Министерство образования Республики Беларусь

Учреждение образования
"Полоцкий государственный университет"

И.В. Матюш, Г.А. Самошенков

ОСНОВЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ C# И РАБОТЫ С БАЗАМИ ДАННЫХ

КОНСПЕКТ ЛЕКЦИЙ

Новополоцк 2008

Министерство образования Республики Беларусь

Учреждение образования
«Полоцкий государственный университет»

М. В. МАТЮШ, Г. А. САМОЩЕНКОВ

ОСНОВЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ C# И РАБОТЫ С БАЗАМИ ДАННЫХ

КОНСПЕКТ ЛЕКЦИЙ

для студентов специальностей 1-40 01 01, 1-40 02 01 и слушателей переподготовки специальности 1-40 01 73 «Программное обеспечение информационных систем», слушателей повышения квалификации учебного центра «Компьютерные технологии и автоматизация»

Новополоцк
ПГУ
2008

УДК 004.432(075.8)
ББК 22.18я73
М33

Рекомендовано методической комиссией радиотехнического факультета в качестве конспекта лекций (протокол № 2 от 15.02.08)

Рецензенты:
Д. О. ГЛУХОВ – канд. техн. наук, проректор по информатизации;
С. В. КУХТА – зав. кафедрой информационных технологий

Матюш, М. В.

М33 Основы языка программирования C# и работы с базами данных: конспект лекций / М. В. Матюш, Г.А. Самощенков. – Новополоцк : ПГУ, 2008. – 92 с.

ISBN 978-985-418-676-4.

Расматриваются основы объектно-орентированного програмирования, основы языка программирования C# и его взаимодействие с базами данных.

Предназначен для для студентов специальностей 1-40 01 01, 1-40 02 01 и слушателей переподготовки специальности 1-40 01 73 «Программное обеспечение информационных систем», слушателей повышения квалификации учебного центра «Компьютерные технологии и автоматизация», а также студентов технических специальностей, изучающих языки программирования факультативно, начинающих программистов, использующих язык C#.

УДК 004.432(075.8)
ББК 22.18я73

ISBN 978-985-418-676-4

1 ОСНОВЫ ЯЗЫКА C#

1.1 СРЕДА РАЗРАБОТКИ

Окна документов обычно представлены вкладками в центре интегрированной среды разработки приложений – Visual Studio Integrated Development Environment (IDE) и содержат редакторы, дизайнеры. Окна инструментов – это окна IDE, которые предоставляют программистам обслуживающие функции: Solution Explorer, Class View, Properties и др. Окна инструментов отличаются от редакторов и дизайнеров тем, что находятся по бокам IDE (рис. 1.1).

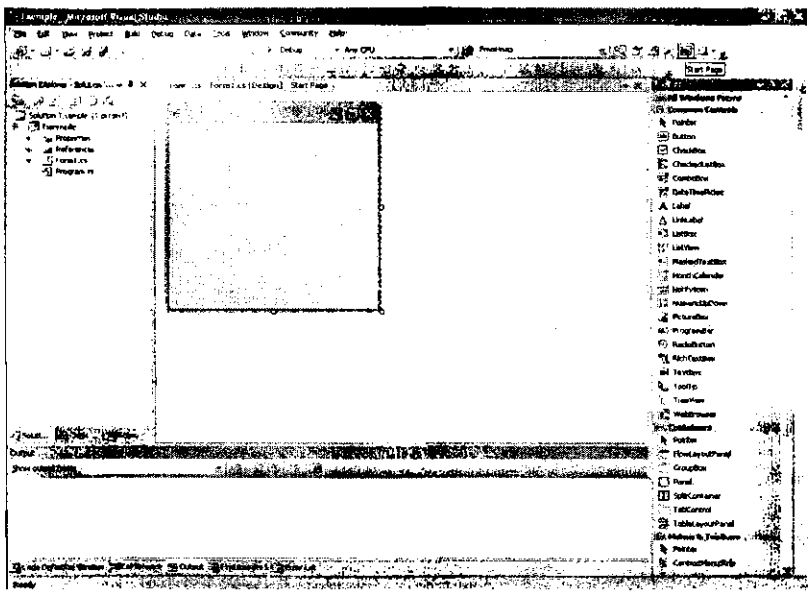

Рис. 1.1. Типичный вид решение в Visual Studio .NET

Окно Solution Explorer (Ctrl+Alt+L)

Solution Explorer (рис. 1.2) – самое важное окно инструментов в Visual Studio .NET. Решение – это набор проектов, каждый из которых является набором файлов. Solution Explorer обеспечит доступ к файлам ваших проектов. Именно здесь вы будете добавлять к проектам новые классы и файлы, и даже новые проекты.

Дизайнеры

Visual Studio .NET предлагает четыре главных типа дизайнеров: дизайнеры Windows Forms, позволяющие визуально создавать приложения Windows Forms; дизайнеры Web Forms, помогающие создавать приложения Web Forms; дизайнер компонентов, позволяющий создавать компоненты для серверных решений масштаба предприятия, и XML-дизайнер, облегчающий программистам работу с файлами определения схемы XML.

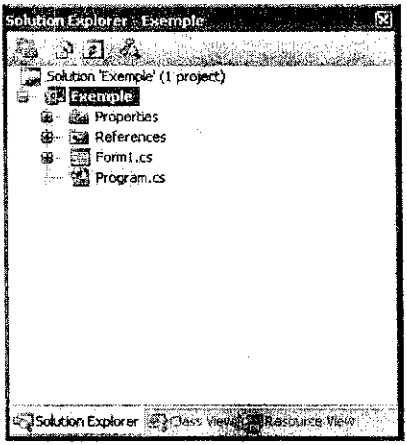

Рис. 1.2. Управляемый проект в Solution Explorer

Окна инструментов

Окна инструментов – это окна IDE, обеспечивающие вас информацией и служебными функциями во время работы. Среда IDE имеет массу разных окон инструментов, к которым позволяют обратиться клавиши быстрого доступа, окно Command и команды меню. Ниже приведены наиболее часто используемые в Visual Studio .NET окна инструментов.

Окно Class View (Ctrl+Shift+C)

Окно Class View (рис. 1.3) позволяет увидеть иерархию классов решения. Если ведется работа над большими проектами, удобнее перемещаться по решениям с помощью Class View, а не с помощью Solution Explorer.

Окно Properties (F4)

Это окно позволяет узнать или установить свойства элементов пользовательского интерфейса, добавляемых к приложениям Windows Forms и

Web Forms. В этом окне можно также установить свойства решений, проектов и файлов, предварительно выделив их в Solution Explorer. Окно Properties для установочного проекта проиллюстрировано на рис. 1.4.

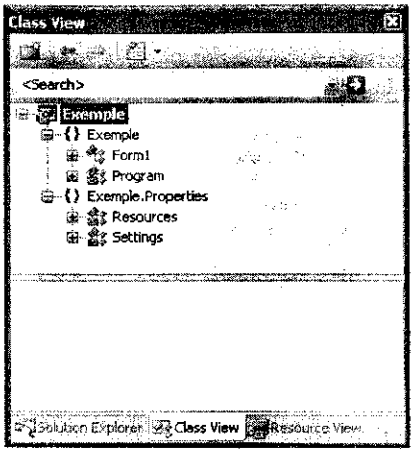

Рис. 1.3. Окно Class View

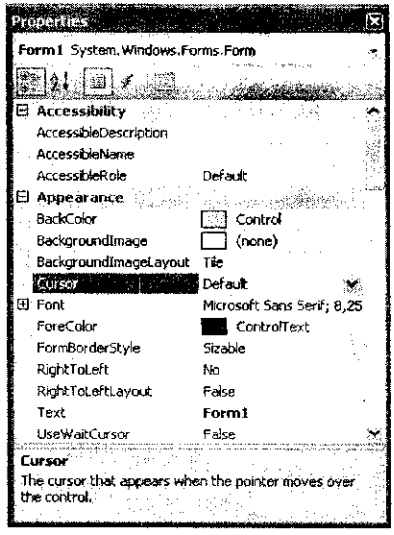

Рис. 1.4. Окно Properties

Окно Server Explorer (Ctrl+Alt+S)

Окно Server Explorer служит для доступа к источникам данных и информации на вашем локальном компьютере и на удаленном сервере. Из этого окна можно подключаться к данным, получать показания счетчиков производительности и информацию из журнала событий, а также управлять службами системы. Даже при локальном использовании этот инструмент способен экономить ваше время, позволяя легко запускать и останавливать системные службы и работать с системными журналами. На рис. 1.5 отображена база данных, доступная в Server Explorer на удалееном компьютере, и локальный на котором работает Visual Studio .NET.

Окно Toolbox (Ctrl+Alt+X)

В основном окно Toolbox в Visual Studio .NET используется для размещения элементов управления, добавляемых к форме в приложениях Windows Forms и Web Forms. Оно позволяет также размещать там часто используемые фрагменты своего кода или фрагменты справочных файлов

и Web-страниц, которые необходимо сохранить при их чтении. Clipboard Ring позволяет возвращаться к ранее сохраненным фрагментам текста и использовать их. Можно добавлять к этому окну свои вкладки и тем самым упорядочивать элементы управления и разрабатываемый код. На рис. 1.6 представлено окно Toolbox, которое открывает доступ к элементам управления вкладки Dialogs.

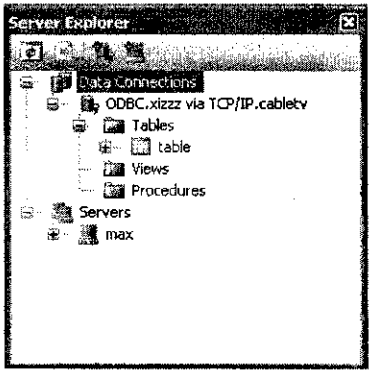

Рис. 1.5. Окно Server Explorer

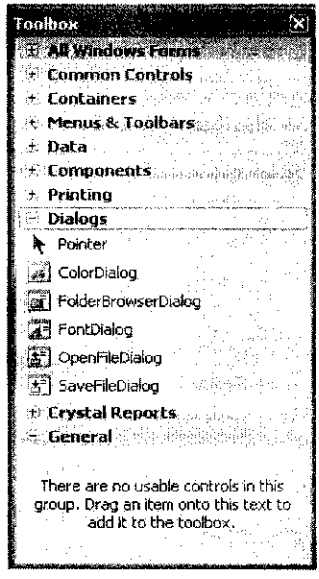

Рис. 1.6. Окно Toolbox

1.2 ТИПЫ ДАННЫХ, ЛИТЕРАЛЫ И ПЕРЕМЕННЫЕ

Типы данных имеют в языке C# особое значение, поскольку C# – строго типизированный язык. Это значит, что все операции проверяются компилятором на соответствие типов. Некорректные операции не компилируются. Для обеспечения контроля типов необходимо, чтобы все переменные, выражения и значения имели определенный тип.

Типы значений

Язык C# содержит две категории встроенных типов данных: типы значений и ссылочные типы. Ссылочные типы определяются в классах.

Ядро языка C# составляют 13 типов, перечисленных в табл. 1.1. Это – встроенные типы, которые определяются ключевыми словами языка C# и доступны для использования в любой сопрограмме. В языке C# строго определяется диапазон и поведение каждого типа значения.

Таблица 1.1

Типы значений в C#

Ключевое слово	Тип
Bool	Логический, или булевый, представляет значения ИСТИНА/ЛОЖЬ
Byte	8-разрядный целочисленный без знака
Char	Символьный
decimal	Числовой тип для финансовых вычислений
double	С плавающей точкой двойной точности
Float	С плавающей точкой
Int	Целочисленный
Long	Тип для представления длинного целого числа
sbyte	8-разрядный целочисленный со знаком
short	Тип для представления короткого целого числа
Uint	Целочисленный без знака
ulong	Тип для представления длинного целого числа без знака
ushort	Тип для представления короткого целого числа без знака

Размер значений в битах и диапазоны представления для каждого из этих восьми типов приведены в табл. 1.2.

Таблица 1.2

Характеристики целочисленных типов

Тип	Размер в битах	Диапазон
byte	8	от 0 до 255
sbyte	8	от – 128 до 127
short	16	от – 32 768 до 32 767
ushort	16	от 0 до 65 535
int	32	от – 2 147 483 648 до 2 147 483 647
uint	32	от 0 до 4 294 967 295
long	64	от – 9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
ulong	64	от 0 до 18 446 744 073 709 551 615

Согласно этой таблице в языке C# определены обе версии всех целочисленных типов: как со знаком, так и без него.

Символы

Для представления символов в языке C# используется Unicode (уникод), 16-разрядный стандарт кодирования символов, позволяющий представлять алфавиты всех существующих в мире языков.

Тип *bool* представляет значения ИСТИНА/ЛОЖЬ, которые в языке C# определяются зарезервированными словами *true* и *false*. Таким образом, переменная или выражение типа *bool* будет иметь одно из этих двух значений. В языке C# не определено ни одно преобразование значения типа *bool* в целочисленное значение. Например, число 1 не преобразуется в значение *true*, а число 0 – в значение *false*.

Литералы

В языке C# литералами называются фиксированные значения, представленные в понятной форме. Литералы также называют *константами.* Способ их представления зависит от их типа. Как упоминалось выше, символьные константы заключаются между двумя одинарными кавычками. Например, как «а» так и «%» – символьные константы. Целочисленные литералы задаются как числа без дробной части. Например, 10 и –100 – это целочисленные константы. Константы с плавающей точкой должны обязательно иметь десятичную точку, а за ней – дробную часть числа. Примером константы с плавающей точкой может служить число 11,123. Для вещественных чисел язык C# позволяет также использовать экспоненциальное представление (в виде мантиссы и порядка).

Поскольку язык C# – строго типизированный язык, литералы в нем также имеют тип.

Если тип, задаваемый по умолчанию в языке C#, не соответствует вашим намерениям в отношении типа конкретного литерала, вы можете явно определить его с помощью нужного суффикса. Чтобы задать литерал типа *long*, присоедините к его концу букву «*l*» или «*L*». Например, если значение 12 автоматически приобретает тип *int*, но значение 12*L* имеет тип *long*. Чтобы определить целочисленное значение без знака, используйте суффикс «*u*» или «*U*». Так, если значение 100 имеет тип *int*, но значение 100*U* – тип *uint*. Для задания длинного целого без знака используйте суффикс 'ul' или 'UL' (например, значение 987 654*UL* будет иметь тип *ulong*). Чтобы задать литерал типа *float*, используйте суффикс «*f*» или «*F*» (например, 10.19F). Чтобы задать литерал типа *decimal*, используйте суффикс «*m*» или «*M*» (например, 9,95*M*). Шестнадцатеричный литерал должен начинаться с пары символов «0*x*» (нуля и буквы «*x*»).

Управляющие последовательности символов

Среди множества символьных констант, образующихся в результате заключения символов в одинарные кавычки, помимо печатных символов есть такие (например, символ возврата каретки), которые создают проблему при использовании текстовых редакторов. Некоторые символы, например одинарная или двойная кавычка, имеют в языке C# специальное значение, поэтому их нельзя использовать непосредственно. По этим причинам в языке C# предусмотрено несколько управляющих последовательностей символов (ESC-последовательностей), перечисленных в табл. 1.3. Эти последовательности используются вместо символов, которые они представляют.

Таблица 1.3

Управляющие последовательности символов

ESC-последовательность	Описание
\a	Звуковой сигнал (звонок)
\b	Возврат на одну позицию
\f	Подача страницы (для перехода к началу следующей страницы)
\n	Новая строка
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\0	Нуль-символ
\f	Одинарная кавычка (апостроф)
\"	Двойная кавычка
\\	Обратная косая черта

Строковые литералы

Язык C# поддерживает еще один тип литерала: строковый. *Строка* – это набор символов, заключенных в двойные кавычки. Например, фрагмент кода «*Это текст*» представляет собой строку. Помимо обычных символов строковый литерал может содержать одну или несколько управляющих последовательностей.

Форматирование

Если данные встроенного типа требуется представить в форме, удобной для восприятия человеком, необходимо создать их строковое представление. Спецификаторы формата, определенные для числовых данных, описаны в табл. 1.4.

Таблица 1.4

Спецификаторы формата

Специфи-катор	Формат	Значение спецификатора точности
C c	Денежный	Задает количество десятичных разрядов
D d	Целочисленный (используется только с целыми числами)	Минимальное количество цифр. При необходимости результат дополняется начальными нулями
E e	Экспоненциальное представление чисел (с использованием прописной буквы *E* или *e* соответсвенно)	Задает количество десятичных разрядов. По умолчанию используется шесть
F f	Представление чисел с фиксированной точкой	Задает количество десятичных разрядов
G g	Используется более короткий из E- либо F-форматов	См. спецификаторы E и F
N n	Представление чисел с фиксированной точкой (и запятой в качестве разделителя групп разрядов)	Задает количество десятичных разрядов
P p	Процентный	Задает количество десятичных разрядов
R r	Числовое значение, которое можно с помощью метода *Parse()* преобразовать в эквивалентную «внутреннюю» форму (это так называемый формат «кругового преобразования»)	Не используется
X x	Шестнадцатеричный (использует прописные буквы $A - F$ или $a - f$ соответственно)	Минимальное количество цифр. При необходимости результат дополняется начальными нулями

Каждый спецификатор формата может включать необязательный спецификатор точности. Общая форма записи команд форматирования имеет такой вид:

{номер_аргумента, ширина: формат}

Несмотря на то что встроенные спецификаторы формата весьма полезны, в языке C# программист может определить собственный формат, используя средство, называемое форматом изображения (picture format). При создании пользовательского формата задается пример или изображение того, как должны выглядеть данные. Для этого используются символы, приведенные в табл. 1.5.

Таблица 1.5

Символы-заполнители, используемые для создания пользовательского формата

Заполнитель	Описание
#	Цифра
.	Десятичная точка
,	Разделитель групп разрядов
%	Процент
0	Используется для дополнения начальными и концевыми нулями
;	Отделяет разделы, которые описывают формат для положительных, отрицательных и нулевых значений
E0 E+o E–o e0 e+0 e–0	Экспоненциальное представление чисел

Общая форма записи спецификатора формата с использованием символа-заполнителя «;» такова:

плюс-формат; *минус-формат*; *нуль-формат*

Форматирование часто применяется к значениям такого типа данных, как DateTime, который представляет дату и время. Форматирование значений даты и времени осуществляется с помощью соответствующих спецификаторов формата, которые приведены в табл. 1.6.

Таблица 1.6

Спецификаторы формата для представления значений даты и времени

Спецификатор	Формат
D	Дата в длинной форме
d	Дата в короткой форме
T	Время в длинной форме
t	Время в короткой форме
F	Дата и время в длинной форме
F	Дата и время в короткой форме
G	Дата в краткой форме, а время – в длинной
g	Дата в краткой форме и время – в краткой
M m	Месяц и день
R r	Дата и время в стандартной форме по Гринвичу
s	Сортируемый формат представления даты и времени
U	Длинная форма универсального сортируемого формата представления даты и времени; время отображается как универсальное синхронизированное время (Universal Time Coordinated – UTC)
u	Краткая форма универсального сортируемого формата представления даты и времени
Y y	Месяц и год

Несмотря на то что в большинстве случаев достаточно стандартных спецификаторов формата для отображения даты и времени, существует возможность создания собственных форматов. Этот процесс аналогичен созданию пользовательских форматов для числовых типов, которое было описано выше. По сути, программист просто создает пример (изображение) того, как должна выглядеть информация, содержащая дату и время. Для создания пользовательского формата отображения значений даты и времени используйте один или несколько символов-заполнителей, перечисленных в табл. 1.7.

