Министерство образования и науки
Донецкой Народной Республики
ГОУ ВПО «Донецкий национальный университет»
Кафедра теории упругости и вычислительной математики

СОВРЕМЕННЫЕ КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ

УЧЕБНОЕ ПОСОБИЕ

Издание второе

для студентов направления подготовки
01.04.02 Прикладная математика и информатика

ДОНЕЦК 2019

УДК 004.9(07)

ББК 3973я73

С56

Рекомендовано к изданию Ученым советом

ГОУ ВПО «Донецкий национальный университет»

(протокол № 1 25.01.2019 г.)

Современные компьютерные технологии: учебное пособие / Сост.: Е.В. Авдюшина.– – Изд. 2-е.- Донецк: ДонНУ, 2019. – 188 с.

Рецензенты:

Щепин Н.Н., кандидат физико-математических наук, доцент, ГОУ ВПО «Донецкий национальный университет»;

Машаров П.А., кандидат физико-математических наук, доцент, ГОУ ВПО «Донецкий национальный университет»

В учебном пособии изложен методика обучения по курсу «Современные компьютерные технологии»: структурирование учебного материала по темам, теоретический материал, вопросы для контроля знаний и список литературы. Учебный материал содержит основные сведения о современных подходах разработки информационных систем, визуализации и обработки данных.

Учебное пособие предназначено для студентов специальности «Прикладная математика и информатика» и может быть использовано студентами других направления прикладной математики, информатики и информационных технологий.

УДК 004.9(07)

ББК 3973я73

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Учебная дисциплина «Современные компьютерные технологии» относится к вариативной части по выбору студента профессионального блока дисциплин подготовки студентов по направлению подготовки 01.04.02 Прикладная математика и информатика.

Содержание дисциплины является логическим продолжением дисциплин «Распределенная обработка данных в современных СУБД», «Распределенные информационные системы», и формирует основу для освоения дисциплин научно-исследовательская и Ассистенстская практики, а также может быть использована при написании магистерской диссертации.

Целью дисциплины является ознакомление магистрантов со специальными компьютерными технологиями, имеющими применение в области моделирования и создания специализированного программного обеспечения для решения прикладных задач в различных сферах жизнедеятельности.

Задачами являются формирование понимания студентами ключевых положений компьютерных технологий, структуры, связи с другими науками, понятий многоуровневое приложение, компоненты приложения, уровни данных, бизнес-логики и представления, целостного представления о видах информации, мировых информационных ресурсов, способах обработки информации.

Процесс изучения дисциплины направлен на формирование элементов следующих компетенций в соответствии с ГОС ВПО по данному направлению подготовки: способности к абстрактному мышлению, анализу, синтезу; готовности к саморазвитию, самореализации, использованию творческого потенциала; способности самостоятельно приобретать с помощью информационных технологий и использовать в практической деятельности новые знания и умения, в том числе в новых областях знаний, непосредственно не связанных со сферой деятельности, расширять и углублять свое научное

мировоззрение; способности использовать и применять углубленные знания в области прикладной математики и информатики; способности проводить научные исследования и получать новые научные и прикладные результаты самостоятельно и в составе научного коллектива; способности разрабатывать и применять математические методы, системное и прикладное программное обеспечение для решения задач научной и проектно-технологической деятельности; способности управлять проектами, планировать научно-исследовательскую деятельность, анализировать риски, управлять командой проекта; способности разрабатывать корпоративные стандарты и профили функциональной стандартизации приложений, систем, информационной инфраструктуры; способности разрабатывать аналитические обзоры состояния области прикладной математики и информационных технологий.

В результате изучения учебной дисциплины студент приобретает знания тенденций и направлений развития современных компьютерных технологий, программирования и разработки приложений; основных тенденций развития современных компьютерных технологий; существующих методов и стандартов управления проектами. Умениями, которые формирует дициплина являются выполнение различных математические расчеты с использованием современных компьютерных средств, работа с современными операционными системами и важнейшими прикладными программами обработки информации, представления информации, с базами данных, Интернет, разрабатка структуры приложения и его уровни, использую объектно-ориентированное программирование, приобретение навыков совместного использования различных языков программирования для создания приложений различных видов, разработка сетевые информационные приложения с использованием современных технологий программирования, применение принципов объектно-ориентированного программирования на языке C#, проектирования многоуровневую иерархию объектов, использования классов технологии ADO.Net для работы с базами данных и язык интегрированных запросов LINQ для работы с различными данными, создания приложений с использованием

языка C# и Asp.Net, реализовывать аутентификацию пользователей веб-приложений. Также студент должен владеть техническими и программными средствами, обеспечивающими применение компьютерных технологий, навыками практического применения современных информационных систем в различных сферах деятельности, методикой оценки эффективности проектов в информационных технологиях.

В рамках изучения дисциплины предусмотрены следующие формы организации учебного процесса: лекции, лабораторные занятия, самостоятельную работу студента.

Лекционные занятия предполагают овладение теоретическими основами дисциплины, лабораторные – для овладения методами разработки компьютерных систем.

Самостоятельная работа студентов предусматривает выполнение домашних заданий, подготовку к лабораторным занятиям, изучение учебно-методической литературы, составление конспектов, подготовку презентаций и докладов.

Текущий контроль осуществляется путем написания самостоятельных и контрольных работ для проверки текущих знаний теории и практики, модульной контрольной работы по проверке знаний теоретических и практических положений.

Тема 1. Объектно-ориентированный язык программирования C#

1.1. Синтаксис языка программирования C#

Язык программирования C# появился сравнительно недавно. Он является наследником языков C и C++, которые признаны одними из самых удачных языков программирования. С другой стороны он имеет родственные связи и с языком Java.

Язык C# тесно связан со средой .Net Framework. Она обеспечивает возможность совместного использования различных языков программирования, контроль безопасности и переносимости программ. Общим между языком и средой являются

– общеязыковая среда выполнения (Common Language Runtime – CLR);

– библиотека классов .Net.

Главной задачей среды CLR является управление кодом. В результате компиляции программы на языке C# создаётся файл с псевдокодом на промежуточном языке MSIL (Microsoft Intermediate Language). Этот файл содержит инструкции, которые не зависят от типа процессора. В дальнейшем среда CLR вызывая JIT компилятор, который преобразует псевдокод в исполняемый код для каждой части программы. Таким образом, описанное преобразоваание может выполнено на любой операционной системе, где установлена CLR. Это обеспечивает переносимость программ.

Все программы, написанные на языке C# являются объектно-ориентированными [1]. Для реализации этого используются инкапсуляция, наследование и полиморфизм.

В C# идентификатор представляет собой имя, которое используется для переменной, функции, любому определяемому пользователем элементу программы. Идентификаторы могут состоять из одного или нескольких символов, начинаться с любой буквы латинского алфавита или знака подчеркивания, далее может следовать буква, цифра или знак подчеркивания.

Идентификатор не может начинаться с цифры. Но идентификаторы, содержащие два знака подчеркивания подряд, зарезервированы для применения в компиляторе. Прописные и строчные буквы в C# различаются. Несмотря на то что зарезервированные ключевые слова нельзя использовать в качестве идентификаторов, в C# разрешается применять ключевое слово с предшествующим знаком @ в качестве допустимого идентификатора.

Как все языки программирования C# имеет ключевые слова. Определены два общих типа ключевых слов: зарезервированные и контекстные. Зарезервированные ключевые слова, которые называют еще зарезервированными словами или зарезервированными идентификаторами, нельзя использовать в именах переменных, классов или методов. Их можно использовать только в качестве ключевых слов. В настоящее время определено 77 зарезервированных ключевых слов (табл. 1.1).

Таблица 1.1. Ключевые слова, зарезервированные в языке C#

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

sealed	string	true	unchecked	volatile
void	while			

В версии C# определены 18 контекстных ключевых слов, которые приобретают особое значение в определенном контексте, т.е. они выполняют роль ключевых слов, а вне его они могут использоваться в именах других элементов программы, например в именах переменных. Формально эти слова не считаются зарезервированными, но их применение в виде элемента программы может привести к путанице. Поэтому рекомендуется их использовать только по назначению. Контекстные ключевые слова приведены в табл. 1.2.

Таблица 1.2. Контекстные ключевые слова в C#

add	group	partial	var	dynamic
into	remove	where	from	join
select	yield	get	let	set
global	orderby	value		

Язык C# является строго типизированным языком [8]. Это позволяет компилятору подвергать все операции контролю и обеспечить большую надежность работы программ. Только для типа dynamic контроль осуществляется на этапе выполнения программы.

Встроенные типы данных делятся на: типы значений и ссылочные типы. Они отличаются по содержимому переменной. Переменная относится к типу значения, если она содержит само значение. Переменная относится к ссылочному типу, если она содержит ссылку на значение (например, объект класса).

В основу языка C# положены 13 типов значений, перечисленных в табл.1.3. Все они называются простыми типами, поскольку состоят из единственного значения. Простые типы данных иногда еще называют примитивными.

Целочисленные типы бываю со знаком и без знака (перед названием типа добавляется префикс u). Целочисленные типы со знаком отличаются от аналогичных типов без знака способом интерпретации старшего разряда целого числа. Старший разряд целого числа со знаком используется в качестве флага знака. Число считается положительным, если флаг знака равен 0, и отрицательным, если он равен 1.

Таблица 1.3 - Типы значений в C#

Тип	Значение	Разрядность	Диапазон
bool	Логический, предоставляет два значения: "истина" или "ложь"		
byte	8-разрядный целочисленный без знака	8	0 – 255
char	Символьный (Unicode)	16	0 – 65535
decimal	Десятичный (для финансовых расчетов)	128	1E-28 – 7,9E+28
double	С плавающей точкой двойной точности	64	5E-324 – 1,7E+308
float	С плавающей точкой одинарной точности	32	-5E-45 до 3,4E+38
int	Целочисленный	32	-2147483648– 2147483647
long	Длинный целочисленный	64	-9223372036854775808 – 9223372036854775807
sbyte	8-разрядный целочисленный со знаком	8	-128 – 127
short	Короткий целочисленный	16	-32768 – 32767
uint	Целочисленный без знака	32	0 – 4294967295
ulong	Длинный целочисленный без знака	64	0 – 18446744073709551615
ushort	Короткий целочисленный без знака	16	-32768 – 32767

В C# имеются две разновидности типов данных с плавающей точкой: float и double. Они представляют числовые значения с одинарной и двойной точностью соответственно. Математические функции определены в библиотеке System.Math. Все эти функции возвращают значение и имеют аргумент типа double.

Среди всех числовых типов данных в C# для финансовых расчетов используется тип decimal. В обычных арифметических вычислениях с плавающей точкой характерны ошибки округления десятичных значений. Тип decimal позволяет представить числа с точностью до 28 (а иногда и 29) десятичных разрядов, т.е способен представлять десятичные значения без ошибок округления. Для констант этого типа в конце необходимо добавить букву m или M.

Символьный тип представлен в виде Unicode. Для задания константы этого типа необходимо заключить ее в одинарные апострофы. В C# отсутствует автоматическое преобразование символьного типа в целочисленный и обратно.

Логический тип определен двумя значениями true и false. Логический тип не преобразовывается автоматически в целочисленный.

Целочисленный литерал должен быть самый мелкий целочисленный тип, которым они могут быть представлены, начиная с типа int, затем uint, long или ulong в зависимости от значения литерала. Литералы с плавающей точкой относятся к типу double.

Чтобы указать тип литерала необходимо для

- типа long к литералу присоединяется суффикс l или L,

- целочисленного типа без знака типа uint к литералу присоединяется суффикс u или U,

- длинного целочисленного типа без знака к литералу типа ulong присоединяется суффикс ul или UL;

- шестнадцатеричные литералы записываются из цифр 0-9 и букв A-F, которым предшествует префикс 0X или 0x.

Кроме того, для указания типа float к литералу присоединяется суффикс F или f. Можете даже указать тип double, присоединив к литералу суффикс d или D, хотя это излишне.

Управляющими последовательностями символов являются

\a - звуковой сигнал (звонок);

\b - возврат на одну позицию;

\f - перевод страницы (переход на новую страницу);

\n - новая строка (перевод строки);

\r - возврат каретки;

\t - горизонтальная табуляция;

\v - вертикальная табуляция;

\0 - пустой символ;

\’ - одинарная кавычка;

\" - двойная кавычка;

\\ - обратная косая черта.

Строковые литералы – это последовательности символов, заключенных в кавычки [20]. Существует еще одна форма строкового литерала, которая называется буквальный строковый литерал. Такой литерал начинается с символа @, после которого следует строка в кавычках. Содержимое строки в кавычках воспринимается без изменений и может быть расширено до двух и более строк. Т.е. буквальные строковые литералы выводятся в том же виде, в каком они введены в исходном тексте программы. Это означает, что в буквальный строковый литерал можно включить символы новой строки, табуляции и прочие, не прибегая к управляющим последовательностям. Единственное исключение составляют двойные кавычки ("), для указания которых необходимо использовать две двойные кавычки подряд (" ").

```
Console.WriteLine(@"Эта строка будет
Располагаться на
нескольких строках. ");
```

Переменная – это именованная область памяти, для которой может быть установлено значение.

Все переменные должны быть объявлены до своего использования. Для этого используется оператор

тип имя_переменной1, имя_переменной2, имя_переменнойN;

где тип — это конкретный тип объявляемой переменной, а имя_переменной — имя самой переменной.

Можно также описать переменную с инициализацией

тип имя_переменной1=значение_переменной1, …имя_переменнойN;

В этом случае значение переменной должно совпадать с ее типом.

Хотя все переменные должны быть типизированы, но начиная с версии C# 3.0, компилятору предоставляется возможность самому определить тип локальной переменной, исходя из значения, которым она инициализируется. Такая переменная называется неявно типизированной. Неявно типизированная переменная объявляется с помощью ключевого слова var и должна быть непременно инициализирована. Для определения типа этой переменной компилятору служит тип ее инициализатора, т.е. значения, которым она инициализируется. Рассмотрим такой пример.

```
var e = 2.7183;
```

В данном примере переменная e инициализируется литералом с плавающей точкой, который по умолчанию имеет тип double, и поэтому она относится к типу double. Если бы переменная e была объявлена следующим образом:

```
var e = 2.7183F;
```

то она была бы отнесена к типу float.

После того, как тип переменной объявлен он не может быть изменен. Таким оператором можно объявить только одну неявно типизированную

перемену. Такие переменные применяются в частных случаях, например, в LINQ.

Областью действия переменной является блок ее видимости. Блок видимости – часть программы, заключенная в фигурные скобки. Если переменная объявлена во внешнем блоке, то она доступна и во внутреннем блоке. Нельзя перекрывать имя переменной из внешнего блока, т.е. описывать во внутреннем блоке переменную с таким же именем. (В C и C++ это допускалось).

Приведение или преобразование типов. Существуют явное и неявное преобразование типов.

Когда данные одного типа присваиваются переменной другого типа, неявное преобразование типов происходит автоматически при следующих условиях:

• оба типа совместимы;

• диапазон представления чисел целевого типа шире, чем у исходного типа.

Если оба эти условия удовлетворяются, то происходит расширяющее преобразование. Например, тип int достаточно крупный, чтобы вмещать в себя все действительные значения типа byte, а кроме того, оба типа, int и byte, являются совместимыми целочисленными типами, и поэтому для них вполне возможно неявное преобразование.

Числовые типы, как целочисленные, так и с плавающей точкой, вполне совместимы друг с другом для выполнения расширяющих преобразований.

В язык не допускается неявное взаимное преобразование типов decimal и float или double, а также числовых типов и char или bool. Кроме того, типы char и bool несовместимы друг с другом.

Но не всегда возможно использование только неявного приведения типов. Поэтому существует операция приведение — это команда компилятору преобразовать результат вычисления выражения в указанный тип. А для этого

требуется явное преобразование типов. Ниже приведена общая форма приведения типов

(целевой_тип) выражение

Если приведение типов приводит к сужающему преобразованию, то часть информации может быть потеряна.

Преобразование типов может применяться и в выражениях. В этом случае применяются правила продвижения типов. Ниже приведен алгоритм, определяемый этими правилами для операций с двумя операндами:

если один операнд имеет тип decimal, то и второй операнд приводится к типу decimal (но если второй операнд имеет тип float или double, результат будет ошибочным);

если один операнд имеет тип double, то и второй операнд продвигается к типу double;

если один операнд имеет тип float, то и второй операнд продвигается к типу float;

если один операнд имеет тип ulong, то и второй операнд продвигается к типу ulong (но если второй операнд имеет тип sbyte, short, int или long, результат будет ошибочным);

если один операнд имеет тип long, то и второй операнд продвигается к типу long;

если один операнд имеет тип uint, а второй — тип sbyte, short или int, то оба операнда продвигаются к типу long;

если один операнд имеет тип uint, то и второй операнд продвигается к типу uint;

иначе оба операнда продвигаются к типу int.

```
byte b = 10;
b = (byte) (b * b); // Необходимо приведение типов!!
char chl = 'a', ch2 = 'b';
```

chl = (char) (chl+ ch2);

Арифметические операции. Арифметические операторы

Арифметические операторы, представленные в языке C#, следующие:

+ − сложение;

- − вычитание, унарный минус;

* − умножение;

/ − деление;

% − деление по модулю;

-- − декремент;

++ − инкремент.

Операции +, -, *, / можно применять к любому встроенному числовому типу данных. При деление целочисленных операндов результат будет целочисленный. Операцию % – получения остатка от деления можно применять к целочисленному и вещественному типам, но результат будет целочисленный.

В C# определен полный набор операторов отношения, которые можно использовать в логических выражениях. Эти операторы следующие

< − меньше;

<= − меньше или равно;

> − больше;

>= − больше или равно;

== − равно;

!= − не равно.

Логические операторы имеют вид:

& − побитовое И;

| − побитовое ИЛИ;

^ – исключающее ИЛИ;

&& − логическое И;

|| − логическое ИЛИ;

! – логическое НЕ.

Результатом операций отношений и логических операций является тип bool.

Поразрядные операторы. В C# предусмотрены поразрядные операторов, которые воздействуют на отдельные двоичные разряды (биты) своих операндов. Они определены только для целочисленных операндов, поэтому их нельзя применять к данным типа bool, float или double.

Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ обозначаются следующим образом: &, |, ^ и ~. Они выполняют те же функции, что и их логические аналоги. Но в отличие от логических операторов, поразрядные операторы действуют на уровне отдельных двоичных разрядов.

В C# имеется возможность сдвигать двоичные разряды, составляющие целое значение, влево « или вправо » на заданную величину. Для этой цели в C# определены два приведенных оператора сдвига двоичных разрядов

значение « число_битов

значение » число_битов

где число_битов — это число двоичных разрядов, на которое сдвигается указанное зна чение.

При сдвиге влево все двоичные разряды в указываемом значении сдвигаются на одну позицию влево, а младший разряд сбрасывается в нуль. При сдвиге вправо все двоичные разряды в указываемом значении сдвигаются на одну позицию вправо. Если вправо сдвигается целое значение без знака, то старший разряд сбрасывается в нуль. А если вправо сдвигается целое значение со знаком, то разряд знака сохраняется. Напомним, что для представления отрицательных чисел старший разряд целого числа устанавливается в 1. Так, если сдвигаемое значение является отрицательным, то при каждом сдвиге вправо старший разряд числа устанавливается в 1. А если сдвигаемое значение является положительным, то при каждом сдвиге вправо старший разряд числа сбрасывается в нуль. При сдвиге влево и вправо крайние двоичные разряды теряются. Восстановить потерянные при сдвиге двоичные разряды нельзя, поскольку сдвиг в данном случае не является циклическим.

Оператор присваивания обозначается одиночным знаком равенства (=). В C# оператор присваивания действует таким же образом, как и в других языках программирования – значение выражения, стоящего справа пересылается в поле переменной. Общая форма такая

имя_переменной = выражение;

Здесь имя_переменной должно быть совместимо с типом выражения.

Можно использовать также составные операторы присваивания для арифметических и логических операций: +=, -=, &=, * =, |=, /=, ^=.

Условный оператор часто называют тернарным, т.к. в нем участвуют три операнда. Ниже приведена общая форма этого оператора.

Выражение 1 ? Выражение2 : Выражение3;

Здесь Выражение1 должно относиться к типу bool, а Выражение2 и Выражение3 – к одному и тому же типу. Значение выражения ? определяется следующим образом. Сначала вычисляется Выражение1. Если оно истинно, то вычисляется Выражение2, а полученный результат определяет значение всего Выражения2 в целом. Если же Выражение1 оказывается ложным, то вычисляется Выражение3, и его значение становится общим для всего выражения.

Оператор условного перехода имеет различные формы

```
if(условие) оператор;
else оператор;
```

где условие – это некоторое условное выражение, а оператор – адресат операторов if и else. Оператор else не является обязательным. Адресатом обоих операторов, if и else, могут также служить блоки операторов. Ниже приведена общая форма оператора if, в котором используются блоки операторов.

```
if (условие)
{
последовательность операторов
}
else
```

```
{
последовательность операторов
}
```

Здесь условие представляет собой булево, т.е. логическое, выражение, принимающее одно из двух значений: "истина" или "ложь". Если условие истинно, то оператор выполняется. А если условие ложно, то выполнение программы происходит, минуя оператор.

Оператором множественного выбора в C# является оператор switch, который обеспечивает многонаправленное ветвление программы. Следовательно, этот оператор позволяет сделать выбор среди нескольких альтернативных вариантов дальнейшего выполнения программы. Несмотря на то что многонаправленная проверка может быть организована с помощью последовательного ряда вложенных операторов if, во многих случаях более эффективным оказывается применение оператора switch. Этот оператор действует следующим образом. Значение выражения последовательно сравнивается с константами выбора из заданного списка. Как только будет обнаружено совпадение с одним из условий выбора, выполняется последовательность операторов, связанных с этим условием. Ниже приведена общая форма оператора switch.

```
switch{выражение) {
case константа1 : последовательность операторов break;
case константа2: последовательность операторов break;
case константа3: последовательность операторов break;
default: последовательность операторов break;
}
```

Заданное выражение в операторе switch должно быть целочисленного типа (char, byte, short или int), перечислимого или же строкового. А выражения других типов, например с плавающей точкой, в операторе switch не допускаются. Зачастую выражение, управляющее оператором switch, просто сводится к одной переменной. Кроме того, константы выбора должны иметь

тип, совместимый с типом выражения. В одном операторе switch не допускается наличие двух одинаковых по значению констант выбора.

Последовательность операторов из ветви default выполняется в том случае, если ни одна из констант выбора не совпадает с заданным выражением. Ветвь default не является обязательной. Если же она отсутствует и выражение не совпадает ни с одним из условий выбора, то никаких действий вообще не выполняется. Если же происходит совпадение с одним из условий выбора, то выполняются операторы, связанные с этим условием, вплоть до оператора break. В последовательности операторов отдельной ветви case встречается оператор break, происходит выход не только из этой ветви, но из всего оператора switch, а выполнение программы возобновляется со следующего оператора, находящегося за пределами оператора switch. Последовательность операторов в ветви default также должна быть лишена "провалов", поэтому она завершается, как правило, оператором break. Но несколько case подряд могут ссылаться на одну и ту же последовательность операторов.

Операторы цикла имеют 4 вида. Оператор for

for (инициализация; условие; итерация) оператор;

В самой общей форме в части инициализация данного оператора задается начальное значение переменной управления циклом. Часть условие представляет собой булево выражение, проверяющее значение переменной управления циклом. Если результат проверки истинен, то цикл продолжается. Если же он ложен, то цикл завершается. В части итерация определяется порядок изменения переменной управления циклом на каждом шаге цикла, когда он повторяется.

Еще одним оператором цикла в C# является **оператор while**:

while (условие) оператор;

где оператор — это единственный оператор или же блок операторов, а условие означает конкретное условие управления циклом и может быть любым логическим выражением. В этом цикле оператор выполняется до тех пор, пока условие истинно. Как только условие становится ложным, управление программой передается строке кода, следующей непосредственно после цикла. Как и в цикле for, в цикле while проверяется условное выражение, указываемое в самом начале цикла.

Третьим оператором цикла в C# является **оператор do-while**. В отличие от операторов цикла for и while, в которых условие проверялось в самом начале цикла, в операторе do-while условие выполнения цикла проверяется в самом его конце. Это означает, что цикл do-while всегда выполняется хотя бы один раз. Ниже приведена общая форма оператора цикла do-while.

```
do {
операторы;
} while (условие);
```

При наличии лишь одного оператора фигурные скобки в данной форме записи необязательны.

Оператор цикла foreach служит для циклического обращения к элементам коллекции, которая представляет собой группу объектов, и имеет вид

```
foreach (ТипЭлемента коллекции ИмяПереременой in ИмяКоллекции)
    {операторы с ИмяПереременой }
```

В C# определено несколько видов коллекций, к числу которых относится массив.

С помощью оператора break можно специально организовать немедленный выход из цикла в обход любого кода, оставшегося в теле цикла, а также минуя проверку условия цикла. Когда в теле цикла встречается оператор break, цикл завершается, а выполнение программы возобновляется с оператора, следующего после этого цикла.

С помощью оператора continue можно организовать преждевременное завершение шага итерации цикла в обход обычной структуры управления циклом. Оператор continue осуществляет принудительный переход к следующему шагу цикла, пропуская любой код, оставшийся невыполненным.

В циклах while и do-while оператор continue вызывает передачу управления непосредственно условному выражению, после чего продолжается процесс выполнения цикла. А в цикле for сначала вычисляется итерационное выражение, затем условное выражение, после чего цикл продолжается.

В C# поддерживаются **три стиля комментариев**.

Один из них приводится в самом начале программы и называется многострочным комментарием. Этот стиль комментария должен начинаться символами /* и оканчиваться символами */. Все, что находится между этими символами, игнорируется компилятором. Как следует из его названия, многострочный комментарий может состоять из нескольких строк.

Второй стиль комментариев, поддерживаемых в C#, однострочный комментарий начинается и оканчивается символами //. Несмотря на различие стилей комментариев, программисты нередко пользуются многострочными комментариями для более длинных примечаний и однострочными комментариями для коротких, построчных примечаний к программе.

Третий стиль комментариев, поддерживаемых в C#, применяется при создании документации.

Массивы. Массив представляет собой совокупность переменных одного типа с общим для обращения к ним именем. В C# массивы могут быть как одномерными, так и многомерными. Главное преимущество массива — в организации данных таким образом, чтобы ими было проще манипулировать. Например, массивы позволяют организовать данные таким образом, чтобы легко отсортировать их. Массивами в C# можно пользоваться практически так же, как и в других языках программирования. Тем не менее у них имеется одна особенность: они реализованы в виде объектов.

Реализация массивов в виде объектов дает ряд существенных преимуществ, и далеко не самым последним среди них является возможность утилизировать неиспользуемые массивы средствам "сборки мусора".

Одномерный массив представляет собой список связанных переменных. Для использования массивом необходимо объявить переменную, которая может обращаться к массиву и создать экземпляр массива, используя оператор new. Так, для объявления одномерного массива обычно применяется следующая общая форма:

```
тип[ ] имя_массива = new тип[размер] ;
```

где тип объявляет конкретный тип элемента массива. Тип элемента определяет тип данных каждого элемента, составляющего массив. Обратите внимание на квадратные скобки, которые сопровождают тип. Они указывают на то, что объявляется одномерный массив. А размер определяет число элементов массива.

Объявление массива можно разделить на два отдельных оператора.

```
тип[ ] имя_массива;
имя_массива = new тип[размер] ;
```

В данном случае переменная имя_массива не ссылается на какой-то определенный физический объект. Только после выполнения второго оператора эта переменная ссылается на массив.

Доступ к отдельному элементу массива осуществляется по индексу: Индекс обозначает положение элемента в массиве. В языке C# индекс первого элемента всех массивов оказывается нулевым.

Инициализировать массив можно различными способами:

– вручную, присваивая каждому элементу значение;

– при описании, например, для одномерного массива:

```
тип[] имя_массива = {vail, val2, val3, ..., valN} ;
```

где val1-valN обозначают первоначальные значения, которые присваиваются по очереди, слева направо и по порядку индексирования. Для хранения инициализаторов массива в C# автоматически распределяется достаточный объем памяти. А необходимость пользоваться оператором new явным образом отпадает сама собой.

– при описании уже созданной ссылки, используя оператор new

тип[] имя_массива = new тип [] { значение1, значение2, …, значениеN};

или в форме

тип[] имя_массива;

имя_массива = new тип [] { значение1, значение2, …, значениеN};

или

тип[] имя_массива = new тип [размер]

{значение1, значение2, …, значениеN};

Но размер должен совпадать с количеством значений в фигурных скобках.

Границы массива в C# строго соблюдаются. Если границы массива не достигаются или же превышаются, то возникает ошибка, исключительная ситуация типа IndexOutOf RangeException, связанная с выходом за пределы индексирования массива, и программа преждевременно завершится.

Можно создать также многомерные массивы. Многомерным называется такой массив, который отличается двумя или более измерениями, причем доступ к каждому элементу такого массива осуществляется с помощью определенной комбинации двух или более индексов. Для объявления многомерного массива используется форма

тип[, . . .,] имя_массива = new тип[размер1, размер2, . . . размерN] ;

Количество запятых в первых квадратных скобках равно N-1.

Для инициализации многомерного массива достаточно заключить в фигурные скобки список инициализаторов каждого его размера. Ниже в качестве примера приведена общая форма инициализации двумерного массива:

```
тип[,] имя_массива = {
{val, val, val, ..., val),
{val, val, val, ..., val),
{val, val, val, ..., val}
};
```

где val обозначает инициализирующее значение, а каждый внутренний блок – отдельный ряд. Первое значение в каждом ряду сохраняется на первой позиции в массиве, второе значение — на второй позиции и т.д. Обратите внимание на то, что блоки инициализаторов разделяются запятыми, а после завершающей эти блоки закрывающей фигурной скобки ставится точка с запятой.

В языке существуют также ступенчатые массивы, т.е. массивы с переменной длиной. Ступенчатые массивы являются массивами в массиве. Такие массивы объявляются с помощью ряда квадратных скобок, в которых указывается их размерность. Например, для объявления двумерного ступенчатого массива служит следующая общая форма:

```
тип[] [] имя_массива = new тип [размер] [];
```

где размер обозначает число строк в массиве.

Память для самих строк распределяется индивидуально, и поэтому длина строк может быть разной. Для обращения к элементам таких массивов для каждой размерности используются свои квадратные скобки.

Присваивание значения одной переменной ссылки на массив другой переменной, по существу, означает, что обе переменные ссылаются на один и тот же массив, и в этом отношении массивы ничем не отличаются от любых других объектов. Такое присваивание не приводит ни к созданию копии массива, ни к копированию содержимого одного массива в другой.

