POST. Такой способ кэширования может оказаться крайне неэффективным. В атрибуте VaryByParam можно указать имя (или список имен) переменных запроса. В этом случае уникальные копии страницы будут сохраняться в кэше вывода, если в запросе значения перечисленных переменных разные.

Атрибуты VaryByHeader и VaryByCustom управляют сохранением версий страницы в кэше на основе значений заголовков HTTP-запроса и версий браузера соответственно. Например, если страница выводится по-разному в зависимости от заголовка Accept-Language, то нужно обеспечить кэширование отдельных копий для каждого языка, предпочитаемого клиентом:

```
<%@ OutputCache Duration="360" VaryByParam="none"
                VaryByHeader="Accept-Language" %>
```

Рассмотрим вопросы, связанные с кэшированием данных. Для доступа к кэшу разработчик может использовать свойство Cache класса Page. Область видимости кэша данных ограничена приложением. Его возможности во многом идентичны возможностям хранилища HttpApplicationState при двух важных отличиях. Во-первых, по умолчанию не гарантируется, что информация, размещенная в кэше данных, сохраняется в нем. Разработчик всегда должен быть готов к тому, что в кэше нет нужной информации. Во-вторых, кэш не предназначен для хранения обновляемых пользователями данных. Кэш позволяет читать данные многим потокам и записывать одному потоку, но не существует методов, подобных Lock() и UnLock(), позволяющих разработчику управлять блокировками. Следовательно, кэш данных предназначен для хранения данных в режиме «только для чтения» в целях облегчения доступа к ним.

Простейший вариант использования свойства Cache заключается в работе с ним как с коллекцией, аналогично работе со свойствами страницы Session и Application. Однако при каждом добавлении некоторого объекта в кэш одновременно в кэш заносится экземпляр класса CacheEntry, описывающий кэшируемый объект. Основные свойства класса CacheEntry представлены в таблице 56. Обратите внимание на возможность связывания кэшируемых объектов с другими элементами кэша или файлами:

Таблица 56

Свойства класса CacheEntry

Свойство	Тип	Описание
Key	Строка	Уникальный ключ для идентификации раздела кэша
Dependency	CacheDependency	Зависимость данного раздела от файла, каталога или другого раздела (при их изменении раздел кэша будет удален)
Expires	DateTime	Фиксированные дата и время, при достижении которых раздел будет удален
SlidingExpiration	TimeSpan	Интервал времени между последним обращением к разделу и его удалением
Priority	CacheItemPriority	Важность сохранения данного раздела по сравнению с другими разделами (используется при чистке кэша)
OnRemoveCallback	CacheItem-RemovedCallback	Делегат, который может запускаться при удалении раздела

Если для помещения объекта в кэш используется индексатор по умолчанию, то свойства соответствующего объекта CacheEntry принимают следующие значения: время жизни устанавливается бесконечным, скользящее время жизни равно нулю, значение Priority равно Normal, а OnRemoveCallback – null. При этом объект в кэше может оставаться как угодно долго, пока его не удалит процедура очистки (обычно вследствие нехватки памяти) или пока он не будет удален явно.

Если возникла необходимость управлять свойствами CacheEntry, то можно воспользоваться одной из перегруженных версий методов Insert() или Add(). В следующем примере показан файл global.asax, добавляющий содержимое файла в кэш при запуске приложения. Раздел кэша становится неправильным, когда изменяется содержимое файла, использованного для заполнения раздела, поэтому в кэш добавлено свойство CacheDependency. В сценарии также зарегистрирован метод обратного вызова, позволяющий получить извещение об удалении данных из кэша. Кроме этого, для добавленного раздела отключен срок действия, отключен режим скользящего времени жизни и задан приоритет по умолчанию:

```
<%@ Application Language="C#" %>
<script runat="server">
  public void OnRemovePi(string key, object val,
                         CacheItemRemovedReason r) {
    // Некоторые действия, выполняемые при удалении элемента.
```

```
    // Например, помещение элемента снова в кэш.
  }

  void Application_Start(object sender, EventArgs e) {
    System.IO.StreamReader sr =
                new System.IO.StreamReader(Server.MapPath("pi.txt"));
    string pi = sr.ReadToEnd();
    CacheDependency piDep =
                new CacheDependency(Server.MapPath("pi.txt"));
    Context.Cache.Add("pi", pi, piDep,
                      Cache.NoAbsoluteExpiration,
                      Cache.NoSlidingExpiration,
                      CacheItemPriority.Default,
                      new CacheItemRemovedCallback(OnRemovePi));
  }
</script>
```

Любая страница этого приложения может обращаться к кэшу данных по ключу `pi`. Кроме этого, гарантируется согласованность раздела `pi` с содержимым файла `pi.txt`.

```
<!-- Файл: default.aspx -->
<%@ Page Language="C#" %>
<script runat="server">
    protected void Page_Load(object src, EventArgs e) {
        if (Cache["pi"] == null) pi.Text = "Undefined!";
        else pi.Text = (string)Cache["pi"];
    }
</script>

<html>
<body>
    <form runat="server">
        <h1> Страница pi </h1>
        <asp:TextBox ID="pi" runat="server" />
    </form>
</body>
</html>
```

5.13. БЕЗОПАСНОСТЬ В WEB-ПРИЛОЖЕНИЯХ

Аутентификацией (*authentication*) называется процесс идентификации пользователей приложений. *Авторизация* (*authorization*) – это процесс предоставления доступа пользователям на основе их идентификационных данных. Аутентификация наряду с авторизацией представляют собой средства защиты web-приложений от несанкционированного доступа.

Для аутентификации пользователей может применяться сервер IIS. По умолчанию IIS не идентифицирует пользователей, а обрабатывает все запросы как анонимные с учетной записью `IUSR_`*`имя_компьютера`*. Просмотр и изменение привилегий учетной записи для анонимного доступа выполняется с помо-

щью оснастки *Computer Management* (Управление компьютером). В частности, можно изменить членство этой учетной записи в группах.

Для отдельного виртуального каталога можно задать особый режим аутентификации, использовав окно свойств IIS (*Безопасность каталога*).