Таблица 1.7

Символы-заполнители, используемые при создании пользовательских форматов даты и времен

Заполнитель	Значение
d	День месяца как число, лежащее в диапазоне 1 – 31
dd	День месяца как число, лежащее в диапазоне 1 – 31. Значения из диапазона 1 – 9 дополняются начальным нулем
ddd	Сокращенное название дня недели
dddd	Полное название дня недели
f, ff, fff, ffff, fffff, ffffff, fffffff	Дробная часть значения секунд. Количество десятичных разрядов определяется числом заданных букв «f»
g	Эра
h	Часы в диапазоне 1 – 12
hh	Часы в диапазоне 1 – 12. Значения из диапазона 1 – 9 дополняются начальным нулем
H	Часы в диапазоне 0 – 23
HH	Часы в диапазоне 0 – 23. Значения из диапазона 1 – 9 дополняются начальным нулем
m	Минуты
mm	Минуты. Значения из диапазона 1 – 9 дополняются начальным нулем
M	Месяц в виде числа из диапазона 1 – 12
MM	Месяц в виде числа из диапазона 1 – 12. Значения из диапазона 1 – 9 дополняются начальным нулем
MMM	Сокращенное название месяца
MMMM	Полное название месяца
s	Секунды
ss	Секунды. Значения из диапазона 1-9 дополняются начальным нулем
t	Символ «А» или «Р», означающий А.М. (до полудня) или Р.М. (после полудня), соответственно
tt	А.М. или Р.М.
y	Год в виде двух цифр, если недостаточно одной
yy	Год в виде двух цифр. Значения из диапазона 1 – 9 дополняются начальным нулем
yyyy	Год в виде четырех цифр

Окончание табл. 1.7

Заполнитель	Значение
z	Смещение часового пояса в часах
zz	Смещение часового пояса в часах. Значения из диапазона 1 – 9 дополняются начальным нулем
zzz	Смещение часового пояса в часах и минутах
:	Разделитель для компонентов значения времени
/	Разделитель для компонентов значения даты
%fmt	Стандартный формат, соответствующий спецификатору формата *fmt*

Язык C# позволяет форматировать значения, определенные в перечислении. В общем случае значения перечислений можно отображать с использованием их имен или чисел. Спецификаторы формата, предназначенные для перечислений, приведены в табл. 1.8.

Таблица 1.8

Спецификаторы формата для перечислений

Спецификатор	Значение
G g	Отображает имя значения. Если форматируемое перечисление предваряется атрибутом Flags, спецификатор отображает имена всех битовых составляющих заданного значения (при условии, что оно допустимо)
F f	Отображает имя значения. Если это значение можно создать, применив операцию ИЛИ к двум или более полям, определенным перечислением, спецификатор отображает имена всех битовых составляющих заданного значения, причем независимо от того, задан ли атрибут Flags
D d	Отображает значение в виде десятичного целого числа
X x	Отображает значение в виде шестнадцатеричного целого числа. Для гарантированного отображения восьми цифр значение при необходимости дополняется начальными нулями

Инициализация переменной

Для объявления переменной необходимо использовать инструкцию следующего формата: *тип имя_переменной;*.

Здесь с помощью элемента *тип* задается тип объявляемой переменной, а с помощью элемента *имя_переменной* – ее имя. Можно объявить переменную любого допустимого типа. При создании переменной создается экземпляр соответствующего типа.

Переменная до использования должна получить значение. Это можно сделать с помощью инструкции присваивания. Можно также присвоить переменной начальное значение одновременно с ее объявлением. Общий формат инициализации переменной имеет такой вид: *тип имя_переменной = значение;*.

Здесь элемент *значение* – это начальное значение, которое получает переменная при создании. Значение инициализации должно соответствовать заданному типу переменной.

Вот несколько примеров:

int count = 10 ; // Присваиваем переменной count начальное значение 10.

char ch = 'X'; // Инициализируем ch буквой X.

float f = 1.2F; // Переменная f инициализируется числом 1.2.

int a, b = 8, c = 1 9 , d; // Переменные b и c инициализируются числами.

Язык C# позволяет инициализировать переменные динамически, с помощью любого выражения, действительного на момент объявления переменной.

Если в инструкции присваивания смешиваются совместимые типы, значение с правой стороны автоматически преобразуется в значение «левостороннего» типа. Не все преобразования типов разрешены в неявном виде. Например, типы *bool* и *int* несовместимы. Тем не менее, с помощью операции *приведения типов* все-таки возможно выполнить преобразование между несовместимыми типами.

Приведение к типу – это явно заданная инструкция компилятору преобразовать один тип в другой. Инструкция приведения записывается в следующей общей форме:

(тип_приемника) выражение

Здесь элемент *тип_приемника* определяет тип для преобразования заданного выражения. Например, если вам нужно, что бы выражение *x/y* имело тип *int*, напишите следующие программные инструкции:

double x=5.45, y=2.345;

int z;

z = (int) (x / y);

Если приведение приводит к сужающему преобразованию, возможна потеря информации. Например, в случае приведения типа *long* к типу *int* информация будет утеряна, если значение типа *long* больше максимально возможного числа, которое способен представить тип *int*, поскольку будут «усечены» старшие разряды *long*-значения. При выполнении операции приведения типа с плавающей точкой к целочисленному будет утеряна

дробная часть простым ее отбрасыванием. Например, при присвоении переменной целочисленного типа числа 1,23 в действительности будет присвоено число 1. Дробная часть (0,23) будет утеряна.

1.3 ОПЕРАЦИИ

Операция – это некоторое действие, обозначающееся символом. Базовые типы языка C# поддерживают целый ряд операций, таких как присваивание, инкрементирование и др.

В языке C# существует пять математических операций. Четыре – это стандартные арифметические действия, а пятая возвращает остаток от деления нацело. Арифметические операции: сложение «+», вычитание «–», умножение «*», деление «/». Для получения остатка от деления целых чисел предусмотрена специальная операция – деление по модулю «%».

В программе часто приходится прибавлять к переменной какое-либо значение (или вычитать, или менять переменную как-то иначе) и затем присваивать результат той же переменной. Данные операции описываются как *++val, val--, --val, val--*.

Операции отношения служат для сравнения двух значений и возврата логического значения (*true* или *false*). Операции отношения, принятые в языке C#, перечислены в табл. 1.9.

Таблица 1.9

Операции отношения в языке C#

Название	Операция	Пример	Результат
Равно	==	bigValue ==100	true
		bigValue ==80	false
Не равно	!=	bigValue !=100	false
		bigValue !=80	true
Больше	>	bigValue > smolValue	true
Больше или равно	>=	bigValue >= smolValue	true
		smolValue >= bigValue	false
Меньше	<	bigValue < smolValue	false
Меньше либо равно	<=	bigValue <= smolValue	false
		smolValue <= bigValue	true

Нередко возникает необходимость убедиться, что два условия истинны одновременно или что истинно хотя бы одно, или, например, не истинно ни одно из них. На этот случай язык C# предоставляет целый набор логических операций, приведенных в табл. 1.10.

Таблица 1.10

Логические операции языка C# (предполагается, что $x = 5$, а $y = 7$)

Название	Операция	Пример	Результат	Логика
И	&&	(x==3) && (y==7)	false	Оба операнда должны быть истинны
ИЛИ	||	(x==3) || (y==7)	true	Один из двух или оба операнда должны быть истинны
НЕ	!	! (x==3)	true	Выражение должно быть ложным

В языке C# порядок выполнения операций можно изменить с помощью скобок, подобно тому, как это делается в алгебре. Результат рассматриваемого примера можно изменить, написав *(5+7) * 3*. Группирование элементов оператора присваивания такой командой заставит компилятор сложить 5 и 7, умножить сумму на 3 и получить результат 36. В табл. 1.11 приводятся арифметические операции языка C#.

Таблица 1.11

Арифметические операции

Категория	Операция
Первичные	*(x) x.y f(x) a[x] x-- new type of sizeof checked unchecked stackalloc*
Унарные	*+ - ! ++x --x (T)x x&x*
Мультипликативные	** / %*
Аддитивные	*+ -*
Сдвиги	*<< >>*
Операции отношения	*< > <= => is as*
Равенство	*== !=*
Поразрядное И	*&*
Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ	^
Поразрядное ИЛИ	|
Логическое И	*&&*
Логическое ИЛИ	||
Условная операция (тернарная)	*?:*
Присваивание	*= *= /= %= += -= <<= >>= &= ^= |=*

В некоторых сложных уравнениях приходится вкладывать скобки друг в друга, чтобы гарантировать правильный порядок выполнения операций.

1.4 ИНСТРУКЦИИ УПРАВЛЕНИЯ

В языке C# завершенная инструкция программы называется оператором (statement). Программа состоит из последовательности операторов языка C#, каждый из которых заканчивается **точкой с запятой**.

Операторы языка C# выполняются последовательно. В программе на языке C# возможны два вида переходов: безусловные и условные.

Операторы безусловного перехода

Безусловный переход реализуется одним из двух способов. Первый это вызов метода. Когда компилятор встречает имя метода, он прекращает выполнение текущего метода и переходит к выполнению только что вызванного. Когда этот метод возвратит значение, выполнение вызвавшего метода возобновится со строчки, следующей за вызовом. Пример:

```
static void Main()
{
Console.WriteLine("В Main! Вызов SomeMethod()...");
SomeMethod();
Console.WriteLine("Вернулись в Main!...");
}
static void SomeMethod()
{
Console.WriteLine("Привет из SomeMethod");
}
```

Выполнение программы начинается с метода *Main()* и продолжается вплоть до вызова *SomeMethod()*. В этой точке программа переходит к указанному методу. Когда он закончится, выполнение программы будет возобновлено со строчки, следующей за вызовом.

Вторым способом реализации безусловного перехода является применение одного из следующих ключевых слов: *goto*, *break*, *continue*, *return* и *throw*.

Инструкция *return* обеспечивает возврат из метода. Ее можно использовать для возвращения методом значения.

Операторы условного перехода

Условный переход реализуется условным оператором, который обозначается ключевыми словами *if*, *else* или *switch*. Условный переход происходит только при выполнении определенного условия.

Операторы *if-else* осуществляют переход по условию. Условие – это выражение, проверяемое в заголовке оператора *if*. Если оно имеет значение *true*, выполняется оператор (или блок операторов) в теле оператора *if*.

Оператор *if* может сопровождаться необязательным оператором *else*. Оператор *else* выполняется, только если условие в заголовке оператора *if* имеет значение *false*:

if (выражение)
оператор 1
[else
оператор2]

Здесь показано, что в заголовке оператора *if* стоит логическое *выражение* (выражение, результатом которого является *true* или *false*), заключенное в круглые скобки. Если его значение равно *true*, выполняется конструкция *оператор 1*, которая может быть как одиночным оператором, так и блоком операторов в фигурных скобках.

В языке C# для обработки сложных условий допустимо вложение операторов *if*. Пример:

if (выражение1)
оператор 1
else if (выражение2)
оператор2]

При наличии большого списка вариантов выбора оператор *switch* является более мощным альтернативным инструментом.

switch (выражение){
    case (значение1):
оператор1
break;
case (значение2):
оператор2
break;
default: оператор3;
break;
}

Как вы можете видеть, аналогично оператору *if*, в заголовке оператора *switch* в круглых скобках приводится выражение. В каждом операторе *case* должно присутствовать константное выражение, то есть литеральная или символьная константа либо перечисление. За выполняемым оператором должен следовать оператор перехода. Как правило, это оператор *break*, который прекращает выполнение оператора *switch*. Если константное выражение не соответствует ни одному из операторов *case* будет выполнятся оператор *default*.

Язык C# поддерживает операторы перехода *goto*, *break*, *continue* и *return*. Оператор *goto* родоначальник всех остальных операторов перехода. Последовательность использования:

1. Создайте метку.

2. Укажите эту метку в операторе *goto*.

В языке C# существует широкий выбор операторов цикла. Сюда входят *for*, *while* и *do...while*, а также оператор *foreach*.

Семантика цикла *while* такова: пока это условие истинно, выполнять эту работу.

Как правило, выражение – это какой-нибудь оператор, возвращающий значение. У операторов *while* выражение обязательно должно возвращать логическое значение (*true* или *false*). Иногда необходимо изменить семантику с *пока условие истинно, работать* на *работать, пока условие остается истинным*. Тонкое различие состоит в том, что во втором случае в начале предпринимается действие, а затем, по его окончании, проверяется условие. В таких случаях необходимо применять цикл *do...while*.

Цикл *for* позволяет собрать все эти шаги в одном операторе:

for ([инициализация]', [выражение]; [итерация]) оператор

Оператор *foreach* применяется для циклического перебора элементов массива или коллекции:

foreach(тип переменная in коллекция) оператор

Бывают ситуации, когда необходимо начать новый проход цикла, не выполнив текущий проход до конца. Оператор *continue* передает управление на начало тела цикла, после чего выполнение продолжается. А для прерывания выполнения цикла и немедленного выхода из него применяется оператор *break*.

В то время как у большинства операций один или два операнда, существует одна операция, требующая три операнда. Она называется тернарной, обозначается символами «? :» и имеет следующий синтаксис:

условное-выражение ? выражение 1 : выражение 2

В этой операции анализируется *условное-выражение* (выражение, возвращающее значение типа *bool*) и выполняется одно из двух других выражений. Если условное выражение возвратило *true*, выполняется *выражение1*, если *false – выражение2*.

1.5 МАССИВЫ

Массив (array) – это коллекция переменных одинакового типа, обращение к которым происходит с использованием общего для всех имени. В языке C# массивы могут быть одномерными или многомерными.

Одномерные массивы

Одномерный массив – это список связанных переменных. Для объявления одномерного массива используется следующая форма записи:

тип[] имя__массива = new тип [размер];

Здесь с помощью элемента записи *тип* объявляется базовый тип массива. *Базовый тип* определяет тип данных каждого элемента, составляющего массив. Обратите внимание на *одну* пару квадратных скобок за элементом записи *тип*. Это означает, что определяется одномерный массив. Количество элементов, которые будут храниться в массиве, определяется элементом записи *размер*. Сначала объявляется ссылочная переменная на массив, а затем для него выделяется память, и переменной массива присваивается ссылка на эту область памяти. Таким образом, в языке C# массивы динамически размещаются в памяти с помощью оператора *new*.

int[] sample = new int [10];

Индекс описывает позицию элемента внутри массива. В языке C# первый элемент массива имеет нулевой индекс. Поскольку массив sample содержит 10 элементов, его индексы изменяются от 0 до 9.

```
int[] sample = new int[10];
for(int i = 0 ; i < 10; i = i+1)
sample[i] = i;
```

Инициализация массива

Существует простой путь инициализации **массива: массивы можно инициализировать** в момент их создания:

тип[] имя_массива = [vail, val2, . . . , valN];

Многомерные массивы

Многомерным называется такой массив, который характеризуется двумя или более измерениями, а доступ к отдельному элементу осуществляется посредством двух или более индексов.

Простейший многомерный массив – двумерный. Если представить двумерный массив в виде таблицы данных, то один индекс означает строку, а второй – столбец.

int[,] table = new int[10, 20];

Синтаксис первой части этого объявления означает, что создается ссылочная переменная двумерного массива.

Массивы трех и более измерений

В языке C# можно определять массивы трех и более измерений. Вот как объявляется многомерный массив:

тип[, ...,] имя = new тип[размер1, ..., размеры];

Присвоение значений ссылочным переменным массивов

Присваивая одной ссылочной переменной массива другую, вы просто изменяете объект, на который ссылается эта переменная. При этом не делается копия массива и не копируется содержимое одного массива в другой.

Использование свойства Length

С каждым массивом связано свойство Length, содержащее количество элементов, которое может хранить массив. Оно содержит количество элементов, которое массив способен хранить.

Строки

В языке C# текстовые строки имеют собственный тип – *string*. Тип данных *siring*, конечно же, связан с типом данных *char*, объект *string* можно создать из массива символов или превратить в массив символов. Но тип данных *string* и массив *char* – это разные типы данных. В языке C# строки не заканчиваются 0. Каждая строка имеет длину, и длина созданной строки изменяться не может. Строковую переменную позволяет определить литерал:

string str = "Hello, world";

Если перед строковым литералом поставить символ @, то обратная косая черта не будет рассматриваться как символ перехода. Это удобно при указании путей:

string str = @"c:\temp\iny file";

Массив строк можно определить так:

string[] astr = new string[5];

Создается массив строк, каждая из которых равна *NULL*. Можно создать массив инициализированных строк:

string[] astr = { "abc", "defghi", "jkl" };

В этом выражении создается строковый массив из трех элементов, таким образом, *astr.Length* возвращает 3 – у каждой строки своя длина, например, *astr[l].Length* возвращает 6.

Класс *string* можно использовать в выражении *foreach*, чтобы обращаться к символам строки по одному. Пример:

foreach (char ch in str)

{...}

1.6 АНАТОМИЯ ПРОГРАММЫ

```
//Простейшая программа
class ConsoleHelloWorld
{
static void Main()
{
System.Console.WriteLine("Hello World");
}
}
```

Пара символов «//» является началом однострочного комментария. Все, что справа от них, не учитывается при компиляции программы. Язык C# поддерживает и многострочные комментарии, заключаемые в комбинации символов «/*» и «*/».

Точка входа в программу «hello-world» на языке C# – функция *Main* внутри первой пары фигурных скобок. C# чувствителен к регистру. Имя *Main* точки входа в программу на языке C# пишется с заглавной буквы. Пустые скобки показывают, что *Main* не имеет параметров, а ключевое слово *void* – что не возвращает значения. Можно также указать, чтобы *Main* принимала массив строк символов в качестве входного параметра и возвращала целое значение.

Main располагается внутри определения *класса*. Класс – основной структурный и организационный элемент объектно-ориентированных языков программирования, подобных C#. Функция *System.ConsoleWriteLine* принимает один параметр – текстовую строку – и показывает ее на консоли в окне командной строки. Если вы скомпилируете и запустите программу, она покажет строку:

Hello world

и завершит работу.

Длинное имя функции *System.Console.WriteLine* состоит из таких частей:

1) *System* – пространство имен;
2) *Console* – класс, определенный в этом пространстве имен;
3) *WriteLine* – метод, определенный в этом классе (метод – то же самое, что и функция, процедура или подпрограмма).

2 ОСНОВЫ ООП

2.1 КЛАССЫ И СТРУКТУРЫ

Структуры

Язык программирования C# предоставляет программисту возможность создания новых типов данных как ссылочных, так и типов-значений. Ссылочные типы создаются с помощью зарезервированного слова *class*, а типы-значения – *struct*. Рассмотрим порядок определения типа *struct*

```
Struct ValType
{
        public int i;
        public double d;
}
public ValType(int i, double d)
{
        this.i = i;
        this.d = d;
}
```

Первой в этом примере объявляется структура *ValType*. Она имеет два поля: *i* и *d* типа *int* и *double* соответственно. Структура имеет конструктор с именем, совпадающим с именем самой структуры. Зарезервированное слово *this* используется для получения ссылки на экземпляр структуры.

Классы

Класс – это шаблон, который определяет форму объекта. Он задает как данные, так и код, который оперирует этими данными. Язык C# использует спецификацию класса для создания объекта. Объекты – это экземпляры класса. Таким образом, класс – это множество намерений (планов), определяющих, как должен быть построен объект. Важно четко понимать следующее: класс – это логическая абстракция. О ее реализации нет смысла говорить до тех пор, пока не создан объект класса, и в памяти не появилось физическое его представление. Вспомните, что методы и переменные, составляющие класс, называются членами класса.

Общая форма определения класса

Определяя класс, вы определяете данные, которые он содержит, и код, манипулирующий этими данными. Данные содержатся в переменных экземпляров, определяемых классом, а код – в методах. Класс создается с помощью ключевого слова *class*. Общая форма определения класса, кото-

рый содержит только переменные экземпляров и методы, имеет следующий вид:

class имя_класса {
// Объявление переменных экземпляров.
доступ тип переменная1;
доступ тип переменная2;
// Объявление методов.
доступ тип_возврата метод 1 (параметры) {
// тело метода}
доступ тип_возврата метод2{параметры) {
// тело метода}
}

Классы, которые мы использовали до сих пор, содержали только один метод – *Main()*.Однако заметьте, что в общей форме определения класса метод *Main()* не задан. Он нужен только в том случае, если определяемый класс является отправной точкой программы.