С каждым массивом связано свойство Length, содержащее число элементов, из которых может состоять массив. Следовательно, у каждого массива имеется специальное свойство, позволяющее определить его длину. Для обращения к этому свойству используется точка имя_массива.Length. Это

свойство можно использовать для массивов любой размерности, но нельзя определить длину каждой размерности.

```
string[,] mass = new string[3,4];
// Число строк в данном случае 3
mass.GetLength(0)
// Число столбцов в данном случае 4
mass.GetLength(1);
```

Если массив является многомерным, то функция вернет число элементов по всем измерениям. Если нужно знать число элементов внутри измерения, можно использовать вместо этого метод GetLength().

Свойство Rank позволяет получить количество измерений массива.

Для ступенчатых массивов можно обращаться к каждой строке со свойством Length следующим образом

```
имя_массива.Length
```

то в нем определяется число массивов, хранящихся в ступенчатом массиве; для получения длины любого отдельного массива, составляющего ступенчатый массив, служит следующее выражение.

```
имя_массива [номер_строки].Length
```

Существуют неявно типизированные массивы, которые объявляется с помощью ключевого слова var, но без последующих квадратных скобок [ ]. Кроме того, неявно типизированный массив должен быть непременно инициализирован, поскольку по типу инициализаторов определяется тип элементов данного массива. Все инициализаторы должны быть одного и того же согласованного типа.

```
var имя_массива = new[] { значение1, значени2, …, значениеN };
var имя_массива = new[,] { {Значения}, {Значения}};
var имя_массива = new[] {
new[] { Значения_1_строки },
new[] {Значения_2_строки },
new[] { Значения_3_строки} };
```

Оператор цикла foreach для массивов. Оператор foreach служит для циклического обращения к элементам массива. Общая форма оператора имеет вид

foreach (ТипМассива имя_переменной_цикла in ИмяМассива) оператор;

Здесь тип обозначает тип элемиента массива, который может также обозначаться как var; имя_переменной_цикла обозначает имя переменной управления циклом, которая получает значение следующего элемента коллекции на каждом шаге выполнения цикла foreach.

Оператор цикла foreach действует следующим образом. Когда цикл начинается, первый элемент массива выбирается и присваивается переменной цикла. На каждом последующем шаге итерации выбирается следующий элемент массива, который сохраняется в переменной цикла. Цикл завершается, когда все элементы массива окажутся выбранными. Следовательно, оператор foreach циклически опрашивает массив по отдельным его элементам от начала и до конца.

Переменная цикла в операторе foreach служит только для чтения. Это означает, что, присваивая этой переменной новое значение, нельзя изменить содержимое массива. Этот цикл можно предварительно завершить с помощью оператора break.

При использовании foreach для многомерных массивов возвращаются по порядку следования строки от первой до последней.

Перегрузка операторов. В языке C# допускается определять назначение оператора по отношению к создаваемому классу, т.е допускается перегрузка операторов. Когда оператор перегружается, ни одно из его первоначальных назначений не теряется. Он просто выполняет еще одну, новую операцию относительно конкретного объекта. Главное преимущество перегрузки операторов заключается в том, что она позволяет плавно интегрировать класс

нового типа в среду программирования. Как только для класса определяются операторы, появляется возможность оперировать объектами этого класса, используя обычный синтаксис выражений в C#. Перегрузка операторов является одной из самых сильных сторон языка C#.

Для перегрузки оператора служит ключевое слово operator, определяющее операторный метод, задающий действие оператора относительно своего класса.

Существуют две формы операторных методов (operator): одна — для унарных операторов, другая — для бинарных.

```
// Общая форма перегрузки унарного оператора.
public static возвращаемый_тип operator op(тип_параметра операнд)
{
// операции
}
// Общая форма перегрузки бинарного оператора.
public static возвращаемый_тип operator op
    (тип_параметра1 операнд1, тип_параметра1 операнд2)
{
// операции
}
```

Здесь вместо op подставляется перегружаемый оператор, например + или /; а возвращаемый_тип обозначает конкретный тип значения, возвращаемого указанной операцией. Это значение может быть любого типа, но зачастую оно указывается такого же типа, как и у класса, для которого перегружается оператор. Такая корреляция упрощает применение перегружаемых операторов в выражениях. Обратите внимание на то, что операторные методы должны иметь оба типа, public и static. Тип операнда унарных операторов должен быть таким же, как и у класса, для которого перегружается оператор. А в бинарных операторах хотя бы один из операндов должен быть такого же типа, как и у его

класса. Следовательно, в C# не допускается перегрузка любых операторов для объектов, которые еще не были созданы. Например, назначение оператора + нельзя переопределить для элементов типа int или string.

И еще одно замечание: в параметрах оператора нельзя использовать модификатор ref и out.

Операторы отношения, например == и <, могут также перегружаться, причем очень просто. Как правило, перегруженный оператор отношения возвращает логическое значение true и false. Это вполне соответствует правилам обычного применения подобных операторов и дает возможность использовать их перегружаемые разновидности в условных выражениях. Если же возвращается результат другого типа, то тем самым сильно ограничивается применимость операторов отношения.

На перегрузку операторов отношения накладывается следующее важное ограничение: они должны перегружаться попарно. Так, если перегружается оператор <, то следует перегрузить и оператор >, и наоборот. Основные пары следующие: == и !=, <= и >=, < и >.

И еще одно замечание: если перегружаются операторы == и ! =, то для этого обычно требуется также переопределить методы Object.Equals() и Object.GetHashCode().

Ключевые слова true и false можно также использовать в качестве унарных операторов для целей перегрузки. Перегружаемые варианты этих операторов позволяют определить назначение ключевых слов true и false специально для создаваемых классов. После перегрузки этих ключевых слов в качестве унарных операторов для конкретного класса появляется возможность использовать объекты этого класса для управления операторами if, while, for и do-while или же в условном выражении ?.

Операторы true и false должны перегружаться попарно, а не раздельно. Ниже приведена общая форма перегрузки этих унарных операторов.

```
public static bool operator true(тип_параметра операнд)
{
```

```
// Возврат логического значения true или false.
public static bool operator false(тип_параметра операнд)
{
// Возврат логического значения true или false.
}
```

Обратите внимание на то, что и в том и в другом случае возвращается результат типа bool.

Из определенных в C# логических операторов: &, |, !, && и || можно перегрузить только операторы &, | и !. Перегрузка этих операторов осуществляется также как и для бинарных операций, только возвращаемое значение чаще всего имеет тип bool, т.е. они применяются в логических операциях.

Для того чтобы применение укороченных логических операторов && и || стало возможным, необходимо соблюсти следующие четыре правила:

1) в классе должна быть произведена перегрузка логических операторов & и |;

2) перегружаемые методы операторов & и | должны возвращать значение того же типа, что и у класса, для которого эти операторы перегружаются;

3) каждый параметр должен содержать ссылку на объект того класса, для которого перегружается логический оператор;

4) для класса должны быть перегружены операторы true и false.

Если все эти условия выполняются, то укороченные логические операторы автоматически становятся пригодными для применения.

Они действуют следующим образом. Первый операнд проверяется с помощью операторного метода operator true (для оператора | |) или же с помощью операторного метода operator false (для оператора &&). Если удается определить результат данной операции, то соответствующий перегружаемый оператор (& или |) далее не выполняется. В противном случае перегружаемый оператор (& или | соответственно) используется для определения конечного результата. Следовательно, когда применяется укороченный логический

оператор & & или | |, то соответствующий логический оператор & или | вызывается лишь в том случае, если по первому операнду невозможно определить результат вычисления выражения. Благодаря перегрузке операторов true и false для класса компилятор получает разрешение на применение укороченных логических операторов, не прибегая к явной их перегрузке. Это дает также возможность использовать объекты в условных выражениях.

Операторы преобразования. Иногда объект определенного класса требуется использовать в выражении, включающем в себя данные других типов. Для подобных ситуаций в C# предусмотрена специальная разновидность операторного метода, называемая оператором преобразования. Такой оператор преобразует объект исходного класса в другой тип.

Операторы преобразования помогают полностью интегрировать типы классов в среду программирования на C#, разрешая свободно пользоваться классами вместе с другими типами данных, при условии, что определен порядок преобразования в эти типы.

Существуют две формы операторов преобразования: явная и неявная.

```
public static explicit operator целевой_тип(исходный_тип v)
        {return значение;}
public static implicit operator целевой_тип(исходный_тип v)
        {return значение;}
```

где целевой_тип обозначает тот тип, в который выполняется преобразование; исходный_ тип — тот тип, который преобразуется; значение — конкретное значение, приобретаемое классом после преобразования. Операторы преобразования возвращают данные, имеющие целевой_тип, причем указывать другие возвращаемые типы данных не разрешается.

Если оператор преобразования указан в неявной форме (implicit), то преобразование вызывается автоматически, например, в том случае, когда объект используется в выражении вместе со значением целевого типа. Если же

оператор преобразования указан в явной форме (explicit), то преобразование вызывается в том случае, когда выполняется приведение типов. Для одних и тех же исходных и целевых типов данных нельзя указывать оператор преобразования одновременно в явной и неявной форме. Преобразование вызывается автоматически, например, в том случае, когда объект используется в выражении вместе со значением целевого типа.

Оператор неявного преобразования применяется автоматически в следующих случаях: когда в выражении требуется преобразование типов; методу передается объект; осуществляется присваивание и производится явное приведение к целевому типу. С другой стороны, можно создать оператор явного преобразования, вызываемый только тогда, когда производится явное приведение типов. В таком случае оператор явного преобразования не вызывается автоматически.

На операторы преобразования накладывается ряд следующих ограничений:

1) исходный или целевой тип преобразования должен относиться к классу, для которого объявлено данное преобразование. В частности, нельзя переопределить преобразование в тип int, если оно первоначально указано как преобразование в тип double;

2) нельзя указывать преобразование в класс object или же из этого класса;

3) для одних и тех же исходных и целевых типов данных нельзя указывать одновременно явное и неявное преобразование;

4) нельзя указывать преобразование базового класса в производный класс;

5) нельзя указывать преобразование в интерфейс или же из него.

Во избежание подобных ошибок неявные преобразования должны быть организованы только в том случае, если удовлетворяются следующие условия. Во-первых, информация не теряется, например, в результате усечения, переполнения или потери знака. И во-вторых, преобразование не приводит к

исключительной ситуации. Если же неявное преобразование не удовлетворяет этим двум условиям, то следует выбрать явное преобразование.

На перегрузку операторов накладывается ряд ограничений. В частности, нельзя изменять приоритет любого оператора или количество операндов, которое требуется для оператора, хотя в операторном методе можно и проигнорировать операнд. Кроме того, имеется ряд операторов, которые нельзя перегружать. А самое главное, что перегрузке не подлежит ни один из операторов присваивания, в том числе и составные, как, например, оператор +=. Ниже перечислены операторы, которые нельзя перегружать:

&&	??	=>	default
typeof	0	[]	->
is	unchecked	1 1	as
new	?	=	checked
sizeof			

Несмотря на то что оператор приведения () нельзя перегружать явным образом, имеется все же возможность создать упоминавшиеся ранее операторы преобразования, выполняющие ту же самую функцию.

1.2. Инкапсуляция и элементы класса на языке C#

Все программы, написанные на языке C# являются объектно-ориентированными [4]. Для реализации этого используются инкапсуляция, наследование и полиморфизм.

Инкапсуляция - это техника программирования, объединяющая данные и код. При этом код должен обеспечить обработку данных, их защиту от внешнего вмешательства и неправильного использования. Объединение данных и программного кода характеризуют объект.

Язык C# является строго объектно-ориентированным, поэтому в классе определяется данные и код, который будет оперировать с этими данными. Класс представляет собой шаблон, по которому определяется форма объекта. Класс, по существу, представляет собой ряд схематических описаний способа построения объекта. При этом очень важно подчеркнуть, что класс является

логической абстракцией. Физическое представление класса появится в оперативной памяти лишь после того, как будет создан объект этого класса.

Данные содержатся в членах данных, определяемых классом, а код – в функциях-членах [17]. Члены данных также называют полями, к ним относятся переменные экземпляра и статические переменные, а к функциям-членам – методы, конструкторы, деструкторы, индексаторы, события, операторы и свойства.

Форма определения простого класса, содержащая только переменные экземпляра и методы, следующая

```
class имя_класса {
// Объявление переменных экземпляра.
специф_доступа тип переменная1;
специф_доступа тип переменная2;
//...
специф_доступа тип переменнаяN;
// Объявление методов.
специф_доступа возвращаемый_тип метод1 (параметры) {
// тело метода
}
специф_доступа возвращаемый_тип метод2 (параметры). {
// тело метода
}
//...
специф_доступа возвращаемый_тип методN (параметры) {
// тело метода
}
}
```

Определение class обозначает создание нового типа данных. Указывать спецификатор доступа не обязательно, но если он отсутствует, то объявляемый

член считается закрытым в пределах класса. Члены с закрытым доступом могут использоваться только другими членами их класса. Метод Main() указывается в классе, с которого начинается выполнение программы. Для функций список_параметров — это последовательность пар, состоящих из типа и идентификатора и разделенных запятыми. Параметры представляют собой переменные, получающие значение аргументов, передаваемых методу при его вызове. Если у метода отсутствуют параметры, то список параметров оказывается пустым.

Для того чтобы создать конкретный объект созданного типа необходимо воспользоваться следующим оператором.

имя_класса переменная_класса = new имя_класса ();

После выполнения этого оператора объект переменная_класса станет экземпляром класса имя_класса, т.е. обретет "физическую" реальность. Сама переменная_класса является ссылкой на объект. Оператор new динамически распределяет память во время выполнения программы.

Для переменных не ссылочного типа также может применяться оператор new для инициализации нулевым начальным значением.

int i = new int () ;

Если так не записать, то переменная не будет проинициализирована. Но такой оператор применяется редко.

Каждый раз, когда получается экземпляр класса, создается также объект, содержащий собственную копию каждой переменной экземпляра, определенной в данном классе. Для доступа к переменным служит оператор доступа к члену класса, который называется оператором-точкой. Оператор-точка связывает имя объекта с именем члена класса

объект.членКласса

Объекты класса доступны по ссылке, классы являются ссылочными типами.

Методы класса. Метод состоит из одного или нескольких операторов. В грамотно написанном коде C# каждый метод выполняет только одну функцию. У каждого метода имеется свое имя, по которому он вызывается. В общем, методу в качестве имени можно присвоить любой действительный идентификатор. Следует, однако, иметь в виду, что идентификатор Main () зарезервирован для метода, с которого начинается выполнение программы. Кроме того, в качестве имен методов нельзя использовать ключевые слова C#. Если в методе используется переменная экземпляра, определенная в его классе, то делается это непосредственно, без указания явной ссылки на объект и без помощи оператора-точки. Ведь метод всегда вызывается относительно некоторого объекта его класса. Как только вызов произойдет, объект становится известным. Поэтому объект не нужно указывать в методе еще раз.

Выход из метода происходит при достижении закрывающей фигурной скобки или по оператору return. Имеются две формы оператора return: одна – для методов типа void, т.е. тех методов, которые не возвращают значения, а другая – для методов, возвращающих конкретные значения. Для возврата значения из метода в вызывающую часть программы служит следующая форма оператора return:

return значение;

где значение — это конкретное возвращаемое значение.

При вызове метода ему можно передать одно или несколько значений. Значение, передаваемое методу, называется аргументом. А переменная, получающая аргумент, называется формальным параметром, или просто параметром. Параметры объявляются в скобках после имени метода. Синтаксис объявления параметров такой же, как и у переменных. А областью действия параметров является тело метода. За исключением особых случаев передачи аргументов методу, параметры действуют так же, как и любые другие переменные. Функции могут иметь несколько параметров.

Конструкторы. Конструктор инициализирует объект при его создании. У конструктора такое же имя, как и у его класса. Класс – это аналог метода, но

конструктор не имеет возвращаемого значения, указанного явно. Ниже приведена общая форма конструктора.

```
доступ имя_класса(список_параметров) {
// тело конструктора
}
```

Конструктор используется для задания первоначальных значений переменных экземпляра, определенных в классе, или же для выполнения любых других действий, которые требуются для создания полностью сформированного объекта. Кроме того, доступ обычно представляет собой модификатор доступа типа public, поскольку конструкторы чаще всего вызываются вне класса. А список_параметров может быть как пустым, так и состоящим из одного или более указываемых параметров.

У всех классов имеется конструктор, используемый по умолчанию и инициализирующий все переменные экземпляра их значениями по умолчанию. Для большинства типов данных значением по умолчанию является нулевое, для типа bool – значение false, а для ссылочных типов – пустое значение. Но если определить свой собственный конструктор, то конструктор по умолчанию больше не используется.

Если конструктор имеет параметры, то для создания объекта класса необходимо использовать форму

```
new имя_класса (список_аргументов)
```

где имя_класса обозначает имя класса, реализуемого в виде экземпляра его объекта.

А имя_класса с последующими скобками обозначает конструктор этого класса. Если в классе не определен его собственный конструктор, то в операторе new будет использован конструктор, предоставляемый в C# по умолчанию. Следовательно, оператор new может быть использован для создания объекта, относящегося к классу любого типа.

При использовании оператора new свободная память для создаваемых объектов динамически распределяется из доступной буферной области оперативной памяти. Свободно доступная память рано или поздно исчерпывается. Это может привести к неудачному выполнению оператора new из-за нехватки свободной памяти для создания требуемого объекта. Именно по этой причине одной из главных функций любой схемы динамического распределения памяти является освобождение свободной памяти от неиспользуемых объектов, чтобы сделать ее доступной для последующего перераспределения. Во многих языках программирования освобождение распределенной ранее памяти осуществляется вручную. Например, в C++ для этой цели служит оператор delete. Но в C# применяется другой, более надежный подход: "сборка мусора".

Система "сборки мусора" в C# освобождает память от лишних объектов автоматически, действуя незаметно и без всякого вмешательства со стороны программиста. Если ссылки на объект отсутствуют, то такой объект считается ненужным, и занимаемая им память в итоге освобождается и накапливается. Эта утилизированная память может быть затем распределена для других объектов. "Сборка мусора" происходит лишь время от времени по ходу выполнения, поэтому, нельзя заранее знать или предположить, когда именно произойдет "сборка мусора".

Для четко контроля действий при окончания жизни объекта может применяться деструктор. Он будет вызываться непосредственно перед окончательным уничтожением объекта системой "сборки мусора". Но деструкторы применяются только в редких случаях. Общая форма деструктора:

```
~имя_класса() {
// код деструктора
}
```

где имя_класса означает имя конкретного класса. У деструктора отсутствуют возвращаемый тип и передаваемые ему аргументы.

Деструктор не вызывается, например, в тот момент, когда переменная, содержащая ссылку на объект, оказывается за пределами области действия этого объекта. Это означает, что заранее нельзя знать, когда именно следует вызывать деструктор. Кроме того, программа может завершиться до того, как произойдет "сборка мусора", а следовательно, деструктор может быть вообще не вызван.

Ключевое слово this. Когда метод вызывается, ему автоматически передается ссылка на вызывающий объект, т.е. тот объект, для которого вызывается данный метод. Эта ссылка обозначается ключевым словом this. Следовательно, ключевое слово this обозначает именно тот объект, по ссылке на который действует вызываемый метод.

Управление доступом к членам класса. Ограничение доступа к членам класса является основополагающим этапом объектно-ориентированного программирования, поскольку позволяет исключить неверное использование объекта. Для соблюдения принципов инкапсуляции правильно реализованный класс образует некий "черный ящик", которым можно пользоваться, но внутренний механизм его действия закрыт для вмешательства извне.

Управление доступом в языке C# организуется с помощью четырех модификаторов доступа: public, private, protected и internal. Для обычных классов основные модификаторы доступа public и private. Модификатор protected применяется только в тех случаях, которые связаны с наследованием. Модификатор internal служит в основном для сборки, которая в широком смысле означает в C# разворачиваемую программу или библиотеку.

Модификатор protected internal - доступен всем из данной сборки, а также типам, производным от данного, т. е. перед нами что-то вроде объединения модификаторов доступа protected и internal.

Член класса со спецификатором public, становится доступным из любого другого кода в программе, включая и методы, определенные в других классах. Когда же член класса обозначается спецификатором private, он может быть

доступен только другим членам этого класса. Модификатор private является значение по умолчанию, т.е. его указывать необязательно.

Правильная организация закрытого и открытого доступа может быть достигнута, если соблюдать ряд общих принципов:

• члены, используемые только в классе, должны быть закрытыми;

• данные экземпляра, не выходящие за определенные пределы значений, должны быть закрытыми, а при организации доступа к ним с помощью открытых методов следует выполнять проверку диапазона представления чисел;

• если изменение члена приводит к последствиям, распространяющимся за пределы области действия самого члена, т.е. оказывает влияние на другие аспекты объекта, то этот член должен быть закрытым, а доступ к нему – контролируемым;

• члены, способные нанести вред объекту, если они используются неправильно, должны быть закрытыми. Доступ к этим членам следует организовать с помощью открытых методов, исключающих неправильное их использование;

• методы, получающие и устанавливающие значения закрытых данных, должны быть открытыми;

• переменные экземпляра допускается делать открытыми лишь в том случае, если нет никаких оснований для того, чтобы они были закрытыми.

Пример с классом точка на плоскости: три конструктора, функция расстояния.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Project1
{
   class Point
```

```
    {
        private double x, y;
        public Point()
        { this.x = 0; this.y = 0; }
        public Point(double x, double y)
        { this.x = x; this.y = y; }
        public Point(Point p)
        { this.x = p.x; this.y = p.y; }
        public double dist(Point b)
        { return Math.Sqrt((this.x - b.x) * (this.x - b.x) +
              (this.y - b.y) * (this.y - b.y)); }
        public static void Main()
        {
            Point p1 = new Point(),
                  p2 = new Point(2, 4.5),
                  p3 = new Point(3.1, 6.7);
            Console.WriteLine("первое расстояние=" + p1.dist(p2));
            Console.WriteLine("второе расстояние=" + p2.dist(p3));
            Console.WriteLine("третье расстояние=" + p3.dist(p1));
            Point p4 = p2;
            Console.WriteLine("расстояние p4=p2 равно" + p4.dist(p2));
        }
    }
}
```

Объект обладает свойствами. В некоторых объектных языках свойства - это просто переменные, которые принадлежат классу. В C# свойства отличаются от переменных и являются отдельной структурой данных. Пусть в классе описаны переменные и к ним невозможно получить доступ извне, если по умолчанию (отсутствуют модификаторы доступа), переменные и методы создаются закрытыми и извне недоступны.

Переменные должны оставаться закрытыми (private или protected), а вот чтобы сделать их открытыми, нужно объявить их свойства следующим образом:

```
public тип ИмяСвойства
{
get return имяПеременной;
set width = value; }
```

Свойства создаются для того, чтобы они были открытыми, поэтому объявление начинается с модификатора доступа public. После этого идет тип данных и имя.

В .NET принято именовать переменные с маленькой буквы, а соответствующие свойства - с большой. Некоторые любят начинать переменные с символа подчеркивания, это тоже нормально, главное - именовать одинаково. В отличие от объявления переменной, тут нет в конце имени точки с запятой, а открываются фигурные скобки, внутри которых нужно реализовать два аксессора (accessor): get и set.

Аксессоры позволяют указать доступ к свойству на чтение (get) и запись (set). После каждого аксессора в фигурных скобках указываются также действия свойства. Для get нужно в фигурных скобках выполнить оператор return и после него указать, какое значение должно возвращать свойство. Значение свойства находится в виртуальной переменной value, которая всегда существует внутри фигурных скобок после ключевого слова set и имеет такой же тип данных, как и свойство. Получается, что свойство просто является оберткой для переменной объекта.

А что, если у вас множество переменных, которые не нуждаются в защите, и им надо всего лишь создать обертку? Писать столь большое количество кода достаточно накладно. Тут можно поступить одним из двух способов: прибегнуть к рефакторингу или применить сокращенный метод объявления свойств. Рассмотрим каждый из этих способов.

Рефакторинг доступен только тем пользователям, которые используют современные средства разработки, да и то не везде есть реализация этой функции, потому что она является не частью платформы или языка, а лишь функцией среды разработки. В Visual Studio, чтобы превратить переменную в свойство, надо выделить переменную и выбрать в меню пункт Refactor 1 Encapsulate Field (Улучшение 1 Искапсулировать поле). В ответ откроется окно, в котором нужно ввести имя будущего свойства. Напомню - свойства желательно именовать так же, как и переменные, но только с большой буквы.

Второй метод - сокращенное объявление свойства. Давайте сокращенным методом объявим глубину сарая:

public тип ИмяСвойства { get; set; }

В фигурных скобках после ключевых слов get и set нет никакого кода, а сразу стоят точки с запятой. Это значит, что свойство будет являться одновременно свойством для внешних источников и переменной для внутреннего использования. В этом случае нельзя добавить код защиты, поэтому данный метод используется там, где свойства являются просто оберткой для переменной.

В общем виде доступ к свойству выглядит следующим образом:

ИмяОбъекта.Свойство

Одно важное замечание - слева от точки указывается именно имя объекта, а не класса, т. е. имя определенного экземпляра класса.

Способы передачи аргументов методу. Существует два способа передачи аргументов методу. Первым способом является вызов по значению. В этом случае значение аргумента копируется в формальный параметр метода. Следовательно, изменения, вносимые в параметр метода, не оказывают никакого влияния на аргумент, используемый для вызова. А вторым способом передачи аргумента является вызов по ссылке. В данном случае параметру метода передается ссылка на аргумент, а не значение аргумента. В методе эта ссылка используется для доступа к конкретному аргументу, указываемому при

вызове. Это означает, что изменения, вносимые в параметр, будут оказывать влияние на аргумент, используемый для вызова метода.

По умолчанию в C# используется вызов по значению.

При передаче методу ссылки на объект сама ссылка по-прежнему передается по значению. Следовательно, создается копия ссылки, а изменения, вносимые в параметр, не оказывают никакого влияния на аргумент. Все изменения, происходящие с объектом, на который ссылается параметр, окажут влияние на тот объект, на который ссылается аргумент. Попытаемся выяснить причины подобного влияния.

Напомним, что при создании переменной типа класса формируется только ссылка на объект. Поэтому при передаче этой ссылки методу принимающий ее параметр будет ссылаться на тот же самый объект, на который ссылается аргумент. Это означает, что и аргумент, и параметр ссылаются на один и тот же объект и что объекты, по существу, передаются методам по ссылке. Таким образом, объект в методе будет оказывать влияние на объект, используемый в качестве аргумента.

Аргументы простых типов передаются методу по значению. Это означает, что изменения, вносимые в параметр, принимающий значение, не будут оказывать никакого влияния на аргумент, используемый для вызова.

Перегрузка методов. В C# допускается совместное использование одного и того же имени двумя или более методами одного и того же класса, при условии, что их параметры объявляются по-разному. В этом случае говорят, что методы перегружаются, а сам процесс называется перегрузкой методов. Перегрузка методов относится к одному из способов реализации полиморфизма в C#.

Для правильной реализации перегрузки функций необходимо соблюдать условие: тип или число параметров у каждого метода должны быть разными. Два метода могут также отличались типами возвращаемых значений, но этого недостаточно для перегрузки. Они должны также отличаться типами или числом своих параметров. Когда вызывается перегружаемый метод, то

выполняется тот его вариант, параметры которого соответствуют (по типу и числу) передаваемым аргументам.

Перегрузка методов поддерживает свойство полиморфизма, поскольку именно таким способом в C# реализуется главный принцип полиморфизма: один интерфейс –множество методов. Для того чтобы стало понятнее, как это делается, обратимся к конкретному примеру. В языках программирования, не поддерживающих перегрузку методов, каждому методу должно быть присвоено уникальное имя. Допустим, что требуется функция, определяющая абсолютное значение. В языках, не поддерживающих перегрузку методов, обычно приходится создавать три или более вариантов такой функции с несколько отличающимися, но все же разными именами. Например, в C функция abs () возвращает абсолютное значение целого числа, функция labs () — абсолютное значение длинного целого числа, а функция fabs() — абсолютное значение числа с плавающей точкой обычной (одинарной) точности. В C перегрузка не поддерживается, и поэтому у каждой функции должно быть свое, особое имя, несмотря на то, что все упомянутые выше функции, по существу, делают одно и то же — определяют абсолютное значение. Но это принципиально усложняет положение, поскольку приходится помнить имена всех трех функций, хотя они реализованы по одному и тому же основному принципу. Подобные затруднения в C# не возникают, поскольку каждому методу, определяющему абсолютное значение, может быть присвоено одно и то же имя. И действительно, в состав библиотеки классов для среды .NET Framework входит метод Abs(), который перегружается в классе System. Math для обработки данных разных числовых типов. Компилятор C# сам определяет, какой именно вариант метода Abs() следует вызывать, исходя из типа передаваемого аргумента. Главная ценность перегрузки заключается в том, что она обеспечивает доступ к связанным вместе методам по общему имени.

Конструктор, как и функция, может быть перегружен. Подходящий конструктор вызывается каждый раз, исходя из аргументов, указываемых при

выполнении оператора new. Перегрузка конструктора класса предоставляет пользователю этого класса дополнительные преимущества в конструировании объектов. Одна из самых распространенных причин для перегрузки конструкторов заключается в необходимости предоставить возможность одним объектам инициализировать другие.

Когда приходится работать с перегружаемыми конструкторами, то иногда очень полезно предоставить возможность одному конструктору вызывать другой. В C# это дается с помощью ключевого слова this. Ниже приведена общая форма такого вызова.

```
имя_конструктора{список_параметров1) :
    this (список_параметров2) {
/1 ... Тело конструктора, которое может быть пустым.
}
```

В исходном конструкторе сначала выполняется перегружаемый конструктор, список параметров которого соответствует критерию список_параметров2, а затем все остальные операторы, если таковые имеются в исходном конструкторе.