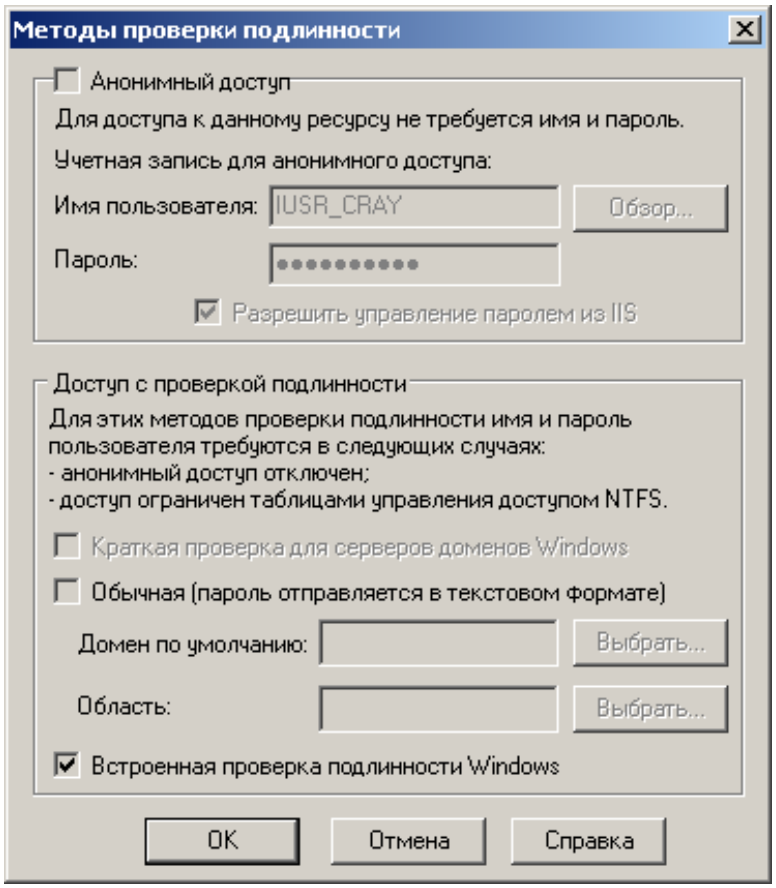

Рис. 25. Свойства безопасности виртуального каталога

Заметим, что большинство web-приложений не использует аутентификацию на основе IIS, так как для этого нужно, чтобы клиент выполнил некие дополнительные действия, например, получил сертификат или учетную запись на сервере. Поэтому в данном параграфе подобный способ обеспечения безопасности подробно рассматриваться не будет.

Рассмотрим средства аутентификации ASP.NET[46]. Отметим, что если клиент аутентифицируется, то информация о нем доступна посредством свойства `User` класса `Page` или класса `HttpContext`. Свойство `User` указывает на реализацию интерфейса `IPrincipal`. Этот интерфейс содержит одно свойство и один метод. Свойство `Identity` является указателем на реализацию интерфейса `IIdentity`, а метод `IsInRole()` проверяет членство клиента в указанной группе. Ниже приведено описание интерфейса `IIdentity`:

[46] Методы аутентификации ASP.NET применяются для файлов, являющихся частью Web-приложения. HTML-страницы (`*.htm` или `*.html`) не включаются в число этих файлов автоматически. Такие страницы обрабатываются IIS, а не ASP.NET. Чтобы зарегистрировать HTML-страницы для обработки рабочим процессом ASP.NET, требуется соответствующим образом настроить IIS.

```
public interface IIdentity {
    string AuthenticationType { get; }
    bool IsAuthenticated { get; }
    string Name { get; }
}
```

Свойство IsAuthenticated используется для различения аутентифицируемых и анонимных клиентов. Свойство Name можно применить для выяснения идентичности клиента. Если клиент аутентифицируется, то свойство AuthenticationType позволит выяснить тип аутентификации. В следующем листинге представлена страница, которая выводит аутентификационную информацию клиента:

```
<%@ Page Language="C#"%>
<script runat="server">
    void Page_Load(object src, EventArgs e)
    {
        AuthUser.Text = User.Identity.Name;
        AuthType.Text = User.Identity.AuthenticationType;
    }
</script>

<html>
<body>
    <h1>Страница тестирования безопасности</h1>
    <h3>пользователь:</h3>
    <asp:Label ID="AuthUser" runat="server" />
    <h3>тип аутентификации:</h3>
    <asp:Label ID="AuthType" runat="server" />
</body>
</html>
```

Задать требования аутентификации клиентов web-приложения можно путем добавления соответствующих элементов в конфигурационный файл приложения web.config. Тип аутентификации клиентов задается с помощью элемента authentication, атрибут mode которого может принимать одно из четырех значений: Windows (по умолчанию), Forms, Passport и None. Если задан режим Windows, то все задачи аутентификации перекладываются на web-сервер.

Сконфигурировав тип аутентификации на использование ASP.NET, нужно продумать цели аутентификации. Аутентификация выполняется для ограничения доступа клиентов ко всему приложению или к его частям с помощью элемента authorization. В этот элемент добавляются подэлементы allow и deny, задающие предоставление или отказ в доступе отдельным пользователям и ролям. Метасимвол * используется для представления всех пользователей, а ? – для представления анонимных пользователей. Следующий пример конфигурационного файла отказывает анонимным пользователям в праве доступа к сайту:

```
<configuration>
  <system.web>
    <authorization>
      <deny users="?" />
```

```
    </authorization>
  </system.web>
</configuration>
```

Элементы allow и deny поддерживают три атрибута: users, roles и verbs. Значениями этих атрибутов могут быть разделенные запятыми списки пользователей, ролей и команд. Это позволяет сконфигурировать сложные требования безопасности. Например, в следующем фрагменте пользователям MyDomain\alice и MyDomain\bob предоставлен полный доступ к приложению, все «именованные» пользователи имеют право передавать запросы GET, анонимным пользователям доступ к приложению запрещен:

```
<authorization>
  <allow users="MyDomain\alice, MyDomain\bob"/>
  <allow users="*" verbs="GET"/>
  <deny users="?"/>
</authorization>
```

Вернемся к рассмотрению различных типов аутентификации и остановимся на режиме аутентификации Forms. Принцип действия аутентификации в этом режиме следующий:

1. Когда пользователь первый раз запрашивает ресурс, требующий аутентификации, сервер перенаправляет запрос в выделенную страницу регистрации.
2. Регистрационная страница принимает данные пользователя (как правило, имя и пароль), а затем приложение аутентифицирует пользователя (предположительно с помощью базы данных).
3. Если пользователь успешно зарегистрировался, сервер предоставляет ему аутентификационный файл cookie в зашифрованном виде, который действителен на протяжении сеанса (но может быть сохранен на клиентском компьютере для использования в последующих сеансах).
4. Пользователь перенаправляется к интересующему его ресурсу, однако теперь он предоставляет в запросе идентификационный cookie и получает доступ.