Пастулатами объектно-ориентированного программирования являются:

1) *инкапсуляция* – это механизм программирования, который связывает код (действия) и данные, которыми он манипулирует, и при этом предохраняет их от вмешательства извне и неправильного использования. Код, данные или оба эти составляющие объекта могут быть закрытыми внутри него или открытыми. Закрытый код или закрытые данные известны лишь остальной части этого объекта и доступны только ей. Если код или данные являются открытыми, к ним могут получить доступ другие части программы;
2) *полиморфизм* – это качество, которое позволяет одному интерфейсу получать доступ к целому классу действий. Простым примером полиморфизма может послужить руль автомобиля. Руль (интерфейс) остается рулем независимо от того, какой тип рулевого механизма используется в автомобиле. Достоинство такого единообразного интерфейса состоит, безусловно, в том, что, если вы знаете, как обращаться с рулем, вы сможете водить автомобиль любого типа. Концепцию полиморфизма часто выражают такой фразой: «один интерфейс – много методов». Полиморфизм позволяет понизить степень сложности программы, предоставляя программисту возможность использовать один и тот же интерфейс для задания общего класса действий. Конкретное действие выбирается компилятором. Программисту нет необходимости де-

лать это вручную. Его задача – правильно использовать общий интерфейс;

3) *наследование* – это процесс, благодаря которому один объект может приобретать свойства другого. Благодаря наследованию поддерживается концепция иерархической классификации и объекту нужно доопределить только те качества, которые делают его уникальным внутри его класса, поскольку он (объект) наследует общие атрибуты своего родителя. Следовательно, именно механизм наследования позволяет одному объекту представлять конкретный экземпляр более общего класса.

2.2 МЕТОДЫ

Методы – это процедуры, которые манипулируют данными, определенными в классе, и во многих случаях обеспечивают доступ к этим данным. Обычно различные части программы взаимодействуют с классом посредством его методов. Любой метод содержит одну или несколько инструкций. Каждый метод имеет имя, и именно это имя используется для его вызова. Формат записи метода такой:

доступ тип_возврата имя(список_параметров)

{

// тело метода

}

С помощью элемента *тип_возврата* указывается тип значения, возвращаемого методом. Это может быть любой допустимый тип, включая типы классов, создаваемые программистом. Если метод не возвращает никакого значения, необходимо указать тип void. В качестве имени метода можно использовать любой допустимый идентификатор, отличный от тех, которые уже использованы для других элементов программы в пределах текущей области видимости. Элемент *список_параметров* представляет собой последовательность пар (состоящих из типа данных и идентификатора), разделенных запятыми. Параметры – это переменные, которые получают значения аргументов, передаваемых методу при вызове. Если метод не имеет параметров, *список_параметров* остается пустым. Методы возвращают значения вызывающим их процедурам, используя следующую форму инструкции *return*:

return значение;

Здесь элемент *значение* и представляет значение, возвращаемое методом.

Нередко требуется, чтобы одно и то же имя было сразу у нескольких функций. Самым распространенным случаем является наличие нескольких конструкторов. Такой способ называется перегрузка методов. *Сигнатура (signature)* метода определяется его именем и списком параметров. Два метода отличаются по сигнатурам, если у них разные имена или списки параметров:

public void SetTime(int hour,int min);

public void SetTime(int hour,int min,int sec);

Процедура доступа get

Тело процедуры доступа *get* похоже на метод класса, возвращающий объект того же типа, что и свойство. Процедура доступа *get* возвращает значение закрытой переменной класса, в которой хранится значение свойства.

Процедура доступа set

Процедура доступа *set* изменяет значение свойства и аналогична методу, не возвращающему значения. Когда программист определяет процедуру доступа *set*, он должен использовать ключевое слово *value*, представляющее собой аргумент, значение которого передается и сохраняется в свойстве.

```
Class time
{
        private int min;
        private int hour;
        private int sec;
        set
        {
                min=value;
        }
        get
        {
                return hour;
        }
}
void main
{
time t;
...
int i=t.hour;
t.min=3;
}
```

Делегаты

Технически делегат – это ссылочный тип, инкапсулирующий метод с указанной сигнатурой и возвращаемым типом. В делегате можно инкапсулировать любой подходящий метод.

Делегат создается ключевым словом *delegate*, за которым указывается возвращаемый тип и сигнатура делегируемых методов:

public delegate int MyDelegate(object obj1, object obj2);

В этом объявлении определяется делегат с именем MyDelegate , который инкапсулирует любой метод, принимающий два параметра типа *Object*, и возвращает значение целого типа. Когда делегат определен, программист может инкапсулировать метод, создав экземпляр делегата путем передачи ему метода, соответствующего по сигнатуре и возвращаемому типу.

2.3 ОБЛАСТЬ ВИДИМОСТИ И УРОВНИ ДОСТУПА

Управление доступом к членам класса достигается за счет использования четырех спецификаторов доступа: *public*, *private*, *protected* и *internal*. Спецификатор *public* разрешает доступ к соответствующему члену класса со стороны другого кода программы, включая методы, определенные внутри других классов. Спецификатор *private* разрешает доступ к соответствующему члену класса только для методов, определенных внутри того же класса. Таким образом, методы других классов не могут получить доступ к private-члену не их класса. При отсутствии спецификатора доступа член класса является закрытым (*private*) по умолчанию. Следовательно, при создании закрытых членов класса спецификатор *private* необязателен. Спецификатор *protected* разрешает доступ к соответствующему члену класса только для методов, определенных внутри того же класса и классов унаследованных от данного. Для спецификатора *internal* доступ разрешен только для соответствующей программы, а для *protected internal* доступ разрешен только для соответствующей программы и производным того типа, который содержит данный член.

Область видимости переменных класса объявленных в методе распространяется только на этот метод. При объявлении переменной в цикле соответственно область видимости этой переменной распространяется только на этот цикл.

2.4 НАСЛЕДОВАНИЕ

Язык C# поддерживает наследование, позволяя в объявление класса встраивать другой класс. Это реализуется посредством задания базового класса при объявлении производного. Если один класс наследует другой, то имя базового класса указывается после имени производного, причем имена классов разделяются двоеточием. В языке C# синтаксис наследования класса очень прост для запоминания и использования:

class имя_наследуемого : имя_базового {}

В новом классе возможно как определение новых переменных и методов, так и переопределение методов базового класса.

Наследование класса не отменяет ограничения, связанные с закрытым доступом. Таким образом, несмотря на то, что производный класс включает все члены базового класса, он не может получить доступ к тем из них, которые объявлены закрытыми.

2.5 ПЕРЕГРУЗКА ОПЕРАТОРОВ

Язык C# позволяет определить значение оператора относительно создаваемого класса. Этот процесс называется перегрузкой операторов. Перегружая оператор, вы расширяете его использование для класса. Результат действия оператора полностью находится в ваших руках, и может быть разным при переходе от класса к классу.

Для перегрузки операторов используется ключевое слово *operator*, позволяющее создать операторный метод, который определяет действие оператора, связанное с его классом:

```
public static тип_возврата operator оператор(...)
{
...
}
```

2.6 ИНТЕРФЕЙСЫ, СТРУКТУРЫ, ПЕРЕЧИСЛЕНИЯ

Интерфейсы

Интерфейс определяет набор методов, которые будут реализованы классом. Сам интерфейс не реализует методы. Таким образом, интерфейс – это логическая конструкция, которая описывает методы, не устанавливая жестко способ их реализации. Интерфейсы синтаксически подобны абстрактным классам. Однако в интерфейсе ни один метод не может включать тело, т.е. интерфейс, в принципе, не предусматривает какой бы то ни было

реализации. Он определяет, что должно быть сделано, но не уточняет как. Для реализации интерфейса класс должен обеспечить тело (способы реализации) методов, описанных в интерфейсе. Каждый класс может определить собственную реализацию. Таким образом, два класса могут реализовать один и тот же интерфейс различными способами, но все классы поддерживают одинаковый набор методов.

Интерфейс языка C# позволяет в полной мере использовать аспект полиморфизма, выражаемый как «один интерфейс – много методов». Интерфейсы объявляются с помощью ключевого слова *interface*. Вот как выглядит упрощенная форма объявления интерфейса:

interface имя
{
тип_возврата имя_метода1 {список_параметров);
тип_возврата имя_метода2 {список_параметров);
...
}

Чтобы реализовать интерфейс, нужно указать его имя после имени класса подобно тому, как при создании производного указывается базовый класс. Формат записи класса, который реализует интерфейс, таков:

class имя_класса : имя__интерфейса {
// тело класса
}

Структуры

Структура подобна классу, но она относится к типу значений, а не к ссылочным типам.

Структуры объявляются с использованием ключевого слова *struct* и синтаксически подобны классам. Формат записи структуры таков:

struct имя : интерфейсы {
// объявления членов
}

Структуры не могут наследовать другие структуры или классы. Структуры не могут использоваться в качестве базовых для других структур или классов. Как и у классов, членами структур могут быть методы, поля, индексаторы, свойства, операторные методы и события. Структуры могут также определять конструкторы, но не деструкторы. Однако для структуры нельзя определить конструктор по умолчанию (без параметров). Дело в том, что конструктор по умолчанию автоматически определяется для всех структур, и его изменить нельзя.

```
struct str
{
        int i;
}
void main()
{
        str temp;
        temp.i=10;
}
```

Поскольку структуры – это типы значений, они обрабатываются напрямую, а не через ссылки. Таким образом, тип *struct* не требует отдельной ссылочной переменной. Это означает, что при использовании структур расходуется меньший объем памяти.

Перечисления

Перечисление (enumeration) – это множество именованных целочисленных констант. Ключевое слово *enum* объявляет перечислимый тип. Формат записи перечисления таков:

enum *имя {список_перечисления)*;

Здесь с помощью элемента *имя* указывается имя типа перечисления. Элемент *список_перечисления* представляет собой список идентификаторов, разделенных запятыми:

enum apple {Jonathan, GoldenDel, RedDel, Winsap,Cortland, McIntosh};

Здесь важно понимать, что каждый символ списка перечисления означает целое число, причем каждое следующее число (представленное идентификатором) на единицу больше предыдущего. Поскольку значение первого символа перечисления равно нулю, следовательно, идентификатор Jonathan имеет значение 0, GoldenDel – значение 1 и т.д. Константу перечисления можно использовать везде, где допустимо целочисленное значение:

Console.WriteLine(apple.RedDel+"имеет значение"+(int)apple.RedDel)

По умолчанию перечисления используют тип *int*, но можно также создать перечисление любого другого целочисленного типа, за исключением типа *char*. Чтобы задать тип, отличный от *int*, укажите этот базовый тип после имени перечисления и двоеточия:

enum apple : byte {Jonathan, GoldenDel, RedDel, Winsap,Cortland, McIntosh}

2.7 ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Под исключительной ситуацией понимается ошибка в ходе выполнения программы. Для борьбы с ошибками применяется блок обработки исключений, который будет автоматически выполняться при возникновении определенной ошибки.

В языке C# исключения представляются классами. Все классы исключений должны быть выведены из встроенного класса исключений *Exception*, который является частью пространства имен *System*.

Основы обработки исключений

Управление механизмом обработки исключений базируется на четырех ключевых словах: *try, catch, throw* и *finally*.

Программные инструкции, которые нужно проконтролировать на предмет исключений, помещаются в try-блок. Исключение может быть перехвачено программным путем с помощью catch-блока и обработано соответствующим образом. Системные исключения автоматически генерируются системой динамического управления. Чтобы сгенерировать исключение вручную, используется ключевое слово *throw*. Любой код, который должен быть обязательно выполнен при выходе из try-блока, помещается в блок *finally*.

Использование try- и catch-блоков

Ядром обработки исключений являются блоки *try* и *catch*. Эти ключевые слова работают «в одной связке»; нельзя использовать слово *try* без *catch* или *catch* без *try*. Вот каков формат записи try-catch-блоков обработки исключений:

```
try {
// Блок кода, подлежащий проверке на наличие ошибок.
}
catch (ExcepType1 exOb)
 {
// Обработчик для исключения типа ExcepType1.
}
catch (ExcepType2 exOb) {
// Обработчик для исключения типа ExcepType2
}
```

Пример:

```
try {
Console.WriteLine(numer[i]/denom[i]);
}
catch (DivideByZeroException) {
// Перехватываем исключение.
Console.WriteLine("Делить на нуль нельзя!");
}
```

2.8 ПРОСТРАНСТВА ИМЁН

Каждая C# программа так или иначе использует некоторое пространство имен. Пространство имен определяет декларативную область, которая позволяет отдельно хранить множества имен. Имена, объявленные в одном пространстве имен, не будут конфликтовать с такими же именами, объявленными в другом.

Библиотека .NET Framework (которая является C#-библиотекой) использует пространство имен *System*. Поэтому в начало каждой программы мы включали следующую инстукцию:

using System;

Пространство имен объявляется с помощью ключевого слова *namespace*. Общая форма объявления пространства имен имеет следующий вид:

```
namespace имя {
// Члены
}
```

Ключевое слово using

Нетрудно предположить, что директиву *using* можно также использовать для объявления действующими пространств имен, создаваемых программистом:

using имя;

Здесь элемент имя означает имя пространства имен, к которому необходимо получить доступ.

```
// Демонстрация использования пространства имен.
using System;
// Делаем текущим пространство имен Counter,
using Counter;
// Объявляем пространство имен для счетчиков,
namespace Counter {
// Простой счетчик для счета в обратном направлении.
class CountDown {
...
}
}
class NSDemo {
public static void Main() {
// Теперь класс CountDown можно использовать
// без указания имени пространства имен.
CountDown cdl = new CountDown(10);
}
}
```

2.9 ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ КЛАССОВ

В языке C# предусмотрена возможность создания пользовательских классов-контейнеров, к внутренним элементам которых можно обращаться при помощи индекса. Такой метод получил название *индексатора*.

```
class Program
 {
   static void Main(string[] args)
   {
     car temp=new car();
     temp[2] = new car();
     temp[4] = new car();
     temp[2].i = 45;
   }
 }
 class car
 {
   private car[] carArray;
   public int i;
   public car()
   {
     carArray = new car[10];
   }
   public car this[int pos]
   {
     get
     {
       return (carArray[pos]);
     }
     set
     {
       carArray[pos] = value;
     }
   }
```

В простейшем виде индексатор создается с помощью синтаксиса *this*.

3 БИБЛИОТЕКА ЯЗЫКА C#

Пространство имен *System* определяет ядро библиотеки языка C#, также содержит множество вложенных пространств имен, предназначенных для поддержки таких подсистем.

3.1 СТРУКТУРЫ ТИПОВ ЗНАЧЕНИЙ

System.Object

Каждый объект в языке C# прямо или косвенно является производным от *System.Object*. Благодаря этому, любой объект любого типа гарантированно имеет минимальный набор методов. Открытые экземплярные методы класса *System.Object* таковы (табл. 3.1).

Таблица 3.1

Открытые методы *System.Object*

Метод	Описание
public bool Equals(object v)	Возвращает значение ИСТИНА, если значение вызывающего объекта равно значению параметра *v*
public int GetHashCode()	Возвращает хеш-код для вызывающего объекта
public string ToString()	Возвращает строковое представление значения вызывающего объекта
public Type GetType ()	Возвращает экземпляр объекта, производного от *Type*, который идентифицирует тип объекта. Возвращаемый объект *Type* может использоваться с классами, реализующими отражение для получения информации о типе в виде метаданных

Структуры типов значений лежат в основе C#-типов значений. Используя члены, определенные этими структурами, можно выполнять операции, разрешенные для определенных типов значений.

К структурам целочисленных типов относятся следующие: *byte*, *sbyte*, *short*, *ushort*, *int*, *uint*, *lomg*, *ulong*. Все эти структуры содержат одни и те же методы (табл. 3.2) которые отличаются лишь типом значения, возвращаемого методом *Parse()*.

Целочисленные типы

Таблица 3.2.

Методы, поддерживаемые структурами целочисленных типов

Метод	Описание
public int CompareTo(object v)	Сравнивает числовое значение вызывающего объекта со значением параметра *v*. Возвращает нуль, если сравниваемые значения равны. Возвращает отрицательное число, если вызывающий объект имеет меньшее значение, и – положительное, если вызывающий объект имеет большее значение
public TypeCode GetTypeCode()	Возвращает значение перечисления *TypeCode* для эквивалентного типа
public static mun_возврата Parse(string str)	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре *str*. Если содержимое строки не представляет числовое значение в соответствии с определением типа структуры, генерируется исключение
public static mun_возврата Parse(string str, IFormatProvider fmtpvdr)	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре *str* с использованием форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра *fmtpvdr*. Если содержимое строки не представляет числовое значение в соответствии с определением типа структуры, генерируется исключение
public static mun_возврата Parse(string str, NumberStyles styles)	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре *str*, с использованием информации стилевого характера, заданной в параметре *styles*. Если содержимое строки не представляет числовое значение в соответствии с определением типа структуры, генерируется исключение
public static mun_возврата Parse(string str, NumberStyles styles, IFormatProvider fmtpvdr)	Возвращает двоичный эквивалент строкового представления числа, заданного в параметре *str*, с использованием информации стилевого характера, заданной в параметре *styles*, а также форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра *fmtpvdr*
public string ToString(string format)	Возвращает строковое представление значения вызывающего объекта в соответствии с требованиями форматирующей строки, переданной в параметре *format*

Окончание табл. 3.2

Метод	Описание
public string ToString(IFormatProvider fmtpvdr)	Возвращает строковое представление значения вызывающего объекта с использованием форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра *fmtpvdr*
public string ToString(string format, IFormatProvider fmtpvdr)	Возвращает строковое представление значения вызывающего объекта с использованием форматов данных (присущих конкретному естественному языку, диалекту или территориальному образованию), заданных посредством параметра *fmtpvdr*, а также форматирующей строки, переданной в параметре *format*

Типы данных с плавающей точкой

В языке C# определены только две структуры типов данных с плавающей точкой: *double* и *float*. Для них характерны методы целочисленных типов, плюс свои методы (табл. 3.3), а также собственные поля (табл. 3.4).

Таблица 3.3

Методы, поддерживаемые структурами *float* и *double*

Методы	Описание
public static bool IsInfinity(float v) *public static bool IsInfinity(double v)*	Возвращает значение ИСТИНА, если значение *v* представляет бесконечность
public static bool IsNaN(float v) *public static bool IsNaN(double v)*	Возвращает значение ИСТИНА, если значение *v* – не число
public static bool IsPositiveInfinity(float v) *public static bool IsPositiveInfinity(double v)*	Возвращает значение ИСТИНА, если значение *v* представляет бесконечность со знаком «плюс»
public static bool IsNegativeInfinity(float v) *public static bool IsNegativeInfinity(double v)*	Возвращает значение ИСТИНА, если значение *v* представляет бесконечность со знаком «минус»

Таблица 3.4

Поля, поддерживаемые структурой *float* и *double*

Поля	Описание
public const float Epsilon *public const double Epsilon*	Наименьшее ненулевое положительное значение
public const float MaxValue *public const double MaxValue*	Наибольшее значение, которое можно хранить с помощью типа *float* или *double*

Окончание табл. 3.4

Поля	Описание
public const float MinValue *public const double MinValue*	Наименьшее значение, которое можно хранить с помощью типа *float* или *double*
public const float NaN *public const double NaN*	Значение, которое не является числом
public const float NegativeInfinity *public const double NegativeInfinity*	Значение, представляющее минус бесконечность
public const float PositiveInfinity *public const double PositiveInfinity*	Значение, представляющее плюс бесконечность

Денежный тип данных

Структура *decimal* содержит множество конструкторов, полей, методов и операторов, которые способствуют совместному использованию типа *decimal* и других числовых C#-типов. Методы, определенные в структуре *decimal*, приведены в табл. 3.5, а поля – в табл. 3.6.