```
public Point(double x, double y)
{ this.x = x; this.y = y; }
public Point():this(0,0)
{    }
public Point(Point p):this(p.x, p.y)
{      }
```

Метод Main (). Главный метод имеет различные перегруженные формы, которые позволяют учитывать особенности приложения. Он может быть без параметра

```
static void Main()
static int Main()
```

static void Main(string[ ] args)

static int Main(string[ ] args)

По описанию видно, что метод Main может не возвращать (тип void) и возвращать (тип int) значение; получать (string[ ] args) и не получать аргументы. Как правило, значение, возвращаемое методом Main (), указывает на нормальное завершение программы или на аварийное ее завершение из-за сложившихся ненормальных условий выполнения. Условно нулевое возвращаемое значение обычно указывает на нормальное завершение программы, а все остальные значения обозначают тип возникшей ошибки.

Применение ключевого слова static. Чаще всего доступ к членам класса осуществляется через имя созданного объекта. Для создания члена класса, к которому можно обращаться без ссылки на конкретный экземпляр объекта, его объявления ключевое слово static. С помощью ключевого слова static можно объявлять как переменные, так и методы. Наиболее характерным примером члена типа static служит метод Main (), который объявляется таковым потому, что он должен вызываться операционной системой в самом начале выполняемой программы.

Для того чтобы воспользоваться членом типа static за пределами класса, достаточно указать имя этого класса с оператором-точкой. Переменные, объявляемые как static, по существу, являются глобальными, потому что копия переменной типа static не создается. Вместо этого все экземпляры класса совместно пользуются одной и той же переменной типа static. Такая переменная инициализируется перед ее применением в классе. Когда же ее инициализатор не указан явно, то она инициализируется нулевым значением, если относится к числовому типу данных, пустым значением, если относится к ссылочному типу, или же логическим значением false, если относится к типу bool. Таким образом, переменные типа static всегда имеют какое-то значение. Поля типа static не зависят от конкретного объекта, и поэтому они удобны для хранения информации, применимой ко всему классу.

Метод типа static отличается от обычного метода тем, что его можно вызывать по имени его класса, не создавая экземпляр объекта этого класса. Пример такого вызова уже приводился ранее. Это был метод Sqrt () типа static, относящийся к классу System.Math из стандартной библиотеки классов C#.

На применение методов типа static накладывается ряд следующих ограничений:

• в методе типа static должна отсутствовать ссылка this, поскольку такой метод не выполняется относительно какого-либо объекта;

• в методе типа static допускается непосредственный вызов только других методов типа static, но не метода экземпляра из того самого же класса. Дело в том, что методы экземпляра оперируют конкретными объектами, а метод типа static не вызывается для объекта. Следовательно, у такого метода отсутствуют объекты, которыми он мог бы оперировать;

• аналогичные ограничения накладываются на данные типа static. Для метода типа static непосредственно доступными оказываются только другие данные типа static, определенные в его классе. Он, в частности, не может оперировать переменной экземпляра своего класса, поскольку у него отсутствуют объекты, которыми он мог бы оперировать.

Поля типа static не зависят от конкретного объекта, и поэтому они удобны для хранения информации, применимой ко всему классу.

Конструктор можно также объявить как static. Статический конструктор, как правило, используется для инициализации компонентов, применяемых ко всему классу, а не к отдельному экземпляру объекта этого класса. Поэтому члены класса инициализируются статическим конструктором до создания каких-либо объектов этого класса.

Обратите внимание на то, что конструктор типа static вызывается автоматически, когда класс загружается впервые, причем до конструктора экземпляра. Из этого можно сделать более общий вывод: статический конструктор должен выполняться до любого конструктора экземпляра. Более того, у статических конструкторов отсутствуют модификаторы доступа – они

пользуются доступом по умолчанию, а следовательно, их нельзя вызывать из программы.

Статические классы. Класс можно объявлять как static. Статический класс обладает двумя основными свойствами. Во-первых, объекты статического класса создавать нельзя. И во-вторых, статический класс должен содержать только статические члены. В таком классе все члены должны быть объявлены как static. Ведь если класс становится статическим, то это совсем не означает, что статическими становятся и все его члены. Для объявления статического класса перед его описанием надо добавить ключевое слова static.

Статические классы применяются главным образом в двух случаях. Во-первых, статический класс требуется при создании метода расширения (связаны в основном с языком LINQ). И во-вторых, статический класс служит для хранения совокупности связанных друг с другом статических методов. Таким образом, статический класс может выполняет организационную роль, предоставляя удобные средства для группирования логически связанных методов.

Несмотря на то, что для статического класса не допускается наличие конструктора экземпляра, у него может быть статический конструктор.

Индексаторы. Индексаторы и свойства используются для более успешной интеграции создаваемых классов с классами языка C#.

Для индексирования массива применяется оператор [ ]. Для создаваемых классов можно определить оператор [ ], но с этой целью вместо операторного метода создается индексатор, который позволяет индексировать объект, подобно массиву. Индексаторы применяются, главным образом, в качестве средства, поддерживающего создание специализированных массивов, на которые накладывается одно или несколько ограничений. Тем не менее индексаторы могут служить практически любым целям, для которых выгодным оказывается такой же синтаксис, как и у массивов. Индексаторы могут быть одно- или многомерными.

Ниже приведена общая форма одномерного индексатора:

```
тип_элемента this[int индекс] {
// Аксессор для получения данных,
get {
// Возврат значения, которое определяет индекс.
}
// Аксессор для установки данных,
set {
// Установка значения, которое определяет индекс.
}
}
```

где тип_элемента обозначает конкретный тип элемента индексатора.

Следовательно, у каждого элемента, доступного с помощью индексатора, должен быть определенный тип_элемента. Этот тип соответствует типу элемента массива. Параметр индекс получает конкретный индекс элемента, к которому осуществляется доступ. Формально этот параметр совсем не обязательно должен иметь тип int, но поскольку индексаторы, как правило, применяются для индексирования массивов, то чаще всего используется целочисленный тип данного параметра.

В теле индексатора определены два аксессора (т.е. средства доступа к данным): get и set. Аксессор подобен методу, за исключением того, что в нем не объявляется тип возвращаемого значения или параметры. Аксессоры вызываются автоматически при использовании индексатора, и оба получают - индекс в качестве параметра. Так, если индексатор указывается в левой части оператора присваивания, то вызывается аксессор set и устанавливается элемент, на который указывает параметр индекс. В противном случае вызывается аксессор get и возвращается значение, соответствующее параметру индекс. Кроме того, аксессор set получает неявный параметр value, содержащий значение, присваиваемое по указанному индексу. Преимущество индексатора заключается, в частности, в том, что он позволяет полностью управлять доступом к массиву, избегая нежелательного доступа.

Индексатор может быть перегружен. В этом случае для выполнения выбирается тот вариант индексатора, в котором точнее соблюдается соответствие его параметра и аргумента, указываемого в качестве индекса.

Следует особо подчеркнуть, что индексатор совсем не обязательно должен оперировать массивом. Его основное назначение — предоставить пользователю функциональные возможности, аналогичные массиву. В качестве примера в приведенной ниже программе демонстрируется индексатор, выполняющий роль массива только для чтения, содержащего степени числа 2 от 0 до 15. Обратите внимание на то, что в этой программе отсутствует конкретный массив. Вместо этого индексатор просто вычисляет подходящее значение для заданного индекса.

На применение индексаторов накладываются два существенных ограничения:

1) значение, выдаваемое индексатором, нельзя передавать методу в качестве параметра ref или out, поскольку в индексаторе не определено место в памяти для его хранения.

2) индексатор должен быть членом своего класса и поэтому не может быть объявлен как static.

Необязательные аргументы. В версии C# 4.0 внедрено новое средство, повышающее удобство указания аргументов при вызове метода, которое называется необязательными аргументами и позволяет определить используемое по умолчанию значение для параметра метода. Данное значение будет использоваться по умолчанию в том случае, если для параметра не указан соответствующий аргумент при вызове метода. Применение необязательного аргумента разрешается при создании необязательного параметра. Синтаксис описания метода следующий

```
спец_доступа тип имя_функции (тип параметр1=значение1,
    …, тип параметрN=значениеN)
{/* тело функции */}
```

Значение параметра по умолчанию должно быть постоянной соответствующего типа. Следует иметь в виду, что все необязательные аргументы должны непременно указываться справа от обязательных.

Преимущество необязательных аргументов заключается упрощенном обращении к сложным методам и конструкторам. Это означает, что передавать нужно лишь те аргументы, которые важны в данном конкретном случае, а не все аргументы, которые в противном случае должны быть обязательными.

В некоторых случая (например, количество параметров) можно заменить перегрузку методов введением в описание функции необязательных параметров.

Именованные аргументы. Еще одним средством, связанным с передачей аргументов методу, является именованный аргумент, который был внедрен в версии C# 4.0. При передаче фактических параметров функции их порядок должен соответствовать списку формальных параметров, такие параметры называются позиционными. Чтобы не учитывать это ограничение можно использовать именованные аргументы. Именованный аргумент позволяет указать имя того параметра, которому присваивается его значение. И в этом случае порядок следования аргументов уже не имеет никакого значения. Таким образом, именованные аргументы в какой-то степени похожи на упоминавшиеся ранее инициализаторы объектов, хотя и отличаются от них своим синтаксисом.

При вызове функции используется форма

имя_функции(имя_параметра : значение, …, имя_параметра : значение)

Здесь имя_параметра обозначает имя того параметра, которому передается значение, причем имя_параметра должно обозначать имя действительного параметра для вызываемого метода.

Преимущества в использовании именованных аргументов:

– независимость от порядка следования;

– позиционные аргументы можно указывать вместе с именованными в одном и том же вызове.

1.3. Парадигмы ООП – наследование и полиморфизм

Наследование [13] является одним из трех основополагающих принципов объектно-ориентированного программирования, поскольку оно допускает создание иерархических классификаций. Благодаря наследованию можно создать общий класс, в котором определяются характерные особенности, присущие множеству связанных элементов. От этого класса могут затем наследовать другие, более специализированные классы, добавляя свои индивидуальные особенности. Классическое наследование позволяет строить новые определения классов, расширяющие функциональность существующих классов.

Существующий класс, который будет служить основой для нового класса, называется базовым или родительским классом. Назначение базового класса состоит в определении всех общих данных и членов для классов, которые расширяют его. Расширяющие классы формально называются производными или дочерними классами.

Следовательно, производный класс представляет собой специализированный вариант базового класса. Он наследует все переменные, методы, свойства и индексаторы, определяемые в базовом классе, добавляя к ним свои собственные элементы [24].

Поддержка наследования в C# состоит в том, что в объявление одного класса разрешается вводить другой класс. Для этого при объявлении производного класса указывается базовый класс

```
class производный_класс : базовый_класс {
//  тело класса
```

```
}
```

Для любого производного класса можно указать **только один базовый** класс. В C# не предусмотрено наследование нескольких базовых классов в одном производном классе. Тем не менее можно создать иерархию наследования, в которой производный класс становится базовым для другого производного класса. Главное преимущество наследования заключается в следующем: как только будет создан базовый класс, в котором определены общие для множества объектов атрибуты, он может быть использован для создания любого числа более конкретных производных классов. А в каждом производном классе может быть точно выстроена своя собственная классификация.

Наследование класса не отменяет ограничения, накладываемые на доступ к закрытым членам класса [22]. Поэтому если в производный класс и входят все члены его базового класса, в нем все равно оказываются недоступными те члены базового класса, которые являются закрытыми.

Для преодоления данного ограничения в C# предусмотрены разные способы. Один из них состоит в использовании защищенных (protected) членов класса, а второй - в применении открытых свойств для доступа к закрытым данным.

Закрытый член базового класса недоступен для производного класса. Из этого можно предположить, что для доступа к некоторому члену базового класса из производного класса этот член необходимо сделать открытым. Но если сделать член класса открытым, то он станет доступным для всего кода, что далеко не всегда желательно. Правда, упомянутое предположение верно лишь отчасти, поскольку в C# допускается создание защищенного члена класса. Защищенный член является открытым в пределах иерархии классов, но закрытым за пределами этой иерархии.

Защищенный член создается с помощью модификатора доступа protected. Если член класса объявляется как protected, он для самого класса становится

закрытым, но за исключением одного случая, когда защищенный член наследуется. В этом случае защищенный член базового класса становится защищенным членом производного класса, а значит, доступным для производного класса. Таким образом, используя модификатор доступа protected, можно создать члены класса, являющиеся закрытыми для своего класса, но все же наследуемыми и доступными для производного класса.

Свойства в классах чаще всего определяются как public, а поля закрытыми. В этом случае свойством можно будет воспользоваться в производном классе, но нельзя будет получить непосредственный доступ к его базовой закрытой переменной.

Аналогично состоянию public и private, состояние protected сохраняется за членом класса независимо от количества уровней наследования. Поэтому когда производный класс используется в качестве базового для другого производного класса, любой защищенный член исходного базового класса, наследуемый первым производным классом, наследуется как защищенный и вторым производным классом.

Конструктор [15]. В классах конструкторы обычно имеют спецификатор public, но производный класс никогда не наследует конструкторы своего базового класса.

В иерархии классов допускается, чтобы у базовых и производных классов были свои собственные конструкторы. При создании объекта конструктор базового класса конструирует базовую часть объекта, а конструктор производного класса – производную часть этого объекта. И в этом есть своя логика, поскольку базовому классу неизвестны и недоступны любые элементы производного класса, а значит, их конструирование должно происходить раздельно. Если конструктор определен только в производном классе, то конструируется объект производного класса, а базовая часть объекта автоматически конструируется конструктором по умолчанию базового класса.

Когда конструкторы определяются как в базовом, так и в производном классе, должны выполняться конструкторы обоих классов. Сначала вызывается

конструктор базового класса, а потом производного класса. При необходимости можно вызвать конкретный базовый конструктор, используя ключевое слово base.

Ключевое слово языка C# base применяется

- для вызова конструктора базового класса;

- для доступа к члену базового класса, скрывающегося за членом производного класса.

С помощью формы расширенного объявления конструктора производного класса и ключевого слова base в производном классе может быть вызван конструктор, определенный в его базовом классе. Ниже приведена общая форма этого расширенного объявления:

```
конструктор_производного_класса(список_параметров) :
                base (список_аргументов) {
// тело конструктора
}
```

где список_аргументов обозначает любые аргументы, необходимые конструктору в базовом классе.

С помощью ключевого слова base [12] можно вызвать конструктор любой формы, определяемой в базовом классе, причем выполняться будет лишь тот конструктор, параметры которого соответствуют переданным аргументам.

Ключевое слово base всегда обращается к базовому классу, стоящему в иерархии непосредственно над вызывающим классом. Это справедливо даже для многоуровневой иерархии классов. Если же ключевое слово отсутствует, то автоматически вызывается конструктор, используемый в базовом классе по умолчанию.

Сокрытие имен. В производном классе можно определить член с таким же именем, как и у члена его базового класса. В этом случае член базового класса скрывается в производном классе. И хотя формально в C# это не

считается ошибкой, компилятор все же выдаст сообщение, предупреждающее о том, что имя скрывается. Если член базового класса требуется скрыть намеренно, то перед его именем следует указать ключевое слово new, чтобы избежать появления подобного предупреждающего сообщения. Следует, однако, иметь в виду, что это совершенно отдельное применение ключевого слова new, не похожее на его применение при создании экземпляра объекта.

Для обращения к скрытому элементу родительского класса применяется форма base.член_класса, где член_класса - обозначать метод или переменную экземпляра. Например,

```
using System;
class A {
public int i = 0;
}
// Производный класс
class B : A {
new int i; // этот член скрывает член i из класса A
public B(int a, int b) {
base.i = a; // здесь обнаруживается скрытый член из класса A
i = b; // член i из класса B
}
public void Show() {
// Выводится член i из класса A
Console.WriteLine("Член i в базовом классе" + base.i);
// Выводится член i из класса B
Console.WriteLine("Член i в производном классе" + i);
}
}
class UncoverName {
static void Main() {
```

```
B ob = new B(1, 2) ;
ob.Show ();
}
}
```

У принципа строгого соблюдения типов в C# имеется одно важное исключение: переменной ссылки на объект базового класса может быть присвоена ссылка на объект любого производного от него класса. Такое присваивание считается вполне допустимым, поскольку экземпляр объекта производного типа инкапсулирует экземпляр объекта базового типа. Следовательно, по ссылке на объект базового класса можно обращаться к объекту производного класса.

Следует особо подчеркнуть, что доступ к конкретным членам класса определяется типом переменной ссылки на объект, а не типом объекта, на который она ссылается. Это означает, что если ссылка на объект производного класса присваивается переменной ссылки на объект базового класса, то доступ разрешается только к тем частям этого объекта, которые определяются базовым классом.

Виртуальные методы и их переопределение [23]. Виртуальным называется такой метод, который объявляется как virtual в базовом классе. Виртуальный метод отличается тем, что он может быть переопределен в одном или нескольких производных классах. Следовательно, у каждого производного класса может быть свой вариант виртуального метода. Кроме того, виртуальные методы интересны тем, что именно происходит при их вызове по ссылке на базовый класс. В этом случае средствами языка C# определяется именно тот вариант виртуального метода, который следует вызывать, исходя из типа объекта, к которому происходит обращение по ссылке, причем это делается во время выполнения. Поэтому при ссылке на разные типы объектов выполняются разные варианты виртуального метода. Иными словами, вариант выполняемого виртуального метода выбирается по типу объекта, а не по типу

ссылки на этот объект. Так, если базовый класс содержит виртуальный метод и от него получены производные классы, то при обращении к разным типам объектов по ссылке на базовый класс выполняются разные варианты этого виртуального метода.

Метод объявляется как виртуальный в базовом классе с помощью ключевого слова virtual, указываемого перед его именем. Когда же виртуальный метод переопределяется в производном классе, то для этого используется модификаторы override или new. А сам процесс повторного определения виртуального метода в производном классе называется переопределением метода. При переопределении имя, возвращаемый тип и сигнатура переопределяющего метода должны быть точно такими же, как и у того виртуального метода, который переопределяется. Кроме того, виртуальный метод не может быть объявлен как static или abstract.

При использовании ключевого слова new создается новый метод в классе, а не переопределяется метод базового класса. Ключевое слово override переопределяет метод базового класса.

Переопределение метода служит основанием для воплощения одного из самых эффективных в C# принципов: динамической диспетчеризации методов, которая представляет собой механизм разрешения вызова во время выполнения, а не компиляции. Значение динамической диспетчеризации методов состоит в том, что именно благодаря ей в C# реализуется динамический полиморфизм.

```
using System;
class Base {
// Создать виртуальный метод в базовом классе.
public virtual void Who() {
Console.WriteLine("Метод Who() в классе Base");
}
}
```

```
class Derived1 : Base {
// Переопределить метод Who() в производном классе.
public override void Who() {
Console.WriteLine("Метод Who() в классе Derived1");
}
}
class Derived2 : Base {
// Вновь переопределить метод Who() в еще одном производном классе.
public override void Who() {
Console.WriteLine("Метод Who() в классе Derived2");
}
}
class OverrideDemo {
static void Main() {
Base baseOb = new Base();
Derived1 dObl = new Derived1();
Derived2 dOb2 = new Derived2();
Base baseRef; // ссылка на базовый класс
baseRef = baseOb;
baseRef.Who();
baseRef = dObl;
baseRef.Who() ;
baseRef = dOb2;
baseRef.Who();
}
}
```

Вот к какому результату приводит выполнение этого кода.

Метод Who() в классе Base.

Метод Who() в классе Derived1

Метод Who() в классе Derived2

Если при наличии многоуровневой иерархии виртуальный метод не переопределяется в производном классе, то выполняется ближайший его вариант, обнаруживаемый вверх по иерархии.

Индексаторы и свойства также подлежат модификации ключевым словом virtual и переопределению ключевым словом override.

Благодаря переопределению методов в C# поддерживается динамический полиморфизм. В объектно-ориентированном программировании полиморфизм играет очень важную роль, потому что он позволяет определить в общем классе методы, которые становятся общими для всех производных от него классов, а в производных классах — определить конкретную реализацию некоторых или же всех этих методов.

Переопределение методов — это еще один способ воплотить в C# главный принцип полиморфизма: один интерфейс — множество методов.

Удачное применение полиморфизма отчасти зависит от правильного понимания той особенности, что базовые и производные классы образуют иерархию, которая продвигается от меньшей к большей специализации. При надлежащем применении базовый класс предоставляет все необходимые элементы, которые могут использоваться в производном классе непосредственно. А с помощью виртуальных методов в базовом классе определяются те методы, которые могут быть самостоятельно реализованы в производном классе. Таким образом, сочетая наследование с виртуальными методами, можно определить в базовом классе общую форму методов, которые будут использоваться во всех его производных классах.

Предотвращение наследования. В C# имеется возможность предотвратить наследование класса с помощью ключевого слова sealed.

Для того чтобы предотвратить наследование класса, достаточно указать ключевое слово sealed перед определением класса. Как и следовало ожидать, класс не допускается объявлять одновременно как abstract и sealed, поскольку сам абстрактный класс реализован не полностью и опирается в этом отношении

на свои производные классы, обеспечивающие полную реализацию. Объявления класса типа sealed

```
sealed class A {
// ...
}
// Следующий класс недопустим.
class B : A { // ОШИБКА! Наследовать класс A нельзя
// ...
}
```

Ключевое слово sealed может быть также использовано в виртуальных методах для предотвращения их дальнейшего переопределения.

Допустим, что имеется базовый класс B и производный класс D. Метод, объявленный в классе B как virtual, может быть объявлен в классе D как sealed. Благодаря этому в любом классе, наследующем от класса D предотвращается переопределение данного метода. Подобная ситуация демонстрируется в приведенном ниже фрагменте кода.

```
class B {
public virtual void MyMethod() { /* ... */ }
}
class D : B {
// Здесь герметизируется метод MyMethod() и
// предотвращается его дальнейшее переопределение.
sealed public override void MyMethod() { /* ... */ }
}
class X : D {
// Ошибка! Метод MyMethod() герметизирован!
public override void MyMethod() { /* ... */ }
}
```

Метод MyMethod () герметизирован в классе D, и поэтому не может быть переопределен в классе X.

Класс object [14]. В C# предусмотрен специальный класс object, который неявно считается базовым классом для всех остальных классов и типов, включая и типы значений. Иными словами, все остальные типы являются производными от object [9]. Это, в частности, означает, что переменная ссылочного типа object может ссылаться на объект любого другого типа. Кроме того, переменная типа object может ссылаться на любой массив, поскольку в C# массивы реализуются как объекты. Формально имя object считается в C# еще одним обозначением класса System.Object, входящего в библиотеку классов для среды .NET Framework. Методы определенные в классе имют следующий синтаксис

```
public virtual bool Equals(object ob)
public satic bool Equals(object ob1, object ob2)
protected Finalize()
public virtual int GetHashCode()
public Type GetType()
protected object MemberwiseClone()
public satic bool ReferenceEquals(object ob1, object ob2)
public virtual sring ToString()
```

По умолчанию метод Equals (object) определяет, ссылается ли вызывающий объект на тот же самый объект, что и объект, указываемый в качества аргумента этого метода, т.е. он определяет, являются ли обе ссылки одинаковыми. Метод Equals (object) возвращает логическое значение true, если сравниваемые объекты одинаковы, в противном случае — логическое значение false. Он может быть также переопределен в создаваемых классах. Это позволяет выяснить, что же означает равенство объектов для создаваемого

класса. Например, метод Equals (object) можно определить таким образом, чтобы в нем сравнивалось содержимое двух объектов.

Метод GetHashCode () возвращает хеш-код, связанный с вызывающим объектом. Этот хеш-код можно затем использовать в любом алгоритме, где хеширование применяется в качестве средства доступа к хранимым объектам. Следует, однако, иметь в виду, что стандартная реализация метода GetHashCode () не пригодна на все случаи применения.

Если перегружается оператор ==, то обычно приходится переопределять методы Equals(object) и GetHashCode(), поскольку чаще всего требуется, чтобы метод Equals (object) и оператор == функционировали одинаково. Когда переопределяется метод Equals(object), то следует переопределить и метод GetHashCode(), чтобы оба метода оказались совместимыми.

Метод ToString() возвращает символьную строку, содержащую описание того объекта, для которого он вызывается. Кроме того, метод ToString() автоматически вызывается при выводе содержимого объекта с помощью метода WriteLine (). Этот метод переопределяется во многих классах, что позволяет приспосабливать описание к конкретным типам объектов, создаваемых в этих классах.

Как пояснялось выше, все типы в C#, включая и простые типы значений, являются производными от класса object. Следовательно, ссылкой типа object можно воспользоваться для обращения к любому другому типу, в том числе и к типам значений. Когда ссылка на объект класса object используется для обращения к типу значения, то такой процесс называется упаковкой. Упаковка приводит к тому, что значение простого типа сохраняется в экземпляре объекта, т.е. "упаковывается" в объекте, который затем используется как и любой другой объект. Но в любом случае упаковка происходит автоматически. Для этого достаточно присвоить значение переменной ссылочного типа object, а об остальном позаботится компилятор C#.

Распаковка представляет собой процесс извлечения упакованного значения из объекта. Это делается с помощью явного приведения типа ссылки

на объект класса object к соответствующему типу значения. Попытка распаковать объект в другой тип может привести к ошибке во время выполнения.

Ниже приведен простой пример, демонстрирующий упаковку и распаковку.

```
// Простой пример упаковки и распаковки.
using System;
class BoxingDemo { .
static void Main() {
int x;
object obj;
x = 10;
obj = x; // упаковать значение переменной x в объект
int y = (int)obj; // распаковать значение из объекта, доступного по
// ссылке obj, в переменную типа int
Console.WriteLine(y);
}
}
```

В этом примере кода выводится значение 10. Обратите внимание на то, что значение переменной x упаковывается в объект простым его присваиванием переменной obj, ссылающейся на этот объект. А затем это значение извлекается из объекта, доступного по его ссылке obj, и далее приводится к типу int.

Упаковка и распаковка позволяют полностью унифицировать систему типов в C#. Благодаря тому что все типы являются производными от класса object, ссылка на значение любого типа может быть просто присвоена переменной ссылочного типа object, а все остальное возьмут на себя упаковка и распаковка. Более того, методы класса object оказываются доступными всем типам, поскольку они являются производными от этого класса.

Если object является базовым классом для всех остальных типов и упаковка значений простых типов происходит автоматически, то класс object можно вполне использовать в качестве "универсального" типа данных. Пример программы, в которой сначала создается массив типа object, элементам которого затем присваиваются значения различных типов данных.

А самое главноедля программирования на C# доступны подлинно обобщенные типы данных - обобщения. Внедрение обобщений позволило без труда определять классы и алгоритмы, автоматически обрабатывающие данные разных типов, соблюдая типовую безопасность. Благодаря обобщениям отпала необходимость пользоваться классом object как универсальным типом данных при создании нового кода. Универсальный характер этого класса лучше теперь оставить для применения в особых случаях. Обобщения представляют аналог шаблонов в C++.

Для расммотрения применения изученного материала создан класс обыкновенные дроби и класс, использующий его для демонстрации использования интерфейса класса.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Text.RegularExpressions;
namespace Fraction
{
   /// <summary>
   /// Класс Fraction предоставляет функции для работы с дробями
   /// </summary>
   class Fraction
   {
      #region Fields
      /// <summary>
```

```
        /// Поле для хранения числителя дробной части
        /// </summary>
        private int _Numerator = 0;
        /// <summary>
        /// Поле для хранения знаменателя дробной части
        /// </summary>
        private int _Denominator = 1;
        #endregion
        #region Properties
        /// <summary>
        /// Числитель дробной части
        /// </summary>
        public int Numerator
        {
            get { return _Numerator; }
            set {if (value==0) Denominator=1; _Numerator=value;simplify();}
        }
        /// <summary>
        /// Знаменатель дробной части
        /// </summary>
        public int Denominator
        {
            get { return _Denominator; }
            set { if (value == 0) Console.WriteLine("Divide by zero!"); else { if
(value < 0) { Numerator *= -1; simplify(); } _Denominator = Math.Abs(value); } }
        }
        #endregion
        #region Constructors
        /// <summary>
```

```
        /// Инициализирует нновый экземпляр класса Fraction с заданной целой и дробной частью
        /// </summary>
        /// <param name="integral">Целая часть числа</param>
        /// <param name="numerator">Числитель дробной части</param>
        /// <param name="denominator">Знаменатель дробной части</param>
        public Fraction(int integral, int numerator, int denominator)
        {
            Numerator = (numerator + Math.Abs(integral) * denominator) * (integral < 0 ? -1 : 1);
            Denominator = denominator;
            simplify();
        }
        /// <summary>
        /// Инициализирует нновый экземпляр класса Fraction с заданной целой частью
        /// </summary>
        /// <param name="integral">Целая часть числа</param>
        public Fraction(int integral)
            : this(integral, 0, 1)
        { }
        /// <summary>
        /// Инициализирует нновый экземпляр класса Fraction с заданной дробной частью
        /// </summary>
        /// <param name="numerator">Числитель дробной части</param>
        /// <param name="denominator">Знаменатель дробной части</param>
        public Fraction(int numerator, int denominator)
            : this(0, numerator, denominator)
        { }
```

```
        /// <summary>
        /// Инициализирует нновый экземпляр класса Fraction, равный 0
        /// </summary>
        public Fraction()
            : this(0, 0, 1)
        { }
        #endregion
        #region Methods
        /// <summary>
        /// Функция сокращения дроби
        /// </summary>
        public void simplify()
        {
            int gcd = Gcd(_Numerator, _Denominator);
            _Numerator /= gcd;
            _Denominator /= gcd;
            if (_Denominator < 0)
            {
                _Numerator = -_Numerator;
                _Denominator = -_Denominator;
            }
        }
        /// <summary>
        /// Функция нахождения наибольшего общего делителя двух чисел
        /// </summary>
        /// <returns>Возвращает число, равное наибольшему общему делителю заданных чисел</returns>
        public static int Gcd(int a, int b)
        {
            while (b != 0)
```

```
            b = a % (a = b);
        return a;
    }
    /// <summary>
    /// Позволяет получить целую часть
    /// </summary>
    /// <returns>Возвращает целую часть от числа</returns>
    public int GetIntegral()
    {
        return this.Numerator / this.Denominator;
    }
    /// <summary>
    /// Позволяет получить строковое представление в формате [Целая_часть] [Числитель]/[Знаменатель]
    /// </summary>
    /// <returns>Возвращает строковое представление дроби</returns>
    public override string ToString()
    {
        if (Numerator % Denominator == 0)
            return
                String.Format("{0}", Numerator / Denominator);
        else
            if (Numerator / Denominator == 0)
                return String.Format("{0}/{1}", Numerator, Denominator);
            else
                return String.Format("{0} {1}/{2}", Numerator / Denominator, Math.Abs(Numerator - Numerator / Denominator * Denominator), Denominator);
    }
    /// <summary>
```

```
        /// Возвращает значение, указывающее, равен ли данный экземпляр заданному объекту
        /// </summary>
        /// <param name="obj">Объект, сравниваемый с этим экземпляром</param>
        /// <returns>Возвращает значение true, если дроби совпадают и false - в противном случае</returns>
        public override bool Equals(object obj)
        {
            if (((Fraction)obj).Numerator == this.Numerator && ((Fraction)obj).Denominator == this.Denominator)
                return true;
            else
                return false;
        }
        /// <summary>
        /// Возвращает хэш-код строкового представления данного экземпляра
        /// </summary>
        /// <returns>Возвращает хэш-код данного экземпляра</returns>
        public override int GetHashCode()
        {
            return this.ToString().GetHashCode();
        }
        /// <summary>
        /// Возвращает копию данного экземпляра
        /// </summary>
        /// <returns>Возвращает объект типа Fraction, равный данному экземпляру</returns>
        public Fraction Copy()
```

```
        {
            return (Fraction)this.MemberwiseClone();
        }
        /// <summary>
        /// Преобразует строковое представление дроби в объект класса Fraction. Возвращает значение, указывающее успешно или нет выполнено преобразование
        /// </summary>
        /// <param name="path">Строка, содержащая преобразуемую дробь</param>
        /// <param name="frac">Содержит объект класса Fraction, соответствующий заданной строке в случае, если преобразование выполнено успешно, и 0 - в противном случае</param>
        /// <returns>Возвращает значение true, если преобразование выполнено успешно и false - в противном случае</returns>
        public static bool TryParse(string path, out Fraction frac)
        {
            bool result;
            int temp_int = 0, temp_num = 0, temp_denom = 1;
            Regex r = new Regex(@"^(?<integral>[+-]?\d+)
                  ( +(?<numerator>\d+)/(?<denominator>\d+))?$");
            Match m = r.Match(path);
            if (m.Success)
            {
           temp_int = int.Parse(m.Groups["integral"].ToString());
         bool f = int.TryParse(m.Groups["numerator"].ToString(), out temp_num);
        if (!int.TryParse(m.Groups["denominator"].ToString(), out temp_denom))
              {
                  temp_denom = 1;
                  result = true;
```

```
        }
        else
            if (temp_denom == 0)
                result = false;
            else
                result = true;
    }
    else
    {
     r = new Regex(@"^(?<numerator>[+-]?\d+)/(?<denominator>\d+)?$");
        m = r.Match(path);
        if (m.Success)
        {
            temp_int = 0;
bool f = int.TryParse(m.Groups["numerator"].ToString(), out temp_num);
 if (!int.TryParse(m.Groups["denominator"].ToString(), out temp_denom))
            {
                temp_denom = 1;
                result = true;
            }
            else
                if (temp_denom == 0)
                    result = false;
                else
                    result = true;
        }
        else
            result = false;
    }
    frac = new Fraction(temp_int, temp_num, temp_denom);
```

```
        return result;
    }
    #endregion
    #region Operators
    public static Fraction operator +(Fraction a, Fraction b)
    {
        int num = 0, denom = 1;
        if (a.Denominator != b.Denominator)
        {
         denom = b.Denominator * a.Denominator;
          num = a.Denominator * b.Numerator + a.Numerator * b.Denominator;
        }
        else
        {
            num = a.Numerator + b.Numerator;
            denom = a.Denominator;
        }
        return new Fraction(num, denom);
    }
    public static Fraction operator -(Fraction a, Fraction b)
    {
        int num = 0, denom = 1;
        if (b.Denominator != a.Denominator)
        {
            denom = b.Denominator * a.Denominator;
            num = a.Numerator * b.Denominator - a.Denominator *
b.Numerator;
        }
        else
        {
```

```
            num = a.Numerator - b.Numerator;
            denom = a.Denominator;
        }
        return new Fraction(num, denom);
    }
    public static Fraction operator *(Fraction a, Fraction b)
    {
        int num, denom;
        denom = a.Denominator * b.Denominator;
        num = a.Numerator * b.Numerator;
        return new Fraction(num, denom);
    }
    public static Fraction operator /(Fraction a, Fraction b)
    {
        int num, denom;
        denom = a.Denominator * b.Numerator;
        num = a.Numerator * b.Denominator;
        return new Fraction(num, denom);
    }
    #endregion
  }
}
```

Программа, использующая класс Fracion

```
        using System;
        using System.Collections.Generic;
        using System.Linq;
        using System.Text;
        namespace Fraction
        {
            class Program
```

```
    {
        static void Main(string[] args)
        {
            Fraction a,b;
            do
            {
                if (Fraction.TryParse(Console.ReadLine(), out a))
                    Console.WriteLine(a.ToString());
                Console.WriteLine("Целая часть:" + a.GetIntegral());
                if (Fraction.TryParse(Console.ReadLine(), out b))
                    Console.WriteLine(b.ToString());
                Console.WriteLine("Сумма: " + (a + b).ToString());
                Console.WriteLine("Разность: " + (a - b).ToString());
                Console.WriteLine("Умножение: " + (a * b).ToString());
                Console.WriteLine("Деление: " + (a / b).ToString());
            }
            while (true);
        }
    }
}
```

Иногда требуется создать базовый класс, в котором определяется лишь самая общая форма для всех его производных классов, а наполнение ее деталями предоставляется каждому из этих классов. В таком классе определяется лишь характер методов, которые должны быть конкретно реализованы в производных классах, а не в самом базовом классе. Подобная ситуация возникает, например, в связи с невозможностью получить содержательную реализацию метода в базовом классе.