Если в файле web.config задать аутентификацию Forms, то дополнительные настройки режима можно выполнить во вложенном элементе forms, атрибуты которого содержит таблица 57.

Таблица 57

Атрибуты элемента forms

Атрибут	Значения	Значение по умолчанию	Описание
name	Строка	.ASPXAUTH	Имя файла cookie
loginUrl	Адрес URL	login.aspx	Адрес URL страницы регистрации
protection	All, None, Encryption, Validation	All	Режим защиты файла cookie
timeout	Минуты	30	Скользящее время жизни файла cookie (переустанавливается при каждом запросе)
path	Маршрут	/	Маршрут файла cookie

Пример файла `web.config`, конфигурирующего аутентификацию в режиме `Forms`, показан в следующем листинге:

```
<configuration>
  <system.web>
    <authorization>
      <deny users="?"/>
    </authorization>

    <authentication mode="Forms">
      <forms loginUrl="login.aspx"/>
    </authentication>
  </system.web>
</configuration>
```

Теперь, чтобы аутентификация на основе cookie заработала, нужно реализовать страницу регистрации. Для реализации страницы регистрации платформа ASP.NET предоставляет класс `FormsAuthentication`. Этот класс состоит главным образом из статических методов, управляющих аутентификационными файлами cookie. Например, чтобы предоставить такой файл клиенту, нужно вызвать метод `SetAuthCookie()`. Метод `RedirectFromLoginPage()` предоставляет cookie клиенту и перенаправляет клиента в запрошенную им страницу.

Приведем пример страницы регистрации. Обратите внимание на наличие флажка, позволяющего запомнить данные клиента для следующих сеансов.

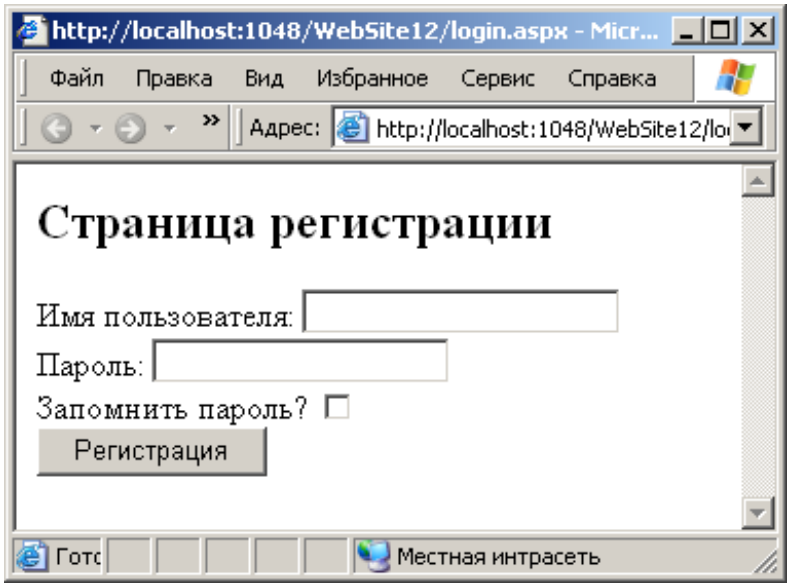

Рис. 26. Страница регистрации

```
<!-- Файл login.aspx -->
<%@ Page Language="C#" %>
<script runat="server">
  void OnClick_Login(object src, EventArgs e) {
    if ((m_usrname.Text == "Alex") && (m_pass.Text == "pass"))
      FormsAuthentication.RedirectFromLoginPage(m_usrname.Text,
                                                m_save_pass.Checked);
    else msg.Text = "Неверно. Попытайтесь еще раз";
```

```
  }
</script>

<html>
<body>
  <form runat="server">
    <h2>Страница регистрации</h2>
    Имя пользователя:
    <asp:TextBox ID="m_usrname" runat="server" /><br />
    Пароль:
    <asp:TextBox ID="m_pass" TextMode="Password"
                 runat="server"/> <br/>
    Запомнить пароль?
    <asp:CheckBox ID="m_save_pass" runat="server" /><br />
    <asp:Button Text="Регистрация" OnClick="OnClick_Login"
                runat="server" /><br />
     <asp:Label ID="msg" runat="server" />
  </form>
</body>
</html>
```

Рассмотрим еще один пример. Достаточно часто клиенту предоставляется возможность идентифицировать себя, если он этого хочет, или возможность анонимного доступа в противном случае. В этом случае имеет смысл интегрировать форму регистрации в страницу. При регистрации вызывается метод `FormsAuthentication.SetAuthCookie()` и выполняется перенаправление на эту же страницу. При загрузке страницы проверяется, идентифицирован ли пользователь, и если да, то выводиться специфичная для пользователя информация. Текст страницы `default.aspx` представлен ниже:

```
<!-- Файл default.aspx -->
<%@ Page Language="C#"%>
<script runat="server">
  void Page_Load(object src, EventArgs e) {
    if (User.Identity.Name == "Alex") msg.Text = "Hello Alex";
    else msg.Text = "General Info";
  }
  void OnClick_Login(object src, EventArgs e) {
    if((m_usrname.Text == "Alex") && (m_pass.Text == "pass"))
      FormsAuthentication.SetAuthCookie(m_usrname.Text,false);
    Response.Redirect("default.aspx");
  }
</script>

<html>
<body>
  <form id="Form1" runat="server">
    <h2>Регистрация</h2>
    Имя :
    <asp:TextBox ID="m_usrname" runat="server" /><br />
    Пароль:
    <asp:TextBox ID="m_pass" TextMode="Password"
```

```
                runat="server"/> <br/>
    <asp:Button ID="Button1" Text="Войти"
                OnClick="OnClick_Login" runat="server"/><br />
     <asp:Label ID="msg" runat="server" />
  </form>
</body>
</html>
```

Файл `web.config`:

```
<configuration>
  <system.web>
    <authentication mode="Forms" />
  </system.web>
</configuration>
```

5.14. СОЗДАНИЕ ПОЛЬЗОВАТЕЛЬСКИХ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ

Среда ASP.NET позволяет программисту создавать собственные элементы управления. Выделяют два вида элементов: *пользовательские элементы управления* (*User Controls*, *UC*) и *серверные элементы* (*Server Custom Controls*, *SCC*). Фактически, UC представляют собой фрагменты обыкновенной `aspx`-страницы, помещенные в специальную именованную оболочку. Процесс их создания в целом подобен процессу визуального дизайна. SCC – это полноценные классы, размещенные в отдельных сборка. Процесс создания SCC является невизуальным, и возможностей для тонкой настройки они предоставляют больше.