Таблица 3.5

Методы, определенные в структуре *decimal*

Методы	Описание
public static decimal Add(decimal v1, decimal v2)	Возвращает значение *v1* + *v2*
public static int CompareTo(decimal v1, decimal v2)	Возвращает нуль, если сравниваемые значения равны. Возвращает отрицательное число, если *v1* меньше *v2*, и – положительное, если *v1* больше *v2*
public static decimal Divide(decimal v1, decimal v2)	Возвращает значение *v1* / *v2*
public static bool Equals(decimal v1, decimal v2)	Возвращает значение ИСТИНА, если *v1* равно *v2*
public static decimal Floor(decimal v)	Возвращает наибольшее целое число (представленное в виде значения типа *decimal*), которое не больше параметра *v*. Например, при *v*, равном 1,02, метод Floor() возвратит 1,0. А при *v*, равном –1,02 , метод Floor() возвратит –2
public static decimal FromOACurrency(long v)	Преобразует значение, предоставленное приложением OLE Automation и переданное в параметре *v*, в его *decimal* –эквивалент и возвращает результат
public static int[] GetBits(decimal v)	Возвращает двоичное представление значения параметра *v* и возвращает его в массиве int-элементов. Организация этого массива описана в тексте этого раздела

Окончание табл. 3.5

Методы	Описание
public static decimal Multiply(decimal vl, decimal v2)	Возвращает значение *v1 * v2*
public static decimal Negate (decimal v)	Возвращает значение –*v*
public static decimal Remainder(decimal vl, decimal v2)	Возвращает остаток от целочисленного деления *vl / v2*
public static decimal Round(decimal v, int decPiaces)	Возвращает значение *v*, округленное до числа, количество цифр дробной части которого равно значению параметра *decPiaces*, которое должно находиться в диапазоне 0 – 28
public static decimal Sutract(decimal v1, decimal v2)	Возвращает значение (*v1 – v2*)
public static byte ToByte(decimal v)	Возвращает byte-эквивалент параметра *v*. Дробная часть отбрасывается
public static double ToDouble(decimal v)	Возвращает *double*-эквивалент параметра *v*. При этом возможна потеря точности, поскольку тип *double* имеет меньше значащих цифр, чем тип *decimal*
public static short Tolnt16(decimal v)	Возвращает *short*-эквивалент параметра *v*. Дробная часть отбрасывается
public static int Tolnt32(decimal v)	Возвращает *int*-эквивалент параметра *v*. Дробная часть отбрасывается
public static long Tolnt64(decimal v)	Возвращает *long*-эквивалент параметра *v*. Дробная часть отбрасывается.
public static long ToOACurrency(decimal v)	Преобразует значение параметра *v* в эквивалентное значение OLE Automation и возвращает результат
public static sbyte ToSByte(decimal v)	Возвращает *sbyte*-эквивалент параметра *v*. Дробная часть отбрасывается
public static float ToSingle(decimal v)	Возвращает *float*-эквивалент параметра *v*. При этом возможна потеря точности, поскольку тип *float* имеет меньше значащих цифр, чем тип *decimal*
public static ushort ToUlntl6(decimal v)	Возвращает *ushort*-эквивалент параметра *v*. Дробная часть отбрасывается
public static uint ToUlnt32(decimal v)	Возвращает *uint*-эквивалент параметра *v*. Дробная часть отбрасывается
public static ulong ToUlnt64(decimal v)	Возвращает *ulong*-эквивалент параметра *v*. Дробная часть отбрасывается
public static decimal Truncate(decimal v)	Возвращает целую часть числа, заданного параметром *v*. Соответственно, любая дробная часть при этом отбрасывается

Таблица 3.6

Поля, поддерживаемые структурой *decimal*

Поле	Описание
public static readonly decimal MaxValue	Наибольшее значение, которое позволяет хранить ТИП *decimal*
public static readonly decimal MinusOne	Представление числа –1 в формате *decimal* значения
public static readonly decimal MinValue	Наименьшее значение, которое позволяет хранить тип *decimal*
public static readonly decimal One	Представление числа 1 в формате *decimal* значения
public static readonly decimal Zero	Представление числа 0 в формате *decimal* значения
public static double GetNumericValue(char ch)	Возвращает числовое значение параметра *ch*, если *ch* – цифра. В противном случае возвращает –1
public static double GetNumericValue(string str, int idx)	Возвращает числовое значение символа *str[idx]*, если он является цифрой. В противном случае возвращает –1
public static UnicodeCategory GetUnicodeCategory(char ch)	Возвращает значение перечисления UnicodeCategory для параметра *ch*. UnicodeCategory – это перечисление, определенное в пространстве имен System.Globalization, в котором символы Unicode разделены по категориям
public static UnicodeCategory GetUnicodeCategory(string str, int idx)	Возвращает значение перечисления UnicodeCategory для символа *str[idx]*. *UnicodeCategory* – это перечисление, определенное в пространстве имен *System.Globalization*, в котором символы Unicode разделены по категориям
public static bool IsControl(char ch)	Возвращает значение ИСТИНА, если параметр *ch* является управляющим символом

Если значение параметра не попадает в диапазон представления чисел, соответствующий представляемому типу, генерируется исключение типа *OverflowException*.

Char

Наиболее используемой является структура *char*. Она предоставляет большое количество методов, которые позволяют обрабатывать символы и определять, к какой категории они относятся. Методы, определяемые в структуре *char*, перечислены в табл. 3.7.

Таблица 3.7

Методы, определенные в структуре *char*

Методы	Описание
public static bool IsControl(string str, int idx)	Возвращает значение ИСТИНА, если символ *str [idx]* является управляющим
public static bool IsDigit(char ch)	Возвращает значение ИСТИНА, если параметр *ch* является цифрой
public static bool IsDigit(string str, int idx)	Возвращает значение ИСТИНА, если символ *str[idx]* является цифрой
public static bool IsLetter(char ch)	Возвращает значение ИСТИНА, если параметр *ch* является буквой алфавита
public static bool IsLetter(string str, int idx)	Возвращает значение ИСТИНА, если символ *str[idx]* является буквой алфавита
public static bool IsLetterOrDigit(char ch)	Возвращает значение ИСТИНА, если параметр *ch* является буквой алфавита или цифрой
public static bool IsLetterOrDigit(string str, int idx)	Возвращает значение ИСТИНА, если символ *str[idx]* является буквой алфавита или цифрой
public static bool IsLower(char ch)	Возвращает значение ИСТИНА, если параметр *ch* является строчной буквой алфавита
public static bool IsLower(string str, int idx)	Возвращает значение ИСТИНА, если символ *str[idx]* является строчной буквой алфавита
public static bool IsNumber(char ch)	Возвращает значение ИСТИНА, если параметр *ch* является шестнадцатиричной цифрой (0 – 9 или A – F)
public static bool IsNumber(string str, int idx)	Возвращает значение ИСТИНА, если символ *str[idx]* является шестнадцатиричной цифрой (0 – 9 или A – F)
public static bool IsPunctuation(char ch)	Возвращает значение ИСТИНА, если параметр *ch* является знаком пунктуации
public static bool IsPunctuation(string str, int idx)	Возвращает значение ИСТИНА, если символ *str [idx]* является знаком пунктуации
public static bool IsSeparator(char ch)	Возвращает значение ИСТИНА, если параметр *ch* является разделительным знаком, например пробелом
public static bool IsSeparator(string str, int idx)	Возвращает значение ИСТИНА, если символ *str[idx]* является разделительным знаком, например пробелом
public static bool IsSurrogate(char ch)	Возвращает значение ИСТИНА, если параметр *ch* является псевдосимволом Unicode
public static bool IsSurrogate(string str, int idx)	Возвращает значение ИСТИНА, если символ *str[idx]* является псевдосимволом Unicode

Окончание табл. 3.7

Методы	Описание
public static bool IsSymbol(char ch)	Возвращает значение ИСТИНА, если параметр *ch* является символическим знаком, например валютным символом
public static bool IsSymbol(string str, int idx)	Возвращает значение ИСТИНА, если символ *str[idx]* является символическим знаком, например валютным символом
public static bool IsUpper(char ch)	Возвращает значение ИСТИНА, если параметр *ch* является прописной буквой алфавита
public static bool IsUpper(string str, int idx)	Возвращает значение ИСТИНА, если символ *str[idx]* является прописной буквой алфавита
public static bool IsWhiteSpace(char ch)	Возвращает значение ИСТИНА, если параметр *ch* является пробелом, символом табуляции или пустой строки
public static bool IsWhiteSpace(string str, int idx)	Возвращает значение ИСТИНА, если символ *str[idx]* является пробелом, символом табуляции или пустой строки
public static char ToLower(char ch)	Возвращает строчный эквивалент параметра *ch*, если *ch* – прописная буква. В противном случае возвращает значение *ch* неизменным
public static char ToLower(char ch, CultureInfo c)	Возвращает строчный эквивалент параметра *ch*, если *ch* – прописная буква. В противном случае возвращает значение *ch* неизменным. Преобразование выполняется в соответствии с заданной в параметре информацией о представлении данных, соответствующей конкретному естественному языку, диалекту или территориальному образованию. *CultureInfo* – это класс, определенный в пространстве имен *System.Globalization*
public static char ToUpper(char ch)	Возвращает прописной эквивалент параметра *ch*, если *ch* – строчная буква. В противном случае возвращает значение *ch* неизменным
public static char ***ToUpper(char ch,*** *CultureInfo c)*	Возвращает прописной эквивалент параметра *ch*, если *ch* – строчная буква. В противном случае возвращает значение *ch* неизменным. Преобразование выполняется в соответствии с заданной в параметре информацией о представлении данных, соответствующей конкретному естественному языку, диалекту или территориальному образованию. *CultureInfo* – это класс, определенный в пространстве имен *System.Globalization*

Поля, определяемые в структуре *char*, перечислены в табл. 3.8. А методы идентичны целочисленным структурам, за исключением конечно возвращаемого типа *bool*.

Таблица 3.8

Поля структуры bool

Поле	Описание
public static readonly string FalseString	Возвращает строковое «*False*»
public static readonly string TrueString	Возвращает строковое «*True*»

System.Array

System.Array – это базовый класс для всех массивов в языке C#, его методы можно применять для массивов любого из встроенных типов, а также массивов. Свойства, определенные в классе *Array*, перечислены в табл. 3.9, а методы – в табл. 3.10.

Таблица 3.9

Свойства, определенные в классе *Array*

Свойство	Поле
public virtual bool IsFixedSize { get; }	Принимает значение *true*, если массив имеет фиксированный размер, и – *false*, если массив может динамически его изменять
public virtual bool IsReadOnly { get;}	Принимает значение *true*, если объект класса Array предназначен только для чтения, и – *false* в противном случае
public virtual bool IsSynchronized { get; }	Принимает значение *true*, если массив можно безопасно использовать в многопоточной среде, и – *false* в противном случае
public int Length { get;}	Содержит количество элементов в массиве
public int Rank { get; }	Содержит размерность массива
public virtual object SyncRoot { get;}	Содержит объект, который должен быть использован для синхронизации доступа к массиву

Таблица 3.10

Методы, определенные в классе *Array*

Метод	Описание
public static int BinarySearch(Array a, object v)	В массиве, заданном параметром *a*, выполняет поиск значения, заданного параметром *v*. Возвращает индекс первого вхождения искомого значения. Если оно не найдено, возвращает отрицательное число. Массив *a* должен быть отсортированным и одномерным

Метод	Описание
public static int BinarySearch(Array a, object v, IComparer comp)	В массиве, заданном параметром *a*, выполняет поиск значения, заданного параметром *v*, с использованием метода сравнения, заданного параметром *comp*. Возвращает индекс первого вхождения искомого значения. Если оно не найдено, возвращает отрицательное число. Массив *a* должен быть отсортированным и одномерным
public static int BinarySearch(Array a, int start, int count, object v)	В части массива, заданного параметром *a*, выполняет поиск значения, заданного параметром *v*. Поиск начинается с индекса, заданного параметром *start*, и охватывает *count* элементов. Возвращает индекс первого вхождения искомого значения. Если оно не найдено, возвращает отрицательное число
public static int BinarySearch(Array a, int start, int count, object v, IComparer comp)	В части массива, заданного параметром *a*, выполняет поиск значения, заданного параметром *v*, с использованием метода сравнения, заданного параметром *comp*. Поиск начинается с индекса, заданного параметром *start*, и охватывает *count* элементов. Возвращает индекс первого вхождения искомого значения. Если оно не найдено, возвращает отрицательное число. Массив *a* должен быть отсортированным и одномерным
public static void Clear(Array a, int start, int count)	Устанавливает заданные элементы равными нулю. Диапазон элементов, подлежащих обнулению, начинается с индекса, заданного параметром *start* , и включает *count* элементов
public virtual object Clone()	Возвращает копию вызывающего массива. Эта копия ссылается на те же элементы, что и оригинал, за что получила название «поверхностной». Это означает, что изменения, вносимые в элементы, влияют на оба массива, поскольку они оба используют одни и те же элементы
public static void Copy(Array source, Array dest, int count)	Копирует *count* элементов из массива *source* в массив *dest*. Копирование начинается с начальных элементов каждого массива. Если оба массива имеют одинаковый ссылочный тип, метод *Copy()* создает «поверхностную копию», в результате чего оба массива будут ссылаться на одни и те же элементы
public static void Copy(Array source, int srcStart, Array dest, int destStart, int count)	Копирует *count* элементов из массива *source* (начиная с элемента с индексом *srcstart)* в массив *dest* (начиная с элемента с индексом *deststart)*. Если оба массива имеют одинаковый ссылочный тип, метод *Copy()* создает «поверхностную копию», в результате чего оба массива будут ссылаться на одни и те же элементы
public virtual void CopyTo(Array dest, int start)	Копирует элементы вызывающего массива в массив *dest*, начиная с элемента *dest[start]*
public static Array CreateInstance (Type t, int size)	Возвращает ссылку на одномерный массив, который содержит *size* элементов типа *t*

Продолжение табл. 3.10

Метод	Описание
public static Array CreateInstance(Type t, int size1, int size2)	Возвращает ссылку на двумерный массив размером *size1*size2*. Каждый элемент этого массива имеет тип *t*
public static Array CreateInstance(Type t, int size1, int size2, int size3)	Возвращает ссылку на трехмерный массив размером *size1*size2*size3*. Каждый элемент этого массива имеет тип *t*
public static Array CreateInstance(Type t, int[] sizes)	Возвращает ссылку на многомерный массив, размерности котоого заданы в массиве *sizes*. Каждый элемент этого массива имеет тип *t*
public static Array CreateInstance(Type t, int[] sizes, int[] startIndexes)	Возвращает ссылку на многомерный массив, размерности котоого заданы в массиве *sizes*. Каждый элемент этого массива имеет тип *t*. Начальный индекс каждой размерности задан в массиве *startIndexes*. Таким образом, этот метод позволяет создавать массивы, которые начинаются с некоторого индекса, отличного от нуля
public virtual IEnumerator GetEnumerator()	Возвращает нумераторный объект для массива. Нумератор позволяет опрашивать массив в цикле.
public int GetLength(int dim)	Возвращает длину заданной размерности. Отсчет размерностей начинается с нуля; следовательно, для получения длины первой размерности необходимо передать методу значение 0, а для получения длины второй – значение 1
public int GetLowerBound(int dim)	Возвращает начальный индекс заданной размерности, который обычно равен нулю. Параметр *dim* ориентирован на то, что отсчет размерностей начинается с нуля; следовательно, для получения начального индекса первой размерности передается 0, а для начального индекса второй – значение 1
public int GetUpperBound(int dim)	Возвращает конечный индекс заданной размерности, который обычно равен нулю. Параметр *dim* ориентирован на то, что отсчет размерностей начинается с нуля; следовательно, для получения конечного индекса первой размерности необходимо передать методу значение 0, а для получения конечного индекса второй – значение 1
public object GetValue(int idx)	Возвращает значение элемента вызывающего массива с индексом *idx*. Массив должен быть одномерным
public object GetValue(int idx1, int idx2)	Возвращает значение элемента вызывающего массива с индексами [*idx1, idx2]*. Массив должен быть двумерным
public object GetValue(int idx1, int idx2, int idx3)	Возвращает значение элемента вызывающего массива с индексами [*idx1, idx2, idx3]*. Массив должен быть трехмерным

Продолжение табл. 3.10

Метод	Описание
public object GetValue(int[] idxs)	Возвращает значение элемента вызывающего массива с индексами, заданными с помощью параметра *idxs*. Вызывающий массив должен иметь столько размерностей, сколько элементов в массиве *idxs*
public static int IndexOf(Array a, object v)	Возвращает индекс первого элемента одномерного массива *a*, который имеет значение, заданное параметром *v*. Возвращает –1, если искомое значение не найдено. (Если массив имеет нижнюю границу, отличную от 0, признак неудачного выполнения метода будет равен значению нижней границы, уменьшенному на 1)
public static int IndexOf(Array a, object v, int start)	Возвращает индекс первого элемента одномерного массива *a*, который имеет значение, заданное параметром *v*. Поиск начинается с элемента *a [start]*. Возвращает –1 , если искомое значение не найдено. (Если массив имеет нижнюю границу, отличную от 0, признак неудачного выполнения метода будет равен значению нижней границы, уменьшенному на 1)
public static int IndexOf(Array a, object v, int start, int count)	Возвращает индекс первого элемента одномерного массива *a*, который имеет значение, заданное параметром *v*. Поиск начинается с элемента *a* [*start*] и охватывает *count* элементов. Метод возвращает –1 , если внутри заданного диапазона искомое значение не найдено
public void Initialize()	Инициализирует каждый элемент вызывающего массива посредством вызова конструктора по умолчанию, соответствующего конкретному элементу. Этот метод можно использовать только для массивов нессылочных типов
public static int LastIndexOf(Array a, object v)	Возвращает индекс последнего элемента одномерного массива *a*, который имеет значение, заданное параметром *v*. Возвращает –1, если искомое значение не найдено. (Если массив имеет нижнюю границу, отличную от 0, признак неудачного выполнения метода будет равен значению нижней границы, уменьшенному на 1)
public static int LastIndexOf(Array a, object v, int start)	Возвращает индекс последнего элемента заданного диапазона одномерного массива *a*, который имеет значение, заданное параметром *v*. Поиск осуществляется в обратном порядке, начиная с элемента а *[start]* и заканчивая элементом *a [0]*. Метод возвращает число –1 , если искомое значение не найдено. (Если массив имеет нижнюю границу, отличную от 0 признак неудачного выполнения метода будет равен значению нижней границы, уменьшенному на 1)
public static int LastIndexOf(Array a, object v, int start, int count)	Возвращает индекс последнего элемента заданного диапазона одномерного массива *a*, который имеет значение, заданное параметром *v*. Поиск осуществляется в обратном порядке, начиная с элемента *a [start]*, и охватвает *count* элементов. Метод возвращает –1, если внутри заданного диапазона искомое значение не найдено

Метод	Описание
public static void Reverse(Array a)	Меняет на обратный порядок следования элементов в массиве *a*
public static void Reverse(Array a, int start, int count)	Меняет на обратный порядок следования элементов в заданном диапазоне массива *a*. Упомянутый диапазон начинается с элемента *a [start]* и включает *count* элементов
public void SetValue(object v, int idx)	Устанавливает в вызывающем массиве элемент с индексом *idx* равным значению *v*. Массив должен быть одномерным
public void SetValue(object v, int idx1, int idx2)	Устанавливает в вызывающем массиве элемент с индексами *[idx1, idx2]* равным значению *v*. Массив должен быть двумерным
public void SetValue(object v, int idx1, int idx2, int idx3)	Устанавливает в вызывающем массиве элемент с индексами *[idx1, idx2, idx3]* равным значению *v*. Массив должен быть трехмерным
public void SetValue(object v, int[] idxs)	Устанавливает в вызывающем массиве элемент с индексами, заданными параметром *idxs*, равным значению *v*. Вызывающий массив должен столько размерностей, сколько элементов в массиве *idxs*
public static void Sort(Array a)	Сортирует массив *a* в порядке возрастания. Массив должен быть одномерным
public static void Sort(Array a, IComparer comp)	Сортирует массив *a* в порядке возрастания с использованием метода сравнения, заданного параметром *comp*. Массив должен быть одномерным
public static void Sort(Array k, Array v)	Сортирует в порядке возрастания два заданных одномерных массива. Массив *к* содержит ключи сортировки, а массив *v* – значения, связанные с этими ключами. Следовательно, эти два массива должны содержать пары ключ/значение. После сортировки элементы обоих массивов расположены в порядке возрастания ключей
public static void Sort(Array k, Array v, IComparer comp)	Сортирует в порядке возрастания два заданных одномерных массива с использованием метода сравнения, заданного параметром *comp*. Массив *к* содержит ключи сортировки, а массив *v* – значения, связанные с этими ключами. Следовательно, эти два массива должны содержать пары ключ/значение. После сортировки элементы обоих массивов расположены в порядке возрастания ключей
public static void Sort(Array a, int start, int count)	Сортирует заданный диапазон массива в порядке возрастания. Упомянутый диапазон начинается с элемента *a [start]* и включает *count* элементов. Массив должен быть одномерным
public static void Sort(Array a, int start, int count, IComparer comp)	Сортирует заданный диапазон массива в порядке возрастания с использованием метода сравнения, заданного параметром *comp*. Упомянутый диапазон начинается с элемента *a [start]* и включает *count* элементов. Массив должен быть одномерным

Окончание табл. 3.10

Метод	Описание
public static void Sort(Array k, Array v, int start, int count)	Сортирует заданный диапазон двух одномерных массивов в порядке возрастания. В обоих массивах диапазон сортировки начинается с индекса, переданного в параметре *start*, и включает *count* элементов. Массив *к* содержит ключи сортировки, а массив *v* – значения, связанные с этими ключами. Следовательно, эти два массива должны содержать пары ключ/значение
public static void Sort(Array k, Array v, int start, int count, IComparer comp)	Сортирует заданный диапазон двух одномерных массивов в порядке возрастания с использованием метода сравнения, заданного параметром *comp*. В обоих массивах диапазон сортировки начинается с индекса, переданного в параметре *start*, и включает *count* элементов. Массив *к* содержит ключи сортировки, а массив *v* — значения, связанные с этими ключами. Следовательно, эти два массива должны содержать пары ключ/значение

3.2 МАТЕМАТИЧЕСКИЕ ОПЕРАЦИИ

System.Math

В классе *System.Math* определены такие стандартные математические операции, как извлечение квадратного корня, вычисление синуса, косинуса и логарифмов и т.д. Методы, определенные в классе *Math*, перечислены в табл. 3.11. Все углы задаются в радианах. Обратите внимание на то, что все методы, определенные в классе *Math*, являются static-методами. Поэтому для их использования не нужно создавать объект класса *Math*, а значит, нет необходимости и в конструкторах класса *Math*.