Абстрактный метод создается с помощью указываемого модификатора типа abstract. У абстрактного метода отсутствует тело, и поэтому он не

реализуется в базовом классе. Это означает, что он должен быть переопределен в производном классе, поскольку его вариант из базового класса просто непригоден для использования.

Нетрудно догадаться, что абстрактный метод автоматически становится виртуальным и не требует указания модификатора virtual. В действительности совместное использование модификаторов virtual и abstract считается ошибкой.

Для определения абстрактного метода служит форма

abstract тип имя(список_параметров);

У абстрактного метода отсутствует тело. Модификатор abstract может применяться только в методах экземпляра, но не в статических методах (static).

Абстрактными могут быть также индексаторы и свойства.

Класс, содержащий один или больше абстрактных методов, должен быть также объявлен как абстрактный, и для этого перед его объявлением class указывается модификатор abstract. А поскольку реализация абстрактного класса не определяется полностью, то у него не может быть объектов. Следовательно, попытка создать объект абстрактного класса с помощью оператора new приведет к ошибке во время компиляции.

Когда производный класс наследует абстрактный класс, в нем должны быть реализованы все абстрактные методы базового класса. В противном случае производный класс должен быть также определен как abstract. Таким образом, атрибут abstract наследуется до тех пор, пока не будет достигнута полная реализация класса.

В абстрактные классы вполне допускается (и часто практикуется) включать конкретные методы, которые могут быть использованы в своем исходном виде в производном классе. А переопределению в производных классах подлежат только те методы, которые объявлены как abstract.

В абстрактном классе, рассмотренном ранее определяется, что именно должен делать класс, но не как он должен это делать. В абстрактном методе определяются возвращаемый тип и сигнатура метода, но не предоставляется его

реализация. А в производном классе должна быть обеспечена своя собственная реализация каждого абстрактного метода, определенного в его базовом классе. Таким образом, абстрактный метод определяет интерфейс, но не реализацию метода.

Интерфейсы. Конечно, абстрактные классы и методы приносят известную пользу, но положенный в их основу принцип может быть развит далее. В C# предусмотрено разделение интерфейса класса и его реализации с помощью ключевого слова interface.

С точки зрения синтаксиса интерфейсы подобны абстрактным классам. Но в интерфейсе ни у одного из методов не должно быть тела. Это означает, что в интерфейсе вообще не предоставляется никакой реализации. В нем указывается только, что именно следует делать, но не как это делать. Как только интерфейс будет определен, он может быть реализован в любом количестве классов. Кроме того, в одном классе может быть реализовано любое количество интерфейсов.

Для реализации интерфейса в классе должны быть предоставлены тела (т.е. конкретные реализации) методов, описанных в этом интерфейсе. Каждому классу предоставляется полная свобода для определения деталей своей собственной реализации интерфейса. Следовательно, один и тот же интерфейс может быть реализован в двух классах по-разному. Тем не менее в каждом из них должен поддерживаться один и тот же набор методов данного интерфейса. А в том коде, где известен такой интерфейс, могут использоваться объекты любого из этих двух классов, поскольку интерфейс для всех этих объектов остается одинаковым. Благодаря поддержке интерфейсов в C# может быть в полной мере реализован главный принцип полиморфизма: один интерфейс – множество методов.

Интерфейсы объявляются с помощью ключевого слова interface:

```
interface имя{
возвращаемый_тип имя_метода1 (список_параметров) ;
возвраща емый_ тип имя_ме то да 2 (список_параме тров) ;
```

```
// ...
возвращаемый_тип имя_методаЫ{список_параметров) ;
}
```

где имя — это конкретное имя интерфейса. В объявлении методов интерфейса используются только их возвращаемый_тип и сигнатура. Они, по существу, являются абстрактными методами. Как пояснялось выше, в интерфейсе не может быть никакой реализации. Поэтому все методы интерфейса должны быть реализованы в каждом классе, включающем в себя этот интерфейс. В самом же интерфейсе методы неявно считаются открытыми, поэтому доступ к ним не нужно указывать явно.

Ниже приведен пример объявления интерфейса для класса, генерирующего последовательный ряд чисел.

```
public interface ISeries {
int GetNextO; // возвратить следующее по порядку число
void Reset(); // перезапустить
void SetStart(int x); // задать начальное значение
}
```

Этому интерфейсу присваивается имя ISeries. Префикс I в имени интерфейса указывать необязательно, но это принято делать в практике программирования, чтобы как-то отличать интерфейсы от классов. Интерфейс ISeries объявляется как public и поэтому может быть реализован в любом классе какой угодно программы.

Помимо методов, в интерфейсах можно также указывать свойства, индексаторы и события. Интерфейсы не могут содержать члены данных. В них нельзя также определить конструкторы, деструкторы или операторные методы. Кроме того, ни один из членов интерфейса не может быть объявлен как static.

Как только интерфейс будет определен, он может быть реализован в одном или нескольких классах. Для реализации интерфейса достаточно указать

его имя после имени класса, аналогично базовому классу. Ниже приведена общая форма реализации интерфейса в классе.

```
class имя_класса : имя_интерфейса1,
    имя_интерфейса2, …, имя_интерфейсаN {
// тело класса
}
```

где имя_интерфейса1, …, имя_интерфейсаN — это конкретное имя реализуемого интерфейса.

Если уж интерфейс реализуется в классе, то это должно быть сделано полностью. В частности, реализовать интерфейс выборочно и только по частям нельзя. В классе допускается реализовывать несколько интерфейсов. В этом случае все реализуемые в классе интерфейсы указываются списком через запятую. В классе можно наследовать базовый класс и в тоже время реализовать один или более интерфейс.

В таком случае имя базового класса должно быть указано перед списком интерфейсов, разделяемых запятой.

Методы, реализующие интерфейс, должны быть объявлены как public. Дело в том, что в самом интерфейсе эти методы неявно подразумеваются как открытые, поэтому их реализация также должна быть открытой. Кроме того, возвращаемый тип и сигнатура реализуемого метода должны точно соответствовать возвращаемому типу и сигнатуре, указанным в определении интерфейса.

Ниже приведен пример программы, в которой реализуется представленный ранее интерфейс ISeries. В этой программе создается класс ByTwos, генерирующий последовательный ряд чисел, в котором каждое последующее число на два больше предыдущего.

```
// Реализовать интерфейс ISeries,
class ByTwos : ISeries {
int start;
int val;
```

```
public ByTwos () {
start = 0;
val = 0;
}
public int GetNext() {
val += 2;
return val;
}
public void Reset() {
val = start;
}
public void SetStart(int x) {
start = x;
val = start;
}
}
// Продемонстрировать применение класса ByTwos, реализующего интерфейс,
using System;
class SeriesDemo {
static void Main() {
ByTwos ob = new ByTwos(); /
for(int i=0; i < 5; i++)
Console.WriteLine ("Следующее число равно " + ob.GetNext());
Console.WriteLine("ХпСбросить");
ob.Reset () ;
for(int i=0; i < 5; i++)
Console.WriteLine("Следующее число равно " + ob.GetNext());
Console.WriteLine("ХпНачать с числа 100");
ob.SetStartA00);
```

```
for(int i=0; i < 5; i++)
Console.WriteLine("Следующее число равно " + ob.GetNext()) ;
}
}
```

Для того чтобы скомпилировать код класса SeriesDemo, необходимо включить в компиляцию файлы, содержащие интерфейс ISeries, а также классы ByTwos и SeriesDemo. Компилятор автоматически скомпилирует все три файла и сформирует из них окончательный исполняемый файл.

Как пояснялось выше, интерфейс может быть реализован в любом количестве классов. В качестве примера ниже приведен класс Primes, генерирующий ряд простых чисел. Обратите внимание на то, реализация интерфейса ISeries в этом классе коренным образом отличается от той, что предоставляется в классе ByTwos.

```
// Использовать интерфейс ISeries для реализации
// процесса генерирования простых чисел,
class Primes : ISeries {
int start;
int val;
public Primes() {
start = 2;
val = 2;
}
public int GetNextO {
int i, j;
bool isprime;
val++;
for(i = val; i < 1000000; i++) {
isprime = true;
for(j = 2; j <= i/j; j++) {
if((i%j)==0) { isprime = false; break; }
```

```
}
if (isprime) { val = i; break; }
}
return val;
}
public void Reset() {
val = start;
}
public void SetStart(int x) {
start = x;
val = start;
}
}
```

Как это ни покажется странным, но в C# допускается объявлять переменные ссылочного интерфейсного типа, т.е. переменные ссылки на интерфейс. Такая переменная может ссылаться на любой объект, реализующий ее интерфейс. При вызове метода для объекта посредством интерфейсной ссылки выполняется его вариант, реализованный в классе данного объекта. Этот процесс аналогичен применению ссылки на базовый класс.

И еще одно замечание: переменной ссылки на интерфейс доступны только методы, объявленные в ее интерфейсе. Поэтому интерфейсную ссылку нельзя использовать для доступа к любым другим переменным и методам, которые не поддерживаются объектом класса, реализующего данный интерфейс.

Аналогично методам, свойства указываются в интерфейсе вообще без тела. Ниже приведена общая форма объявления интерфейсного свойства.

```
// Интерфейсное свойство
тип имя{
get;
set;
```

```
}
```

Очевидно, что в определении интерфейсных свойств, доступных только для чтения или только для записи, должен присутствовать единственный аксессор: get или set соответственно.

Несмотря на то что объявление свойства в интерфейсе очень похоже на объявление автоматически реализуемого свойства в классе, между ними все же имеется отличие. При объявлении в интерфейсе свойство не становится автоматически реализуемым. В этом случае указывается только имя и тип свойства, а его реализация предоставляется каждому реализующему классу. Кроме того, при объявлении свойства в интерфейсе не разрешается указывать модификаторы доступа для аксессоров. Например, аксессор set не может быть указан в интерфейсе как private.

В интерфейсе можно также указывать индексаторы. Ниже приведена общая форма объявления интерфейсного индексатора.

```
// Интерфейсный индексатор
тип_элемента this[int индекс]{
get;  set;
}
```

Как и прежде, в объявлении интерфейсных индексаторов, доступных только для чтения или только для записи, должен присутствовать единственный аксессор: get или set соответственно.

Один интерфейс может наследовать другой. Синтаксис наследования интерфейсов такой же, как и у классов. Когда в классе реализуется один интерфейс, наследующий другой, в нем должны быть реализованы все члены, определенные в цепочке наследования интерфейсов.

Когда один интерфейс наследует другой, то в производном интерфейсе может быть объявлен член, скрывающий член с аналогичным именем в базовом интерфейсе. Такое сокрытие имен происходит в том случае, если член в производном интерфейсе объявляется таким же образом, как и в базовом интерфейсе. Но если не указать в объявлении члена производного интерфейса

ключевое слово new, то компилятор выдаст соответствующее предупреждающее сообщение.

При реализации члена интерфейса имеется возможность указать его имя полностью вместе с именем самого интерфейса. В этом случае получается явная реализация члена интерфейса, или просто явная реализация. Так, если объявлен интерфейс IMylF

```
interface IMylF {
int MyMeth(int x) ;
}
```

то следующая его реализация считается вполне допустимой:

```
class MyClass : IMylF {
int IMylF.MyMeth(int x) {
return x / 3;
}
}
```

Как видите, при реализации члена MyMeth () интерфейса IMylF указывается его полное имя, включающее в себя имя его интерфейса.

Для явной реализации интерфейсного метода могут быть две причины. Во-первых, когда интерфейсный метод реализуется с указанием его полного имени, то такой метод оказывается доступным не посредством объектов класса, реализующего данный интерфейс, а по интерфейсной ссылке. Следовательно, явная реализация позволяет реализовать интерфейсный метод таким образом, чтобы он не стал открытым членом класса, предоставляющего его реализацию. И во-вторых, в одном классе могут быть реализованы два интерфейса с методами, объявленными с одинаковыми именами и сигнатурами. Но неоднозначность в данном случае устраняется благодаря указанию в именах этих методов их соответствующих интерфейсов.

Одна из самых больших трудностей программирования на C# состоит в правильном выборе между интерфейсом и абстрактным классом в тех случаях, когда требуется описать функциональные возможности, но не реализацию. В подобных случаях рекомендуется придерживаться следующего общего правила: если какое-то понятие можно описать с точки зрения функционального назначения, не уточняя конкретные детали реализации, то следует использовать интерфейс. А если требуются некоторые детали реализации, то данное понятие следует представить абстрактным классом.

Структуры. Классы относятся к ссылочным типам данных. Это означает, что объекты конкретного класса доступны по ссылке, в отличие от значений простых типов, доступных непосредственно. Но иногда прямой доступ к объектам как к значениям простых типов оказывается полезно иметь, например, ради повышения эффективности программы. Ведь каждый доступ к объектам (даже самым мелким) по ссылке связан с дополнительными издержками на расход вычислительных ресурсов и оперативной памяти. Для разрешения подобных затруднений в C# предусмотрена структура, которая подобна классу, но относится к типу значения, а не к ссылочному типу данных.

Структуры объявляются с помощью ключевого слова struct и с точки зрения синтаксиса подобны классам. Ниже приведена общая форма объявления структуры:

```
struct имя : интерфейсы {
// объявления членов
}
```

где имя обозначает конкретное имя структуры.

Одни структуры не могут наследовать другие структуры и классы или служить в качестве базовых для других структур и классов. Структуры, как и все остальные типы данных в C#, наследуют класс object. Тем не менее в структуре можно реализовать один или несколько интерфейсов, которые указываются после имени структуры списком через запятую. Как и у классов, у каждой структуры имеются свои члены: методы, поля, индексаторы, свойства,

операторные методы и события. В структурах допускается также определять конструкторы, но не деструкторы. В то же время для структуры нельзя определить конструктор, используемый по умолчанию, т.е. конструктор без параметров. Дело в том, что конструктор, вызываемый по умолчанию, определяется для всех структур автоматически и не подлежит изменению. Такой конструктор инициализирует поля структуры значениями, задаваемыми по умолчанию.

А поскольку структуры не поддерживают наследование, то их члены нельзя указывать как abstract, virtual или protected.

Объект структуры может быть создан с помощью оператора new таким же образом, как и объект класса, но в этом нет особой необходимости. Ведь когда используется оператор new, то вызывается конструктор, используемый по умолчанию. А когда этот оператор не используется, объект по-прежнему создается, хотя и не инициализируется. В этом случае инициализацию любых членов структуры придется выполнить вручную.

Структура может быть инициализирована с помощью оператора new для вызова конструктора или же путем простого объявления объекта. Так, если используется оператор new, то поля структуры инициализируются конструктором, вызываемым по умолчанию (в этом случае во всех полях устанавливается задаваемое по умолчанию значение), или же конструктором, определяемым пользователем. А если оператор new не используется, то объект структуры не инициализируется, а его поля должны быть установлены вручную перед тем, как пользоваться данным объектом.

Когда одна структура присваивается другой, создается копия ее объекта. В этом заключается одно из главных отличий структуры от класса. Когда ссылка на один класс присваивается ссылке на другой класс, в итоге ссылка в левой части оператора присваивания указывает на тот же самый объект, что и ссылка в правой его части. А когда переменная одной структуры присваивается переменной другой структуры, создается копия объекта структуры из правой части оператора присваивания.

Основным преимуществом структур является то, что они относятся к типам значений, и поэтому ими можно оперировать непосредственно, а не по ссылке. Следовательно, для работы со структурой вообще не требуется переменная ссылочного типа, а это означает в ряде случаев существенную экономию оперативной памяти. Более того, работа со структурой не приводит к ухудшению производительности, столь характерному для обращения к объекту класса. Ведь доступ к структуре осуществляется непосредственно, а к объектам – по ссылке, поскольку классы относятся к данным ссылочного типа. Косвенный характер доступа к объектам подразумевает дополнительные издержки вычислительных ресурсов на каждый такой доступ, тогда как обращение к структурам не влечет за собой подобные издержки. И вообще, если нужно просто сохранить группу связанных вместе данных, не требующих наследования и обращения по ссылке, то с точки зрения производительности для них лучше выбрать структуру.

Перечисления. Перечисление представляет собой множество именованных целочисленных констант. Перечислимый тип данных объявляется с помощью ключевого слова enum. Ниже приведена общая форма объявления перечисления:

enum имя { список_перечисления) ;

где имя — это имя типа перечисления, а список_перечисления — список идентификаторов, разделяемый запятыми.

В приведенном ниже примере объявляется перечисление Apple различных сортов яблок.

enum Apple { Jonathan, GoldenDel, RedDel, Winesap, Cortland, Mcintosh };

Следует особо подчеркнуть, что каждая символически обозначаемая константа в перечислении имеет целое значение. Тем не менее неявные

преобразования перечислимого типа во встроенные целочисленные типы и обратно в C# не определены, а значит, в подобных случаях требуется явное приведение типов. Кроме того, приведение типов требуется при преобразовании двух перечислимых типов. Но поскольку перечисления обозначают целые значения, то их можно, например, использовать для управления оператором выбора switch или же оператором цикла for.

Для каждой последующей символически обозначаемой константы в перечислении задается целое значение, которое на единицу больше, чем у предыдущей константы.

По умолчанию значение первой символически обозначаемой константы в перечислении равно нулю. Следовательно, в приведенном выше примере перечисления Apple константа Jonathan равна нулю, константа GoldenDel — 1, константа RedDel — 2 и т.д.

Доступ к членам перечисления осуществляется по имени их типа, после которого следует оператор-точка. Например, Apple.RedDel

```
using System;
class EnumDemo {
enum Apple { Jonathan, GoldenDel, RedDel, Winesap,
Cortland, Mcintosh };
static void Main() {
string[] color = {
"красный",
"желтый",
"красный",
"красный",
"красный",
"красновато-зеленый"
};
Apple i; // объявить переменную перечислимого типа
```

```
// Использовать переменную i для циклического
// обращения к членам перечисления.
for(i = Apple.Jonathan; i <= Apple.Mcintosh; i++).
Console.WriteLine(i + " имеет значение " + (int)i);
Console.WriteLine ();
// Использовать перечисление для индексирования массива.
for(i = Apple.Jonathan; i <= Apple.Mcintosh; i++)
Console.WriteLine("Цвет сорта " + i + " - " +
color[(int)i]);
}
}
```

Как упоминалось выше, в C# не предусмотрены неявные преобразования перечислимых типов в целочисленные и обратно, поэтому для этой цели требуется явное приведение типов.

И еще одно замечание: все перечисления неявно наследуют от класса System. Enum, который наследует от класса System.ValueType, а тот, в свою очередь, — от класса object.

Значение одной или нескольких символически обозначаемых констант в перечислении можно задать с помощью инициализатора. Для этого достаточно указать после символического обозначения отдельной константы знак равенства и целое значение. Каждой последующей константе присваивается значение, которое на единицу больше значения предыдущей инициализированной константы. Например, в приведенном ниже фрагменте кода константе RedDel присваивается значение 10:

```
enum Apple { Jonathan, GoldenDel, RedDel = 10, Winesap, Cortland, Mcintosh };
```

По умолчанию в качестве базового для перечислений выбирается тип int, тем не менее перечисление может быть создано любого целочисленного типа,

за исключением char. Для того чтобы указать другой тип, кроме int, достаточно поместить этот тип после имени перечисления, отделив его двоеточием. В качестве примера ниже задается тип byte для перечисления Apple:

```
enum Apple : byte{ Jonathan, GoldenDel, RedDel, Winesap,Cortland, Mcintosh };
```

Теперь константа Apple.Winesap, например, имеет количественное значение типа byte.

На первый взгляд перечисления могут показаться любопытным, но не очень нужным элементом C#, но на самом деле это не так. Перечисления очень полезны, когда в программе требуется одна или несколько специальных символически обозначаемых констант. Допустим, что требуется написать программу для управления лентой конвейера на фабрике. Для этой цели можно создать метод Conveyor (), принимающий в качестве параметров следующие команды: "старт", "стоп", "вперед" и "назад". Вместо того чтобы передавать методу Conveyor () целые значения, например, 1 — в качестве команды "старт", 2 — в качестве команды "стоп" и так далее, что чревато ошибками, можно создать перечисление, чтобы присвоить этим значениям содержательные символические обозначения. Ниже приведен пример применения такого подхода.

```
// Имитировать управление лентой конвейера,
using System;
class ConveyorControl {
// Перечислить команды конвейера.
public enum Action { Start, Stop, Forward, Reverse };
public void Conveyor(Action com) {
switch(com) {
case Action.Start: Console.WriteLine("Запустить конвейер."); break;
case Action.Stop: Console.WriteLine("Остановить
        конвейер."); break;
```

```
case Action.Forward: Console.WriteLine("Переместить
        конвейер вперед."); break;
case Action.Reverse: Console.WriteLine("Переместить
        конвейер назад."); break;
}
}
}
class ConveyorDemo {
static void Main() {
ConveyorControl c = new ConveyorControl();
c.Conveyor(ConveyorControl.Action.Start);
c.Conveyor(ConveyorControl.Action.Forward);
c.Conveyor(ConveyorControl.Action.Reverse);
c.Conveyor(ConveyorControl.Action.Stop);
}
}
```

В приведенном выше примере обращает на себя внимание еще одно интересное обстоятельство: перечислимый тип используется для управления оператором switch. Как упоминалось выше, перечисления относятся к целочисленным типам данных, и поэтому их вполне допустимо использовать в операторе switch.

1.4. Вопросы для самоконтроля

1. С каких символов начинается однострочный комментарий в C#?
   а) #
   б) \\
   в) //
   г) &

2. В какие символы заключаются многострочные комментарии?

д) /* и *\

е) */ и /*

ж) /- и -/

з) /* и */

3. sbyte, byte, short, ushort, int, uint, long, ulong, float, double, char все это является ... ?

и) единицами компиляции

к) структурированными типами

л) элементарными типами

м) типы-делегаты

5. Модификатор неуправляемого кода?

н) Static

о) Public

п) New

р) Extern

6. Использование одного имени или идентификатора для метода внутри одной иерархии класса таким образом, чтобы для разных классов этой иерархии этот метод реализовывал различные операции это - ...?

а) Пространство имен

б) Наследование

в) Инкапсуляция

г) Полиморфизм

8. Создание новых классов, которые строятся на базе структур данных и методов уже существующих классов (базовых) - это ...?

а) Полиморфизм

б) Наследование

в) Инкапсуляция

г) Пространство имен

9. Совмещение структур данных с функциями (методами),

манипулирующими этими данными это - ...?

а) Наследование

б) Полиморфизм

в) Инкапсуляция

г) Пространство имен

10. Объект, инкапсулирующий ссылку на метод (аналог указатель указателя на функцию) это - ...?

а) Делегат

б) Полиморфизм

в) Виртуальный класс

г) Абстрактный класс

11. Базовый класс, который не предполагает создания экземпляров - это ...?

а) Абстрактный класс

б) Виртуальный класс

в) Родительский класс

г) Делегат

12. Какой синтаксис используется для указания класса родителя в C#?

а) class ChildClass :: ParentClass

б) class ChildClass : ParentClass

в) class ChildClass = ParentClass

г) class ChildClass == ParentClass

13. Какие типы можно использовать в предложении foreach?

а) Методы;

б) Переменные;

в) Массивы, коллекции;

г) Модификаторы.

14. Один или несколько файлов, содержащий логический набор функциональности (код и другие данные, связанные с кодом) это - ...?

а) Коллекция

б) Подборка

в) Сборка

г) Метод

15. Кому доступны переменные с модификатором protected на уровне класса?

а) Только родительскому классу

б) Любому классу

в) Любому классу-наследнику

г) Никому

16. Что такое assembly? Что такое приватные и совместные сборки?

17. Какие простые типы данныхопределены в C#?

18. Что такое и для чего используется пространство имён?

19. Для чего предназначен оператор цикла for?

20. Что вам известно об операторе цикла while?

21. Массивы. Какие виды массивов вы знаете?

22. Спецификаторы доступа?

23. Опишите методы и способы их определения.

24. Конструкторы: виды, доступ.

25. Деструкторы и их функция.

26. Что вам известно о свойствах?

27. Перегрузка операторов и их назначение.

28. Что представляет собой событие?

29. Способы передачи параметров при вызове метода.

30. Типы передачи параметров в функцию.

Тема 2. Интерфейс пользователя в технологии .Net

2.1. Пользовательский интерфейс в оконных приложениях

Для создания простейшего оконного приложения [2] понадобится минимум два класса из пространства имен System.Windows.Forms:

1. Класс **Application**, предназначенный для управления приложением;
2. Класс **Form**, описывающий форму.

Существует два обязательных требования для создания оконного приложения:

1. Определить класс, порожденный от класса System.Windows.Forms.**Form**, иерархия приведена на рис. 2.1;
2. В методе Main вызвать статический метод Run класса System.Windows.Forms.**Application**, передав в качестве параметра экземпляр класса, порожденного от класса System.Windows.Forms.Form.

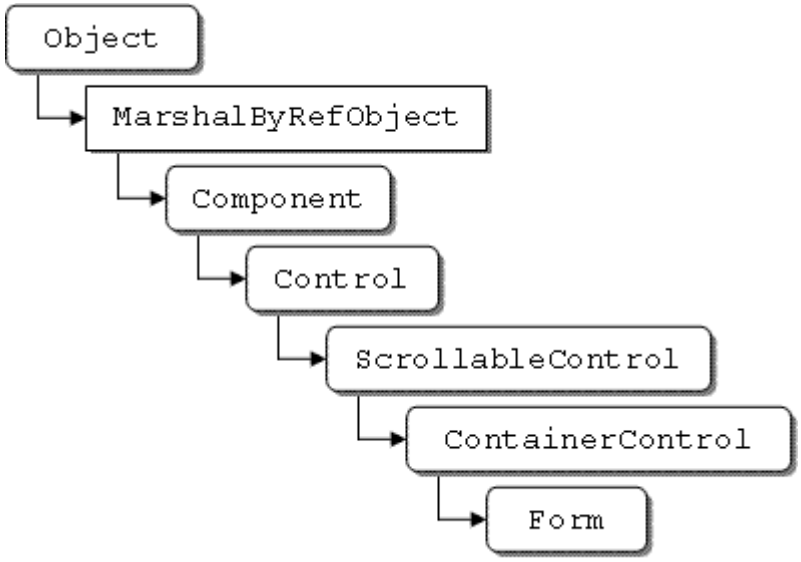

Рис. .2.1 - Иерархия классов System.Windows.Forms.Form

Событием (event) [19] называется некоторое действие, вызванное либо действиями пользователя, например, нажатие клавиши мыши или клавиатуры, либо некоторой программой, например, завершение копирования файла. Работа с событиями в C# соответствует модели "издатель - подписчик", согласно

которой некоторый класс "публикует" (определяет) событие, которое он может инициировать, а другие классы могут "подписаться" на это событие. Класс-подписчик должен реализовать метод, который будет вызван при возникновении события. Такой метод называется обработчиком события (event handler).