Пользовательский элемент управления может быть создан декларативно в текстовом или HTML редакторе. Декларативный синтаксис пользовательского элемента управления очень похож на синтаксис, используемый при создании страниц Web Forms. Основным отличием является то, что пользовательский элемент управления не включает элементы `<html>`, `<body>` и `<form>`. В качестве примера UC приведем компонент для ввода имени пользователя и пароля. Ниже приведен код компонента (файл `LogonForm.ascx`[47]).

```
<script language="C#" runat="server">
    public string UserId {
        get { return User.Text; }
        set { User.Text = value; }
    }
    public string Password {
        get { return Pass.Text; }
        set { Pass.Text = value; }
    }
</script>

<table style="BORDER-RIGHT: black 1px solid;
              BORDER-TOP: black 1px solid;
              BORDER-LEFT: black 1px solid;
              BORDER-BOTTOM: black 1px solid;
```

[47] В Visual Studio можно начать новый web-проект, добавить в проект новый элемент `Web User Control` (файл `LogonForm.ascx`) и размещать элементы в дизайнере или ввести код.

```
                FONT: 10pt verdana" cellspacing="15">
      <TR>
        <TD><B>Login: </B></TD>
        <TD>
          <asp:TextBox id="User" runat="server" Width="144px" />
        </TD>
      </TR>
      <TR>
        <TD><B>Password: </B></TD>
        <TD>
          <asp:TextBox id="Pass" runat="server" Width="144px"
                       TextMode="Password" />
        </TD>
      </TR>
      <TR>
        <TD></TD>
        <TD>
          <asp:Button id="Button1" runat="server" Text="Submit" />
        </TD>
      </TR>
    </table>
```

Создадим страницу, на которой будет использоваться элемент управления. При написании такой страницы необходимо использовать директиву `@Register` для указания следующих параметров:

- `Src` – имя ascx-файла, содержащего элемент управления;
- `TagName` – имя тэга, идентифицирующего элемент управления;
- `TagPrefix` – префикс для тэгов, ссылающихся на элемент управления.

При размещении элемента на странице он должен быть снабжен атрибутом `runat="server"`.

```
<%@ Register TagPrefix="EPAM" TagName="Login"
             Src="~\LogonForm.ascx" %>
<html>
  <body>
    <form id="Form1" method="post" runat="server">
      <EPAM:Login id="Login1" runat="server" />
    </form>
  </body>
</html>
```

Внешний вид созданной страницы показан на рисунке 27.

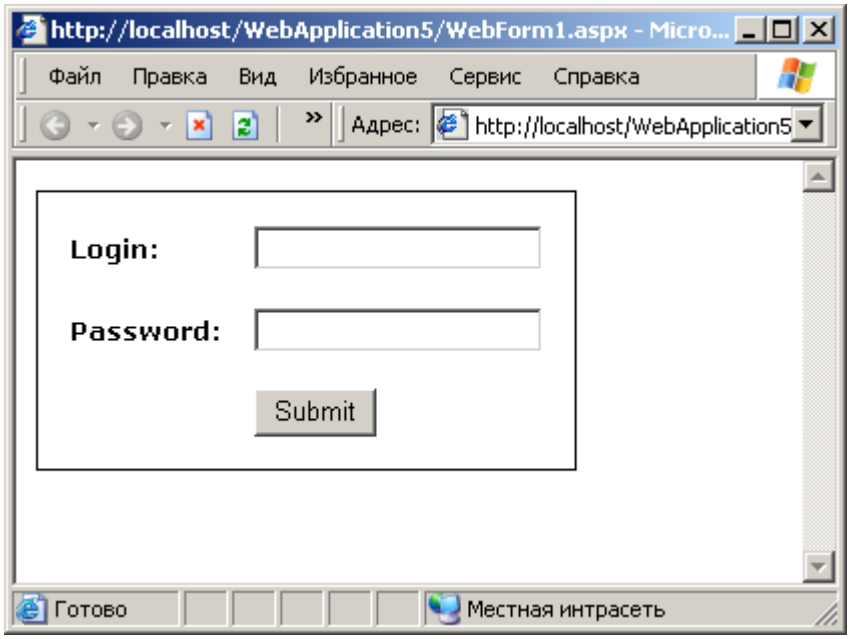

Рис. 27. Страница с элементом `LogonForm.ascx`

Вообще говоря, существует универсальный «рецепт» преобразования любой aspx-страницы в пользовательский элемент управления. Чтобы сделать это:

1. Удалите все элементы `<html>`, `<body>` и `<form>` из страницы.
2. Если в aspx-странице есть директива `@Page`, замените ее директивой `@Control`. Чтобы избежать ошибки синтаксического разбора при преобразовании страницы в элемент управления, удалите все атрибуты, поддерживаемые директивой `@Page`, которые не поддерживаются директивой `@Control`.
3. Включите атрибут `className` в директиву `@Control` (при желании). Это даст возможность присвоить строгий тип пользовательскому элементу управления при программном добавлении на страницу или к другому серверному элементу управления.
4. Присвойте элементу управления имя файла, которое отражает планы по его использованию, и измените расширение имени файла на `.ascx`.