Таблица 3.11

Методы, определенные в классе Math

Метод	Описание
public static double Abs(double v) *public static float Abs(float v)* *public static decimal Abs(decimal v)* *public static int Abs(int v)* *public static short Abs(short v)* *public static long Abs(long v)* *public static sbyte Abs(sbyte v)*	Возвращает абсолютную величину параметра *v*
public static double Acos(double v)	Возвращает арккосинус параметра *v*. Значение *v* должно находиться в диагнозе между –1 и 1

Продолжение табл. 3.11

Метод	Описание
public static double Asin(double v)	Возвращает арксинус параметра *v*. Значение *v* должно находиться в диагнозе между –1 и 1
public static double Atan(double v)	Возвращает арктангенс параметра *v*
public static double Atan2(double y, double x)	Возвращает арктангенс частного *y/x*
public static double Cos(double v)	Возвращает косинус параметра *v*
public static double Cosh(double v)	Возвращает гиперболический косинус параметра *v*
public static double Exp(double v)	Возвращает основание натурального логарифма
public static double Floor(double v)	Возвращает наибольшее целое (представленное в виде значения с плавающей точкой), которое не больше параметра *v*
public static double IEEERemainder(double dividend, double divisor)	Возвращает остаток от деления *dividend/division*
public static double Log(double v)	Возвращает натуральный логарифм *v*
public static double Log(double v, double base)	Возвращает логарифм *v* по основанию base
public static double Log10(double v)	Возвращает логарифм *v* по основанию 10
public static double Max(double v1, double v2) *public static float Max(float v1, float v2)* *public static decimal Max(decimal v1, decimal v2)* *public static int Max(int v1, int v2)* *public static short Max(short v1, short v2)* *public static long Max(long v1, long v2)* *public static uint Max(uint v1, uint v2)* *public static ulong Max(ulong v1, ulong v2)* *public static byte Max(byte v1, byet v2)* *public static sbyte Max(sbyte v1, sbyet v2)*	Возвращает большее из значений *v*1 и *v*2
public static double Min(double v1, double v2) *public static float Min(float v1, float v2)* *public static decimal Min(decimal v1, decimal v2)* *public static int Min(int v1, int v2)* *public static short Min(short v1, short v2)* *public static long Min(long v1, long v2)* *public static uint Min(uint v1, uint v2)* *public static ulong Min(ulong v1, ulong v2)* *public static byte Min(byte v1, byet v2)* *public static sbyte Min(sbyte v1, sbyet v2)*	Возвращает меньшее из значений *v*1 и *v*2
public static double Pow(double base, double exp)	Возвращает значение *base* возведенное в степень *exp*

Окончание табл. 3.11

Метод	Описание
public static double Round(double v) *public static decimal Round(decimal v)*	Возвращает значение *v*, округленного до ближайшего целого числа
public static double Round(double v, int frac) *public static decimal Round(decimal v, int frac)*	Возвращает значение *v*, округленного до числа, количество дробной части которого равно значению *frac*
public static int Sign(double v) *public static int Sign(float v)* *public static int Sign(decimal v)* *public static int Sign(int v)* *public static int Sign(short v)* *public static int Sign(long v)* *public static int Sign(sbyte v)*	Возвращает –1, если значение *v* меньше нуля; 0, если *v* равно нулю и 1, если *v* больше нуля
public static double Sin(double v)	Возвращает синус параметра *v*
public static double Sinh(double v)	Возвращает гиперболический синус параметра *v*
public static double Tan(double v)	Возвращает тангенс параметра *v*
public static double Tanh(double v)	Возвращает гиперболический тангенс параметра *v*

BitConverter

При написании программ часто приходится преобразовывать данные встроенных типов в массив байтов. Для решения подобных проблем преобразования данных в языке C# и предусмотрен класс *System.BitConverter*. Класс *BitConverter* содержит методы, представленные в табл. 3.12.

Таблица 3.12

Методы, определенные в классе *BitConverter*

Метод	Описание
public static long *DoubleToInt64Bits(double v)*	Преобразует значение параметра *v* в целочисленное значение типа *long* и возвращает результат
public static double *Int64BitsToDouble(long v)*	Преобразует значение параметра *v* в значение с плавающей точкой типа *double* и возвращает результат
public static byte[] GetBytes(bool v) *public static byte[] GetBytes(char v)* *public static byte[] GetBytes(double v)* *public static byte[] GetBytes(float v)* *public static byte[] GetBytes(int v)* *public static byte[] GetBytes(long v)* *public static byte[] GetBytes(short v)* *public static byte[] GetBytes(uint v)* *public static byte[] GetBytes(ulong v)* *public static byte[] GetBytes(ushort v)*	Преобразует значение параметра *v* в соответствующий массив и возвращает результат

Окончание табл. 3.12

Метод	Описание
public static bool ToBoolean(byte[] a, int idx)	Преобразует элемент *a [idx]* байтового массива *a* в его bool-эквивалент и возвращает результат. Ненулевое значение преобразуется в значение *true*, а нулевое – в *false*
public static char ToChar(byte[] a, Int start)	Преобразует два байта, начиная с элемента *a[start]*, в соответствующий char-эквивалент и возвращает результат
public static double ToDouble(byte[] a, int start)	Преобразует восемь байтов, начиная с элемента *a[start]*, в соответствующий double-эквивалент и возвращает результат
public static short ToIntl6(byte[] a, int start)	Преобразует два байта, начиная с элемента *a[start]*, в соответствующий short-эквивалент и возвращает результат
public static int ToInt32(byte[] a, int start)	Преобразует четыре байта, начиная с элемента *a[start]*, в соответствующий int-эквивалент и возвращает результат
public static long ToInt64(byte[] a, int start)	Преобразует восемь байтов, начиная с элемента *a[start]*, в соответствующий long-эквивалент и возвращает результат
public static float ToSingle(byte[] a, , int star)	Преобразует четыре байта, начиная с элемента *a[start]*, в соответствующий float-эквивалент и возвращает результат
public static string ToString(byte[] a) *public static string ToString(byte[] a, int start)* *public static string ToString(byte[] a, int start, int count)*	Преобразует байты массива *a* в строку. Строка содержит шестнадцатеричные значения (связанные с этими байтами), разделенные дефисами
public static ushort ToUIntl6(byte[] a, int start)	Преобразует два байта, начиная с элемента *a[start]*, в соответствующий *ushort* эквивалент и возвращает результат
public static uint ToUInt32(byte[] a, int start)	Преобразует четыре байта, начиная с элемента *a[start]*, в соответствующий uint-эквивалент и возвращает результат
public static ulong ToUInt64(byte[] a, int start)	Преобразует восемь байтов, начиная с элемента *a[start]*, в соответствующий ulong-эквивалент и возвращает результат

В нем также определено следующее поле: *public static readonly bool IsLittleEndian*. Это поле принимает значение true, если текущая операционная среда обрабатывает сначала слово с младшим (наименее значимым), а затем со старшим (наиболее значимым) байтом.

Класс *BitConverter* является sealed-классом, т.е. не может иметь производных классов.

System.Random

Чтобы сгенерировать последовательность псевдослучайных чисел, используйте класс *System.Random*. В классе *Random* определены следующие два конструктора:

public Random()

public Random(int seed)

С помощью первой версии конструктора создается объект класса *Random*, который для вычисления начального числа последовательности случайных чисел использует системное время. При использовании второй версии конструктора начальное число задается в параметре *seed*.

Методы, определенные в классе *Random*, перечислены в табл. 3.13.

Таблица 3.13

Методы, определенные в классе *Random*

Метод	Свойство
public virtual int Nexto	Возвращает случайное число типа *int*, которое будет находиться в диапазоне 0 – *int32.MaxVaiue* –1, включительно
public virtual int Next (int upperBound)	Возвращает следующее случайное число типа *int*, которое будет находиться в диапазоне 0 – *upperBound*, включительно
public virtual int Next (int lowerBound, int upperBound)	Возвращает следующее случайное число типа *int*, которое будет находиться в диапазоне *lowerBound* – *upperBound* включительно
public virtual void NextBytes (byte [ ] bur)	Заполняет буфер *buf* последовательностью случайных целых чисел. Каждый байт в массиве будет находиться в диапазоне 0 – *Byte. MaxValue*–1, включительно
public virtual double NextDoubie ()	Возвращает следующее случайное число из последовательности (представленное в форме с плавающей точкой), которое будет больше или равно числу 0,0 и меньше 1,0
protected virtual double sample()	Возвращает следующее случайное число из последовательности (представленное в форме с плавающей точкой), которое будет больше или равно числу 0,0 и меньше 1,0. Чтобы создать несимметричное или специализированное распределение, этот метод необходимо переопределить в производном классе

3.3 ОБРАБОТКА ТЕКСТА

Класс *String* определен в пространстве имен *System*. Он лежит в основе встроенного в C#-типа *String*.

В классе *String* определено несколько конструкторов, которые позволяют создавать строки различными способами.

В классе *String* определено только одно поле: public static readonly string *Empty*, которое определяет пустую строку, т.е. строку, которая не содержит символов.

В классе *String* определен единственный индексатор, предназначенный только для чтения: *public char this[int idx] { get; }*. Этот индексатор позволяет получить символ по заданному индексу.

В классе *String* определено единственное свойство, предназначенное только для чтения: *public int Length { get; }*. Свойство *Length* возвращает количество символов, содержащихся в строке.

В классе *String* также реализована перегрузка двух операторов: = = и !=.

В классе *String* определено множество различных методов. При этом многие из них имеют два или больше перегруженных форматов.

Сравнения

Из всех операций обработки строк, возможно, чаще всего используется операция сравнения одной строки с другой. Поэтому в классе *String* предусмотрен широкий выбор методов сравнения, которые перечислены в табл. 3.14.

Таблица 3.14

Методы сравнения, определенные в классе *String*

Метод	Описание
public static int Compare(string str1, string str2)	Сравнивает строку, адресуемую параметром *str1*, со строкой, адресуемой параметром *str2*. Возвращает положительное число, если строка *str1* больше *str2*, отрицательное число, если *str1* меньше *str2*, и нуль, если строки *str1* и *str2* равны
public static int Compare(string str1, string str2, bool ignoreCase)	Сравнивает строку, адресуемую параметром *str1*, со строкой, адресуемой параметром *str2*. Возвращает положительное число, если строка *str1* больше *str2*; отрицательное число, если *str1* меньше *str2* и – нуль, если строки *str1* и *str2* равны. Если параметр *ignoreCase* равен значению *true*, при сравнении не учитываются различия между прописным и строчным вариантами букв. В противном случае эти различия учитываются

Метод	Описание
public static int Compare(string str1, string str2, bool ignoreCase, CultureInfo ci)	Сравнивает строку, адресуемую параметром *str1*, со строкой, адресуемой параметром *str2*, с использованием специальной информации (связанной с конкретным естественным языком, диалектом или территориальным образованием), переданной в параметре *ci*. Возвращает положительное число, если строка *str1* больше *str2*, отрицательное число, если *str1* меньше *str2*, и – нуль, если строки *str1* и *str2* равны. Если параметр *ignoreCase* равен значению *true*, при сравнении не учитываются различия между прописным и строчным вариантами букв. В противном случае эти различия учитываются. Класс *CultureInfo* определен в пространстве имен *System.Globalization*
public static int Compare(string str1, int startl, string str2, int start2, int count)	Сравнивает части строк, заданных параметрами *str1* и *str2*. Сравнение начинается со строковых элементов *str1 [start1]* и *str2[start2]* и включает count символов. Метод возвращает положительное число, если часть строки *str1* больше части строки *str2* , отрицательное число, если *str1* –часть меньше *str2*-части, и – нуль, если сравниваемые части строк *str1* и *str2* равны
public static int Compare(string str1, int start1, string str2, int start2, int count, bool ignoreCase)	Сравнивает части строк, заданных параметрами *str1* и *str2*. Сравнение начинается со строковых элементов *str1[start1]* и *str2 [start2]* и включает *count* символов. Метод возвращает положительное число, если часть строки *str1* больше части строки *str2*; – отрицательное число, если *str1*-часть меньше *str2*-части, и – нуль, если сравниваемые части строк *str1* и *str2* равны. Если параметр *ignoreCase* равен значению *true* , при сравнении не учитываются различия между прописным и строчным вариантами букв
public static int Compare(string str1, int start1, string str2, int start2, int count, bool ignoreCase, CultureInfo ci)	Сравнивает части строк, заданных параметрами *str1* и *str2* с использованием специальной информации, переданной в параметре *ci*. Сравнение начинается со строковых элементов *str1[start1]* и *str2[start2]* и включает *count* символов. Метод возвращает положительное число, если часть строки *str1* больше части строки *str2* отрицательное число, если *str1*- часть меньше *str2* - части, и нуль, если сравниваемые части строк *str1* и *str2* равны. Если параметр *ignoreCase* равен значению *true*, при сравнении не учитываются различия между прописным и строчным вариантами букв. В противном случае эти различия учитываются. Класс *CultureInfo* определен в пространстве имен *System.Globalization*
public static int CompareOrdinal (string str1, string str2)	Сравнивает строку, адресуемую параметром *str1*, со строкой, адресуемой параметром *str2*, независимо от языка, диалекта или территориального образования. Возвращает положительное число, если строка *str1* больше *str2;* – отрицательное число, если *str1* меньше *str2* и – нуль, если строки *str1* и *str2* равны
public int CompareTo(object str)	Сравнивает вызывающую строку со строкой, заданной параметром *str*. Возвращает положительное число, если вызывающая строка больше строки *str;* – отрицательное число, если вызывающая строка меньше строки *str* и – нуль, если сравниваемые строки равны

Окончание табл. 3.14

Метод	Описание
public static int CompareOrdinal (string str1, int start1, string str2, int start2, int count)	Сравнивает части строк, заданных параметрами *str1* и *str2*, независимо от языка, диалекта или территориального образования. Сравнение начинается со строковых элементов *str1[start1]* и *str2[start2]* и включает *count* символов. Метод возвращает положительное число, если часть строки *str1* больше части строки *str2*; – отрицательное число, если *str1* часть меньше *str2* части и – нуль, если сравниваемые части строк *str1* и *str2* равны
public int CompareTo(string str)	Сравнивает вызывающую строку со строкой, заданной параметром *str*. Возвращает положительное число, если вызывающая строка больше строки *str*, отрицательное число, если вызывающая строка меньше строки *str*, и нуль, если сравниваемые строки равны

Конкатенация

Существует два способа конкатенации (объединения) двух или больше строк. Во-первых, можно использовать для этого оператор «+». Во-вторых, можно применить один из методов конкатенации, определенных в классе String. Несмотря на то, что оператор «+» – самое простое решение во многих случаях, методы конкатенации предоставляют дополнительные возможности.

Метод, который выполняет конкатенацию, именуется *Concat ()*, а его простейший формат таков:

public static string Concat(string str1, string str2)

Метод возвращает строку, которая содержит строку *str2*, присоединенную к концу строки *str1*.

Еще один формат метода *Concat ()* позволяет объединить три строки:

public static string Concat(string str1, string str2, string str3)

При вызове этой версии возвращается строка, которая содержит конкатенированные строки *str1*, *str2* и *str3*. По правде говоря, для выполнения описанных выше операций все же проще использовать оператор «+», а не метод *Concat ()*.

А вот следующая версия метода *Concat ()* объединяет произвольное число строк, что делает ее весьма полезной для программиста:

public static string Concat(params string[] strs)

Здесь метод *Concat ()* принимает переменное число аргументов и возвращает результат их конкатенации.

Некоторые версии метода *Concat ()* принимают не string-, а object-ссылки. Они извлекают строковое представление из передаваемых им объектов и возвращают строку, содержащую конкатенированные строки. Форматы этих версий метода *Concat ()* таковы:

public static string Concat(object v1, object v2)

public static string Concat(object v1, object v2, object v3)

public static string Concat(params object[] v)

Поиск

В классе *string* есть два набора методов, которые позволяют найти заданную строку (подстроку либо отдельный символ). При этом можно заказать поиск первого либо последнего вхождения искомого элемента. Чтобы отыскать первое вхождение символа или подстроки, используйте метод *IndexOf ()*, который имеет два таких формата:

public int IndexOf(char ch)

public int IndexOf(string str)

Метод *IndexOf ()*, используемый в первом формате, возвращает индекс первого вхождения символа *ch* в вызывающей строке. Второй формат позволяет найти первое вхождение строки *str*. В обоих случаях возвращается значение –1, если искомый элемент не найден.

Чтобы отыскать последнее вхождение символа или подстроки, используйте метод *Last IndexOf ()*, который имеет два таких формата:

public int LastIndexOf(char ch)

public int LastIndexOf(string str)

Метод *LastIndexOf()*, используемый в первом формате, возвращает индекс последнего вхождения символа *ch* в вызывающей строке. Второй формат позволяет найти последнее вхождение строки *str*. В обоих случаях возвращается значение – 1, если искомый элемент не найден.

В классе *String* определено два дополнительных метода поиска: *IndexOfAny()* и *LastIndexOfAny()*. Они выполняют поиск первого или последнего символа, который совпадает с любым элементом заданного набора символов. Вот их форматы:

public int IndexOfAny(char[] a)

public int LastIndexOfAny(char[] a)

Метод *IndexOfAny()* возвращает индекспервого вхождения любого символа из массива а, который обнаружится в вызывающей строке. Метод *LastIndexOfAny()* возвращает индекс последнего вхождения любого символа из массива а, который обнаружится в вызывающей строке. В обоих случаях возвращается значение – 1, если совпадение не обнаружится.

При работе со строками часто возникает необходимость узнать: начинается ли строка с заданной подстроки либо оканчивается ею. Для таких ситуаций используются методы *StartsWith ()* и *EndsWith ()*:

public bool StartsWith(string str)

public bool EndsWith(string str)

Метод *StartsWith ()* возвращает значение *true*, если вызывающая строка начинается с подстроки, переданной в параметре *str*. Метод *EndsWith ()* возвращает значение *true*, если вызывающая строка оканчивается подстрокой, переданной в параметре *str*. В случае неудачного исхода оба метода возвращают значение *false*.

Некоторые методы поиска имеют дополнительные форматы, которые позволяют начать поиск с заданного индекса или указать диапазон поиска. Все версии методов поиска символов в строке, которые определены в классе *String*, приведены в табл. 3.15.