Событие является свойством класса, опубликовавшего его. При создании оконного приложения достаточно:

1. Подписаться на событие. Синтаксис:

имя-объекта-издателя.событие **+= new EventHandler(**имя-обработчика**);**

2. Реализовать обработчик события:

private **void** *имя-обработчика***(**object sender, EventArgs e**) {** … **}**

Обработчики события не должны возвращать значение и должны принимать два параметра:

1. Источник события - объект класса System.Object;

2. Объект, содержащий информацию о событии - экземпляр класса EventArgs или любого другого класса, порожденного данным.

Например, класс System.Windows.Forms.Form содержит событие Click, которое инициируется при нажатии кнопки мыши на форме. Чтобы форма реагировала на нажатие кнопки мыши на ней, нужно создать обработчик данного события, для этого необходимо в некотором методе класса формы (определенного пользователем) подписаться на событие и реализовать обработчик события.

Базовым в иерархии классов элементов управления является класс System.Windows.Forms.Control. В классе Control определен вложенный класс ControlCollection. Этот класс представляет коллекцию для хранения списка элементов управления, ассоциированных с данным элементом управления.

Пользовательский интерфейс реализуется с помощью элементов, представленных на рис. 2.1.

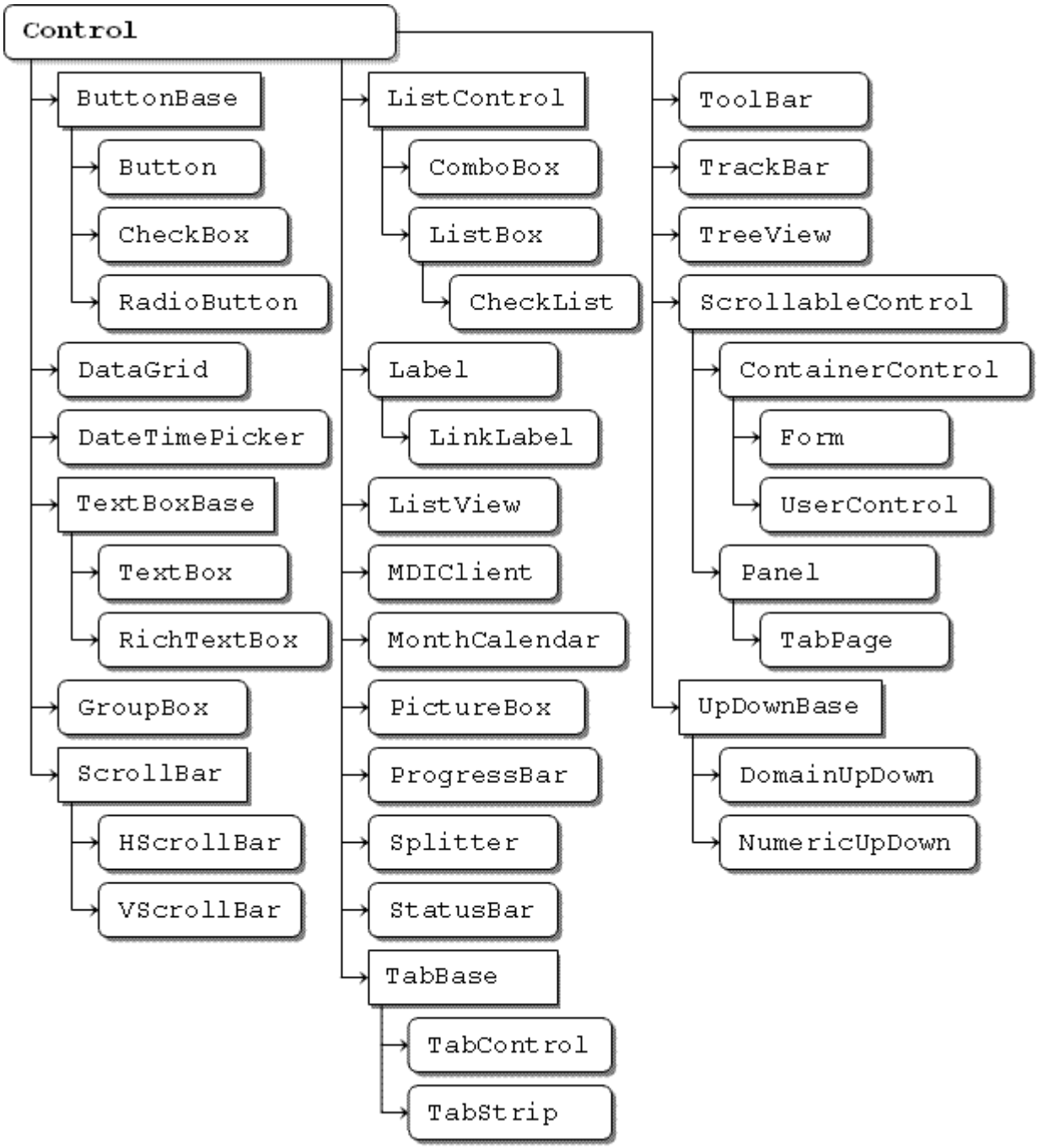

Рис. 2.1 – Диаграмма иерархии наследования элементов управления

Поскольку класс Form [11] является потомком класса Control, то он содержит поле класса ControlCollection. Итак, для того чтобы добавить элемент управления на форму, необходимо:

1. создать элемент управления;
2. добавить созданный элемент управления в коллекцию ControlCollection. Если этого не сделать, элемент управления на форме отображен не будет.

Пример:

```
public class ИмяForm : Form
{  …
  private ТипЭлементаУправления ctrl = new ТипЭлементаУправления ();
   public MainForm()
   {
     Controls.Add(ctrl);  // Добавление в коллекцию
   }
}
```

В рассмотренном примере создан экземпляр класса Control. Если заменить его на конкретный элемент управления, то этот элемент будет отображен на форме.

Пример создания интерфейса. Рассмотрим приложение для интерполяции функциии различными методами. Приложение представлено в виде оконного приложения, которое является отдельным потоком в операционной системе. Использована интегрированная среда Visual Studio.Net.

Форма представляет собой экранный объект, обычно прямоугольной формы, который можно применять для представления информации пользователю и для обработки ввода информации от пользователя. Формы могут иметь вид стандартного диалогового окна, многодокументного интерфейса (MDI) или поверхности для отображения графической информации. Самый простой способ задать интерфейс пользователя для формы – разместить элементы управления на её поверхности.

К основным элементам дизайнера форм относятся:

- Properties Window (пункт меню View /Properties Window);
- Layout Toolbar (пункт меню View /Toolbars/Layout);
- Toolbox (пункт меню View /Toolbox).

Интерфейс приложения реализован с помощью различных элементов.

Для выбора функции создан выпадающий список, который позволяет выбрать функцию или же задать её вручную (рис.2.3).

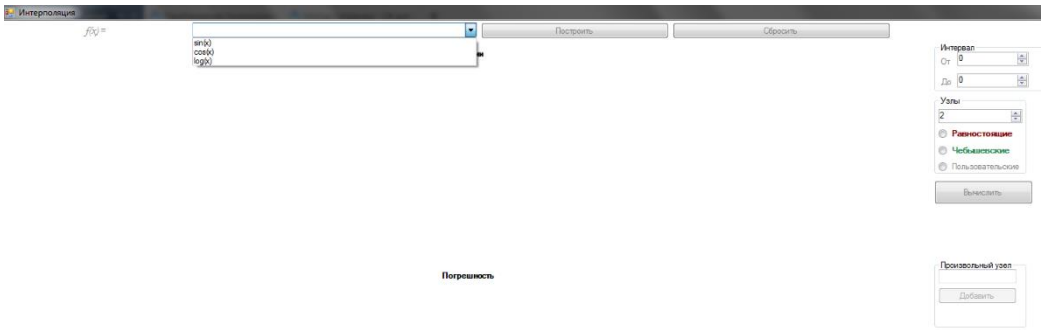

Рис.2.3 – Элемент выбора функции

Для дальнейшего построения графика следует выбрать отрезок интерполирования. В программе присутствует кнопка «Построить», при нажатии на неё выполняется построение графика (рис.2.4).

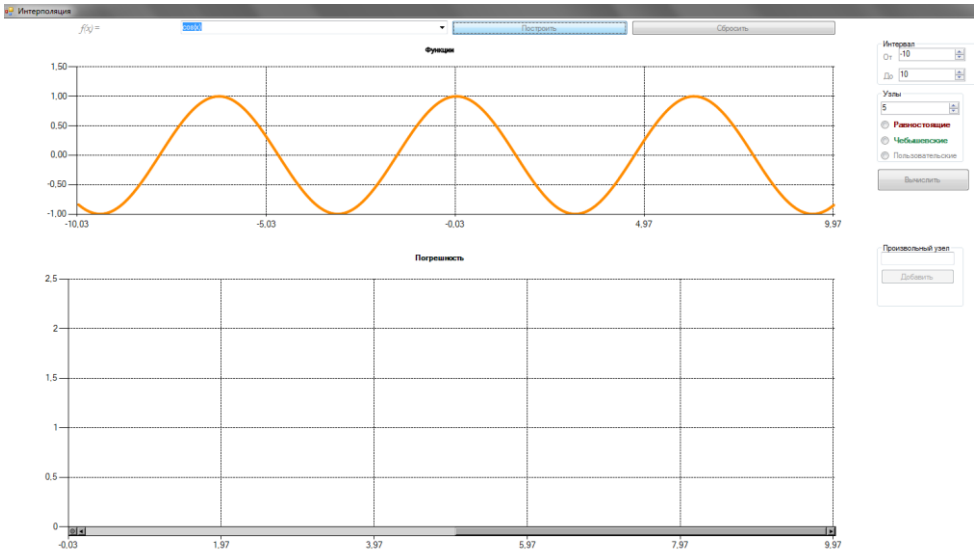

Рис.2.4 – Выбор отрезка интерполирования

Если не задать отрезок интерполирования, то в противном случае программа выдаст ошибку.

Кнопка «Построить» проводит построение графика исходной функции в элементе управления Chart.

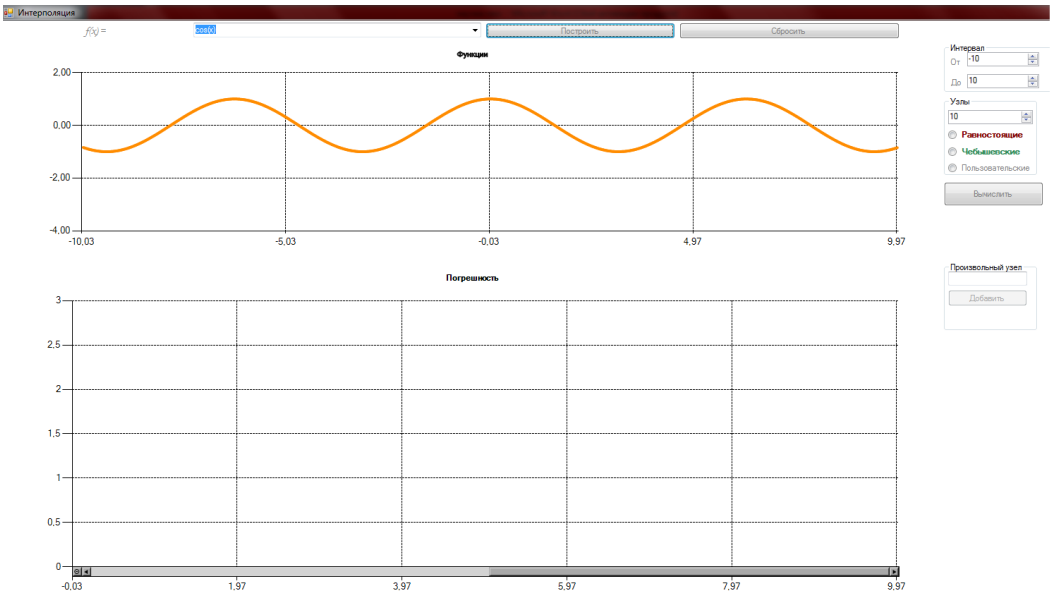

Рис.2.5 – График функции

Элемент Chart позволяет визуализировать график функции.

Дальнейшая работа приложения требует добавления элементов радио-кнопок, которые позволяют выбрать тип узлов. Выбираем метод Лагранжа с равностоящими узлами, в программе необходимо указывать количество узлов. Выполним график следуя вышесказанному и наблюдаем график функции. Происходит вызов функции Лагранжа с равностоящими узлами. Таким образом, наблюдаем точное и приближенное значение функции (рис.2.6). Соответственно внизу видим точное и приближенное значение функции.

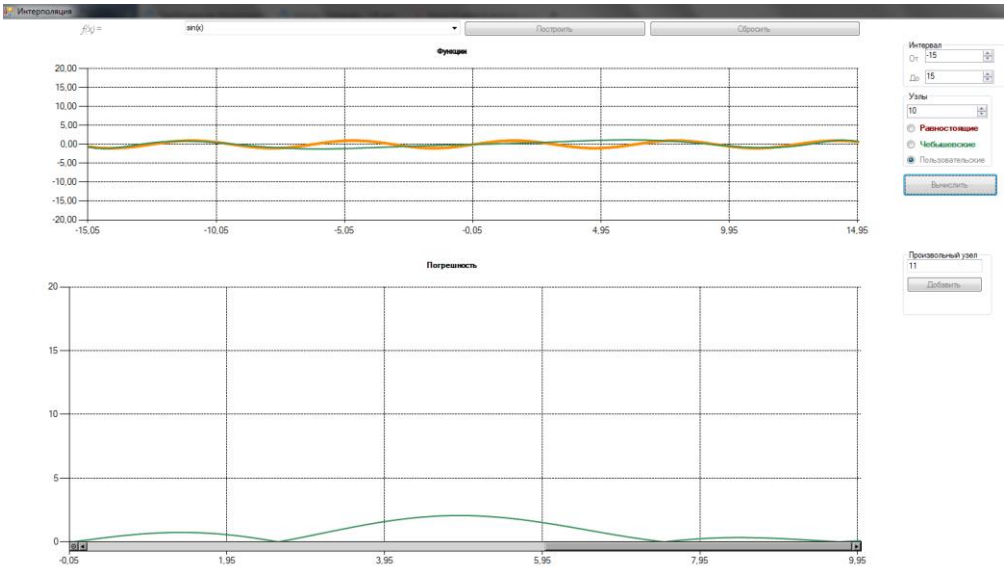

Рис.2.6 – Визуализация точного и приближенного значения функции погрешности метода

Для демонстрации погрешности используется элемент управления Chart, который зеленым цветом отображает разность точного и приближенного значения в точках, которые известны.

Аналогично приложение работает для остальных вариантов узлов и методов интерполирования.

Пример работы программы приведен на рис.2.7.

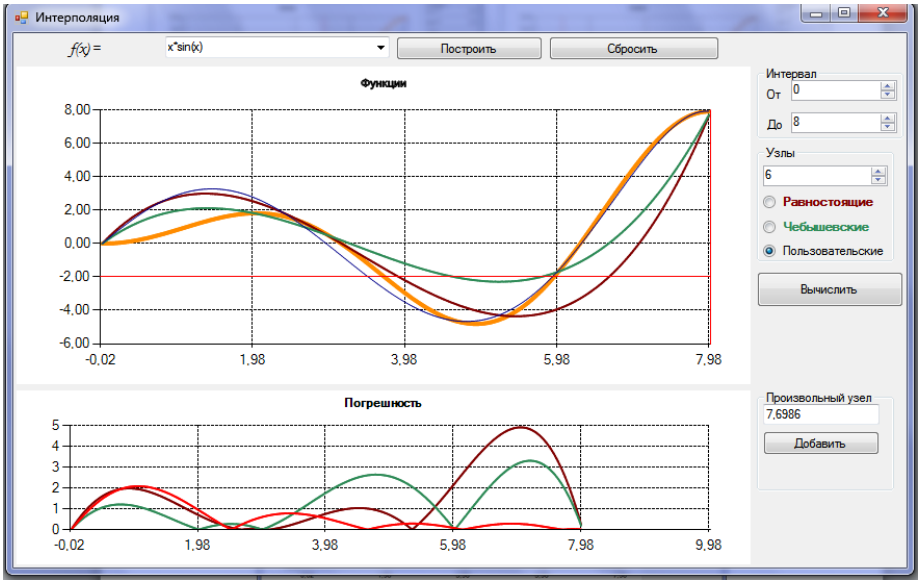

Рис.2.7 – Изображение всех вариантов интерполирования одновременно для сравнения

2.2. Пользовательский нтерфейс веб-приложения

Страницы ASP.NET называются веб-формами. являются важной частью приложения ASP.NET [19]. Веб-формы позволяют создавать веб-приложения с интерфейсом, основанным на элементах управления. При запуске веб-формы ASP.NET считывает весь файл с расширением .aspx, генерирует соответствующие объекты и инициирует ряд событий, а разработчик реагирует на эти события с помощью полностью объектно-ориентированного кода [6].

Модель веб-форм ASP.NET [7]:

• визуализация согласно стандарту XHTML Strict. Страницы веб-форм теперь всегда будут на 100% совместимыми с XHTML (если только не нарушать правила XHTML самостоятельно);

• назначение предсказуемых клиентских идентификаторов. Чтобы каждый элемент управления получал уникальный идентификатор в визуализируемом HTML-коде, в ASP.NET используется система генерации длинных имен. Это приводит к усложнению при использовании таких идентификаторов в клиентском сценарии JavaScript;

• Постоянная переадресация. С целью улучшения поисковой оптимизации в ASP.NET теперь можно перенаправлять запросы с кодом состояния HTTP 301, который обозначает постоянную переадресацию. Когда роботы поисковых систем получают это сообщение, они обновляют свои каталоги.

Так как веб-приложения выполняются на сервере, то обработка страниц выполняется методом обратной отправки, которая посылает на сервер страницу (и всю предоставленную пользователем информацию) при выполнении определенных действий пользователем. Когда система ASP.NET получает страницу, она генерирует соответствующие серверные события для уведомления вашего кода.

Веб-приложения не поддерживают состояния. После визуализации страницы в виде HTML объекты веб-страниц уничтожаются, и вся касающаяся клиента информация отбрасывается. Эта модель хорошо подходит для масштабируемых приложений с интенсивным трафиком, но усложняет настройку на пользовательские предпочтения. В ASP.NET доступно несколько инструментов, помогающих заполнить данный пробел; наиболее заметным является механизм сохранения под названием состояние представления (view state), автоматически встраивающий информацию о странице в скрытое поле сгенерированной HTML-формы.

В ASP.NET все элементы управления помещены в отдельный дескриптор <form>. Этот дескриптор помечен атрибутом runat="server", который позволяет ему работать на сервере. ASP.NET не допускает создание веб-форм,

содержащих более одного серверного дескриптора <f orm>, хотя можно создавать страницы, выполняющие отправку информации другим страницам, с использованием технологии межстраничной отправки.

ASP.NET облегчает ситуацию, предлагая управляемую событиями модель обработки (event-driven model) [16]. В этой модели разработчик сначала добавляет в веб-форму элементы управления, а затем решает, на какие события он хочет реагировать. Каждый обработчик событий представляет собой отдельный метод, благодаря чему код страниц выглядит аккуратно и организованно. Эта модель не является новой, но до появления ASP.NET она применялась исключительно в области программирования оконных пользовательских интерфейсов для многофункциональных клиентских приложений.

Обработка событий в ASP.NET происходит следующим образом [20]:

1. при первом запуске страницы ASP.NET создает объекты этой страницы и ее элементов управления. Далее выполняется код инициализации, после чего страница преобразуется в HTML и возвращается клиенту, а созданные объекты удаляются из памяти сервера;

2. на каком-то этапе пользователь выполняет действие, инициирующее обратную отправку данных, например, щелкает на кнопке. Тогда страница отправляется серверу вместе со всеми данными формы;

3. ASP.NET перехватывает эту возвращаемую страницу и снова воссоздает ее объекты, возвращая их в то состояние, в котором они находились тогда, когда эта страница в последний раз отправлялась клиенту;

4. далее ASP.NET проверяет, какая именно операция привела к обратной отправке данных, и генерирует соответствующие события (например, Button .Click), на которые разработчик может предусмотреть в своем коде определенную реакцию. На этом этапе чаще всего выполняются серверные операции (вроде обновления базы данных или чтения данных из файла), а затем изменять объекты элементов управления так, чтобы они отображали уже новую информацию;

5. измененная страница преобразуется в HTML и возвращается клиенту. Объекты страницы удаляются из памяти. В случае если происходит еще одна обратная отправка данных, ASP.NET повторяет действия, перечисленные в пунктах 2-4.

Обработка быстро генерируемых событий (например, движение мыши) может происходить на стороне клиента и на стороне сервера. Это означает, что ответ на событие всегда влечет за собой определенные накладные расходы. Поэтому быстро генерируемые события в мире ASP.NET являются совершенно непрактичными. Чтобы добиться в пользовательском интерфейсе определенного эффекта, можно создать клиентский сценарий JavaScript. Или, что даже лучше, можно воспользоваться специальным элементом управления ASP.NET со встроенными возможностями подобного рода, например, одним из элементов ASP.NET AJAX. Однако код бизнес-логики должен обязательно выполняться только в безопасной многофункциональной среде сервера.

Чтобы использовать автоматическую обратную отправку, понадобится установить свойство AutoPostBack веб-элемента управления в true (по умолчанию это свойство устанавливается в false, что гарантирует оптимальную производительность в случаях, когда не требуется реагировать ни на какие события изменения). После этого ASP.NET использует клиентские возможности JavaScript для устранения пробела между клиентским и серверным кодом.

В частности, происходит следующее: в случае создания веб-страницы с одним или более веб-элементами управления, для которых настраивается AutoPostBack, ASP.NET добавляет в визуализируемую HTML-страницу JavaScript-функцию по имени doPostBack (). При вызове эта функция инициирует обратную отправку, отправляя страницу обратно веб-серверу со всеми данными формы.

Помимо этого, ASP.NET также добавляет два скрытых поля ввода, которые функция doPostBack () использует для передачи обратно серверу определенной информации.

Этой информацией является идентификатор элемента управления, который инициировал событие, и другие значимые сведения. Первоначально данные поля пусты, как показано ниже:

```
<div class="aspNetHidden">
<input type="hidden" name=" EVENTTARGET"
    id=" EVENTTARGET" value="" />
<input type=,,hidden" name=" EVENTARGUMENT"
    id=" EVENTARGUMENT" value=,,n />
</div>
```

Функция doPostBack () отвечает за установку этих значений в соответствующую информацию о событии и последующую отправку формы.

Не забывайте, что ASP.NET генерирует функцию doPostBack () автоматически. Наконец, любой элемент управления, свойство AutoPostBack которого установлено в true, подключается к функции doPostBack () с использованием атрибута onclick или onchange. Эти атрибуты указывают на то, какое действие следует выполнить браузеру в ответ на клиентские JavaScript-события onclick и onchange.

Все крупномасштабные веб-сайты ASP.NET создаются с помощью Visual Studio. В состав этого профессионального средства для разработки входит развитый набор инструментов для проектирования, в том числе инструменты для отладки и механизм IntelliSense, способный перехватывать ошибки и предлагать варианты по мере ввода. Кроме того, в Visual Studio поддерживается мощная модель отделенного кода, которая позволяет разделять создаваемый код .NET и дескрипторы разметки веб-страницы. И, наконец, в Visual Studio имеет встроенный тестовый веб-сервер, который значительно упрощает процесс отладки веб-сайтов.

В Visual Studio поддерживаются два пути создания веб-приложений на базе ASP.NET:

• разработка на основе проекта. При создании веб-проекта в Visual Studio генерируется файл проекта с расширением .csproj (при условии, что код

пишется на языке C#), в котором фиксируется информация обо всех включаемых в состав проекта файлах и сохраняются кое-какие отладочные параметры. При запуске веб-проекта перед открытием веб-браузера Visual Studio сначала компилирует весь написанный код в одну сборку;

• разработка без использования проекта. Это альтернативный подход, который подразумевает создание просто веб-сайта безо всякого файла проекта. При таком подходе Visual Studio предполагает, что каждый файл в каталоге веб-сайта (и всех его подкаталогах) является частью веб-приложения. В этом случае Visual Studio не требуется предварительно компилировать код. Вместо этого ASP.NET компилирует уже сам веб-сайт при первом запросе какой-нибудь входящей в его состав страницы.

Изменить целевую версию .NET можно после создания веб-сайта. Для этого выполните следующие шаги:

1. выберите в меню Website (Веб-сайт) пункт Start Options (Параметры запуска);

2. в списке слева выберите категорию Build (Сборка);

3. в списке Target Framework (Целевая платформа) выберите желаемую версию .NET.

При изменении версии .NET среда Visual Studio существенно модифицирует соответствующий файл web.config.

Место размещения отвечает за то, где будут храниться файлы веб-сайта. Обычно выбирается вариант File System (Файловая система) и затем указывается либо папка на локальном компьютере, либо сетевой путь. Однако веб-сайт также допускается редактировать и непосредственно через HTTP или FTP (File Transfer Protocol — протокол передачи файлов). Такой подход иногда удобен, когда требуется "вживую" выполнять редактирование веб-сайта на удаленном веб-сервере. С другой стороны, он влечет за собой дополнительные накладные расходы. Конечно, производить редактирование напрямую на производственном сервере не следует никогда, поскольку такие изменения

являются автоматическими и необратимыми. Вместо этого лучше ограничивать свои изменения только тестовыми серверами.

В левой части этого окна есть четыре кнопки, позволяющие выбирать различные варианты для размещения файлов:

• File System (Файловая система). Это самый простой вариант, поскольку он подразумевает просто просмотр дерева дисков и каталогов или общих ресурсов, отображаемых другими компьютерами в сети, и выбор походящего каталога. При желании создать новый каталог нужно всего лишь щелкнуть на значке Create New Folder (Создать новую папку), который отображается в правом верхнем углу дерева каталогов. (Заставить Visual Studio создать новый каталог также можно и путем добавления имени нового каталога в конце пути.);

• Local IIS (Локальный IIS). Этот вариант позволяет просматривать виртуальные каталоги, которые делает доступными предоставляющее веб-хостинг программное обеспечение IIS, при условии, конечно, что таковое установлено на данном компьютере;

• FTP Site (FTP-сайт). Этот вариант является не таким удобным, как поиск нужного каталога, поскольку предполагает ввод всей информации, которая необходима для установки соединения: имени FTP-сайта, номера порта, названия каталога, имени пользователя и пароля;

• Remote Site (Удаленный веб-сайт). Этот вариант позволяет получать доступ к определенному веб-сайту с определенным URL-адресом по протоколу HTTP. Чтобы он работал, на веб-сервере, к которому требуется получить доступ, должен быть установлен компонент FrontPage Extensions. Вдобавок, при подключении потребуется ввести имя пользователя и пароль.

Visual Studio предоставляет три режима для просмотра веб-страницы: Source (Исходный код), Design (Конструктор) и Split (Комбинированный). Выбрать желаемый режим можно, просто щелкнув на одной из трех имеющих соответствующие названия кнопок в нижней части окна с веб-страницей. В режиме Source отображается разметка страницы (т.е. дескрипторы элементов

управления HTML и ASP.NET), в режиме Design — отформатированное изображение того, как страница будет выглядеть в окне веб-браузера, а в режиме Split — комбинированное представление, позволяющее просматривать одновременно и разметку страницы, и ее окончательный внешний вид.

Добавить элемент управления ASRNET на страницу легче всего, перетащив его из находящейся слева панели Toolbox (Панель инструментов).

Перетащить элемент управления можно как в область видимой структуры страницы (в режиме визуального конструктора), так и в определенную позицию в ее разметке (в режиме исходного кода). И в том и в другом случае результат будет одинаковым. Существует также и альтернативный вариант: вручную ввести дескриптор требуемого элемента управления в режиме Source. В таком случае представление, отображаемое в режиме Design, не обновляется до тех пор, пока не будет либо выполнен щелчок в области проектирования окна, либо нажата комбинация клавиш <Ctrl+S> для сохранения веб-страницы.

Добавив элемент управления, можно изменить его размер или сконфигурировать его свойства в окне Properties (Свойства). Многие разработчики предпочитают компоновать новые веб-страницы в режиме Design, но выполнять перегруппировку элементов управления или более детальную настройку — в режиме Source. Исключением являются лишь случаи, когда речь идет об HTML-разметке: хотя в панели Toolbox и имеется вкладка с HTML-элементами, обычно легче ввести дескрипторы необходимых элементов управления вручную, чем перетаскивать их на страницу по одному за раз.

Чтобы сконфигурировать элемент управления, сначала нужно щелкнуть на нем для его выбора или же выбрать его по имени в раскрывающемся списке в верхней части окна Properties (Свойства), а затем изменить все необходимые свойства, например, Text (Текст), ID (Идентификатор) HForeColor (Цвет переднего плана). Эти установки автоматически преобразуются в соответствующие атрибуты дескрипторов элементов управления ASP.NET и определяют исходный внешний вид этих элементов. Visual Studio даже предоставляет специальные "селекторы" (известные как UITypeEditor),

позволяющие выбирать расширенные свойства. Например, можно выбрать из раскрывающегося списка цвет или же сконфигурировать шрифт с помощью стандартного диалогового окна выбора шрифта.

Позиционирование. Для позиционирования элемента управления на странице необходимо использовать все обычные приемы HTML-дизайна, такие как абзацы, разрывы строк, таблицы и стили. По умолчанию Visual Studio предполагает, что элементы должны размещаться с использованием гибкого режима потокового (flow) позиционирования, который позволяет содержимому увеличиваться и уменьшаться динамически без возникновения проблем с компоновкой. Однако у разработчика также есть возможность применить и режим абсолютного (absolute) позиционирования с помощью стандарта CSS. Все, что от него требуется — это добавить для своего элемента управления внутристрочный CSS-стиль с параметрами абсолютного позиционирования. Например, ниже показан CSS-стиль, указывающий, что кнопка должна размещаться в точности в 100 пикселях от левого края и 50 пикселях от верхнего края страницы:

<asp:Button id="cmd" style="P0SITI0N: absolute; left: l00px; top: 50px;" runat="server" ... />

После внесения такого изменения кнопка может перетаскиваться по окну в любое место, и Visual Studio будет автоматически обновлять указанные в стиле координаты соответствующим образом.