UC похож на страницу и тем, что он может содержать обработчики собственных событий. Следующий пример демонстрирует элемент управления, инкапсулирующий `LinkButton`, который отображает текущее время. При щелчке на ссылке или загрузке страницы время обновляется. При разработке этого элемента управления мы поместим требуемый код в code-behind-файл. Также определим собственное свойство, отвечающее за формат отображения времени.

```
<%@ Control Codebehind="TimeDisplay.ascx.cs"
            Inherits="TimeDisplay"%>
<asp:LinkButton id="lnkTime" runat="server"
                OnClick="lnkTime_Click" />

using System;
using System.Web;
using System.Web.UI.WebControls;
public class TimeDisplay : System.Web.UI.UserControl {
```

```
    protected LinkButton lnkTime;
    private string format;
    public string Format {
        get { return format; }
        set { format = value; }
    }
    protected void Page_Load(object sender,
                             System.EventArgs e) {
        if (!Page.IsPostBack) RefreshTime();
    }
    protected void lnkTime_Click(object sender,
                                 System.EventArgs e) {
        RefreshTime();
    }
    public void RefreshTime() {
        if (format == "")
            lnkTime.Text = DateTime.Now.ToLongTimeString();
        else
            lnkTime.Text = DateTime.Now.ToString(format);
    }
}
```

Текст страницы с двумя элементами `TimeDisplay`, а также ее внешний вид приведены ниже

```
<%@ Register TagPrefix="EPAM" TagName="TimeDispl"
             Src="~\TimeDisplay.ascx" %>
<html>
  <body>
    <form id="Form1" method="post" runat="server">
      <EPAM:TimeDispl id="TD1" runat="server"
            Format="dddd, dd MMMM yyyy HH:mm:ss tt (GMT z)" />
      <hr />
      <EPAM:TimeDispl id="TD2" runat="server" />
    </form>
  </body>
</html>
```

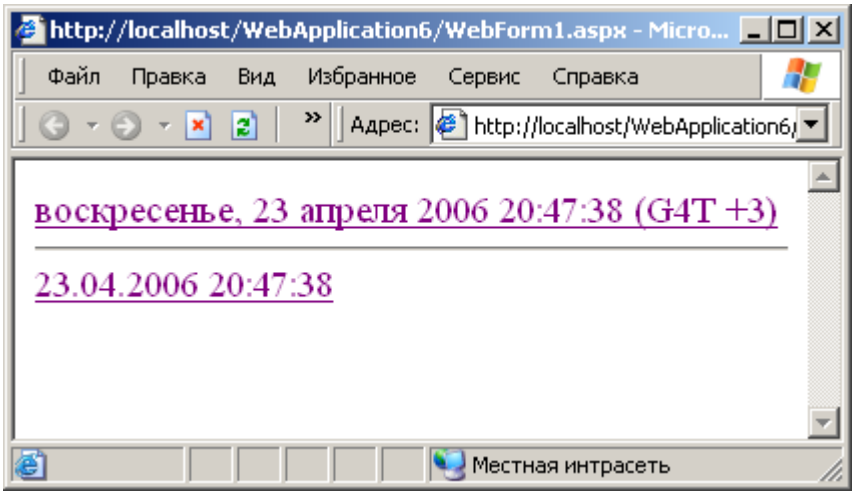

Рис. 28. Страница с элементами `TimeDisplay.ascx`

Последний аспект, который будет продемонстрирован для пользовательских элементов управления, это создание собственных событий. Пусть требуется создать элемент управления, содержащий две опции выбора – Yes и No, а также кнопку, при нажатии на которую генерируется определенное событие. Предполагается, что в это событие в качестве аргумента передается индекс выбранной опции (1 или 2).

Исходный текст и код элемента управления представлен ниже.

```
<%@ Control Codebehind="UserChoiceComponent.ascx.cs"
            Inherits="UserChoice"%>
<table style="BORDER-RIGHT: black 1px solid;
              BORDER-TOP: black 1px solid;
              BORDER-LEFT: black 1px solid;
              BORDER-BOTTOM: black 1px solid;
              FONT: 10pt verdana" cellspacing="15">
  <TR>
    <TD> <B>
      <asp:RadioButtonList id="RBL1" runat="server" />
         </B>
    </TD>
  </TR>
  <TR>
    <TD>
      <asp:Button id="B1" runat="server" Text="Submit"
                  onclick="B1_Click" />
    </TD>
  </TR>
</table>

using System;
using System.Web;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

public class UserChoice : System.Web.UI.UserControl {
    protected RadioButtonList RBL1;
    protected Button B1;
    protected void Page_Load(object sender,
                             System.EventArgs e) {
        if (!Page.IsPostBack) {
            string[] items = {"Yes", "No"};
            RBL1.DataSource = items;
            RBL1.DataBind();
        }
    }

    public event UserChoiceEventHandler UserChoosed;

    protected void B1_Click(object sender,System.EventArgs e){
        if (UserChoosed != null) {
            int item = RBL1.SelectedIndex + 1;
```

```
            UserChoiceEventArgs args =
                                    new UserChoiceEventArgs(item);
            UserChoosed(this, args);
        }
    }
}

public class UserChoiceEventArgs : EventArgs {
    private int selectedItem;
    public int SelectedItem {
        get { return selectedItem; }
    }
    public UserChoiceEventArgs(int item) {
        selectedItem = item;
    }
}

public delegate void UserChoiceEventHandler(object sender,
                                            UserChoiceEventArgs e);
```

Дадим некоторые комментарии. Для генерации события был создан собственный тип `UserChoiceEventArgs`, описывающий аргументы события. Это стандартный подход. Кроме этого, был объявлен соответствующий открытий делегат, а в классе компонента – событие `UserChoosed`. Компонент содержит два обработчика внутренних событий. В обработчике события `Page_Load()` происходит заполнение отображаемого списка. В обработчике события нажатия на кнопку проверяется, назначен ли внешний обработчик для `UserChoosed`. Если это так, то создается требуемый аргумент и генерируется это событие.

Пример кода страницы с элементом `UserChoice` показывает, как назначить обработчик события.

```
<%@ Register TagPrefix="EPAM" TagName="UserChoice"
             Src="~\UserChoiceComponent.ascx" %>
<script runat="server">
void UserChoice1_UserChoosed(object sender,
                             UserChoiceEventArgs e) {
    Label1.Text = e.SelectedItem.ToString();
}
</script>

<HTML>
    <body>
        <form id="Form1" method="post" runat="server">
            <EPAM:UserChoice id="UserChoice1" runat="server"
                    OnUserChoosed="UserChoice1_UserChoosed" />
            <hr>
            <asp:Label id="Label1" runat="server" />
        </form>
    </body>
</HTML>
```

Как видим, назначаются обработчики стандартным способом.