Таблица 3.15

Методы поиска символов в строке, определенные в классе *String*

Метод	Описание
public bool EndsWith(string str)	Возвращает значение *true*, если вызывающая строка оканчивается подстрокой, переданной в параметре *str*. В противном случае метод возвращает значение *false*
public int IndexOf(char ch)	Возвращает индекс первого вхождения символа *ch* в вызывающей строке. Возвращает значение –1, если искомый символ не обнаружен
public int IndexOf(string str)	Возвращает индекс первого вхождения подстроки *str* в вызывающей строке. Возвращает значение –1, если искомая подстрока не обнаружена
public int IndexOf(char ch, int start)	Возвращает индекс первого вхождения символа *ch* в вызывающей строке. Поиск начинается с элемента, индекс которого задан параметром *start*. Возвращает значение –1, если искомый символ не обнаружен
public int IndexOf(string str, int start)	Возвращает индекс первого вхождения подстроки *str* в вызывающей строке. Поиск начинается с элемента, индекс которого задан параметром *start*. Возвращает значение –1, если искомая подстрока не обнаружена
public int IndexOf(char ch, int start, int count)	Возвращает индекс первого вхождения символа *ch* в вызывающей строке. Поиск начинается с элемента, индекс которого задан параметром *start*, и охватывает *count* элементов. Метод возвращает значение –1, если искомый символ не обнаружен
public int IndexOf(string str, int start, int count)	Возвращает индекс первого вхождения подстроки *str* в вызывающей строке. Поиск начинается с элемента, индекс которого задан параметром *start*, и охватывает *count* элементов. Метод возвращает значение –1, если искомая подстрока не обнаружена

Продолжение табл. 3.15

Метод	Описание
public int IndexOfAny(char[] a)	Возвращает индекс первого вхождения любого символа из последовательности *a*, который будет обнаружен в вызывающей строке. Возвращает значение –1, если совпадения символов не найдено
public int IndexOfAny (char[] a, int start)	Возвращает индекс первого вхождения любого символа из последовательности *a*, который будет обнаружен в вызывающей строке. Поиск начинается с элемента, индекс которого задан параметром *start*. Метод возвращает значение –1, если совпадения символов не обнаружено
public int IndexOfAny (char[] a, int start, int count)	Возвращает индекс первого вхождения любого символа из последовательности *a*, который будет обнаружен в вызывающей строке. Поиск начинается с элемента, индекс которого задан параметром *start*, и охватывает *count* элементов. Метод возвращает значение –1, если совпадения символов не обнаружено
public int LastIndexOf(char ch)	Возвращает индекс последнего вхождения символа *ch* в вызывающей строке. Возвращает значение –1, если искомый символ не обнаружен
public int LastIndexOf(string str)	Возвращает индекс последнего вхождения подстроки *str* в вызывающей строке. Возвращает значение –1, если искомая подстрока не обнаружена
public int LastIndexOf(char ch, int start)	Возвращает индекс последнего вхождения символа *ch*, обнаруженного в пределах диапазона вызывающей строки. Поиск выполняется в обратном порядке, начиная с элемента, индекс которого задан параметром *start*, и заканчивая элементом с нулевым индексом. Метод возвращает значение –1, если искомый символ не обнаружен
public int LastIndexOf(string str, int start)	Возвращает индекс последнего вхождения подстроки *str* , обнаруженной в пределах диапазона вызывающей строки. Поиск выполняется в обратном порядке, начиная с элемента, индекс которого задан параметром *start*, и заканчивая элементом с нулевым индексом. Метод возвращает значение –1, если искомая подстрока не обнаружена
public int LastIndexOf(char ch, int start, int count)	Возвращает индекс последнего вхождения символа *ch*, обнаруженного в пределах диапазона вызывающей строки. Поиск выполняется в обратном порядке, начиная с элемента, индекс которого задан параметром *start*, и охватывая *count* элементов. Если искомый символ не обнаружен, возвращает –1
public int LastIndexOf(string str, int start, int count)	Возвращает индекс последнего вхождения подстроки *str*, обнаруженной в пределах диапазона вызывающей строки. Поиск выполняется в обратном порядке, начиная с элемента, индекс которого задан параметром *start*, и охватывая *count* элементов. Метод возвращает значение –1, если искомая подстрока не обнаружена

Метод	Описание
public int LastIndexOfAny (char[] a)	Возвращает индекс последнего вхождения любого символа из последовательности *a*, который будет обнаружен в вызывающей строке. Возвращает значение –1, если совпадения символов не обнаружено
public int LastIndexOfAny (char[] a, int start)	Возвращает индекс последнего вхождения любого символа из последовательности *a*, который будет обнаружен в вызывающей строке. Поиск выполняется в обратном порядке, начиная с элемента, индекс которого задан параметром *start*, и заканчивая элементом с нулевым индексом. Возвращает значение –1, если совпадения символов не обнаружено
public int LastIndexOfAny (char[] a, int start, int count)	Возвращает индекс последнего вхождения любого символа из последовательности *a*, который будет обнаружен в вызывающей строке. Поиск выполняется в обратном порядке, начиная с элемента, индекс которого задан параметром *start*, и охватывая *count* элементов. Возвращает значение –1, если совпадения символов не обнаружено
public bool StartsWith(string str)	Возвращает значение *true*, если вызывающая строка начинается строкой, переданной в параметре *str*. В противном случае метод возвращает значение *false*

С помощью метода *Substring ()* можно получить нужную часть строки. Возможны две формы использования этого метода:

public string Substring(int idx)

public string Substring(int idx, int count)

При использовании первой формы выделяемая подстрока начинается с элемента, заданного индексом *idx*, и продолжается до конца вызывающей строки. Вторая форма позволяет выделить подстроку, которая начинается с элемента, заданного индексом *idx*, и включает *count* символов. В обоих случаях возвращается выделенная подстрока.

Разбиение и сборка

Двумя важными операциями обработки строк являются разбиение (декомпозиция) и сборка. При *разбиении* строка делится на составные части. А при *сборке* строка «собирается» из отдельных частей. Для разбиения строк в классе *String* определен метод *Split ()*, а для сборки – метод *Join ()*.

Форматы использования метода *Split ()* таковы:

public string[] split(params char[] seps)

public string[] split(params char[J seps, int count)

Метод первого формата предназначен для разбиения вызывающей строки на подстроки, которые возвращаются методом в виде строкового массива. Символы, которые отделяют подстроки одну от другой, передаются в массиве *seps*. Если параметр *seps* содержит null-значение, в качестве разделителя подстрок используется пробел. Метод второго формата отличается от первого тем, что ограничивает количество возвращаемых подстрок значением *count*.

Рассмотрим теперь форматы применения метода *Join ()*:

public static string Join(string sep, string[] strs)

public static string Join(string sep, string[] strs, int start, int count)

Метод *Join()* в первом формате возвращает строку, которая содержит конкатенированные строки, переданные в массиве *strs*. Второй формат позволяет собрать строку, которая будет содержать *count* конкатенированных строк, переданных в массиве *strs*, начиная со строки *strs[start]*. В обоих случаях каждая строка, составляющая результат, отделяется от следующей разделительной строкой, заданной параметром *sep*.

Удаление и вставка

Иногда возникает необходимость удалить из строки начальные и концевые пробелы. Чтобы удалить из строки начальные и концевые пробелы, используйте один из следующих вариантов метода Trim ():

public string Trim()

public string Trim(params char[] chrs)

Первый формат метода предназначен для удаления начальных и концевых пробелов из вызывающей строки. Второй позволяет удалить начальные и концевые символы, заданные параметром *chrs*. В обоих случаях возвращается строка, содержащая результат этой операции.

В языке C# предусмотрена возможность дополнить строку заданными символами справа либо слева. Для реализации «левостороннего» дополнения строки используйте один из следующих методов:

public string PadLeft(int len)

public string PadLeft(int lenr char ch)

Первый формат метода предназначен для дополнения строки с левой стороны пробелами в таком количестве, чтобы общая длина вызывающей строки стала равной заданному значению *len*. Второй формат отличается от первого тем, что для дополнения строки вместо пробела используется символ, заданный параметром *ch*. В обоих случаях возвращается строка, содержащая результат этой операции.

Для реализации «правостороннего» дополнения строки используйте один из следующих методов:

public string PadRight(int len)

public string PadRight(int lenf char ch)

Первый формат метода дополняет строку с правой стороны пробелами в таком количестве, чтобы общая длина вызывающей строки стала равной заданному значению *len*. Второй формат отличается от первого тем, что для дополнения строки вместо пробела используется символ, заданный параметром *ch*. В обоих случаях возвращается строка, содержащая результат этой операции.

С помощью метода *Insert ()* можно вставлять одну строку в другую:

public string Insert(int start, string str)

Здесь параметром *str* задается строка, вставляемая в вызывающую. Позиция вставки (индекс) задается параметром *start*. Метод возвращает строку, содержащую результат этой операции.

С помощью метода *Remove ()* можно удалить заданную часть строки:

public string Remove(int start, int count)

Количество удаляемых символов задается параметром *count*. Позиция (индекс), с которой начинается удаление, задается параметром *start*. Метод возвращает строку, содержащую результат этой операции.

С помощью метода *Replace ()* можно заменить часть строки заданным символом либо строкой. Этот метод используется в двух форматах:

public string Replace(char ch1, char ch2)

public string Replace(string str1, string str2)

Первый формат метода позволяет заменить в вызывающей строке все вхождения символа *ch1* символом *ch2*. Второй служит для замены в вызывающей строке всех вхождений строки *str1* строкой *str2*. В обоих случаях возвращается строка, содержащая результат этой операции.

Класс *String* содержит два удобных метода, которые позволяют изменить «регистр», т.е. способ написания букв в строке. Эти методы называются *ToUpper ()* и *ToLower()*:

public string ToLower()

public string ToUpper()

Метод *ToLower ()* заменяет все буквы в вызывающей строке их строчными вариантами, а метод *ToUpper ()* – прописными. Оба метода возвращают строку, содержащую результат операции. Существуют также версии этих методов, которые позволяют задавать форматы данных, присущие конкретному естественному языку, диалекту или территориальному образованию.

Форматирование

Строковое представление значения в форматированном виде можно получить с помощью метода *String.Format ()*.

Форматированное значение можно получить в результате вызова одной из версий метода *Format ()*, определенного в классе *string* (эти версии представлены в табл. 3.16).

Таблица 3.16

Методы *Format()*

Метод	Описание
public static string Format(string str, object v)	Форматирует объект *v* в соответствии с первой командой форматирования, которая содержится в строке *str* . Возвращает копию строки *str*, в которой команда форматирования заменена форматированными данными
public static string Format(string str, object v1, object v2)	Форматирует объект *v1* в соответствии с первой командой форматирования, содержащейся в строке *str*, а объект *v2* – в соответствии со второй. Возвращает копию строки *str* , в которой команды форматирования заменены форматированными данными
public static string Format(string str, object v1, object v2, object v3)	Форматирует объекты *v1, v2* и *v3* согласно соответствующим командам форматирования, содержащимся в строке *str*. Возвращает копию строки *str*, в которой команды форматирования заменены форматированными данными
public static string Format(string str, params object[] v)	Форматирует значения, переданные в параметре *v*, в соответствии с командами форматирования, содержащимися в строке *str*. Возвращает копию строки *str*, в которой команды форматирования заменены форматированными данными
public static string Format(IFormatProvider fmtprvdr, string str, params object [] v)	Форматирует значения, переданные в параметре *v*, в соответствии с командами форматирования, содержащимися в строке *str*, и с использованием провайдера формата, заданного параметром *fmtprvdr*. Возвращает копию строки *str*, в которой команды форматирования заменены форматированными данными

Для получения форматированного нужным образом строкового представления, соответствующего значению встроенного числового типа, можно использовать метод *ToString ()*:

public string ToString(string fmt)

Метод *ToString()* возвращает строковое представление вызывающего объекта в соответствии с заданным спецификатором формата, переданным в параметре *fmt*.

4 СОЗДАНИЕ WINDOWS-ПРИЛОЖЕНИЯ

4.1 ОБЗОР ПРОСТРАНСТВА ИМЕН WINDOWS.FORM

Пространство имен *Windows.Forms* включает в себя огромное число типов. Основные типы, используемые в приложениях представлены в табл. 4.1. Основной типы при работе с окнами *Form* (класс формы окна) и *Application* (класс приложения).

Таблица 4.1

Основные типы *Windows.Forms*

Класс	Назначение
Application	Основа приложения *Windows. Forms*. При помощи этого класса возможна обработка сообщений Windows, запуск приложений, прекращение работы приложений и т.д.
ButtonBase, Button, CheckBox, ComboBox, DataGrid, GroupBox, ListBox, LinkLabel, PictureBox	Классы элементов графического интерфейса
Form	Класс главной формы приложения
ColorDialog, FileDialog, FontDialog, PrintPreviewDialog	Классы диалогов выбора цвета, шрифта, файлов, просмотра печати
Menu, MainMenu, ManuItem, ContextMenu	Классы для создания различных типов меню
Clipboar, Help, Timer, Screen, ToolTip, Cursors	Различные вспомогательные типы для создания интерактивных графических интерфейсов
StatusBar, Splitter, ToolBar, ScrollBar	Дополнительные элементы управления

4.2 СОЗДАНИЕ ПРОЕКТА WINDOWS.FORM

Создание проекта *Windows.Forms* возможно двумя способами – вручную и с помощью шаблона *Windows.Forms*. Второй способ проще и имеется возможность добавлять элементы управления вручную, зная, что код добавится автоматически. Первый способ позволяет понять основы создания и управления приложением.

Формы

Обычно в программах *Windows.Forms* формы создаются для главного окна приложения. Кроме того, формы служат для построения диалоговых

окон приложения. Форма, используемая в качестве главного окна приложения, обычно состоит из заголовка (*caption bar, title bar*) с именем приложения, меню (*menu bar*), расположенного под заголовком, и внутренней области – клиентской области (client area). Форма окружается рамкой, позволяющей изменять ее размеры, или тонкой рамкой, не позволяющей изменять размеры.

Приложение (ручное создание)

Форма определяет лишь вид окна, для запуска самой программы требуется создать приложение и запустить его. Для работы с самим приложением непосредственно служит объект класса приложения *Application*. Для запуска приложения используется метод класса *Application.Run*.

Пример простейшей программы создания формы и запуска приложения:

```
using System. Windows. Forms;
class NewForm
{       public static void Main()
{       form = new Form();//создаем форму
form.Visible = true;//отображаем форму
Application .Run();
}
}
```

В случае передаче методу *Run* объекта формы качестве переменной, не требуется делать форму видимой:

```
Application.Run(form);
```

Когда вы закрываете форму, передаваемую как параметр *Application.Run*, метод *Application.Run* закрывает все остальные формы, созданные программой. Если вы не передадите объект *Form* методу *Application.Run*, то программа должна явно вызвать метод *Application.Exit*, чтобы заставить *Application.Run* вернуть управление.

Приложение (автоматическое создание)

Для создания приложения с помощью шаблона требуется выбрать в меню *New->Project->Visual C#->Windows->Windows Application* (рис. 4.1).

После создания приложения весь код будет написан автоматически:

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

```
namespace WindowsApplication1
{
        static class Program
{
[STAThread]
static void Main()
{
Application.EnableVisualStyles();
Application.SetCompatibleTextRenderingDefault(false);
Application.Run(new Form1());
}
}
}
```

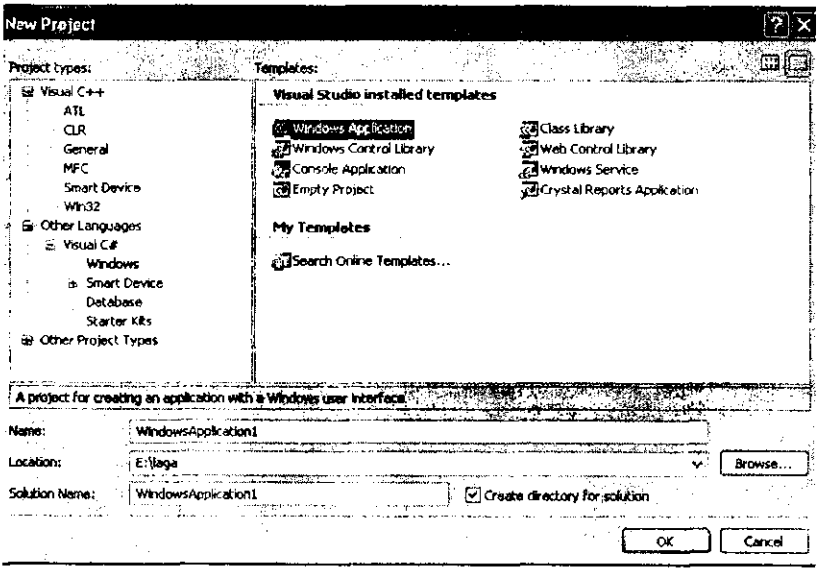

Рис. 4.1 Создание приложения с помощью шаблона *Windows Application*

Класс System.Windows.Forms.Application

Данный класс позволяет управлять поведением приложения Windows.Forms. Основные методы данного класса представлены в табл. 4.2.

Таблица 4.2

Основные методы класса *Application*

Метод	Назначение
AddMessageFilter(), RemoveMessageFilter()	Позволяют приложению перехватывать сообщения и выполнять с этими сообщениями предварительные действия
DoEvents()	Обеспечивает способность приложения обрабатывать сообщения из очереди сообщений во время выполнения какой-либо длительной операции
Exit()	Завершает работу приложения
ExitThread()	Прекращает обработку сообщений для текущего потока и закрывает все окна, владельцем которых является этот поток
OLERequired()	Инициализация библиотеки OLE
Run()	Запускает стандартный цикл работы с сообщениями для текущего потока

Из свойств данного класса можно выделить *StartupPath*, которое возвращает путь и имя выполняемого файла приложения. Основные события класса *Application* занесены в табл. 4.3.

Таблица 4.3

Основные события класса *Application*

Событие	Момент возникновения
ApplicationExit	Закрытие приложения
Idle	Все сообщения в текущем потоке обработаны, и приложение находится в режиме бездействия
ThreadExit	Завершение работы потока в приложении. Если работу завершает главный поток, это событие возникает до события *ApplicationExit*

Ожидание некого события выполняется следующим образом:

Application.ApplicationExit += new EventHandler(Form_Exit);

....

private Form_Exit(object sender, EventArgs evArgs) {}

Функция *Form_Exit* вызовется при возникновении сообщения *ApplicationExit*, первый параметр – объект, отправляющий событие, второй – идентификатор события.

Класс Form

Класс *Form* унаследовал от классов *Control*, *ScrollableControl*, *CountainerControl* много методов. Кроме этих методов, класс *Form* имеет много своих методов представленных в табл. 4.4.

Таблица 4.4

Свойства класса *Form*

Метод	Назначение
Activate()	Активирует указанную форму и помещает её в фокус
Close()	Закрывает форму
CenterToScreen()	Помещает форму в центр экрана
LayoutMDI()	Размещает все дочерние формы на родительском в соответствии со значением перечисления
OnResize()	Может быть замещён для реагирования на событие Resize()
ShowDialog()	Отображает форму как диалоговое окно

4.3 ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

За элементы управления отвечает класс *Control*. Данный класс позволяет определять общие черты всех элементов относящихся к графическому интерфейсу. Наиболее важные свойства данного класса отображены в табл. 4.5.

Таблица 4.5

Наиболее важные свойства класса *Control*

Свойство	Назначение
Top, Left, Right, Bottom	Ориентирование текста в элементе управления
ClientRetangle, Height, Widht	Область элемента управления (прямоугольник), высота и ширина
Created, Disposed, Enabled, Focused, Visible	Возвращает bool, определяет текущее состояние
Handle	Номер HWND для элемента управления
ModiferKeys	Проверка состояния модифицирующих клавиш (Shift, Alt и Ctrl). Результат в виде объекта Keys
MouseButtons	Проверяет состояния клавиш мыши
TabIndex, TabStop	Используются для настройки последовательности перемещения по Tab
Text	Надпись, ассоциированная с элементом управления
Parent	Возвращает родительский объект Control

Для создания элемента управления достаточно перетянуть его на форму либо сделать это вручную. При создании элемента управления вручную требуется провести 3 операции: создать объект, настроить его (задать свойства) и добавить в коллекцию *Controls*. Основные элементы

управления перечислены в табл. 4.1. В следующей программе на созданную ранее форму помещается кнопка. Пока что эта кнопка бездействует:

```
// Добавление кнопки.
using System;
using System.Windows.Forms;
using System.Drawing;
class ButtonForm : Form
{
Button MyButton = new Button();
public ButtonForm()
{
Text = "Реакция на щелчок";
MyButton = new Button();//создание
MyButton.Text = "Щелкните";//настройка
MyButton.Location = new Point(100, 200);
// Добавляем в список обработчик событий кнопки.
MyButton.Click += new EventHandler(MyButtonClick);
this.Controls.Add(MyButton);//добавление в коллекцию
}
[STAThread] %
public static void Main()
{
ButtonForm skel - new ButtonForm();
Application.Run(skel);
}
protected void MyButtonClick(object who, EventArgs e)
{// Обработчик для кнопки MyButton
if (MyButton.Top === 200)
MyButton.Location = new Point (10, 10);
else
MyButton.Location = new Point (100, 200);
}
}
```

В этой программе создается класс *ButtonForm*, который является производным от класса *Form*. Он содержит поле типа *Button* с именем *MyButton*. В конструкторе класса *ButtonForm* кнопка создается, инициализируется и помещается на форму. Чтобы заставить кнопку выполнять какие-либо действия, необходимо добавить в программу обработчик сообщений.