В размещении отдельных элементов управления с использованием режима абсолютного позиционирования редко есть смысл. Такой режим не позволяет схеме компоновки адаптироваться под разные размеры окон браузеров и заканчивается появлением проблем при разворачивании содержимого в одном из элементов, приводя к перекрытию другого имеющего абсолютную позицию элемента. Он также способствует созданию негибких схем компоновки, которые впоследствии очень сложно изменять. Однако режим абсолютного позиционирования, тем не менее, прекрасно подходит для размещения целых контейнеров, внутри которых затем спокойно может

использоваться потоковое содержимое. Например, можно применять режим абсолютного позиционирования для закрепления строки меню в какой-то определенной стороне окна, а для списка ссылок внутри нее — режим обычной потоковой компоновки. Для такого случая прекрасно подходит контейнер , поскольку он не имеет никаких встроенных параметров внешнего вида (хотя с помощью правил стилей для него все-таки может устанавливаться цвет переднего плана, цвет фона и т.д.). Контейнер , по сути, представляет собой "плавающее" поле. В приведенном ниже примере, он имеет фиксированную ширину, которая составляет 200 пикселей, и нефиксированную высоту, которая будет изменяться в соответствии с размером отображаемого внутри него содержимого:

Интеллектуальные дескрипторы (смарт-теги) упрощают конфигурирование сложных элементов управления. Они предлагаются не для всех элементов управления, а только для многофункциональных, таких как GridView, TreeView и Calendar.

О доступности смарт-тега свидетельствует наличие в верхнем правом углу при выборе элемента управления небольшой стрелки. Если щелкнуть на этой стрелке, появится окно со ссылками (и другими элементами управления), позволяющими решать задачи более высокого уровня.

Страницы ASP.NET содержат элементы управления ASP.NET вперемешку с обычными дескрипторами HTML. Добавляются HTML-дескрипторы либо путем ввода, либо перетаскиванием из соответствующей вкладки в панели Toolbox.

В состав Visual Studio входит очень удобный конструктор стилей, который позволяет форматировать любой статический элемент HTML с помощью свойств CSS. Чтобы испробовать его, перетащите на страницу из раздела HTML в панели Toolbox элемент . Этот элемент появится на странице в виде не имеющей границ панели. Далее щелкните на этой панели, чтобы выделить ее, и затем щелкнуть внутри поля Style в окне Properties

(Свойства). После этого в поле Style появится кнопка с изображением троеточия (...). Щелчок на ней приводит к открытию диалогового окна Modify Style (Изменение стиля) с опциями для настройки цветов, шрифта, компоновки и рамки элемента.

Созданный подобным образом новый стиль будет сохраняться как внутристрочный и записываться в отвечающий за стиль атрибут изменяемого элемента. В качестве альтернативы можно определить именованный стиль в текущей странице или в отдельной таблице стилей. При желании сконфигурировать элемент HTML как серверный элемент управления для того, чтобы иметь возможность обрабатывать события и взаимодействовать с ним в коде, нужно переключиться в режим Source и добавить в дескриптор элемента управления требуемый для этого атрибут runat="server".

Окно Solution Explorer на самом базовом уровне представляет собой визуальную файловую систему. Оно позволяет просматривать файлы, которые находятся в каталоге веб-приложения. В каталоге веб-приложения могут находиться файлы изображений, файлы HTML или файлы CSS. Эти ресурсы могут использоваться как на одной из веб-страниц ASP.NET, так и сами по себе.

Visual Studio различает файлы разных типов. При щелчке правой кнопкой мыши на каком-то из представленных в списке файлов появляется контекстное меню с опциями, которые подходят для файлов именно этого типа. Например, в случае выполнения щелчка правой кнопкой мыши на веб-странице, это будут опции, позволяющие создать веб-страницу и запустить ее в окне браузера. Используя окно Solution Explorer, можно переименовывать, переупорядочивать и добавлять файлы. Все эти опции доступны через единственный щелчок правой кнопкой мыши. Чтобы удалить файл, нужно выделить его в окне Solution Explorer и затем нажать клавишу <Delete>.

Помимо этого Visual Studio также отслеживает события, связанные с управлением проектом, наподобие внесения изменений в открытый на текущий

момент файл проекта другим процессом. Когда такое происходит, Visual Studio тут же отображает соответствующее уведомление и предлагает обновить файл.

Окно документа является частью Visual Studio, которая позволяет редактировать различные типы файлов с помощью разных визуальных конструкторов. Для каждого типа файла предусмотрен свой редактор по умолчанию. Узнать, какой редактор является редактором по умолчанию для того или иного файла, можно, щелкнув правой кнопкой мыши на этом файле в окне Solution Explorer и выбрав в контекстном меню пункт Open With (Открыть с помощью). Рядом с редактором по умолчанию будет отображаться слово Default (По умолчанию).

Окно Toolbox работает вместе с окном документа. Его первоначальная функция заключается в предоставлении элементов управления, которые можно перетаскивать в окно проектирования веб-формы. Однако оно также позволяет сохранять фрагменты HTML и другого кода. Содержимое окна Toolbox зависит от используемого в данный момент конструктора, а также от типа проекта. Например, при проектировании веб-страницы в нем отображаются вкладки, перечисленные в табл. 2.3. На каждой из этих вкладок предлагается свой набор кнопок. Чтобы просмотреть ту или иную вкладку, нужно щелкнуть на ее заголовке, после чего сразу станут видны все доступные на ней кнопки. Как вкладки, так и доступные на каждой из них элементы, можно настраивать. Чтобы внести какие-то изменения в набор предлагаемых вкладок, нужно щелкнуть правой кнопкой мыши на заголовке вкладки и выбрать в контекстном меню пункт Rename Tab (Переименовать вкладку), Add Tab (Добавить вкладку) или Delete Tab (Удалить вкладку).

Чтобы добавить на вкладку еще какой-нибудь элемент, нужно щелкнуть правой кнопкой мыши где-то на пустом месте в окне Toolbox и затем выбрать в контекстном меню пункт Choose Items (Выбрать элементы). Кроме того, можно также просто перетаскивать элементы из одной вкладки в другую.

Иногда может потребоваться функциональная возможность в элементе управления, не предоставляемая встроенными серверными веб-элементами

управления ASP.NET.В таких случаях можно создавать собственные элементы управления.Имеются две возможности.

Пользовательские элементы управления — это контейнеры, в которые можно поместить разметку и серверные веб-элементы управления. Пользовательский элемент управления затем можно рассматривать как единое целое и задавать для него свойства и методы.

Настраиваемые элементы управления представляют собой классы с кодом, добавляемым и изменяемым программистом, производные от класса Control или WebControl.

Создавать пользовательские элементы управления значительно проще, так как можно повторно использовать уже существующие элементы управления. Благодаря им очень легко создавать элементы управления со сложными элементами пользовательского интерфейса.

Создание пользовательского элемента управления во многом напоминает создание сначала собственно страницы ASP.NET с последующим добавлением к ней необходимой разметки и необходимых дочерних элементов управления. Пользовательский элемент управления, как и страница, может включать в себя код для управления его же содержимым, включая выполнение задач, например привязку данных.

Пользовательские элементы управления отличаются от веб-страницы ASP.NET:

Расширением имени файла; пользовательский элемента управления имеет расширение ASCX.

Вместо директивы @ Page пользовательский элемент управления содержит директиву @ Control, определяющую конфигурацию и другие свойства.

Пользовательские элементы управления не могут выполняться как автономные файлы.Вместо этого, необходимо добавить их к другим страницам ASP.NET, как в случае с любым элементом управления.

В пользовательском элементе управления можно использовать те же элементы HTML (кроме элементов html, body или form) и веб-элементы управления, что и на веб-странице ASP.NET.Например, при создании пользовательского элемента управления, который используется как панель инструментов, можно поместить в этот элемент управления набор серверных веб-элементов управления Button и задать обработчики событий для кнопок.

Пользовательский элемент управления добавляется на страницу путем регистрации его на странице.При регистрации пользовательского элемента управления необходимо указать ASCX-файл, содержащий его, префикс тега и имя тега, который будет использован для объявления пользовательского элемента управления на странице.

В содержащей его веб-странице ASP.NET создайте директиву @ Register, включающую:

TagPrefix - атрибут, связывающий префикс с пользовательским элементом управления.Префикс включается в открывающий тег элемента пользовательского элемента управления;

TagName - атрибут, связывающий имя с пользовательским элементом управления.Имя включается в открывающий тег элемента пользовательского элемента управления;

Src - атрибут, определяющий виртуальный путь к файлу пользовательского элемента управления, включаемого в страницу. Значением атрибута Src может быть относительный или абсолютный путь к исходному файлу пользовательского элемента управления, задаваемый относительно корневого каталога приложения. Для обеспечения гибкости рекомендуется использовать относительный путь.Тильда (~) представляет корневой каталог приложения.Пользовательские элементы управления не могут располагаться в каталоге App_Code.

Если пользовательский элемент управления содержит серверные веб-элементы управления, то можно написать код в пользовательском элементе управления для обработки событий, вызванных дочерними элементами

управления.Например, если пользовательский элемент управления содержит элемент управления Button, то можно создать обработчик событий для события Click кнопки.

События, вызванные дочерними элементами управления в пользовательском элементе управления, по умолчанию недоступны странице, на которой размещен данный элемент управления.Однако можно определить события для пользовательского элемента управления и вызвать их так, что страница, на которой размещен элемент управления, будет уведомлена о событии.Это делается так же, как определение событий для любого класса.

При выполнении пользовательского элемента управления ссылки на внешние ресурсы, такие как изображения или точки привязки на другие страницы разрешаются с помощью URL-адреса пользовательского элемента управления, который используется как базовый URL-адрес. Например, если пользовательский элемент управления содержит элемент управления Image, свойство ImageUrl которого имеет значение Images/Button1.gif, то URL-адрес изображения добавляется к URL-адресу пользовательского элемента управления для разрешения полного пути к изображению.Если пользовательский элемент управления ссылается на ресурс, находящийся за пределами вложенной папки самого пользовательского элемента управления, то необходимо предоставить путь, разрешающийся в правильную папку во время выполнения.

Для добавления пользовательского элемента в обозревателе решений выберите файл пользовательского элемента управления и перетащите его на страницу. На страницу будет добавлен пользовательский элемент управления ASP.NET.Кроме того, конструктор создает директиву @ Register, которая требуется для распознавания страницей пользовательского элемента управления. Теперь можно приступать к работе с открытыми свойствами и методами элемента управления. В основной части веб-страницы объявите элемент пользовательского элемента управления внутри элемента form.

При необходимости, если пользовательский элемент предоставляет доступ к открытым свойствам, задайте свойства декларативно.

Серверные элементы, кроме этого, реализуют собственное поведение и самостоятельно выводят HTML-код, который их отображает. Они могут наследоваться от WebControl или одного из классов стандартных элементов управления. Их можно использовать в любых проектах и распространять в виде откомпилированной PE (Portable Executable) сборки.

Класс Page наследует класс Control, как и все элементы управления, некоторые прямо, а другие через класс WebControl или HtmlControl. Следовательно, между написанием страницы и разработкой собственного элемента управления должно быть много общего. У них тоже есть свой жизненный цикл. В классе Control определены события Init, Load, DataBinding, PreRender, Unload, Disposed. Свойства, которые Control предоставляет своим наследникам, включают EnableViewState, ID, UniqueID, Page, Parent, SkinID, ViewState и Controls — коллекция дочерних элементов управления.

Класс Control предоставляет возможность помещать элемент управления в дерево элементов управления, которые отображаются на странице .aspx. Класс Control также реализует интерфейс System.ComponentModel. IComponent, который делает компонент конструктивным. Конструктивный компонент может быть добавлен в панель Toolboox визуального дизайнера, может быть помещен на разрабатываемую страницу методом drag-and-drop, может отображать свойства в окне свойств и обеспечивать другие виды поддержки режима разработки (в том числе Smart Tags).

Пользовательские элементы управления можно создавать в визуальном редакторе по той же модели, что и страницы aspx. Как всегда, откроем диалог NewFile и выберем тип страницы Web User Control. Расширение файла с дизайном элемента — ascx, а с кодом класса — ascx.cs. В отличие от страниц aspx, сам по себе пользовательский элемент нельзя увидеть в браузере, для этого он должен находиться на какой-нибудь странице:

<%@ Control Language="C#" AutoEventWireup="true"

```
CodeFile="C1.ascx.cs" Inherits="WebUserControl" %>
```

Класс пользовательского элемента управления - наследник System.Web.UI.UserControl. В остальном он ничем не отличается от файла с классом страницы:

```
public partial class WebUserControl : System.Web.UI.UserControl
{
}
```

Можно добавить в него любые элементы управления и HTML-код:

```
<%@ Control Language="C#" AutoEventWireup="true"
    CodeFile="C1.ascx.cs" Inherits="WebUserControl" %>
<h1><%= Field %></h1>
<asp:TextBox ID="txtField" runat="server"></asp:TextBox><br />
<asp:Button ID="bt1" runat="server"
    Text="Button" OnClick="bt1_Click" />
```

В классе элемента управления определим его свойства:

```
public partial class WebUserControl : System.Web.UI.UserControl
{
   string field;
   public string Field
   {
      get
      {
         return field;
      }
      set
      {
         field = value;
      }
   }
```

```
    protected void Page_Init(object sender, EventArgs e)
    {
        bt1.Text = "Ввдите текст и нажмите кнопку";
    }
    protected void Page_Load(object sender, EventArgs e)
    {
    }
    protected void bt1_Click(object sender, EventArgs e)
    {
        Field = txtField.Text;
    }
}
```

Теперь перетащите название элемента из Solution Explorer на любую страницу.

Чтобы использовать пользовательский элемент на странице, его надо зарегистрировать. Директива Register появляется автоматически:

```
<%@ Register Src="C1.ascx" TagName="C1" TagPrefix="uc1" %>
```

Теперь новый пользовательский элемент управления можно описать так:

```
<uc1:C1 id="C1_1" runat="server">
    </uc1:C1>
```

Свойства, описанные в классе, можно устанавливать в описании на странице, причем они даже будут видны в окне свойств дизайнера.

Пользовательские элементы полностью участвуют в отображении страницы, и вставленные в него элементы ведут себя как обычно. Во время жизненного цикла страницы вызываются события встроенного в нее элемента управления.

В коде страницы можно манипулировать его свойствами:

```
    protected void Page_Load(object sender, EventArgs e)
    {
```

```
            C1_1.Field = "В текстовое поле ничего не введено";
        }
```

2.5. Вопросы для самоконтроля

1. Какой тег html соответствует серверному элементу управления <asp:Label>?

а) <Span>

б) <Input Type="Text">

в) <img>

г) <Table>

2. Какой тег html соответствует серверному элементу управления <asp:ListBox>?

а) <Select>

б) <Input Type=”Text”>

в) <Span>

г) <img>

3. Какой тег html соответствует серверному элементу управления <asp:DropDownList>?

а) <Select>

б) <Input Type=”Text”>

в) <Span>

г) <img>

4. Какой тег html соответствует серверному элементу управления <asp:TextBox>?

а) <input Type=”Text”>

б) <Select>

в) <Input Type="Hidden">

г) <Input Type="Radio">

5. Какой тег html соответствует серверному элементу управления <asp:TextBox>?

а) <input Type="Password”>

б) <Select>

в) <Input Type="Hidden">

г) <Input Type="Radio">

6. Какой тег html соответствует серверному элементу управления <asp:TextBox>?

а) <Textarea>

б) <Select>

в) <Input Type="Hidden">

г) <Input Type="Radio">

7. Какой тег html соответствует серверному элементу управления <asp:HiddenField>?

а) <Input Type="Hidden">

б) <Input Type="Radio">

в) <input Type=”Text”>

г) <Span>

8. Какой тег html соответствует серверному элементу управления <asp:RadioButton>?

а) <Input Type="Radio">

б) <Input Type="Hidden">

в) <Span>

г) <Select>

9. Какой тег html соответствует серверному элементу управления <asp:RadioButtonList>?

а) <Input Type="Radio">

б) <Input Type="CheckBox">

в) <Input Type="button">

г) <Select>

10. Какой тег html соответствует серверному элементу управления <asp:CheckBox>?

а) <Input Type="CheckBox">

б) <Select>

в) <Input Type="Radio">

г) <Span>

11. Какой тег html соответствует серверному элементу управления <asp:CheckBoxList>?

а) <Input Type="CheckBox">

б) <Select>

в) <Input Type="Radio">

г) <Span>

12. Какой тег html соответствует серверному элементу управления <asp:Button>?

а) <Input Type="button">

б) <Input Type="Radio">

в) <Select>

г) <Input Type="CheckBox">

13. Какой тег html соответствует серверному элементу управления <asp:Image>?

а) <img>

б) <picture>

в) <Input Type="Radio">

г) <a>

14. Какой тег html соответствует серверному элементу управления <asp:ImageButton>?

а) <Input Type="image">

б) <img>

в) <Input Type="button">

г) <Input Type="Radio">

15. Какой тег html соответствует серверному элементу управления <asp:Table>?

а) <Table>

б) <img>

в) <Input Type="button">

г) <Select>

16. Какой тег html соответствует серверному элементу управления <asp:Panel>?

а) <Div>

б) <Table>

в) <img>

г) <penel>

17. Какой тег html соответствует серверному элементу управления <asp:BulletedList>?

а)

б) <list>

в) <Table>

г) <img>

18. Какой тег html соответствует серверному элементу управления <asp:BulletedList>?

а)

б) <list>

в) <Table>

г) <img>

19. Какой тег html соответствует серверному элементу управления <asp:HyperLink>?

а) <A Href>

б)

в) <list>

г) <Table>

20. Какой серверный элемент управления соответствует html тегам <Input Type="Text"> <Input Type="Password"> <Textarea>?

а) <asp:TextBox>

б) <asp:ListBox>

в) <asp:Label>

г) <asp:Table>

21. Какой серверный элемент управления соответствует html тегам <Input Type="button"> <Input Type="submit">?

а) <asp:Button>

б) <asp:TextBox>

в) <asp:ListBox>

г) <asp:Label>

22. Какой серверный элемент управления соответствует html тегу <Span>?

а) <asp:Label>

б) <asp:Button>

в) <asp:TextBox>

г) <asp:ListBox>

23. Какой серверный элемент управления соответствует html тегу <img>?

а) <asp:Image>

б) <asp:Table>

в) <asp:Panel>

г) <asp:Picture>

24. Какой серверный элемент управления соответствует html тегу <Table>?

а) <asp:Table>

б) <asp:Panel>

в) <asp:Picture>

г) <asp:Button>

25. Какой серверный элемент управления соответствует тегу <Div>?

а) <asp:Panel>

б) <asp:Table>

в) <asp:Picture>

г) <asp:Button>

26. Определите тип атаки, подходящий под данное описание: атакующий подставляет в поле формы или URL сценарий, используемый приложением при формировании новой страницы:

а) межсайтовый скриптинг

б) атака на отказ

в) подслушивание

г) атака одним щелчком

27. Определите тип атаки, подходящий под данное описание: сервер атакуется ложными запросами, в результате чего система перегружается и настоящий трафик блокируется:

а) а) атака на отказ

б) межсайтовый скриптинг

в)подслушивание

г)атака одним щелчком

28. Определите тип атаки, подходящий под данное описание: атакующий использует специальное ПО для перехвата и чтения незашифрованных пакетов, передаваемых по сети:

а) подслушивание

б) атака на отказ

в) межсайтовый скриптинг

г) подмена значений скрытого поля

29. Определите тип атаки, подходящий под данное описание: атакующий подменяет значение скрытых полей, содержащих критичные для приложения данные:

а) подмена значений скрытого поля

б) подслушивание

в) атака на отказ

г) межсайтовый скриптинг

30. Определите тип атаки, подходящий под данное описание: серверу направляются опасные POST-запросы HTTP, выдаваемые за запросы от его собственных страниц:

а) атака одним щелчком

б) подслушивание

в) атака на отказ

г) межсайтовый скриптинг

31. Определите тип атаки, подходящий под данное описание: атакующий вычисляет или похищает идентификатор сеанса и подключается к серверу, используя сеанс другого пользователя

а) похищение сеанса

б) подслушивание

в) атака на отказ

г) межсайтовый скриптинг

32. Определите тип атаки, подходящий под данное описание: в поле формы, которое предназначено для ввода значений, включаемых в формируемые SQL- команды путем конкатенации, атакующий вводит злонамеренный SQL код, в результате чего команды изменяется и выполняет нужные атакующему действия

а) внедрение SQL кода

б) подслушивание

в) атака на отказ

г) межсайтовый скриптинг

33. Назовите способ аутентификации, при котором запрос возвращается браузеру с определенным кодом состояния ATL:

а) Basic

б) Digest

в) Integrated Window

г) Anonymous

34. Какая из аутентификаций заключается в обмене информации между браузером и Web-сервером:

а) Integrated Window

б) Basic

в) Digest

г) Anonymous

35. Назовите уровень доверия при котором : приложения работают с полным доверием и могут выполнять любой код в контексте того процесса, в котором они работают:

а) Full

б) Hight

в) Medium

г) Low

36. . Назовите уровень доверия, при котором приложения наделяются большинством поддерживаемых разрешений. Такой режим подходит для приложений, которым необходимо достаточно широкие полномочия, в условиях, когда все же требуется снизить возможные риски

а) Hight

б) Full

в) Medium

г) Low

37. Назовите уровень доверия, при котором приложение может выполнять чтение и запись элементов своего каталога и взаимодействовать с базами данных:

а) Medium

б) Hight

в) Full

г) Low

38. Назовите уровень доверия, при котором приложение позволяет считывать его собственные ресурсы, но запрещено обращаться к ресурсам, расположенным вне его пространства:

а) Low

б) Medium

в) Hight

г) Full

39. Назовите уровень доверия, при котором приложение не может взаимодействовать с защищенными ресурсами. Данная установка подходит для непрофессиональных сайтов, которым достаточно поддержки обычного HTML и очень изолированной бизнес логики:

а) Minimal

б) Medium

в) Hight

г) Full

40. При помощи какого объекта можно обладать всей информацией о пользователе и можно программно изменять его пароль и сведения:

а) MembershipUser

б) ChangePasword

в) ResetPassword

г) Pasword

41. Какому из методов передается старый пароль:

а) ChangePasword

б) MembershipUser

в) ResetPassword

г) Pasword

42. Какой метод позволяет удалить старый и автоматически сгенерировать новый пароль:

а) ResetPassword

б) ChangePasword

в) MembershipUser

г) Pasword

43. События элемента управления Login, пользователь аутентифицирован:

а) Authencicate

б) LoggedIn

в) LogingIn

г) LogingOut

44. События элемента управления Login, пользователь успешно зашел на сайт после аутентификации:

а) LoggedIn

б) Authencicate

в) LogingIn

г) LogingOut

45. События элемента управления Login, после ввода пользователем учетных данных, но до аутентификации

а) LogingIn

б) LoggedIn

в) Authencicate

г) LogingOut

46. Какое из свойств класса Membership возвращает максимальное количество попыток ввода пароля перед блокированием пользователя

а) MaxinvalidName

б) ApplicationName

в) PasswordAttemptWindow

г) Provider

47. Какое из свойств класса Membership возвращает минимальную длину пароля:

а) MinRequiredpasswordLenght

б) ApplicationName

в) PasswordAttemptWindow

г) Provider

48. Какой из методов класса Membership генерирует случайный пароль, заданной длины:

а) GenerationPassword

б) GetAllUsers

в) DeletUsers

г) CreateUsers

49. Какое из свойств MambershipProvider указывает, поддерживает ли провайдер восстановление пароля:

а) EnablepasswordRettrieval

б) EnablePasswordReset

в) PasswordAttemptWindow

г) ApplicationName

50. Свойство Application:

а) содержит ссылку на объект типа HttpApplicationState, который ассоциируется с текущем приложением, в котором выполняется страница;

б) свойство содержит ссылку на объект типа HttpServerUtility;

в) это свойство содержит ссылку на объект типа HttpSessionState, который ассоциируется с текущим приложением, в котором выполняется страница;

г) позволяет получить объект типа Cache, принадлежащий приложению, в котором выполняется страница;

Тема 3. Использование технологии ADO.Net для работы с базами данных

3.1. Основы ADO.NET

Платформа .NET Framework включает собственную технологию доступа к данным – ADO.NET. (ActiveX Data Objects.NET). Эта технология состоит из управляемых классов, позволяющих приложениям .NET подключаться к источникам данных (обычно реляционным базам данных), выполнять команды и управлять автономными данными. Использование этой технологии не почти не зависит от вида приложения: веб-приложение, клиент-серверные настольные приложения, однопользовательские приложения, подключаемые к локальной базе данных.

В ASP.NET можно получить информацию из базы данных без использования классов ADO.NET [3]. Для этого используются такие варианты:

• элемент управления SqlDataSource. Элемент SqlDataSource позволяет определять запросы декларативно. Можно подключать SqlDataSource к элементам управления (например, GridView), предоставляя страницам возможность редактирования и обновления данных без необходимости в коде ADO.NET. При этом ADO.NET используется неявно. Недостатком этого способа является размещение логики базы данных в ту часть страницы, которая касается разметки;

• LINQ (Language Integrated Query) to Entities. С помощью LINQ to Entites можно сгенерировать модель данных с поддержкой на этапе проектирования в Visual Studio. Надлежащая логика для доступа к базе данных в таком случае генерируется автоматически. LINQ to Entites поддерживает внесение обновлений, генерирует безопасные и правильно оформленные операторы SQL и предоставляет широкие возможности в плане настройки. Кроме того, LINQ to Entities теперь заменяет собой более простую модель LINQ to SQL, которой разработчики приложений ASP.NET пользовались ранее. Вдобавок LINQ to

Entities служит основой для новой системы поставки данных под названием ASP.NET Dynamic Data.

В ADO.NET используется многоуровневая архитектура, которая обращается вокруг небольшого числа ключевых концепций, таких как объекты Connection, Command и DataSet.

Однако архитектура ADO.NET серьезно отличается от классической архитектуры ADO методом работы с разными поставщиками баз данных [18]. В ADO программисты всегда используют обобщенный набор объектов, независимо от лежащих в их основе источников данных. Например, для извлечения записи из базы данных Oracle используется тот же класс Connection, что применяется для решения той же задачи в SQL Server. Это не касается технологии ADO.NET, которая использует модель поставщиков данных.

Поставщик данных (data provider) — это набор классов ADO.NET, которые позволяют получать доступ к определенной базе данных, выполнять команды SQL и извлекать данные. По сути, поставщик данных – это мост между вашим приложением и источником данных.

Ниже перечислены классы, которые входят в состав любого поставщика данных:

• Connection. Объект этого класса используется для установки соединения с источником данных;

• Command. Объект этого класса используется для выполнения команд SQL и хранимых процедур;

• DataReader. Объект этого класса предоставляет быстрый однонаправленный доступ только для чтения к данным, извлеченным из запроса;

• Data Adapter. Объект этого класса решает две задачи. Первая — наполнение DataSet (автономная коллекция таблиц и отношений) информацией, извлеченной из источника данных. Вторая — применение изменений к источнику данных в соответствии с модификациями, произведенными в DataSet.

ADO.NET не содержит объектов обобщенных поставщиков данных. Вместо этого имеется набор специализированных поставщиков для различных источников данных.

Каждый поставщик данных имеет специфическую реализацию классов Connection, Command, DataReader и DataAdapter, оптимизированных для конкретных реляционных систем управления базами данных (СУБД). Например, для создания подключения к базе данных SQL Server используется класс соединения по имени SqlConnection.

Одной из ключевых идей, лежащих в основе модели поставщиков ADO.NET, является расширяемость. Другими словами, разработчики могут создавать собственные поставщики для патентованных источников данных. В действительности доступно множество подтверждающих это примеров, которые демонстрируют, как создавать настраиваемые поставщики ADO.NET, служащие оболочками для нереляционных хранилищ данных, таких как файловая система или служба каталогов. Некоторые независимые производители также продают собственные поставщики данных для .NET.

В рамках .NET Framework поставляется небольшой набор из четырех поставщиков (рис. 3.1):

• поставщик SQL Server. Предоставляет оптимизированный доступ к базам данных SQL Server;

• поставщик OLE DB. Предоставляет доступ к любому источнику данных, который имеет драйвер OLE DB;

• поставщик Oracle. Предоставляет оптимизированный доступ к базам данных Oracle;

• поставщик ODBC. Предоставляет доступ к любому источнику данных, имеющему драйвер ODBC.

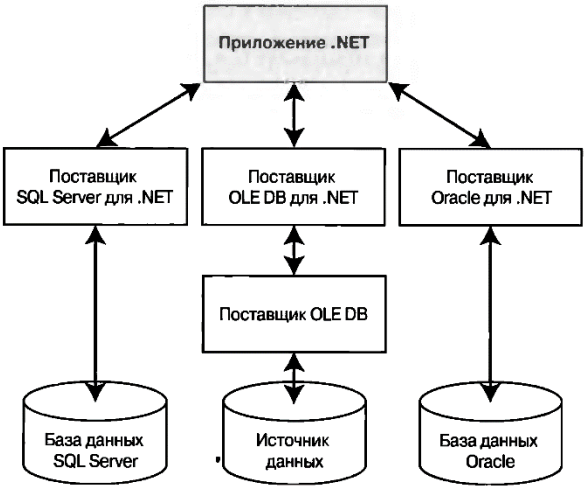

Рис. 3.1 – Поставщики данных

При выборе поставщика сначала пытайтесь найти встроенный поставщик .NET, который предназначен для имеющегося источника данных. Если таковой не найден, можно воспользоваться OLE DB при наличии драйвера OLE DB для источника данных.

Технология OLE DB существует уже много лет как часть ADO, поэтому для большинства источников данных предусмотрены драйверы OLE DB (включая SQL Server, Oracle, Access, MySQL и многие другие). В тех редких случаях, когда найти специализированный поставщик .NET или драйвер OLE DB не удается, можно обратиться к поставщику ODBC, который работает в сочетании с драйвером ODBC.

Разные поставщики данных .NET используют различные классы, но все они некоторым образом стандартизированы. Каждый поставщик основан на одном и том же наборе интерфейсов и базовых классов. Так, например, объект Connection реализует интерфейс IDbConnection, который определяет такие ключевые методы, как Open() и Close(). Подобная стандартизация гарантирует, что каждый класс Connection будет работать одинаковым образом и предоставит один и тот же набор ключевых свойств и методов.

"За кулисами" различные поставщики используют совершенно разные низкоуровневые вызовы и API-интерфейсы. Например, поставщик данных SQL

Server применяет патентованный протокол TDS (Tabular Data Stream — поток табличных данных) для взаимодействия с сервером.

Преимущества этой модели не сразу очевидны, но весьма существенны:

• поскольку каждый поставщик использует одни и те же интерфейсы и базовые классы, можно писать обобщенный код доступа к данным (с приложением небольших дополнительных усилий), работая с интерфейсами, а не классами поставщиков;

• поскольку каждый поставщик реализован отдельно, он может использовать соответствующую оптимизацию (это отличается от модели ADO, где каждый вызов базы данных должен проходить через общий уровень, прежде чем достигнет лежащего в основе драйвера базы данных). Кроме того, специализированные поставщики могут добавлять нестандартные средства, которых не имеют другие поставщики (например, возможность SQL Server выполнять XML-запросы).