Рассмотрим процесс создания серверного элемента управления. По сути, создание такого элемента представляет собой разработку пользовательского класса. Естественно, что такой класс не создается «с нуля», а является наследником некоторых стандартных классов:

- `System.Web.UI.Control` – базовый класс для любого (пользовательского или стандартного) элемента управления;
- `System.Web.UI.WebControls.WebControl` – класс, содержащий методы и свойства для работы со стилем представления. Этот класс является потомком класса `System.Web.UI.Control`, от него, в свою очередь, порождены все элементы управления Web Controls;
- `System.Web.UI.HtmlControls.HtmlControl` – базовый класс для стандартных элементов HTML, таких как `input`;
- `System.Web.UI.TemplateControl` – базовый класс для страниц и пользовательских элементов управления User Controls.

Построим простейший серверный элемент управления. Создадим библиотеку классов и включим в нее следующий код:

```
using System;
using System.Web.UI;

namespace EPAMControls {
    public class FirstControl : Control {
        protected override void Render(HtmlTextWriter writer)
        {
            writer.Write("<h1>ASP.NET Custom Control</h1>");
        }
    }
}
```

Наш элемент управления очень прост. Это класс `EPAMControls.FirstControl`, в котором перекрыт виртуальный метод `Render()`. Именно этот метод вызывается исполняющей средой при необходимости отрисовки страницы и всех элементов управления, расположенных на ней. В методе `Render()` выполняется вывод HTML-кода при помощи средств объекта класса `System.Web.UI.HtmlTextWriter`.

Создадим страницу, на которой будет использоваться элемент `FirstControl`. При написании такой страницы необходимо использовать директиву `@Register` для указания следующих параметров:

- `Assembly` – имя сборки с скомпилированным серверным элементом управления. Если указано слабое имя сборки, то она должна быть расположена в подкаталоге `~/bin` web-приложения;
- `Namespace` – имя пространства имен, содержащего написанный класс;
- `TagPrefix` – префикс для тэгов, ссылающихся на элемент управления.

При размещении элемента на странице он должен быть снабжен атрибутом `runat="server"`.

```
<%@ Page language="c#" %>
<%@ Register TagPrefix="EPAM" Namespace="EPAMControls"
```

```
                Assembly="EPAMControls" %>

<html>
    <body>
        <EPAM:FirstControl runat="server" />
    </body>
</html>
```

Внешний вид страницы в браузере показан на рис. 29.

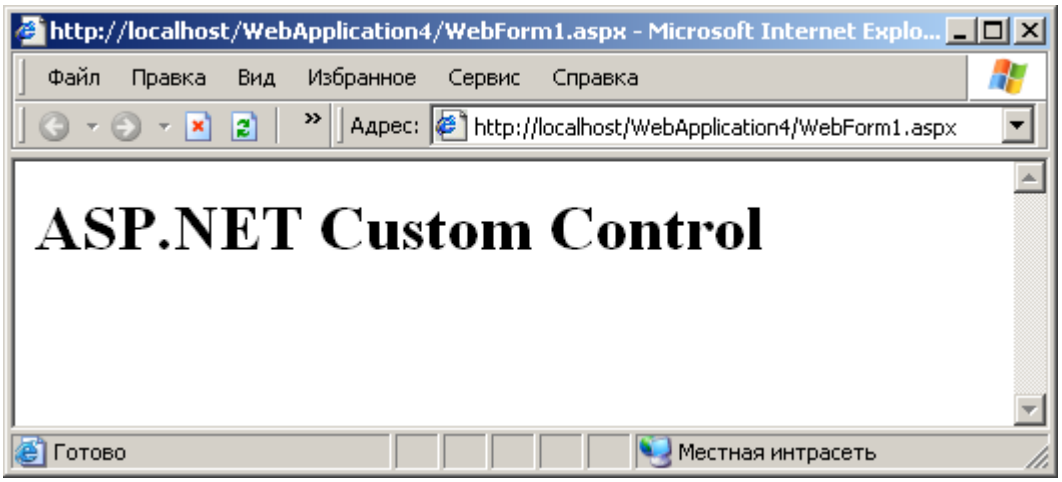

Рис. 29. Страница с пользовательским элементом управления

Как вы уже знаете, настройка любого элемента управления при размещении его на странице выполняется при помощи атрибутов. ASP.NET преобразует значения атрибутов в значения соответствующих открытых свойств класса, представляющего элемент управления. Если тип свойства строковый, то происходит простое копирование значения атрибута в свойство. В случае числовых или булевских типов выполняется преобразование строки в соответствующий тип. Если тип свойства – некое перечисление, то строка атрибута задает имя элемента этого перечисления. Особый случай – использование в качестве типа свойств классов. Для доступа к свойству агрегированного объекта требуется применять формат `Имя сложного свойства-Имя подсвойства`. Например:

```
<EPAM:Control runat="server" Font-Color="Red"/>
```

Добавим некоторые свойства в класс `FirstControl`. Свойство `Text` будет содержать отображаемый текст, свойство `RepeatCount` – это количество повторений текста, свойство `ForeColor` – цвет текста (перечисление `System.Drawing.Color`). Заметим, что после задания всех свойств у элемента управления вызывается виртуальный метод `OnInit()`. Его можно переопределить для задания значения свойств по умолчанию или для контроля диапазона значений свойства.

```
using System;
using System.Web.UI;
using System.Drawing;

namespace EPAMControls {
    public class FirstControl : Control {
```

```
        private string text = "Default Text";
        private int repeatCount = 1;
        private Color foreColor = Color.Blue;

        public string Text {
            get { return text; }
            set { text = value; }
        }

        public int RepeatCount {
            get { return repeatCount; }
            set { repeatCount = value; }
        }

        public Color ForeColor {
            get { return foreColor; }
            set { foreColor = value; }
        }

        protected override void OnInit(EventArgs e) {
            base.OnInit(e);
            if ((repeatCount < 1) || (repeatCount > 10))
                throw new ArgumentException(
                                "RepeatCount is out of range");
        }

       protected override void Render(HtmlTextWriter writer) {
            for(int i = 1; i <= repeatCount; i++) {
                writer.Write("<h1 style='color:" +
                    ColorTranslator.ToHtml(foreColor) +
                    "'>" + text + "</h1>");
            }
        }
    }
}
```