Обработка сообщений

Чтобы программа реагировала на щелчок на кнопке (или на какое-либо другое действие пользователя), необходимо обеспечить обработку сообщения, которое генерирует эта кнопка. В общем случае, когда пользователь воздействует на элемент управления, его действие передается программе в виде сообщения. В C# программе, основанной на применении окон, такие сообщения обрабатываются обработчиками событий. Следовательно, чтобы получить сообщение, программа должна добавить собственный обработчик событий в список обработчиков, вызываемых при генерировании сообщения. Для сообщений, связанных со щелчком на кнопке, это означает добавление обработчика для события *Click*. В примере выше ожидание события представляется следующим кодом:

MyButton.Click += new EventHandler(MyButtonClick);

По нажатию на кнопку вызовется метод *MyButtonClick*. В зависимости от элемента управления методы и события могут быть разными.

TextBox

Элемент *TextBox* предназначен для хранения стоки (строк) текста. Данный элемент унаследован от класса *TextBoxBase*, свойства которого определены в табл. 4.6. Данный элемент взаимодействует с буфером обмена, кнопкой отмена и прочими возможностями редактирования.

Таблица 4.6

Свойства, определённые в классе *TextBoxBase*

Свойство	Назначение
AcceptsTab	Определяет, что будет производиться при нажатии на клавишу Tab: вставка символа табуляции в само поле или переход к другому элементу управления
AutoSize	Определяет, будет ли элемент управления автоматически изменять размер при изменении шрифта в нем
BackColor, ForeColor	Позволяют получить или установить значение цвета фона и переднего плана
HideSelection	Указывает, будет ли текст выделенным после выведения элемента из фокуса
MaxLength	Максимальное количество символов
Modified	Определяет, был ли текст изменен пользователем
Multiline	Определяет наличие нескольких строк
ReadOnly	Помечает элемент «только для чтения»
SelectedText	Содержит выделенный текст в элементе
SelectionStart	Позволяет получить начало выделенного текста
WordWrap	Определяет, будет ли текст автоматически переноситься на новую строчку при достижении предельной длины строки

Для класса *TextBox* определены свои свойства, указанные в табл. 4.7.

Таблица 4.7

Свойства, определенные в классе *TextBox*

Свойство	Назначение
AcceptsReturn	Позволяет определить реакцию программы на нажатие Enter: либо начинается новая строка, либо активизируется кнопка
CharacterCasing	Определяет возможность изменения регистра символов
PasswordChar	Определяет поле как поле ввода пароля
ScrollBars	Определяет присутствие/отсутствие полосы прокрутки
TextAlign	Выравнивание текста(Center, Left, Right)

ButtonBase

Кнопка – самый простой и часто используемый элемент управления. Базовый класс кнопки *ButtonBase*. Основные свойства данного класса представлены в табл. 4.8. От класса *ButtonBase* наследованы классы *Button*, *CheckBox* и *RadioButton*.

Таблица 4.8

Основные свойства класса *ButtonBase*

Свойство	Назначение
FlatStyle	Настройка рельефности кнопки
Image	Задает изображение на кнопке
ImageAlign	Определяет выравнивание изображения на кнопке
ImageIndex, *ImageList*	Используются для работы с набором изображений на кнопке
IsDefault	Определяет, является ли данная кнопка кнопкой «по умолчанию» (срабатывать по нажатию Enter)
TextAlign	Выравнивание текста на кнопке

CheckBox

Класс *CheckBox* определяет флажки. Для этого элемента управления определены три состояния *Checked* (флаг установлен), *Indeterminate* (флаг не определен) и *Unchecked* (флаг снят). Свойства класса *CheckBox*, не определённые в классе *ButtonBase* указаны в табл. 4.9.

RadioButton

Переключатель, как правило, объединяется в группы, имея возможность выбора только одного элемента. Для объединения нескольких *Ra-*

dioButton в группу используется объект класса *GroupBox*. От класса *CheckBox* отличается наличием события *CheckedChanged*(возникает при изменении *Checked*) и не поддерживает свойство *TreeState*.

Таблица 4.9

Основные свойства класса *CheckBox*

Свойство	Назначение
Appearance	Настраивает вид флажка
AutoCheck	Позволяет определить будет ли значение *Checked* и *ChackState* меняться при щелчке на объекте
CheckAlign	Выбор вертикального и горизонтального выравнивания
Checked	Возвращает значение bool, представляющее текущее состояние
ChackState	Позволяет получить или установить значение флажка
ThreeState	Определяет, будет ли для флажка использоваться три значения или только два

CheckedListBox

Визуально представляет собой список с флажками. Унаследован от *Control*; для добавления элементов служат методы *Add* и *AddRange* (для массива элементов).

ListBox

Список строк служит для отображения данных. Основные свойства класса *ListBox* показаны в табл. 4.10.

Таблица 4.10

Основные свойства класс *ListBox*

Свойство	Назначение
ScrollAlwaysVisible	Определяет наличие полосы прокрутки постоянно
SelectedIndex	Индекс выделенного в данный момент элемента в списке (если не выделен никто –1)
SelectedIndices	Набор индексов выделенных в данный момент элементов
SelectedItem	Значение выделенного в данный момент элемента
SelectedItems	Коллекция выделенных в данный момент элементов
SelectionMode	Количество элементов возможных для одновременного выбора
Sorted	Определяет, будет ли отсортирован список по алфавиту
TopIndex	Возвращает индекс первого видимого элемента в списке

ComboBox

Комбинированные списки позволяют пользователю производить выбор элемента из заранее определённых элементов. Данный класс унаследо-

вал большинство своих возможностей от *ListBox*. Собственные важные свойства определены в табл. 4.11.

Таблица 4.11

Свойства класса *ComboBox*

Свойство	Назначение
DroppedDown	Определяет, будет ли список ниспадающим
MaxDropDownItems	Определяет максимальное число элементов, которое будет показано в нижней части ниспадающего списка (1..100)
MaxLength	Определяет максимальную длину текста доступную для ввод.
SelectedIndex	Определяет индекс выбранного элемента
SelectedItem	Возвращает ссылку на выбранный элемент
SelectedText	Возвращает выделенный текст в поле редактирования
SelectionLength	Возвращает длину выделенного текста в поле редактирования
Style	Позволяет установить стиль
Text	Позволяет получить доступ к тексту в поле редактирования

TrackBar

Шкала с ползунком позволяет пользователю производить выбор значений из предложенного диапазона с помощью графических средств.

Panel и GroupBox

Оба элемента управления предназначены для логического объединения элементов управления. Различия заключаются в том, что *Panel* наследован от *ScrollableControl* и может иметь полосу прокрутки, но не может иметь встроенного заголовка. Добавление в оба элемента происходят одинаково:

```
panel1.AutoScroll=true;
panel1.Controls.Add(this.label2);
```

4.4 ПРОСТРАНСТВО ИМЕН WINDOWS.DRAWING

В пространстве *Windows.Drawing* определены основные типы для вывода графики (работа с перьями, кистью, шрифтами и т.д.). Для *Windows.Drawing* имеются внутренние пространства имен. Основное назначение внутренних пространств имен перечислено в табл. 4.12.

Таблица 4.12

Пространства имен *Windows.Drawing*

Пространство имен	Назначение
System.Drawing.Drawing2d	Предоставляет расширенную функциональную возможность векторной и двухмерной графики
System.Drawing.Imaging	Предоставляет расширенные функциональные возможности визуализации изображения GDI+
System.Drawing.Printing	Определяет типы для вывода графики на принтер
System.Drawing.Text	Предоставляет расширенные функциональные возможности GDI+ для работы с текстом

Основные типы пространства имен *System.Drawing* перечислены в табл. 4.13.

Таблица 4.13

Основные типы пространства имен *System.Drawing*

Тип	Назначение
Bitmap	Инкапсулирует файл изображение и определяет набор методов для выполнения различных операций над этим изображением
Brush, Brushes, SolidBrush, SystemBrushes, TextureBrush	Кисти используются для заполнения пространства внутри геометрических фигур
Color, SystemColor, ColorTranslator	Структура определяет набор статических полей, которые могут быть использованы для настройки цвета
Font, FontFamily	Данные классы инкапсулируют характеристики шрифта (наборов шрифтов)
Graphics	Определяет набор методов для вывода текста, изображений и геометрических фигур
Icon, SystemIcons	Классы предназначены для работы со значками
Image, ImageAnimator	*Image* – это абстрактный класс для работы типов *Bitmap*, *Icon* и *Cursor*. *ImageAnimator* – класс для отображения коллекции изображений
Pen, Pens, SystemPens	*Pen* – класс для рисования линий
Point, PointF	Класс для работы с точками (*Point* – с *int*, *PointF* – с *float*)
Rectangle, RectangleF	Структуры для работы с прямоугольными областями
Size, SizeF	Структуры работы с размерами
StringFormat	Используется для форматирования текста (размера, выравнивания и т.д.)
Region	Определяет область, занятую фигурой

System.Drawing.Text

Классы этого пространства имен позволяют создавать и использовать коллекции шрифтов. Основные классы и перечисления представлены в табл. 4.14.

Таблица 4.14

Основные классы и перечисления System.Drawing.Text

Класс/перечисление	Описание
Класс	
FontCollection	Базовый класс установленных и закрытых коллекций шрифтов. В классе определен метод для получения списка семейств шрифтов, хранимых в коллекции
InstalledFontCollection	Представляет установленные в системе шрифты. Этот класс не может быть унаследован
PrivateFontCollection	Коллекция семейств шрифтов, созданных из файлов шрифтов, предоставленных приложением-клиентом
Перечисление	
GenericFontFamilies	Задает общий объект FontFamily
HotkeyPrefix	Задает тип отображения префиксов сочетания клавиш, относящихся к тексту
TextRenderingHint	Задает качество отображения текста

System.Drawing. Imaging

Класс используется для визуализации изображений. Основные классы и перечисления представлены в табл. 4.15.

Таблица 4.15

Основные классы и перечисления System.Drawing.Imaging

Класс/ Перечисление	Описание
Класс	
BitmapData	Определяет атрибуты точечного изображения. Класс *BitmapData* используется методами *LockBits* и *UnlockBits* класса *Bitmap*. Не наследуется
ColorMap	Определяет карту для преобразования цветов. Несколько методов класса *ImageAttributes* настраивают цвета изображения с помощью таблицы преобразования цветов, которая представляет собой массив структур *ColorMap*. Не наследуется
ColorMatrix	Определяет матрицу 5×5, которая содержит координаты для пространства RGBA. Несколько методов класса *ImageAttributes* настраивают цвета изображения с помощью цветовой матрицы. Не наследуется

Продолжение табл. 4.15

Класс/ Перечисление	Описание
	Класс
ColorPalette	Определяет массив цветов, которые составляют цветовую палитру. Цвета – это 32-разрядные цвета ARGB. Не наследуется
Encoder	Объект *Encoder* инкапсулирует глобально уникальный идентификатор GUID, который определяет категорию параметра кодировщика изображения
EncoderParameter	Можно использовать объект *EncoderParameter* для передачи массива значений кодировщику изображений. Также можно использовать объект *EncoderParameter* для получения списка возможных, поддерживаемых определенным параметром определенного кодировщика изображений
EncoderParameters	Инкапсулирует массив объектов EncoderParameter
FrameDimension	Предоставляет свойства, получающие размеры кадра изображения. Не наследуется
ImageAttributes	Объект *ImageAttributes* содержит информацию о том, каким образом обрабатываются цвета точечных рисунков и метафайлов во время визуализации. Объект *ImageAttributes* обслуживает несколько параметров настройки цвета, включая матрицы настройки цвета, матрицы настройки черно-белой палитры, значения гамма-коррекции, таблицы карт цвета и значения цветового порога. Не наследуется
ImageFormat	Указывает формат изображения. Не наследуется
Metafile	Определяет графический метафайл. Метафайл содержит записи, описывающие последовательность графических операций, которые могут быть записаны (созданы) и воспроизведены (выведены на экран). Не наследуется
MetafileHeader	Содержит атрибуты связанного объекта *Metafile*. Не наследуется
MetaHeader	Содержит информацию о метафайле WMF
PropertyItem	Инкапсулирует свойство метаданных, включаемое в файл изображения. Не наследуется
WmfPlaceableFile Header	Определяет размещаемый метафайл. Не наследуется
	Перечисление
ColorAdjustType	Определяет, какие объекты GDI+ используют информацию настройки цвета
ColorChannelFlag	Определяет отдельные каналы в цветовом пространстве CMYK (голубой, пурпурный, желтый, черный). Перечисление, используемое методами *SetOutputChannel Methods*
ColorMapType	Определяет типы карт цветов

Окончание табл. 4.15

Класс/ Перечисление	Описание
	Перечисление
ColorMatrixFlag	Определяет типы изображений и цвета, которые находятся под влиянием параметров настройки цветной или черно-белой палитры объекта *ImageAttributes*
ColorMode	Определяет два режима для значений составляющих цвета
EmfPlusRecordType	Определяет методы, доступные для чтения и записи графических команд
EmfType	Определяет характер записей, размещенных в файле EMF. Перечисление, используемое несколькими конструкторами в классе Metafile
EncoderParameterValueType	GDI+ использует кодировщики изображений для преобразования изображений, хранящихся в объектах Bitmap, в различные форматы файлов. Кодировщики изображений, встроенные в GDI+, поддерживают форматы BMP, JPEG, GIF, TIFF и PNG. Кодировщик вызывается при вызове методов *Save* или *SaveAdd* объекта *Bitmap*
EncoderValue	Когда вызывается метод *Save* или *SaveAdd* объекта *Image*, можно передавать параметры кодировщику изображения путем передачи объекта *EncoderParameters* методу *Save* или *SaveAdd*. Объект *EncoderParameters* содержит массив объектов *EncoderParameter*. Каждый объект *EncoderParameter* имеет массив значений и свойство *Encoder*, указывающее категорию параметра. Перечисление *EncoderValue* предоставляет имена для некоторых значений, которые могут быть переданы кодировщикам изображений JPEG и TIFF
ImageCodecFlags	Предоставляет флаги для использования с кодеками
ImageFlags	Указывает атрибуты данных о точках, содержащихся в объекте *Image*. Свойство *Image.Flags* возвращает члена данного перечисления
ImageLockMode	Указывает флаги, передающиеся параметру флагов метода *Bitmap.LockBits*. Метод *LockBits* блокирует часть изображения так, что можно считывать или записывать данные о точках
MetafileFrameUnit	Указывает единицу измерения прямоугольника, используемого для определения размеров и положения метафайлов. Это указывается во время создания объекта *Metafile*
MetafileType	Указывает типы метафайлов. Свойство *MetafileHeader.Type* возвращает члена данного перечисления
PaletteFlags	Указывает тип данных о цвете в системной палитре. Это могут быть данные о цвете с альфа-составляющей, только черно-белые данные или полутоновые данные
PixelFormat	Указывает формат данных о цвете для каждой точки изображения

Для прорисовки текста на форме используется следующий код:

protected override void OnPaint(PaintEventArgs e)//переопределяем событие //OnPaint возникающее при прорисовке формы

{

CenterToScreen();

Graphics g= e.Graphics;

g.DrawString("Hello",new Font("Times New Roman", 20,

new SolidBrush(Color.Black), 0 ,0);

}

Для перерисовки формы используется функция *Invalidate(Rectangle)*. В табл. 4.16 перечислены основные методы класса *Graphics*.

Таблица 4.16

Основные методы класса *Graphics*

Метод	Назначение
FromHdc(), FromHwnd(), FromImage()	Получают объект *Graphics* из элемента управления или изображения
Clear()	Заполняет объект *Graphics* выбранным цветом, удаляя все его содержимое
DrawArc(), DrawBezier(), DrawEllipse(), DrawIcon(), DrawLine(), DrawPie(), DrawPath(), DrawRectangle(), DrawRectangles(), DrawString()	Прорисовка нужных фигур
FillEllipse(), FillPath(), FillPie(), FillPolygon(), FillRectangle()	Заполнение внутренних полей фигур
MesureString()	Возвращает структуру *Size*, представляющую границы блока

5 ДОСТУП К БАЗАМ ДАННЫХ

5.1 ОПИСАНИЕ ЯЗЫКА SQL

Рассматриваемый ниже непроцедурный язык SQL (Structured Query Language – структуризованный язык запросов) ориентирован на операции с данными, представленными в виде логически взаимосвязанных совокупностей таблиц.

В SQL используются основные типы данных, которые представлены в табл. 5.1.

Таблице 5.1

Основные типы данных SQL

Формат	Описание
INTEGER	Целое число (обычно до 10 значащих цифр и знак)
SMALLINT	«Короткое целое» (обычно до 5 значащих цифр и знак)
DECIMAL(p,q)	Десятичное число, имеющее p цифр ($0 < p < 16$) и знак; с помощью q задается число цифр справа от десятичной точки ($q < p$, если $q = 0$, оно может быть опущено)
DOUBLE FLOAT	Вещественное число с 15 значащими цифрами и целочисленным порядком, определяемым типом СУБД
CHAR(n)	Символьная строка фиксированной длины из n символов ($0 < n < 256$)
VARCHAR(n)	Символьная строка переменной длины, не превышающей n символов ($n > 0$ и различные n в разных СУБД, но не меньше 4096)
DATE	Дата в формате, определяемом специальной командой (по умолчанию *mm/dd/yy*); поля даты могут содержать только реальные даты, начинающиеся за несколько тысячелетий до н.э.
TIME	Время в формате, определяемом специальной командой, (по умолчанию *hh.mm.ss*)
DATETIME	Комбинация даты и времени
MONEY	Деньги в формате, определяющем символ денежной единицы ($, руб, ...) и его расположение (суффикс или префикс), точность дробной части и условие для показа денежного значения
LOGICAL	Логическое значение «ИСТИНА» или «ЛОЖЬ»

SELECT

Все запросы на получение практически любого количества данных из одной или нескольких таблиц выполняются с помощью единственного предложения *SELECT*. Синтаксические конструкции используются обозначения представленные в табл. 5.2.

Таблица 5.2

Синтаксические конструкции предложения SELECT

Конструкция	Описание
*	Употребляется в обычном для программирования смысле, т.е. «все случаи, удовлетворяющие определению»
[]	Означает, что конструкции, заключенные в эти скобки, являются необязательными (т.е. могут быть опущены)
{}	Означает, что конструкции, заключенные в эти скобки, должны рассматриваться как целые синтаксические единицы, т.е. они позволяют уточнить порядок разбора синтаксических конструкций, заменяя обычные скобки, используемые в синтаксисе SQL
...	Указывает на то, что непосредственно предшествующая ему синтаксическая единица факультативно может повторяться один или более раз
|	Означает наличие выбора из двух или более возможностей
;	Завершающий элемент предложений SQL
,	Используется для разделения элементов списков
пробел	Могут вводиться для повышения наглядности между любыми синтаксическими конструкциями предложений SQL
прописные латинские буквы и символы	Используются для написания конструкций языка SQL и должны (если это специально не оговорено) записываться в точности так, как показано
строчные буквы	Используются для написания конструкций, которые должны заменяться конкретными значениями, выбранными пользователем, причем для определенности отдельные слова этих конструкций связываются между собой символом подчеркивания «_»
термины таблица, столбец, ...	Заменяют (с целью сокращения текста синтаксических конструкций) термины имя_таблицы, имя_столбца, ..., соответственно
термин таблица	Используется для обобщения таких видов таблиц, как базовая_таблица, представление или псевдоним; здесь псевдоним служит для временного (на момент выполнения запроса) переименования и (или) создания рабочей копии базовой_таблицы (представления)

В предложении *SELECT* могут использоваться символы представленные в табл. 5.3.