ADO.NET также имеет другой уровень стандартизации — DataSet. Класс DataSet – это контейнер данных общего назначения, которые извлекаются из одной или более таблиц источника данных. DataSet полностью обобщен; другими словами, специализированные поставщики не определяют собственных специализированных версий класса DataSet. Независимо от того, какой поставщик данных применяется, можно извлекать данные и помещать их в полностью автономный DataSet одинаковым образом. Это облегчает отделение кода, извлекающего данные, от кода, обрабатывающего их. В случае смены лежащей в основе базы данных придется изменить только код, извлекающий данные, но если используется DataSet, а информация имеет одну и ту же структуру, модифицировать способ ее обработки не понадобится.

3.2. Фундаментальные классы ADO.NET

ADO.NET имеет два типа объектов: основанные на соединении и основанные на содержимом:

• объекты, основанные на соединении. Существуют объекты поставщика данных, такие как Connection, Command и DataReader. Они позволяют подключаться к базе данных, выполнять операторы SQL, перемещаться по результирующему набору, доступному только для чтения, и наполнять DataSet. Объекты, основанные на соединении, специфичны для типа источника данных и находятся в специфичных для поставщика пространствах имен (таких как System. Data. SqlClient для поставщика SQL Server);

• объекты, основанные на содержимом Эти объекты в действительности лишь "упаковывают" данные. Они включают DataSet, DataColumn, DataRow, DataRelation и др. Они полностью независимы от типа источника данных и определены в пространстве имен System. Data.

Классы ADO.NET группируются в несколько пространств имен. Каждый поставщик имеет свое собственное пространство имен, а обобщенные классы вроде DataSet находятся в пространстве имен System.Data.

Поставщик ADO.NET — это просто набор классов ADO.NET (с реализацией Connection, Command, DataAdapter и DataReader), которые поставляются в сборке типа библиотеки классов. Обычно все эти классы поставщика имеют один и тот же префикс. Например, префикс OleDb применяется для поставщика OLE DB в ADO.NET, который предусматривает реализацию объекта Connection по имени OleDbConnection.

Класс Connection. Класс Connection позволяет устанавливать соединения с источником данных, с которым нужно взаимодействовать. Перед тем, как можно будет делать что-то еще (в том числе извлечение, удаление, вставка или обновление данных), понадобится установить соединение.

Ключевые свойства и методы Connection определены интерфейсом IDbConnection, который реализуют все классы Connection.

При создании объекта Connection понадобится предоставить строку соединения. Строка соединения представляет собой последовательность пар "имя-значение", разделенных точками с запятой (;). Порядок этих настроек, как

и регистр, не важен. Все вместе они указывают базовую информацию, необходимую для установки соединения.

Хотя строки соединений варьируются в зависимости от реляционной СУБД и используемого поставщика данных, несколько фрагментов информации необходимы почти всегда:

• сервер, на котором размещается база данных. Если сервер базы данных всегда расположен на том же компьютере, что и приложение ASP.NET, поэтому вместо имени компьютера применяется псевдоним localhost;

• база данных, которую следует использовать;

• как база данных производит аутентификацию. Поставщики данных Oracle и SQL Server предоставляют возможность выбора — применить определенные учетные данные аутентификации либо подключиться как текущий пользователь системы. Последний вариант обычно более предпочтителен, поскольку в этом случае не понадобится помещать информацию о пароле в код или конфигурационные файлы.

При создании объекта Connection можно передать конструктору в виде параметра строку соединения. В качестве альтернативы можно вручную установить значение свойства ConnectionString, если это делается до попытки открыть соединение. Раздел <connectionString> в файле web.config — самое подходящее место для сохранения строки соединения. Ниже показан пример:

```
<configuration>
<connectionStrings>
<add name="имя_строки" connectionString=
"Data Source=localhost;Initial Catalog=имяБД;
    Integrated Security=SSPI"/>
</connectionStrings>
</configuration>
```

Затем эту строку соединения легко извлечь по имени из коллекции WebConfigurationManager.ConnectionStrings. При условии, что импортировано пространство имен System.Web.Configuration, для этого можно воспользоваться следующим оператором:

```
string connectionString =
WebConfigurationManager.ConnectionStrings["имя_строки"].ConnectionString;
```

Каждый сервер баз данных хранит главный каталог всех установленных на нем баз данных. В этом каталоге содержатся сведения об имени каждой базы данных и месте размещения файлов, в которых находятся данные. При создании базы данных (например, запуском соответствующего сценария или применением инструмента управления) информация об этой базе данных добавляется в главный каталог. При подключении к базе данных ее имя указывается в строке соединения с использованием значения Initial Catalog.

Интересно то, что SQL Server Express располагает удобным средством, которое позволяет обходить главный каталога и подключаться к любому файлу базы данных напрямую даже при отсутствии информации о нем в главном каталоге баз данных. Это средство называется пользовательскими экземплярами (user instances) и в полной редакции SQL Server оно не доступно.

SQL Server Express — это усеченная версия SQL Server, которая распространяется бесплатно. Она обладает определенными ограничениями, например, может использовать только один ЦП и максимум 1 Гбайт ОЗУ и не поддерживает базы данных объемом более 4 Гбайт.

Однако она все равно является удивительно мощной и подходит для многих веб-сайтов среднего масштаба. Даже еще лучше то, что ее можно легко обновить до платной версии SQL Server, если возникнет необходимость в большем количестве функциональных возможностей.

Для присоединения пользовательского экземпляра базы данных необходимо установить параметр User Instances в True (в строке соединения) и

в параметре AttachDBFilename предоставить имя нужной базы данных. Значение Intital Catalog указывать не понадобится.

Вот пример строки соединения в случае применения такого подхода:

```
myConnection.ConnectionString =
 @"Data Source=localhost\SQLEXPRESS;" + "Integrated Security=SSPT;" +
@"AttachDBFilename=|DataDirectory|\имяБД.mdf;User Instance=True";
```

Здесь присутствует еще одна хитрость. Имя файла начинается с | DataDi rectory |. Это автоматически указывает на папку AppData внутри каталога веб-приложения. В таком случае не нужно задавать полный путь к файлу, который может перестать быть корректным после перемещения веб-приложения на веб-сервер. ADO.NET всегда будет искать файл по имени *имяБД*.mdf в каталоге AppData.

Пользовательские экземпляры — удобное средство, когда имеется веб-сервер, обслуживающий много веб-приложений с базами данных, которые часто добавляются и удаляются. Это средство также хорошо работает в сочетании с другими высокоуровневыми компонентами ASP.NET, такими как профили и система членства. По умолчанию эти компоненты создают файловые базы данных для SQL Server Express, которые сокращают работы по конфигурированию:

```
SqlClient     Integrated Security=true;  -- or -- Integrated Security=SSPI;
OleDb         Integrated Security=SSPI;
Odbc  Trusted_Connection=yes;
OracleClient          Integrated Security=yes;
```

Значение Integrated Security=true вызывает исключение при работе с поставщиком OleDb.

После выбора строки соединения управлять подключением очень легко — нужно просто использовать методы Open () и Close ().

Приведенный ниже код в обработчике события Page.Load можно использовать для проверки соединения и вывода его состояния в текст метки. Чтобы код работал, понадобится импортировать пространство имен System.Data.SqlClient. Например,

```
// Создать объект Connection,
string connectionString =
WebConfigurationManager.ConnectionStrings
   [“имя_строки_соединения”].ConnectionString;
SqlConnection con = new SqlConnection(connectionString);
try
{
 // Попытаться открыть соединение.
con.Open();
//Для созданной ранее метки LabelTest
LabelTest.Text = "<b>Server Version:</b> " + con.ServerVersion;
   // Версия сервера
LabelTest.Text += "<br /><b>Connection Is:</b>" + con.State.ToString() ;
// Состояние соединения
}
catch (Exception err)
{ // Обработать ошибку, отобразив соответствующую информацию.
LabelTest.Text = «Ошибка чтения БД. " + err.Message;
// Возникла ошибка при чтении базы данных
}
finally
{ //В любом случае убедиться, что соединение правильно закрыто.
// Даже если оно не было открыто успешно,
```

```
    вызов Close () не приводит к ошибке.
con.Close () ;
LabelTest.Text += "<br /><b>Now Connection Is:</b> "
    + con.State.ToString();
}
```

Классы Command и DataReader. Класс Command позволяет выполнить SQL-оператор любого типа [5]. Хотя класс Command можно использовать для решения задач определения данных (таких как создание и изменение баз данных, таблиц и индексов), все же более вероятно его применение для выполнения задач манипулирования данными (вроде извлечения и обновления записей в таблице).

Специфичные для поставщика классы Command реализуют стандартную функциональность, в точности как классы Connection. В данном случае небольшой набор ключевых свойств и методов, используемых для выполнения команд через открытое соединение, определяется интерфейсом IDbConnection.

Прежде чем использовать команду, необходимо выбрать ее тип, установить ее текст и привязать к соединению. Всю эту работу можно выполнить, установив значения соответствующих свойств (CommandType, CommandText и Connection), либо передать необходимую информацию в аргументах конструктора.

Значения CommandType:

CommandType.Text – команда будет выполнять прямой оператор SQL. Оператор SQL указывается в свойстве CommandText. Это — значение по умолчанию;

CommandType.StoredProcedure – команда будет выполнять хранимую процедуру в источнике данных. Свойство CommandText представляет имя хранимой процедуры ;

CommandType.TableDirect – команда будет опрашивать все записи таблицы. CommandText — имя таблицы, из которой команда извлечет все

записи. Эта опция предназначена только для обратной совместимости с некоторыми драйверами OLE DB. Она не поддерживается поставщиком данных SQL Server и не работает так хорошо, как тщательно направленный запрос.

Фрагмент кода на C# имеет вид:

```
SqlCommand cmd = new SqlCommand();
cmd.Connection = имя_соединения;
cmd.CommandType = CommandType.Text;
cmd.CommandText = "текст_запроса";
```

или

SqlCommand cmd = new SqlCommand(" *текст_запроса* ", *имя_соединения*) ;

Методы Command реализуются с помощью функций:

ExecuteNonQuery() – выполняет команды, отличные от select, такие как SQL-операторы вставки, удаления или обновления записей. Возвращаемое значение указывает количество строк, обработанных командой;

ExecuteNonQuery () –можно использовать для выполнения команд определения данных, которые создают, изменяют и уничтожают объекты базы данных (наподобие таблиц, индексов, ограничений и т.п.);

ExecuteScalar() – Выполняет запрос select и возвращает значение первого поля первой строки из набора строк, сгенерированного командой. Этот метод обычно применяется при выполнении агрегатной команды select, использующей функции вроде COUNT () или SUM () для вычисления единственного значения;

ExecuteReader() – Выполняет запрос SELECT и возвращает объект DataReader, который является оболочкой однонаправленного курсора, доступного только для чтения.

Класс DataReader. Класс DataReader позволяет читать данные, возвращенные командой SELECT, по одной строке за раз, в однонаправленном,

доступном только для чтения потоке. Иногда это называют пожарным курсором. Использование DataReader — простейший путь получения данных, но ему недостает возможностей сортировки и связывания автономного объекта Data Set. Однако DataReader представляет наиболее быстрый способ доступа к данным.

Основные методы DataReader:

Read () – перемещает курсор строки на следующую строку в потоке. Этот метод также должен быть вызван перед чтением первой строки данных. Когда DataReader создается впервые, курсор строки помещается в позицию непосредственно перед первой строкой. Метод Read () возвращает true, если существует следующая строка для чтения, или false, если прочитана последняя строка в наборе;

GetValue () – возвращает значение, сохраненное в поле с указанным именем столбцы или индексом, внутри текущей выбранной строки. Тип возвращенного значения — ближайший тип .NET, наиболее соответствующий встроенному значению, хранимому в источнике данных. Если обратится к полю с неверным индексом (ссылающийся на несуществующее поле), то будет выбрашено исключение IndexOutOfRangeException. Используя индексатор для DataReader, можно получить значение по имени поля. (Другими словами, myDataReader.GetValue(0) и myDataReader ["NameOfFirstField"] эквивалентны.) Поиск по имени более читабелен, но менее эффективен;

GetValues ( ) – сохраняет значения текущей строки в массиве. Количество сохраняемых полей зависит от размеров массива, переданного этому методу. С помощью свойства DataReader. FieldCount можно определить действительное количество полей в строке и воспользоваться этой информацией для создания массива нужного размера, если нужно сохранить в нем все поля;

Getlnt32(), GetChar (), GetDateTimeO, GetXxx() – эти методы возвращают значение поля с указанным индексом в текущей строке, причем тип данных указывается в имени метода. Обратите внимание, что если попытаться присвоить возвращенное значение переменной неверного типа, возникнет

исключение invalidCastException. Кроме того, эти методы не поддерживают типов, допускающих null-значения. Если поле может содержать null, это придется проверить перед вызовом одного из методов. Чтобы проверить на null-значение, сравните непреобразованное значение (которое можно извлечь по позиции методом GetValue () или по имени с помощью индексатора DataReader) С константой DBNull .Value;

NextResult () – если команда, которая сгенерировала DataReader, возвратила более одного набора строк, этот метод перемещает указатель на следующий набор строк и устанавливает его непосредственно перед первой строкой Close () – Закрывает модуль чтения. Если исходная команда запустила хранимую процедуру, возвратившую выходное значение, это значение может быть прочитано из соответствующего параметра после закрытия модуля чтения.

Получив DataReader, можно организовать цикл для прохождения по его записям, вызывая метод Read () в теле цикла. Этот метод перемещает курсор строки на следующую запись (при первом вызове — на первую строку). Метод Read () также возвращает булевское значение, означающее наличие последующих строк для чтения. В следующем примере цикл продолжается до тех пор, пока Read () не вернет false, после чего элегантно завершается.

Информация из каждой записи затем объединяется в одну длинную строку. Чтобы обеспечить быстрое выполнение манипуляций со строками, вместо обычных объектов строк используется StringBuilder (из пространства имен System.Text).

Метод ExecuteNonQuery () выполняет команды, которые не возвращают результирующих наборов, такие как INSERT, DELETE или UPDATE. Метод ExecuteNonQuery () возвращает одну порцию информации — количество обработанных записей (или -1, если команда отлична от INSERT, DELETE или UPDATE).

3.3. Расширенные элементы управления-контейнеры для работы с данными

Познакомитесь поближе с тремя наиболее мощными элементами управления данными: GridView, DetailsView, FormView и ListView.

Grid View — исключительно гибкий табличный элемент управления, предназначенный для демонстрации данных в виде двумерной сетки (grid), или экранной таблицы, состоящей из строк pi столбцов. Он включает в себя широкий диапазон встроенных средств, включая выделение, разбиение на страницы и редактирование. К тому же он может быть расширен с помощью шаблонов. Огромным преимуществом GridView перед DataGrid является его поддержка сценариев без кода. Используя GridView, можно без написания кода решать множество распространенных задач, таких как перемещение по страницам и выделение. В DataGrid для получения тех же средств нужно было кодировать обработку событий.

Свойство GridView AutoGenerateColumns по умолчанию имеет значение true. В этом случае GridView использует рефлексию для исследования объекта данных и нахождения полей (для записи) или свойств (для пользовательского объекта). Затем он создает столбцы для каждого из них в том порядке, в котором их обнаруживает.

Эта автоматическая генерация столбцов хороша для быстрого создания тестовых страниц, но не предоставляет необходимой гибкости, которая обычно требуется.

Если нужно скрыть столбцы, изменить порядок их следования или настроить некоторые аспекты их отображения, такие как форматирование и текст заголовков, то понадобится установить AutoGenerateColumns в false и определить столбцы самостоятельно в разделе <Columns> дескриптора элемента управления GridView.

Этот элемент управления связан с источником данных DataSourse (рис. 3.2). При включенном свойстве AutoGenerateColumns (" True ") элемент

GridView автоматически выбирает из связанного источника данных все столбцы в том порядке, в котором их обнаруживает, и выстраивает их в таблицу с заголовками колонок, равными именам полей. Чтобы иметь возможность самостоятельно выбирать порядок следования полей, а также их количество и вид необходимо установить в настройках GridView атрибут AutoGenerateColumns="False" и определить столбцы самостоятельно в разделе <Columns> дескриптора. Если же явно назначить столбцы таблицы, формируемой GridView, определяя их заголовки, порядок следования, фон, шрифт и другие аспекты отображения, скрывать или отображать, а при этом свойство AutoGenerateColumns = " True ", то вслед за явно определенными столбцами последуют автоматически сгенерированные столбцы. Если свойство AutoGenerateColumns отключить и не определить явно ни одного столбца, то GridView ничего не сгенерирует.

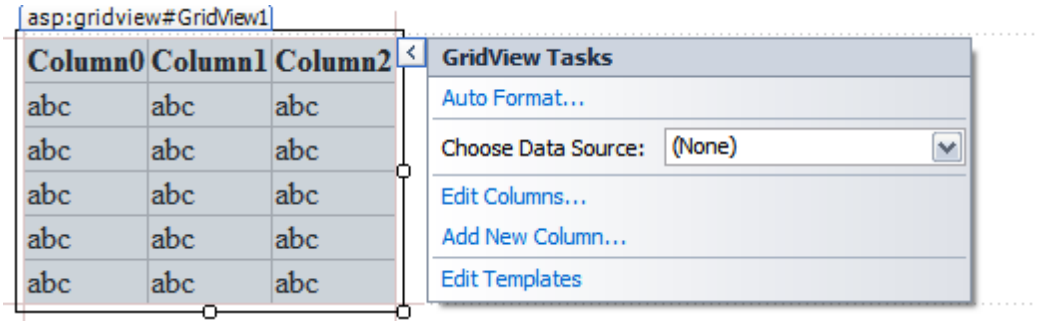

Рис. 3.2 - Источником данных является элемент DataSourse

Для работы с источниками данных часто возникает необходимость редактировать, добавлять и удалять записи. Чтобы иметь такую возможность в расширенных элементах управления для работы с данными необходимо определить эти дополнительные возможности у источника данных DataSourse.

Изменим свойства источника данных изменив его настройки: во вкладке Advanced. Кроме этого необходимо определить PrimaryKey в структуре таблицы базы данных. Также для поля со свойством PrimaryKey необходимо задать Identity Specification равным Yes, начальное значение и шаг изменения.

После внесения таких изменения в источник данных появиться возможность сгенерировать кнопки Edit, Delete, Update для каждой строки в элементе GridView.

Для визуальной настройки столбцов GridView предназначено диалоговое окно Fields, которое можно вызвать командой Edit Columns (Add New Column) через встроенную панель, щелкнув на кнопке с пиктограммой треугольника в правом верхнем углу отображаемого на Web-форме элемента GridView

Диалоговое окно Fields можно вызвать также щелкнув на свойстве Columns в панели Properties (рис. 3.3).

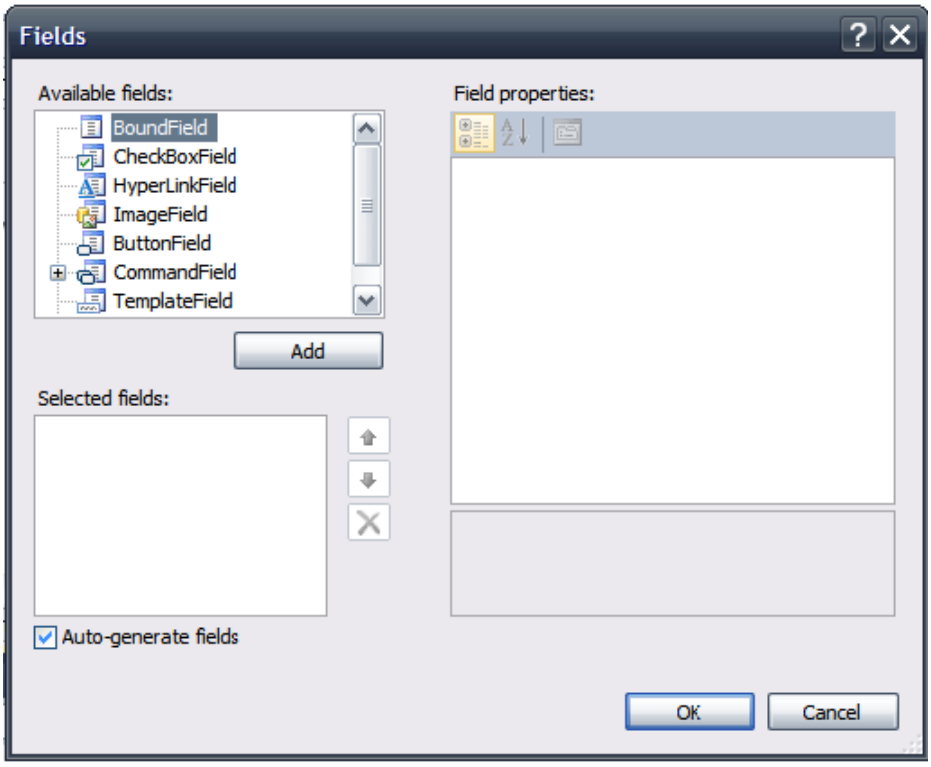

Рис.3.3 – Свойства элемента GridView

Каждый явно определенный столбец GridView относиться к одному из типов, перечисленных в таблице.

Типы столбцов, явно определяемых в GridView, имеют названия:

BoundField - Основной тип, представляющий отображаемые данные поля;

CheckBoxField - Столбец, представленный флажками для отображения логических данных типа bit;

HyperLinkField - Отображает содержимое полей в форме гиперссылки;

ImageField - Отображает графические данные из двоичного поля;

ButtonField - Отображает содержимое полей в форме кнопок;

CommandField - Представляет управляющий столбец с кнопками редактирования;

TemplateField - Специфицирует отображение множественных полей, настраиваемых элементов управления и произвольного HTML, используя шаблон. Обладает наибольшей гибкостью, но требует больших усилий в настройке.

Все действия в диалоговом окне Fields запоминаются в дескрипторном описании элемента GridView в разделе <Columns>, например

```
<asp:GridView ID="GridView1" runat="server"
      AutoGenerateColumns="False">
  <Columns>
    <asp:BoundField />
    <asp:CheckBoxField />
    <asp:HyperLinkField />
    <asp:ImageField>
    </asp:ImageField>
    <asp:ButtonField Text="Button" />
    <asp:CommandField />
    <asp:TemplateField></asp:TemplateField>
  </Columns>
</asp:GridView>
```

Коллекцией столбцов можно управлять и программно обратившись к коллекции Columns, которая отвечает за столбцы. Нумерация элементов в коллекции начинается с 0. Например,

```
GridView1.Columns[1].Visible=false;
```

скрывает второй столбец. Скрытый столбец не генерирует HTML код.

Элемент GridView подстраивается под источник данных, который возвращает определенный набор столбцов. После изменения источника данных так, что он стал возвращать новый набор столбцов, команда Refresh Schema (обновить схему) встроенной панели GridView вернет его в исходное состояние (регенерирует).

При явном объявлении привязанного поля имеется возможность установить другие свойства. BoundField имеет свойства:

DataField - имя поля элемента данных, которое нужно отобразить в данном столбце;

DataFormatString - строка форматирования, удобная для представления чисел и дат;

ApplyFormatInEditMode - булев флаг включения строки форматирования даже применительно к введенному тексту (по умолчанию равно false). Если равно true, то строка формата будет использоваться для форматирования значения, даже если оно отображается в текстовом поле в режиме редактирования;

HeaderText, HeaderImageUrl, FooterText - текст заголовка, рисунок заголовка, нижний колонтитул столбца. Первые два свойства устанавливают текст заголовка и нижнего колонтитула сетки, если сетка имеет заголовок (ShowHeader установлено в true) и нижний колонтитул (ShowFooter установлено в true). Заголовок чаще всего используется для указания описательного имени, такого как имя поля, в то время как нижний колонтитул может содержать динамически вычисляемое. Чтобы показать графическое изображение в заголовке вместо текста, понадобится установить свойство HeaderlmageUrl;

ReadOnly - замыкает столбец от режима редактирования. Применяется, как правило, к первичным ключам;

InsertVisible - булев флаг управления возможностью вставки. Если равно false, то значение этого столбца не может быть установлено в режиме вставки. Если нужно, чтобы значение столбца устанавливалось программно или было

основано на значении по умолчанию, определенном в базе данных, то можно использовать это свойство;

Visible - скрывает столбец от генерации HTML;

SortExpression - определяет выражение, которое может быть добавлено к запросу для выполнения сортировки на базе данного столбца;

HtmlEncode - булев флаг включения URL-кодирования, при котором в генерируемом HTML управляющие символы заменяются на ASCII. Делает генерируемый HTML более безопасным на клиентской стороне и рекомендуется к применению как можно чаще;

NullDisplayText - определяет текст, который будет отображен в таблице на месте null-значения (например, "неопределено"). По умолчанию равно пустой строке, хотя это можно изменить на жестко закодированное значение, такое как (not specified);

ConvertEmptyStringToNull - булев флаг. Если включен, то перед подтверждением обновления в конечном источнике данных все пустые строки будут преобразованы в null-значения;

ControlStyle, HeaderStyle, FooterStyle, ItemStyle - определяет стиль конкретного раздела отдельного столбца.

Каждый столбец BoundField предоставляет свойство DataFormatString, которое можно использовать для настройки внешнего вида чисел и дат, используя форматную строку. Форматные строки обычно состоят из заполнителя и индикатора формата, которые заключены в фигурные скобки. Типичная форматная строка выглядит примерно так: {0:C}. Здесь 0 представляет собой значение, которое будет отформатировано, а буква – предопределенный стиль формата. Буква C означает денежный формат, основанный на установках культуры, касающихся текущего потока. По умолчанию компьютер, настроенный для региона English (United States) работает с локалью en-US и отображает денежные значения с символом доллара (поэтому 3400.34 превращается в $3,400.34).

Форматные строки для чисел имеют вид:

- денежный {0:C} ($1,234.50);

- скобки указывают, что значение является отрицательным: ($1,234.50);

- символ валюты зависит от локали: G1,234.50;

- нучный (экспоненциальный) {0:E} (1.234.50E+004);

- процентный {0:P} (45.6%);

- фиксированный десятичный {0: F?}, зависит от количества десятичных цифр после точки. {0:F3} даст 123.400, а {0:F0} — 123;

- короткая дата {0:d} М/д/гггг (например: 10/30/2008);

- длинная дата и короткое время {0:f} дцдц, ММММ дд, гггг ЧЧ:мм дп (например: Monday, January 28, 2008 10:00 AM);

- длинная дата {0:D} дцдц, ММММ дд, гггг (например: Monday, January 28, 2008);

- длинная дата и длинное время {0:F} дцдц, ММММ дд, гггг ЧЧ:мм:сс дп (например: Monday, January 28, 2008 10:00:23 AM);

- стандарт ISO, поддерживающий сортировку {0:s} гггг- М М - дцТЧ Ч: м м: сС (например: 2008-01-28T10:00:23);

- месяц и день {0:M} ММММ дд (например: January 28);

- общий {0:G} М/д/гггг НН:mm:ss дп (зависит от настроек локали) (например: 10/30/2008 10:00:23 AM).

Стили в GridView. Среда разработки предоставляет возможность применить AutoFormat, который уже создан. Но есть также возможность самостоятельно настраивать вид GridView. Все свойства для форматирования – то не простые однозначные свойства, а составные объекты, имеющие все необходимые настройки. Все они влияют на внешний вид генерируемой HTML-сетки в целом.

Свойства стилей GridView следующие:

HeaderStyle - определяет стиль строки заголовков столбцов, если включено их отображение (т.е. ShowHeader = true);

RowStyle - определяет стиль каждой строки данных;

AlternatingRowStyle - определяет стиль каждой четной строки данных. Это форматирование действует в дополнение к форматированию RowStyle;

SelectedRowStyle - определяет стиль текущей выбранной строки данных;

EditRowStyle - определяет стиль текущей строки данных, находящейся в режиме редактирования;

EmptyDataRowStyle - конфигурирует стиль, используемый для отображения одной пустой строки, когда привязанный объект данных вообще не содержит строк;

FooterStyle - определяет стиль строки нижнего колонтитула столбцов (ShowFooter=true);

PagerStyle - определяет внешний вид ссылок на составные страницы, если включено постраничное разбиение (AllowPaging=true).

Выбор строки означает, что пользователь может выделить или изменить внешний вид строки щелчком на какой-то кнопке или ссылке. Когда пользователь щелкает на кнопке, то не только строка изменяет свой внешний вид, но также появляется шанс обработать событие в коде.

Элемент GridView предоставляет встроенную поддержку выбора (рис.3.4).

	Column0	Column1	Column2
Select	abc	abc	abc
Select	abc	abc	abc
Select	abc	abc	abc
Select	abc	abc	abc
Select	abc	abc	abc

Рис.3.4 – Общий вид элемента GridView без настроек

Нужно просто добавить столбец CommandField со свойством ShowSelect, установленным в true. При этом CommandField может отображаться как гиперссылка, кнопка или фиксированное изображение. Тип указывается с

помощью свойства ButtonType (рис.3.5). Затем можно указать текст в свойстве SelectText или же ссылку на изображение в свойстве SelectlmageUrl.

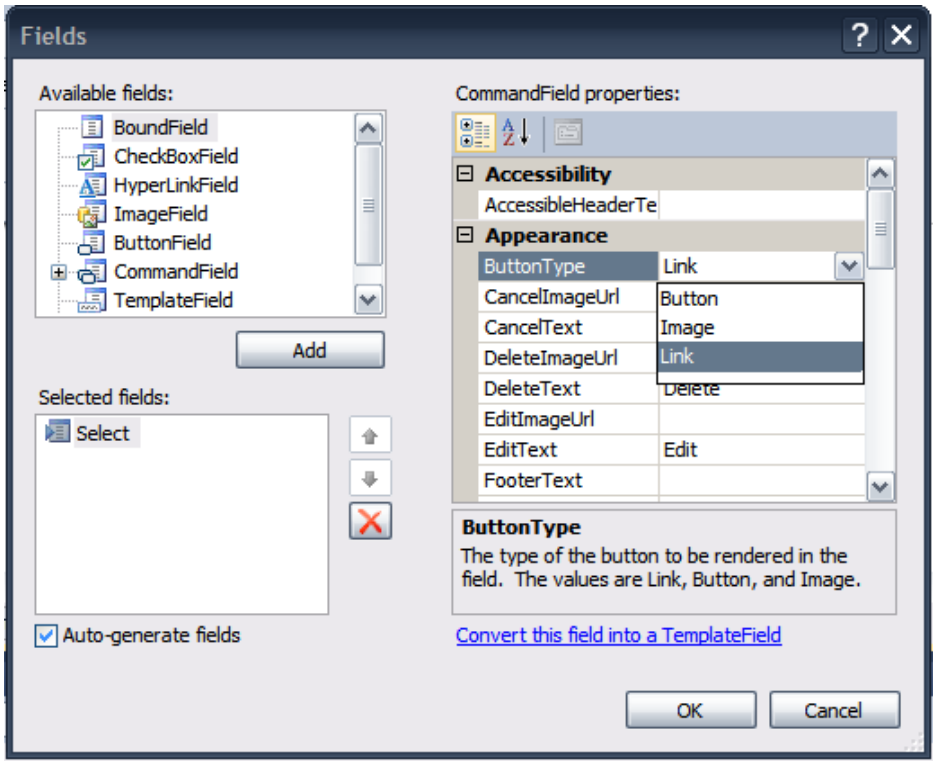

Рис.3.5 – Настройка элемента GridView

При щелчке на кнопке выбора выполняется обратная отправка страницы, а также серия дополнительных шагов. Сначала генерируется событие GridView.SelectedlndexChanging, которое можно перехватить, чтобы отменить операцию.