Исходный код тестовой страницы и ее вид в браузере представлены ниже:

```
<%@ Page language="c#" %>
<%@ Register TagPrefix="EPAM" Namespace="EPAMControls"
             Assembly="EPAMControls" %>

<html>
  <body>
     <EPAM:FirstControl runat="server" />
     <EPAM:FirstControl runat="server" Text="Hello"
                       ForeColor="Red" RepeatCount="3"/>
  </body>
</html>
```

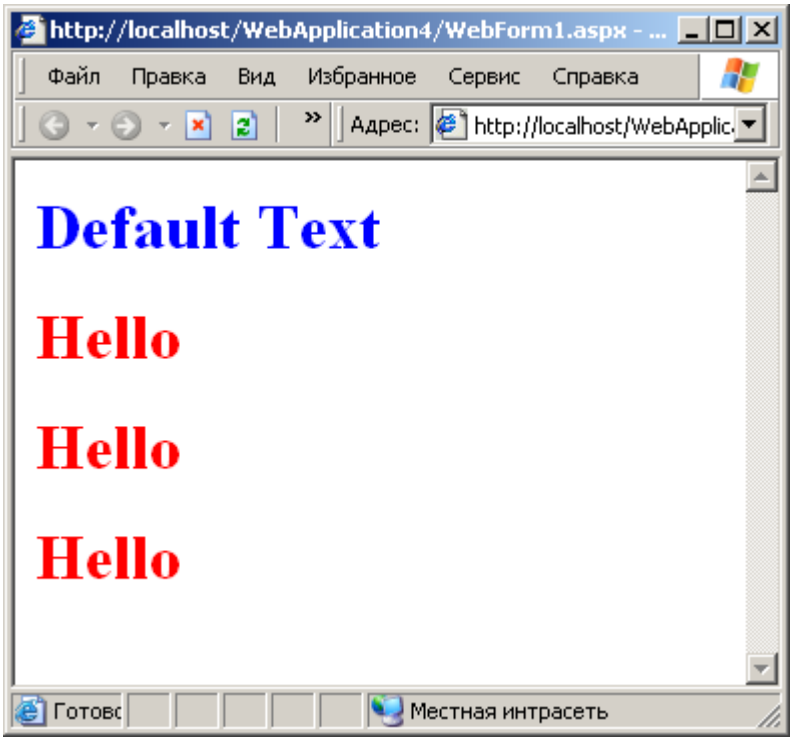

Рис. 30. Пользовательский элемент со свойствами

При сложной схеме отображения, применяемой в методе `Render()`, удобным является использование дополнительных методов класса `HtmlTextWriter`, упрощающих построение HTML-кода. Эти методы перечислены в таблице 58.

Таблица 58

Методы класса `HtmlTextWriter`

Имя метода	Описание
`AddAttribute()`	Добавляет атрибут к следующему элементу HTML, который будет сформирован
`AddStyleAttribute()`	Добавляет соответствующий элемент для атрибута `style`
`RenderBeginTag()`	Формирует открывающий тег HTML-элемента
`RenderEndTag()`	Формирует замыкающий тег HTML-элемента и записывает этот элемент, а также любые атрибуты, формирование которых еще не закончено. После вызова этого метода все атрибуты считаются сформированными
`Write()`	Немедленно записывает строку
`WriteAttribute()`	Немедленно записывает HTML-атрибут
`WriteBeginTag()`	Немедленно записывает открывающий тег HTML-элемента
`WrileEndTag()`	Немедленно записывает замыкающий тег HTML-элемента
`WriteFullBeginTag()`	Немедленно записывает начальный тег вместе с замыкающей скобкой (>)
`WriteLine()`	Немедленно записывает строку содержимого. Эквивалент метода `Write()`, отличающийся лишь добавлением к концу строки символа перехода на новую строку

Применяя описанные методы класса `HtmlTextWriter`, метод `Render()` можно переписать следующим образом:

```
protected override void Render(HtmlTextWriter writer) {
    for(int i = 1; i <= repeatCount; i++)
```

```
        {
            writer.AddStyleAttribute(HtmlTextWriterStyle.Color,
                                     ColorTranslator.ToHtml(foreColor));
            writer.RenderBeginTag("h1");
            writer.Write(text);
            writer.RenderEndTag();
        }
    }
```

Если пользовательский элемент управления использует стили отображения, то имеет смысл рассмотреть в качестве базового класса для такого элемента класс `WebControl` из пространства имен `System.Web.UI.WebControls`. Особенностями данного класса являются:

- возможность задания цвета и шрифта отображения элемента при помощи таких свойств как `Font`, `ForeColor`, `BackColor`;
- поддержка сохранения ViewState элемента управления – в случае с классом `System.Web.UI.Control` такую поддержку необходимо реализовывать самостоятельно, переопределив методы `SaveViewState()` и `LoadViewState()`;
- возможность задать дополнительные атрибуты элемента управления, не соответствующие открытым свойствам, а записываемые непосредственно в выходной поток HTML.

Перепишем рассматриваемый элемент управления, использовав `WebControl`. Для этого:

1. Удалим из класса свойства, связанные со стилем, так как `WebControl` их уже поддерживает.

2. Объявим открытый конструктор, вызывающий конструктор базового класса с указанием HTML-тэга, соответствующего нашему элементу управления.

3. Переопределим метод `RenderContents()`, чтобы задать содержимое HTML-тэга.