Таблица 5.3

Элементы предложения SELECT

Элемент	Описание
SELECT	(выбрать) данные из указанных столбцов и (если необходимо) выполнить перед выводом их преобразование в соответствии с указанными выражениями и (или) функциями

Окончание табл. 5.3

Элемент	Описание
FROM	(из) перечисленных таблиц, в которых расположены эти столбцы
WHERE	(где) строки из указанных таблиц должны удовлетворять указанному перечню условий отбора строк
ORDER BY	(упорядочить) строки результата по значениям первого столбца списка пока не появится несколько строк с одинаковыми значениями данных в этом столбце, такие строки сортируются по значениям следующего столбца из списка
GROUP BY	(группируя по) указанному перечню столбцов с тем, чтобы получить для каждой группы единственное агрегированное значение
HAVING	(имея) в результате лишь те группы, которые удовлетворяют указанному перечню условий отбора групп
UNION	Позволяет объединить различные запросы, при соблюдении количества выборок и их типов

Для исключения дубликатов и одновременного упорядочения перечня необходимо добавить, перед списком столбцов, во фразе *SELECT* ключевое слово *DISTINCT*.

Упорядочение по фразе *ORDER BY* можно производить в порядке возрастания – *ASC col* или убывания – *DESC col*, а по умолчанию принимается *ASC*.

Кроме традиционных операторов сравнения (= | <> | < | <= | > | >=) в *WHERE* используются условия представленные в табл. 5.4.

Таблица 5.4

Специальные операторы сравнения

Операторы	Описание
col BETWEEN val1 AND val2	Истина для *col* в промежутке *val1* и *val2* включительно
col LIKE fmt	Истина, если *fmt* соответствует *col*. *fmt* может использовать символ «%», для формирования маски поиска
col IN (val1, val2, ...)	Истина, если *col* соответствует одному из *val1*, *val2*, ...
col IS NULL	Истина, если *col* неопределенно
EXISTS(выражение)	В качестве *выражения* можно использовать любое логическое выражение ли отдельный запрос. Истина, если истинно *выражение*

Критерий отбора формируется из одного или нескольких условий, соединенных логическими операторами, представленными в табл. 5.5.

Таблица 5.5

Логические операторы условий

Оператор	Описание
AND	Истина, когда должны удовлетворяться оба разделяемых с помощью AND условия
OR	Истина, когда должно удовлетворяться одно из разделяемых с помощью OR условий
NOT	Истина, когда должно удовлетворяться условие в случае его логического невыполнения

GROUP BY инициирует перекомпоновку формируемой таблицы по группам, каждая из которых имеет одинаковое значение в столбцах, включенных в перечень *GROUP BY*. Далее к этим группам применяются агрегирующие функции, представленные в табл. 5.6.

Таблица 5.6

Агрегирующие функции

Функция	Описание
SUM(col)	Сумма значений в столбце *col*
COUNT(col)	Число значений в столбце *col*
MIN(col)	Самое малое значение в столбце *col*
MAX(col)	Самое большое значение в столбце *col*
AVG(col)	Среднее значение в столбце *col*

Фраза *HAVING* играет такую же роль для групп, что и фраза *WHERE* для строк: она используется для исключения групп, точно так же, как *WHERE* используется для исключения строк. Эта фраза включается в предложение лишь при наличии фразы *GROUP BY*, а выражение в *HAVING* должно принимать единственное значение для группы.

DELETE

Предложение *DELETE* имеет формат:

DELETE

FROM базовая таблица | представление

[WHERE фраза];

DELETE позволяет удалить содержимое всех строк указанной таблицы (при отсутствии *WHERE* фразы) или тех ее строк, которые выделяются *WHERE* фразой.

INSERT

Предложение *INSERT* имеет один из следующих форматов:

INSERT

INTO {базовая таблица | представление} [(столбец [,столбец] ...)]

VALUES ({константа | переменная} [,{константа | переменная}] ...);

или

INSERT

INTO {базовая таблица | представление} [(столбец [,столбец] ...)]

подзапрос;

В первом формате в таблицу вставляется строка со значениями полей, указанными в перечне фразы *VALUES* (значения), причем *i*-тое значение соответствует *i*-тому столбцу в списке столбцов (столбцы, не указанные в списке, заполняются NULL-значениями). Если в списке *VALUES* фразы указаны все столбцы модифицируемой таблицы и порядок их перечисления соответствует порядку столбцов в описании таблицы, то список столбцов во фразе *INTO* можно опустить. Однако не советуем этого делать, так как при изменении описания таблицы (перестановка столбцов или изменение их числа) придется переписывать и *INSERT* предложение.

Во втором формате сначала выполняется подзапрос, т.е. по предложению *SELECT* в памяти формируется рабочая таблица, а потом строки рабочей таблицы загружаются в модифицируемую таблицу. При этом *i*-тый столбец рабочей таблицы (*i*-тый элемент списка *SELECT*) соответствует *i*-тому столбцу в списке столбцов модифицируемой таблицы. При выполнении указанных выше условий может быть опущен список столбцов фразы *INTO*.

UPDATE

Предложение *UPDATE* также имеет два формата. Первый из них:

UPDATE (базовая таблица | представление}

SET столбец = значение [, столбец = значение] ...

[WHERE фраза]

где значение - это

столбец | выражение | константа | переменная

UPDATE может включать столбцы лишь из обновляемой таблицы, т.е. значение одного из столбцов модифицируемой таблицы может заменяться на значение ее другого столбца или выражения, содержащего значения нескольких ее столбцов, включая изменяемый.

При отсутствии *WHERE* обновляются значения указанных столбцов во всех строках модифицируемой таблицы. *WHERE* позволяет сократить число обновляемых строк, указывая условия их отбора.

Второй формат описывает предложение, позволяющее производить обновление значений модифицируемой таблицы по значениям столбцов из других таблиц. В ряде СУБД эти форматы отличаются друг от друга и от стандарта.

5.2 ПРОСТРАНСТВА ИМЕН ADO.NET

Все типы ADO.NET предназначены для выполнения единого набора задач: установить соединение с хранилищем данных, создать и заполнить данными объект *DataSet*, отключиться от хранилища данных и вернуть изменения, внесенные в объект *DataSet*, обратно в хранилище данных. Объект *DataSet* – это тип данных представляющий локальный набор таблиц и информацию об отношениях между ними. После создания объекта *DataSet* и его заполнения данными мы можем программными средствами производить запросы к нему и перемещаться по таблицам.

В составе ADO.NET включены два провайдера: провайдер SQL (специально оптимизирован под взаимодействие с Microsoft SQL Server) и провайдер OleDb (для любых хранилищ данных, поддерживающих протокол OLE DB).

Все возможности ADO.NET заключены в типах данных, заключенных в соответствующих пространствах имен, основные из которых представлены в табл. 5.7).

Таблица 5.7

Пространства имен ADO.NET

Пространство имен	**Описание**
System.Data	Главное пространство имен ADO.NET. В нем определены типы, представляющие таблицы, столбцы, записи, ограничения, и самый важный тип *DataSet*. В этом пространстве имен только типы, представляющие сами данные
System.Data.Common	Здесь определены типы, общие для всех управляемых провайдеров
System.Data.OleDb	Определены типы для установления соединений с OLE DB-совместимыми источниками данных, выполнения к ним SQL-запросов и заполнения данными объектов *DataSet*

Окончание табл. 5.7

Пространство имен	Описание
System.Data.SqlClient	В данном пространстве определены типы, которые составляют управляемый провайдер SQL
System.Data.SqlTypes	Представляют собой родные типы данных Microsoft SQL Server. Использование этих типов позволяет добиваться наивысшей производительности

Все пространства имен ADO.NET расположены в одной сборке – *System.Data.dll*. В любом проекте использующим ADO.NET должна быть подключена ссылка на сборку *System.Data.dll*. При использовании ADO.NET требуется использовать *System.Data* в любой ситуации и *System.Data.OleDb* или *System.Data.SqlClient*.

System.Data

Наиболее часто используемые типы пространства имен *System.Data* представлены в табл. 5.8.

Таблица 5.8

Основные типы пространства имен *System.Data*

Тип	Назначение
DataColumn	Представляет один столбец в объекте *DataSet*
DataColumnCollection	Представляет все столбцы в объекте *DataSet*
Constrain	Объектно-ориентированная оболочка вокруг ограничения, наложенного на один или несколько *DataColumn*
ConstrainCollection	Все ограничения в объекте *DataTable*
DataRow	Строка в *DataTable*
DataRowCollection	Все строки в *DataTable*
DataView	Созданное программным образом представление объекта *DataTable*, которое возможно сортировать, фильтровать и т.д.
DataRowView	Позволяет создать настроенное представление одной строки
DataSet *ForeignKeyConstraint*	Объект, создаваемый в памяти компьютера, состоит из множества *DataTable* и информации об отношениях между ними Ограничение, налагаемое на набор столбцов в таблице, связанных отношением первичный-внешний ключ
UniqueConstraint	Ограничение, при помощи которого гарантируется, что в столбце не будет повторяющихся записей
DatarelationCollection, *DataRelation*	Набор всех отношений (объектом *DataRelation*) между таблицами в *DataSet*
DataTableCollection, *DataTable*	Набор всех таблиц (объектов *DataTable*) в *DataSet*

DataTable

Объект *DataTable* может быть создан программно либо в результате запроса к базе данных. Объект *DataTable* обладает рядом открытых свойств, в число которых входит коллекция *Columns*, которая возвращает объект *DataColumnCollection*, состоящий из объектов *DataColumn*. Свойства класса *DataTable* перечислены в табл. 5.9.

Таблица 5.9

Свойсва класса *DataTable*

Свойство	Назначение
CaseSensitive	Будет ли при сравнении символьных данных в таблице учитываться регистр букв
ChildRelations	Возвращает коллекцию подчинённых отношений (если есть) для объекта *DataTable*
Columns	Возвращает набор столбцов для таблицы
Constraints	Позволяет получить коллекцию ограничений, определённых в таблице
DataSet	Ссылка на объект *DataSet*
DefaultView	Возвращает настроенное представление для таблицы
MinimumCapaticy	Исходное количество строк в таблице
ParentRealtions	Позволяет получить коллекцию родительских отношений для данного объекта *DataTable*
PrimaryKey	Массив столбцов, которые являются первичным ключом в таблице
Rows	Набор строк, относящийся к таблице
TableName	Имя таблицы

DataColumn

Объект *DataColumn* представляет собой столбец таблицы. В табл. 5.10 перечислены свойства класс *DataColumn*.

Таблица 5.10

Основные свойства класса *DataColumn*

Свойство	Назначение
AllowDbNull	Может ли столбец содержать значение *NULL*
AutoIncrement, AutoIncrementSeed, AutoIncrementStep	Настройки автоматического приращения значений в столбце
Caption	Заголовок столбца
ColumnMapping	Будет ли столбец при сохранении *DataSet* представлен в XML
ColumnName	Позволяет получить или установить имя столбца в коллекции *Columns* (по умолчанию Column1, Column2 и т.д)
DataType	Тип данных

Окончание табл. 5.10

Свойство	Назначение
DefaultValue	Значение по умолчанию
Expression	Позволяет получить или установить выражение, используемое для фильтрации новых строк, вычисления значения в столбце и т.д.
Ordinal	Позволяет установить порядковый номер столбца в коллекции *Columns*
ReadOnly	Определяет, будет ли столбец только для чтения
Table	Возвращает *DataTable*, к которой принадлежит данный объект *DataColumn*
Unique	Определяет, будет ли в столбце повторяющиеся значения (для ключа *true*)

Rows

Коллекция *Rows* объекта *DataTable* возвращает набор строк указанной таблицы. Эта коллекция используется для изучения результатов запроса к базе данных. Циклический перебор элементов коллекции *Rows* позволяет проанализировать каждую запись. В ADO.NET программист не перебирает элементы *DataSet*; вместо этого он обращается к нужной таблице и перебирает элементы коллекции *Rows* (как правило, в цикле *foreach*).

DataRow

Данная коллекция определяет собственно данные в таблице. Каждый объект *DataRow* представляет собой одну строку из таблицы. Не требуется создавать объекты данного класса, ссылку на них можно получить из класса *DataTable*. Члены класса *DataRow* описаны в табл. 5.11.

DataRelation

Кроме коллекции *Tables* объект *DataSet* имеет свойство *Relations*, которое возвращает коллекцию *DataRelationCollection*, состоящую из объектов *DataReiation*. Каждый такой объект представляет собой отношение между двумя таблицами, выраженное через объекты *DataColumn*.

Адаптер данных

Объект *DataSet* является абстрактным представлением реляционной базы данных. В модели ADO.NET мостом между объектом *DataSet* и источником данных (базой данных) служит *DataAdapter*. Он предоставляет программисту метод *Fill*(), позволяющий извлекать информацию из базы данных и заполнять ею объект *DataSet*.

Таблица 5.11

Члены класса *DataRow*

Член класса	Назначение
AcceptChanges(), RejectChanges()	Для записи в строку (или отказа) всех изменений, произведенных, начиная с момента, когда последний раз был вызван метод *AcceptChanges()*
BeginEdit(), EndEdit(), CanselEdit()	Начать, завершить, прекратить операции редактирования для объекта *DataRow*
Delete()	Помечает строку для удаления при следующем вызове метода *AcceptChanges()*
HasErrors, GetColumnsinErrors(), GetColumnError(), ClearColumn(), RowError	Свойство *HasErrors* определяет, были ли ошибки в значениях столбцов для данной строки. Если такие ошибки есть, то возможно удаление всех ошибок, получение ошибок, настроить значения в столбце с ошибкой
IsNull()	Содержит ли строка значение *NULL*
ItemArray	Позволяет получить и установить значение всех полей строки при помощи массива
RowState	Позволяет получать информацию о текущем состоянии объекта *DataRow*
Table	Указатель на таблицу

DataAdapter

Вместо жесткой привязки объекта *DataSet* к архитектуре базы данных модель ADO.NET предоставляет объект *DataAdapter* в качестве посредника между *DataSet* и базой. Разъединение объекта *DataSet* и базы данных позволяет одному объекту этого класса получать информацию из нескольких баз данных или иных источников информации.

Управляемый провайдер OLE DB

Наиболее важные типы пространства имен *System.Data.OleDb* указаны в табл. 5.12.

При работе с управляемым драйвером OLE DB требуются следующие действия: установить соединения с источником данных при помощи типа *OleDbConnection*. Для *OLEDbConnection* предусмотрено использование строки подключения, состоящей из пар: имя – значение. С ее помощью можно задать имя компьютера, к которому производится подключение, параметры безопасности подключения, имя базы данных, к которой производится подключение и имя самого провайдейра OLE DB.

//Создаем строку соединения

```
OleDbConnection cn= new OleDbConnection();
cn.ConnectionString="Provider = SQLOLEDB.1; " + //имя провайдера
"Integrated security = SSPI ; " +
"Persist Security Info = False; " +
"Initial Catalog = Cars; " + //имя базы данных
"Data Source = BIGMANU;";//имя компьютера
cn.Open();
//действия с БД
cn.Close();
```

Таблица 5.12

Основные типы пространства имен *System.Data.OleDb*

Тип	Описание
OleDbCommand	Представляет SQL запрос к источнику данных
OleDbConnection	Представляет открытое соединение с источником данных
OleDbDataAdapter	Представляет соединение с базой данных и набор команд используемых для заполнения объекта *DataSet*, а также обновление исходной базы данных после внесения изменений в *DataSet*
OleDbDataReader	Обеспечивает метод считывания потока данных из источника в одном направлении (вперед)
OleDbParameter	Используется для передачи параметров процедуре, хранимой на источнике данных. Параметры предоставлены объектами *OleDbParameter*
OleDbParameterCollection	Используется для передачи параметров процедуре, хранимой на источнике данных. Весь набор параметров предоставлен объектом *OleDbParameterCollection*
OleDbError	Ошибка, возвращаемая источником данных
OleDbErrorCollection	Набор ошибок, возвращаемый источником данных
OleDbErrorException	Исключение, генерирующее при возникновении ошибки

Наиболее часто используемые имена провайдеров: *Microsoft.JET.OLEDB.4.0* (БД Access), *MSDAORA* (БД Oracle), *SQLOLEDB* (БД MS SQL Server)

Члены класса *OleDbConnection* перечислены в табл. 5.13.

Таблица 5.13

Члены класса *OleDbConnection*

Член класса	Описание
BeginTransaction(), CommitTransaction(), RollbackTransaction()	Используется для того, чтобы программным образом начать транзакцию, завершить её или отменить
Close()	Закрыть соединение с источником БД
ConnectionString	Строка подключения
ConnectionTimeout	Тайм-аут при установке соединения
Database	Название текущей БД
DataSource	Позволяет получить или установить имя
Open()	Открывает соединение с базой данных, используя текущие настройки свойств соединения
Provider	Имя провайдера
State	Позволяет получить информацию о текущем состоянии соединения

Построение команды SQL

Для выполнения запроса SQL служит класс *OleDbCommand*. Множество типов ADO.NET принимает объект *OleDbCommand* в качестве параметра для передачи запроса к источнику данных. Основные члены класса *OleDbCommand* перечислены в табл. 5.14.

Таблица 5.14

Члены класса *OleDbCommand*

Член класса	Описание
Cancel()	Прекращает выполнение команды
CommandText	Позволяет получить или задать текст команды на SQL, который будет передан источнику
CommandTineout	Время тайм-ута на выполнение команды (30 с – по умолчанию)
CommandType	Позволяет получить или задать значение, определяющее, как именно будет интерпретирован текст запроса, заданный через свойство
Connection	Ссылка на объект *OleDbConnection*
ExecuteReader()	Возвращает объект *OleDbReader*
Parameters	Возвращает коллекцию параметров *OleDbParameterCollection*
Prepare()	Готовит команду к выполнению на источнике данных

OleDbDataReader

Данный класс используется для передачи запроса к источнику данных. Поток передачи однонаправленный, возвращает одну строку в ответ на запрос SQL. Данный класс удобно использовать при обработке большого количества данных, не требующих сохранения.

```
//Создаем строку соединения
OleDbConnection cn= new OleDbConnection();
cn.ConnectionString="Provider = SQLOLEDB.1; " + //имя провайдера
"Integrated security = SSPI ; " +
"Persist Security Info = False; " +
"Initial Catalog = Cars; " + //имя базы данных
"Data Source = BIGMANU;";//имя компьютера
cn.Open();
string strSQL="SELECT Make From Inventory WHERE Color='Red'";
OleDbCommand myCommand=new OleDbCommand(strSQL,cn);
OleDbDataReader myDataReader();
MyDataReader=myCommand.ExecuteReader();
while (myDataReader.Read())
{
        Console.WriteLine(myDataReader["Make"].ToString());
}
myDataReader.Close();
cn.Close();
```

ЛИТЕРАТУРА

1. Троенсел, Э. C# и платформа .NET. / Э. Троенсел – СПб.: Питер, 2005. – 796 с.
2. Шилдт, Г. Полный справочник по C#. / Г. Шилдт – М.: Издательский дом «Вильямс», 2004. – 752 с.
3. Рихтер, Д. Программирование на платформе Microsoft .NET Framework / Д. Рихтер. -- 2-е изд., испр. – М.: Издательско-торговый дом «Русская Редакция», 2003. – 512 с.
4. Либерти, Д. Программирование на C# / Д. Либерти. – 2-е изд. – М.: Издательство «Символ-плюс», 2002. – 236 с.
5. Разработка Windows-приложений на Microsoft Visual Basic .NET и Microsoft Visual C# .NET – пер. с англ. – М.:Издательско-торговый дом «Русская Редакция», 2003. – 512 с.

ОГЛАВЛЕНИЕ

Учебное издание

МАТЮШ Максим Викторович,
САМОЩЕНКОВ Григорий Александрович

ОСНОВЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ C#
И РАБОТЫ С БАЗАМИ ДАННЫХ

КОНСПЕКТ ЛЕКЦИЙ
для студентов специальностей 1-40 01 01, 1-40 02 01 и слушателей переподготовки специальности 1-40 01 73 «Программное обеспечение информационных систем», слушателей повышения квалификации учебного центра «Компьютерные технологии и автоматизация»

Редактор *О.П. Михайлова*

Дизайн обложки *И.С. Васильевой*

Подписано в печать 16.06.08. Формат 60×84 1/16. Гарнитура Таймс. Бумага офсетная. Ризография. Усл. печ. л. 5,34. Уч.-изд. л. 5,12. Тираж 45 экз. Заказ 845.

Издатель и полиграфическое исполнение
учреждение образования «Полоцкий государственный университет»

ЛИ № 02330/0133020 от 30.04.04 ЛП № 02330/0133128 от 27.05.04

211440 г. Новополоцк, ул. Блохина, 29