4. Для реализации свойства `RepeatCount` перекроем базовый метод `Render()`, вызвав в нем отрисовку необходимое количество раз.

```
using System;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace EPAMControls {
    public class SecondControl : WebControl {
        private string text = "Default Text";
        private int repeatCount = 1;

        public string Text {
            get { return text; }
            set { text = value; }
        }

        public int RepeatCount {
            get { return repeatCount; }
```

```
            set { repeatCount = value; }
        }

        protected override void OnInit(EventArgs e) {
            base.OnInit(e);
            if ((repeatCount < 1) || (repeatCount > 10))
                throw new ArgumentException(
                                "RepeatCount is out of range");
        }

        public SecondControl() : base("H1") { }

        protected override void RenderContents(
                                        HtmlTextWriter writer)
        {
            writer.Write(text);
        }

       protected override void Render(HtmlTextWriter writer) {
            for(int i = 1; i <= repeatCount; i++)
                base.Render(writer);
        }
    }
}
```

Серверный элемент управления может быть контейнером для других элементов управления. Класс `Control` имеет свойство `Controls` типа `ControlsCollection`. Когда элемент управления визуализируется, вызывается каждый из его вложенных элементов, чтобы визуализировать себя.

По умолчанию, если серверный элемент управления наследуется от класса `Control`, то вложенные элементы управления, объявленные на странице ASP.NET, будут добавлены в коллекцию `Controls` (при условии, что не был использован атрибут `ParseChildren` для изменения такого поведения).

Пусть объявлен серверный элемент управления, который ничего не делает:

```
using System;
using System.Web.UI;
namespace EPAMControls {
    public class DoNothingControl : Control { }
}
```

Если использовать этот элемент на странице следующим образом, то в коллекцию `Controls` элемента `DoNothingControl` будут добавлены объекты классов `Button` (для кнопки) и `LiteralControl` (для текста `Some Text`).

```
<%@ Page Language="c#" %>
<%@ Register TagPrefix="EPAM" Namespace="EPAMControls"
             Assembly="EPAMControls" %>
<html>
<body>
    <form runat="server">
        <EPAM:DoNothingControl runat="server">
```

```
            <asp:Button Text="Click" runat="server" />
            Some Text
        </EPAM:DoNothingControl>
    </form>
</body>
</html>
```

Обычно то, какие дочерние элементы управления будут у серверного элемента, решает не пользователь, а создатель элемента. Для создания дочерних элементов требуется перекрыть метод `CreateChildControls()` и заполнить коллекцию `Controls`. Метод `CreateChildControls()` автоматически вызывается из метода `Render()`. Кроме этого, рекомендуется реализовать в классе серверного элемента управления интерфейс `INamingContainer`. Это интерфейс-метка, не содержащий свойств и методов. Реализация этого интерфейса обеспечивает получение уникальных идентификаторов (свойство `ID`) дочерними элементами управления.

В следующем примере создается элемент управления с текстом и кнопкой.

```
using System;
using System.Web.UI;
using System.Web.UI.WebControls;
namespace EPAMControls {
    public class CompositControl : Control, INamingContainer {
        protected override void CreateChildControls() {
            LiteralControl text =
                      new LiteralControl("<h1>Header</h1><p>");
            Button button = new Button();
            button.Text = "OK";
            Controls.Add(text);
            Controls.Add(button);
        }
    }
}
```

Схема 31 показывает, какие методы вызываются в процессе отрисовки элемента управления и могут быть перекрыты.

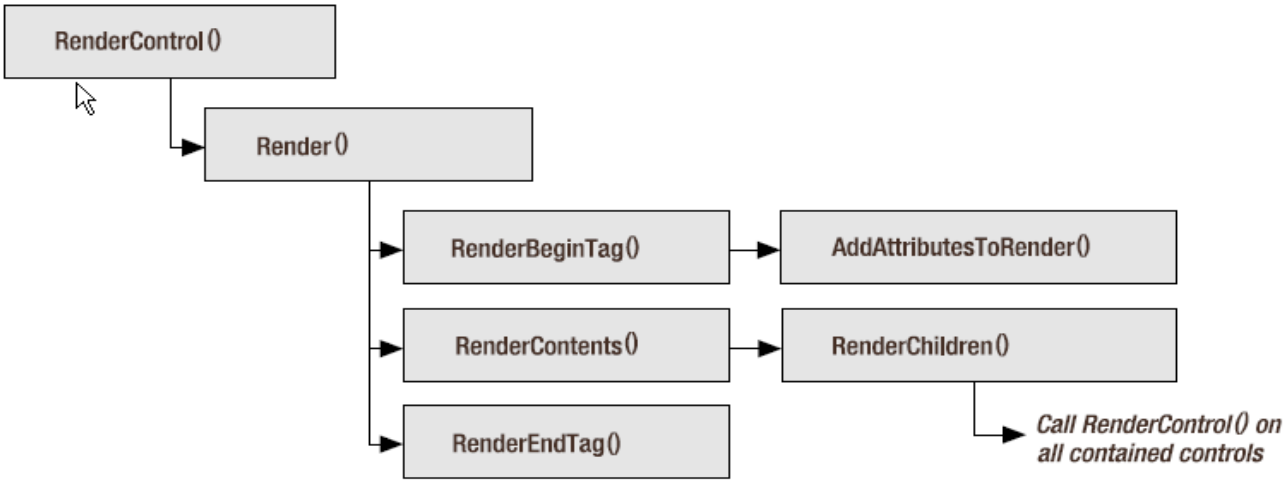

Рис. 31. Последовательность вызовов методов при отрисовке

ЛИТЕРАТУРА

1. R. Cameron, D. Michalk. Building ASP.NET Server Controls – APress, 2004.
2. B. Hamilton, M. MacDonald. ADO.NET in a Nutshell – O'Reilly, 2003.
3. Alex Homer et al. Professional ASP.NET 1.1 – Wrox Press, 2004.
4. Hoang Lam, Thuan L. Thai. .NET Framework Essentials – O'Reilly, 2003.
5. M. MacDonald, M. Szpuszta. Pro ASP.NET 2.0 in C# 2005 – APress, 2005.
6. S. McLean, J. Naftel, K. Williams. Microsoft .NET Remoting – Microsoft Press, 2003.
7. F. Muhammad, M. Milner. Real World ASP.NET Best Practices – APress, 2003.
8. Jeff Prosise. Programming Microsoft .NET (core reference) – Microsoft Press, 2002.
9. К. Ватсон. C#. – М.: Лори, 2005.
10. Дубовцев А. В. Microsoft .NET в подлиннике. – СПб.: БХВ-Петербург, 2004.
11. Рихтер Дж. Программирование на платформе Microsoft .NET Framework. – М.: Русская редакция, 2002.
12. Сеппа Д. Microsoft ADO.NET. – М.: Русская редакция, 2003