Далее изменяется значение свойства GridView.Selectedlndex, чтобы указать на выбранную строку. И, наконец, генерируется событие GridView.SelectedlndexChanged, которое можно обработать, если нужно вручную обновить другие элементы управления, отразив новый выбор. Когда страница отображается, выбранной строке назначается стиль SelectedRowStyle.

Чтобы поддержать возможность выбора строки, не обязательно создавать новый столбец. Вместо этого можно превратить существующий столбец в активную ссылку. Такой прием обычно используется, чтобы дать возможность пользователям выбирать строки таблицы по уникальному значению

идентификатора. Для этого добавьте столбец ButtonField. Затем установите в DataTextField имя поля, которое хотите использовать:

<asp:ButtonField ButtonType="Button" DataTextField="Имя первичного ключа" />

Это поле будет подчеркнуто и превращено в ссылку, щелчок на которой приведет к обратной отправке страницы и генерации события GridView.RowCommand. Это событие можно обработать, определив, на какой строке был совершен щелчок, и программно установить свойство Selectedlndex. Однако можно воспользоваться и более простым методом. Для этого нужно настроить ссылку, чтобы она генерировала событие SelectedlndexChanged, указав для CommandName текст Select:

<asp:ButtonField CommandName=" Select" ButtonType="Button" DataTextField=" Имя первичного ключа " />

Теперь щелчок на поле данных приводит к автоматическому выбору записи. Столбец CommandField, который использовался ранее для отображения ссылки Select, а также столбец BoundField, применяемый для отображения EmployeelD, можно удалить, потому что столбец ButtonField эффективно совмещает эти две детали в одном месте.

Сортировка. Средства сортировки GridView позволяют переупорядочить результирующий набор строк GridView, щелкая на заголовке столбца. Это удобно, и это легко реализовать. Чтобы разрешить сортировку, нужно установить свойство GridView.AllowSorting в true. Далее понадобится определлть SortExpression для каждого столбца, который может быть отсортирован. Теоретически выражение сортировки может использовать любой синтаксис, который понимает элемент управления источником данных. На практике выражение сортировки почти всегда принимает форму, используемую

в конструкции ORDER BY запроса SQL. Это значит, что выражение сортировки может включать единственное поле или список полей, разделенный запятыми, и с необязательным словом ASC или DESC, добавленным после имени столбца, которое позволяет сортировать в восходящем или нисходящем порядке.

Вот как определить столбец FirstName, чтобы строки сортировались в алфавитном порядке по имени:

```
<asp:BoundField DataField="FirstName11
HeaderText="First Name" SortExpression="FirstName"/>
```

Если два раза щелкнуть на заголовке столбца FirstName в строке, то первый щелчок отсортирует его по алфавиту в прямом порядке, а второй — в обратном.

Не все источники данных поддерживают сортировку.

Шаблоны. До сих пор в примерах элемент управления GridView использовался для отображения данных с помощью отдельно привязанных столбцов для каждого поля. Если требуется поместить множественные значения в одну ячейку или иметь неограниченные возможности настройки содержимого ячейки, добавляя HTML-дескрипторы и серверные элементы управления, то для этого нужно применять TemplateField. TemplateField позволяет определять полностью настраиваемый шаблон для столбца. Внутрь шаблона можно добавлять дескрипторы элементов управления, произвольные элементы HTML и выражения привязки данных. Вы получаете полную свободу для отображения чего угодно.

Элементы TemplateField:

HeaderTemplate - определяет внешний вид и содержимое ячейки заголовка;

FooterTemplate - определяет внешний вид и содержимое ячейки нижнего колонтитула;

ItemTemplate - определяет внешний вид и содержимое каждой ячейки данных (если не используется AlternatingltemTemplate) или каждой нечетной ячейки (если используется);

AlternatingltemTemplate - используется в сочетании с ItemTemplate для различного форматирования четных и нечетных строк;

EditItemTemplate - определяет внешний вид и элементы управления, используемые в режиме редактирования;

InsertItemTemplate - определяет внешний вид и элементы управления, используемые при вставке новой записи.

Например, предположим, что необходимо создать столбец, который комбинирует поля имени, фамилии и титула. Чтобы выполнить такой трюк, можно сконструировать примерно такой TemplateField:

```
<asp:GridView ID="GridView1" runat="server" AllowSorting="True"
AutoGenerateColumns="False" DataKeyNames="StudID"
DataSourceID="SqlDataSource1">
        <Columns>
          <asp:CommandField ShowEditButton="True" />
          <asp:TemplateField HeaderText="Фамилия, имя"
              SortExpression="LastName">
            <EditItemTemplate>
              <asp:TextBox ID="TextBox1" runat="server"
              Text='<%# Bind("LastName") %>'></asp:TextBox>
              <asp:TextBox ID="TextBox2" runat="server"
              Text='<%# Bind("FirstName") %>'></asp:TextBox>
            </EditItemTemplate>
            <ItemTemplate>
              <%#Eval("LastName") %> <%#Eval("FirstName") %>
            </ItemTemplate>
          </asp:TemplateField>
```

```
                    <asp:TemplateField HeaderText="kurs" SortExpression="kurs">
                        <EditItemTemplate>
                            <asp:DropDownList ID="DropDownList1" runat="server"
                                SelectedValue='<%# Bind("kurs") %>'>
                                <asp:ListItem text="1"></asp:ListItem>
                                <asp:ListItem text="2"></asp:ListItem>
                                <asp:ListItem text="3"></asp:ListItem>
                                <asp:ListItem text="4"></asp:ListItem>
                                <asp:ListItem text="5"></asp:ListItem>
                            </asp:DropDownList>
                        </EditItemTemplate>
                        <ItemTemplate>
                            <asp:Label ID="Label1" runat="server"
                            Text='<%# Bind("kurs") %>'></asp:Label>
                        </ItemTemplate>
                    </asp:TemplateField>
                </Columns>
            </asp:GridView>
            <asp:SqlDataSource ID="SqlDataSource1" runat="server"
        ConnectionString="<%$ConnectionStrings:StudentConnectionString %>"
            DeleteCommand="DELETE FROM [Stud1g] WHERE [StudID] =
@StudID" InsertCommand="INSERT INTO [Stud1g] ([FirstName], [LastName],
[email], [kurs]) VALUES (@FirstName, @LastName, @email, @kurs)"
            SelectCommand="SELECT * FROM [Stud1g]"
UpdateCommand="UPDATE [Stud1g] SET [FirstName] = @FirstName, [LastName]
= @LastName, [email] = @email, [kurs] = @kurs WHERE [StudID] = @StudID">
            <DeleteParameters>
                <asp:Parameter Name="StudID" Type="Int16" />
            </DeleteParameters>
            <UpdateParameters>
```

```
            <asp:Parameter Name="FirstName" Type="String" />
            <asp:Parameter Name="LastName" Type="String" />
            <asp:Parameter Name="email" Type="String" />
            <asp:Parameter Name="kurs" Type="Int16" />
            <asp:Parameter Name="StudID" Type="Int16" />
        </UpdateParameters>
        <InsertParameters>
            <asp:Parameter Name="FirstName" Type="String" />
            <asp:Parameter Name="LastName" Type="String" />
            <asp:Parameter Name="email" Type="String" />
            <asp:Parameter Name="kurs" Type="Int16" />
        </InsertParameters>
    </asp:SqlDataSource>
```

Если теперь привязать GridView, он извлечет информацию из источника данных и пройдет по коллекции элементов. Он обработает ItemTemplate для каждого элемента, вычислит выражения привязки данных и добавит сгенерированную HTML-разметку к таблице. Этот шаблон довольно прост — он всего лишь определяет три выражения привязки данных. После вычисления эти выражения преобразуются в обычный текст.

Как вы заметили, в этих выражениях используется Eval() — статический метод класса System.Web.UI.DataBinder. Присутствие Eval() здесь обязательно - он автоматически извлекает элемент данных, привязанный к текущей строке, использует рефлексию для нахождения соответствующего поля (для объекта DataRow) или свойства (для пользовательского объекта данных) и извлекает значение. Процесс рефлексии добавляет немного работы. Однако маловероятно, что эти накладные расходы намного увеличат время обработки запроса. Без метода Eval() пришлось бы обращаться к данным через свойство Container.Dataltem и использовать код приведения типа, вроде такого:

<%# ((EmployeeDetails)Container.Dataltem)["FirstName"] %>

Для работы с источников даных применяется функция ИмяGridView.DataBind();

DetailsView. Элемент, который выводит только одну запись в отличие от GridView, поддерживает разбиение на страницы.

В отличие от GridView, DetailsView позволяет вставлять записи. Для этого нужно установить значение свойства AutoGenerateInsert Button= "True". При отображении появится кнопка New. Ее нажатие переводит элемент в режим вставки, по умолчанию для каждого поля генерируются TextBox-ы.

Если источник данных для DetailsView — SqlDataSource, то у него должны быть определены свойство InsertCommand и набор параметров.

У DetailsView имеются пары событий, которые происходят при связывании с данными, при переходе из режима просмотра в режим вставки, при перелистывании страницы.

FormView. Еще один новый элемент FormView похож на DetailsView, но отличается от него тем, что нуждается в шаблоне для своего представления:

3.3. Вопросы для самоконтроля

1. Что чаще всего используется для хранения данных

а) СУБД

б) текстовый документ

в) CompareValidator

г) RangeValidator

2. В виде чего хранятся данные в СУБД

а) в виде таблиц

б) в виде страниц

в) в виде архивов

г) в виде строковых записей

3. Что представляет собой запись

а) строка таблицы

б) столбец таблицы

в) ячейка таблицы

г) поле в таблице

4. Что представляет собой поле

а) столбец таблицы

б) строка таблицы

в) ячейка таблицы

г) запись в таблице

5. Множество таблиц данных, связанных отношениями, составляют

а) базу данных

б) схему данных

в) представление

г) схему

6. Что из ниже перечисленных программ не является СУБД

а) Microsoft Power Point

б) Microsoft Access

в) Microsoft SQL Server 2000

г) FoxPro

7. Что представляет ADO.Net

а) набор классов для работы с внешними данными

б) набор классов для работы с внутренними данными

в) набор классов для работы с графикой

г) набор классов для обработки прерываний

8. Какой объект отвечает за соединение с базой данных

а) объект Connection

б) объект Command

в) объект DataAdapter

г) объект DataBase

9. Что является обязательным атрибутом объекта Connection

а) строка соединения

б) объект Command

в) объект DataAdapter

г) объект DataBase

10. Какое свойство объекта Connection хранит строку соединения с СУБД

а) ConnectionString

б) DataSource

в) Database

г) State

11. Что такое транзакция

а) последовательность команд, которая выполняется как одно целое

б) объект DataSource

в) объект Database

г) объект State

12. Что представляет собой класс DataSet

а) представление в памяти информации, считанной через ADO из баз данных или XML

б) последовательность команд, которая выполняется как одно целое

в) объект класса Database

г) объект класса State

13. Что обеспечивает класс DataAdapter

а) двусторонний обмен информацией между СУБД и приложением

б) последовательность команд, которая выполняется как одно целое

в) *представление* в памяти информации, считанной через ADO из баз данных или XML

г) объект класса State

14. Что делает объект Command

а) исполняет запрос *SQL*

б) обеспечивает двусторонний обмен информацией между СУБД и приложением

в) обеспечивает *представление* в памяти информации, считанной через ADO из баз данных или XML

г) создаёт транзакции - последовательность команд, которая выполняется как одно целое

15. Какая вкладка Visual Studio 2005 позволяет работать с соединениями баз данных

а) вкладка Server Explorer

б) вкладка Solution Explorer

в) вкладка Class Explorer

г) вкладка Project Explorer

16. Использование технологии ADO.Net для работы с базами данных на основе SQL Server. Connection, Command, DataReader.

17. Модель связывания с данными: связывание элементов управления с данными, выражения связанные с данными.

18. Таблицы, связанные с данными: элементы управления DataGrid и GridView.

Тема 4. Технология LINQ для работы с данными

4.1. Введение в Linq

LINQ относится к одним из самых интересных средств языка C#. Аббревиатура LINQ означает Language-Integrated Query, т.е. язык интегрированных запросов [10]. Это понятие охватывает ряд средств, позволяющих извлекать информацию из источника данных. Как вам должно быть известно, извлечение данных составляет важную часть многих программ. Например, программа может получать информацию из списка заказчиков, искать информацию в каталоге продукции или получать доступ к учетному документу, заведенному на работника. Как правило, такая информация хранится в базе данных, существующей отдельно от приложения. Так, каталог продукции может храниться в реляционной базе данных. В прошлом для взаимодействия с такой базой данных приходилось формировать запросы на языке структурированных запросов (SQL). А для доступа к другим источникам данных, например в формате XML, требовался отдельный подход.

LINQ дополняет C# средствами, позволяющими формировать запросы для любого LINQ-совместимого источника данных. При этом синтаксис, используемый для формирования запросов, остается неизменным, независимо от типа источника данных. Это, в частности, означает, что синтаксис, требующийся для формирования запроса к реляционной базе данных, практически ничем не отличается от синтаксиса запроса данных, хранящихся в массиве. Для этой цели теперь не нужно прибегать к средствам SQL или другого внешнего по отношению к C# механизма извлечения данных из источника. Возможности формировать запросы отныне полностью интегрированы в язык C#.

Помимо SQL, LINQ можно использовать вместе с XML-файлами и наборами данных ADO.NET Dataset. Не менее важным является применение LINQ вместе с массивами и коллекциями в C# Таким образом, средства LINQ

предоставляют, в целом, единообразный доступ к данным. И хотя такой принцип уже сам по себе является весьма эффективным и новаторским, преимущества LINQ этим не ограничиваются. LINQ предлагает осмыслить иначе и подойти по-другому к решению многих видов задач программирования, помимо традиционной организации доступа к базам данных. И в конечном итоге многие решения могут быть выработаны на основе LINQ. LINQ поддерживается целым рядом взаимосвязанных средств, включая внедренный в C# синтаксис запросов, лямбда-выражения (является исполняемым объектом), анонимные типы и методы расширения [4].

LINQ в C# - это, по сути, язык в языке. Поэтому предмет рассмотрения LINQ довольно обширен и включает в себя многие средства, возможности и альтернативы. Несмотря на то что в этой главе дается подробное описание средств LINQ, рассмотреть здесь все их возможности, особенности и области применения просто невозможно. Для этого потребовалась бы отдельная книга. В связи с этим в настоящей главе основное внимание уделяется главным элементам LINQ, применение которых демонстрируется на многочисленных примерах.

Формальная структура запроса на LINQ имеет вид:

<переменная запроса> = from <переменная диапазона> in <источник данных> [where <условие, дающее логическое значение>]

[orderby <правило сортировки> [<направление сортировки>]] {select <возвращаемые данные> |

[group <возвращаемые данные> by <ключ группировки>]};

где <переменная запроса> – переменная, в которой хранится ссылка на правила запроса и с использованием которой можно будет выполнить запрос и получить доступ к возвращаемым результатам. Должна быть одной из форм интерфейса IEnumerable<T>. Может быть объявлена как var; <переменная диапазона> – идентификатор, который используется внутри запроса для обращения к

требуемым данным; <источник данных> – должен поддерживать интерфейсIEnumerable<T>. По <источнику данных> автоматически определяется тип <переменной диапазона>; <условие, дающее логическое значение> – условие, позволяющее из общего набора данных отобрать требуемые. Часто в условии используется <переменная диапазона>; <правило сортировки> – данные, по значениям которых будет осуществляться сортировка. Могут получаться на основе построения выражений. Часто использу ется <переменная диапазона>; <направление сортировки> – слова «ascending» или «descending», определяющие сортировку по возрастанию и убыванию, соответственно. По умолчанию используется сортировка по возрастанию, поэтому слово «ascending» можно не использовать; <возвращаемые данные> – данные, которые будут записаны в <переменную запроса>. Часто используется <переменная диапазона>; <ключ группировки> – данные, по которым будут осуществлена группировка. Часто задается с использованием<переменной диапазона>.

Например, построим запрос для отображения из целочисленного массива элементы, значение которых большие 51.

```
string s = "";
int[] Mas1 = { 6, 4, 8, 5, 9 }; var Mas2 = from elem in Mas1
where elem > 5 select elem;
foreach (var elem in Mas2)
s += elem + " "; // s = “6 8 9 ”
```

Формирование запроса и его выполнение – две независимые процедуры.

Так в примере, приведенном выше, команда var Mas2 = ... выполняет только формированием запроса. При этом никакого отбора данных еще не производиться. Чтобы убедиться в этом, изменим программу следующим образом:

```
string s = "";
int[] Mas1 = { 6, 4, 8, 5, 9 };
var Mas2 = from elem in Mas1 where elem > 5 select elem;
```

```
Mas1[0] = 3;
foreach (var elem in Mas2)
s += elem + " "; // s = “8 9 ”
```

Таким образом, выполнение запроса осуществляется только при получении данных из <переменной запроса>.

Одна и та же <переменная запроса> может использоваться несколько раз, и при этом выдавать разные результаты, если<источник данных> между выполнением запросов был изменен. Например:

```
string s = "";
int[] Mas1 = { 6, 4, 8, 5, 9 }; var Mas2 = from elem in Mas1
where elem > 5 select elem;
foreach (var elem in Mas2)
s += elem + " "; // s = “6 8 9 ” Mas1[0] = 3;
s = "";
foreach (var elem in Mas2)
s += elem + " "; // s = “8 9 ”
```

В приведенных выше примерах использовалось неявное типизирование переменных с использованием var (как при описании запроса, так и при его выполнении).

Однако типы всех переменных могут быть описаны явно. При этом <переменная запроса> должна быть описана как одна из форм интерфейса IEnumerable<T>, где<T> вытекает из результата, указанного в блоке select. Тип<переменной диапазона> должен соответствовать элементу<источника данных> и тоже может быть задан явно. Также тип<T> должна иметь переменная, используемая при выполнении запроса. Например:

```
string s = "";
int[] Mas1 = { 6, 4, 8, 5, 9 };
IEnumerable<int> Mas2 = from int elem in Mas1
where elem > 5 select elem;
foreach (int elem in Mas2)
```

```
s += elem + " "; // s = “6 8 9 ”
```

LINQ to SQL. Операции LINQ to DataSet состоят из множества специальных операций, определенных в нескольких сборках и пространствах имен, которые позволяют разработчику решать следующие задачи:

- выполнять операции множеств на последовательностях объектов DataRow;

- извлекать и устанавливать значения DataColumn;

- получать из DataTable стандартные последовательности LINQ типа IEnumerable<T>, так что можно вызывать стандартные операции запросов;

- копировать измененные последовательности из объектов DataRow в DataTable.

В дополнение к этим операциям LINQ to DataSet, операция AsEnumerable, то сможете вызывать стандартные операции запросов из LINQ to Objects на возвращенных последовательностях объектов DataRow, достигая большей мощи и гибкости.

Типизированные DataSet могут быть опрошены с использованием LINQ, как это возможно и с нетипизированными. Однако типизированные DataSet позволяют упростить код запросов LINQ и сделать его более читабельным. Поскольку существует класс для DataSet, при запросе к типизированному DataSet можно обращаться к именам таблиц и столбцов, используя свойства типизированного класса DataSet, вместо индексации в коллекции Tables или применения операций Field<T> и SetField<T>.

Операцию AsEnumerable специально предназначена для класса DataTable и возвращает последовательность объектов DataRow. В статическом классе System.Data.DataTableExtensions есть операция AsEnumerable. Назначение этой операции - возвращать последовательность типа IEnumerable<DataRow> из объекта DataTable. Эта операция, будучи вызванной на объекте DataTable, возвращает последовательность объектов DataRow. Обычно так выглядит первый шаг при выполнении запроса LINQ to DataSet на DataTable объекта DataSet. За счет вызова этой операции можно получить последовательность

IEnumerable<T>, где T является DataRow, что позволяет вызывать множество операций LINQ, которые могут быть вызваны на последовательности типа IEnumerable<T>.

В то время как большинство обычных запросов LINQ выполняются на массивах и коллекциях, реализующих интерфейсы IEnumerable<T> или IEnumerable, запрос LINQ to SQL выполняется на классах, реализующих интерфейс IQueryable<T>, таком как TaЬ1e<T>. Это значит, что запросам LINQ to SQL доступны дополнительные операции запросов, наряду со стандартными операциями запросов, поскольку IQueryable<T> реализует IEnumerable<T>.

В отличие от запросов LINQ, которые выполняются в памяти локальной машины, запросы LINQ to SQL транслируются в вызовы SQL, которые в действительности выполняются в базе данных. Из-за этого происходит некоторое расхождение, такое как способ обработки проекции, которая не может в действительности случаться в базе данных, поскольку база ничего не знает о сущностных классах, как и о любых других классах.

4.2. Использование средств языка XML в SQL

Тесная связь Web-технологий с технологиями баз данных сложилась еще на ранних этапах развития сети Интернет. Она сводилась к обеспечению теледоступа к системам баз данных через среду Web. В настоящее время создано и функционирует огромное количество приложений такого рода в самых различных областях деятельности. Однако до появления технологии XML не удавалось обеспечить реальную интеграцию информационных ресурсов Web и баз данных. Система базы данных выступали здесь по отношению к Web как «черный ящик». Только с развитием технологии XML стали проявляться более глубокие связи между этими двумя направлениями информационных технологий. Стремление к обеспечению в Web полноценных возможностей управления данными, поддерживаемыми в этой среде в рамках XML-технологий, объективно привело к необходимости использования

подходов и принципов, аналогичных тем, которые на протяжении десятилетий прошли испытание временем в технологиях баз данных.

В результате использования этих подходов и принципов в Web-технологиях в лексиконе спецификаций стандартов платформы XML появились такие ключевые термины технологий баз данных, как модель данных, схема, ограничение целостности, язык запросов.

Со временем эта тенденция привела к тому, что было создано несколько коммерческих компаний, занявшихся разработкой баз данных на основе XML. Данные в этих базах данных хранились в виде XML-документов либо непосредственно в текстовом виде.

Производители баз данных формата XML высказывают в пользу своих продуктов те же аргументы, которые в свое время приводили производители объектно-ориентированных баз данных [12].

1. Поскольку огромное количество внешних данных представлено в формате XML, в базах данных удобнее всего использовать этот же формат и соответствующую модель данных.

2. Так как все большее количество пользователей осваивает HTML и XML, базы данных XML-формата также доступны для пользователей, как и реляционные базы данных SQL-типа.

На сегодняшний день базы данных XML-формата являются пока новым направлением рынка СУБД, и время покажет, будут ли они иметь успех. Однако история развития строго объектно-ориентированных баз данных показала, что производители реляционных СУБД способны достаточно быстро расширять свои продукты, включая в них важнейшие элементы новых моделей данных, благодаря чему их продукты сохраняют доминирующую роль в области обработки данных.

Если сервер СУБД выполняет множество дополнительных функций по ведению базы данных (поддерживает транзакции, блокирует таблицу или запись от конфликтных изменений, сохраняет ее целостность, выполняет различные действия по оптимизации запросов), то работа с XML-файлами

таких возможностей не дает. При этом надо учитывать, что полная открытость XML-файлов делает их беззащитными от внешнего просмотра, поэтому вряд ли разумно хранить в них конфиденциальную информацию .

Основной недостаток использования XML-файлов в качестве базы данных заключается в том, что организовать корректную работу множества пользователей с одним файлом практически невозможно. Как только одна из клиентских программ начинает модифицировать такой файл-базу, все остальные пользователи будут либо ждать окончания этого процесса, либо пытаться одновременно внести в файл противоречивые данные, модифицированные разными пользователями.

Поэтому лучше всего задействовать XML-файлы в интеграционных приложениях, когда данные из одних баз и систем передаются во временное хранилище. При этом интеграция реляционных СУБД с XML будет возрастать и реляционные продукты будут включать все больше XML-ориентированных функций.

Сходства XML и SQL. Поскольку язык XML произошел от SGML, он обладает рядом полезных характеристик, сближающих его с языком SQL:

– *описательный подход*. В XML принят такой подход к определению структуры документов, при котором задается структура и содержимое каждого элемента документа, а не то, как он должен обрабатываться. Этот же подход используется и в SQL, ориентированном на определение запрашиваемых данных, а не на то, как они должны извлекаться;

– *строительные блоки*. XML-документ состоит из небольшого количества базовых строительных блоков, в число которых входят такие фундаментальные компоненты, как *элементы* и *атрибуты*. В SQL этим понятиям соответствуют таблицы и столбцы;

– *типы документов*. В XML каждый документ воспринимается не сам по себе, а в качестве представителя определенного типа, соответствующего документам реального мира, например заказ, ответ на деловое письмо или

резюме кандидата на должность. И здесь тоже очевидна параллель с SQL, поскольку таблицы также представляют типы сущностей реального мира.

Различия XML и SQL. Однако несмотря на очевидные параллели между XML и SQL, между ними есть и значительные различия:

– *ориентация на документы или на данные***.** Базовые концепции XML сформированы на основе структуры типовых документов. Это язык с ориентацией на текст, и в нем строго различается содержимое (элементы документа) и характеристики этого содержимого (атрибуты). В то же время базовые концепции SQL сформированы на основе структур, типичных для обработки данных. SQL ориентирован на данные, поддерживает широкий диапазон типов данных (в двоичном представлении), и его структуры (таблицы и столбцы) ориентированы на содержимое (данные). Это расхождение между фундаментальными моделями SQL и XML приводит к некоторым сложностям их совместного использования;

– *иерархическая или табличная структура*. Естественные структуры XML имеют иерархическую природу и отражают иерархию элементов большинства распространенных типов документов (например, книга содержит главы, главы включают параграфы, параграфы содержат заголовки, абзацы и рисунки). Эти структуры не являются жесткими. Например, один параграф содержит пять абзацев и один рисунок, а в следующем параграфе будет три абзаца и два рисунка и т. Д. В противоположность этому структуры SQL имеют табличную, а не иерархическую организацию. Более того, это жесткие структуры - все строки таблицы SQL содержат одинаковый набор столбцов в одинаковом порядке. Эти отличия также затрудняют совместное использование SQL и XML;

– *объекты или операции*. Основной задачей языка XML является представление объектов. Если выделить осмысленный фрагмент текста XML и спросить, что он представляет, выяснится, что представляет он какой-нибудь объект: абзац, заказ товаров, адрес клиента и т. П. У языка SQL более широкие задачи, но в первую очередь он ориентирован на *обработку объектов.* Если

выделить осмысленный фрагмент текста SQL и спросить, что он представляет, выяснится, что он представляет *операцию* над объектом: создание объекта, удаление объекта, поиск одного и/или более объектов, либо обновление содержимого объекта. Эти отличия делают назначение и использование языков XML и SQL взаимодополняющими. Как уже было отмечено, стремительный рост популярности XML привел к тому, что производители баз данных стали включать его поддержку в свои продукты. Формы поддержки XML различаются, но все их можно условно разделить на пять следующих категорий;

– *хранение данных в формате XML*. Реляционные базы данных могут принимать XML-документ как символьную строку переменой длины (VARCHAR) или данные большого символьного объекта (CLOB). В этом случае XML документ является содержимым одного столбца одной строки базы данных. При усиленной поддержке XML, по сравнению с этим элементарным уровнем СУБД может позволять явно объявлять столбцы как относящиеся к типу данных XML;

– *вывод в формате XML*. Данные одной или более строк результата запроса легко представить в виде XML-документа. Поддержка выходных данных в формате XML означает, что в ответ на SQL-запрос СУБД вместо обычного набора строк и столбцов может генерировать XML-документ;

– *ввод в формате XML*. XML-документ может содержать данные, предназначенные для вставки в одну или более новых строк таблицы базы данных, или же в нем могут содержаться данные, предназначенные для обновления строки таблицы, либо данные, идентифицирующие удаляемую строку. Поддержка входных данных в формате XML означает, что вместо SQL-запросов СУБД может принимать в качестве входных данных XML документы;

– *обмен данными в формате XML*. XML представляет собой очень удобный и естественный способ выражения данных для обмена данными между разными СУБД или серверами баз данных. Данные исходной базы

данных преобразуются в XML-документ и направляются в принимающую базу данных, где они вновь преобразуются в формат базы данных;

– *интеграция данных XML*. Это более высокий уровень поддержки интегрированного хранения данных в формате XML, суть которого состоит в том, что СУБД может выполнить синтаксический анализ XML-документа, разделить его на составляющие, и сохранить отдельные элементы в отдельных столбцах. После этого для поиска данных в полученной таблице может использоваться обычный SQL- таким образом, реализуется поддержка поиска элементов и XML-документе. В ответ на запрос СУБД может снова собрать XML-документ из хранящихся в таблице составляющих элементов.

Хранение данных в формате XML. Ввод, вывод и обмен данными в формате XML открывают очень эффективный путь интеграции существующих реляционных баз данных с расширяющимся миром XML. Формат XML используется во внешнем по отношению к базам данных мире для представления структурированных данных, но данные в самой базе данных сохраняют табличную структуру, состоящую из строк и столбцов. Очевидно, что следующим шагом в развитии этой интеграции является хранение XML - документов прямо в базе данных.

Если СУБД на базе SQL поддерживает большие объекты, это означает, что она уже содержит элементарные средства поддержки, хранения и извлечения XML-Документов. Некоторые коммерческие базы данных хранят и извлекают большие текстовые документы при помощи двух типов данных: больших символьных объектов (CLOB) и больших двоичных объектов (BLOB). Во многих коммерческих продуктах поддерживаются значения типа BLOB или CLOB объемом до 4 Гбайт, что достаточно для хранения подавляющего большинства XML-документов.

Для хранения XML-документа в базе данных с использованием этой технологии нужно определить таблицу с одним столбцом типа BLOB или CLOB для хранения текста документа и несколькими вспомогательными столбцами (стандартных типов данных) для хранения атрибутов,

идентифицирующих данный документ. Например, если в таблице должны храниться документы с заказами товаров, в дополнение к столбцу типа CLOB для хранения XML-документов можно определить вспомогательные столбцы для хранения номера заказчика, даты заказа и номера заказа, используя тип данных INTEGER, VARCHAR или DATE. Тогда можно будет выполнять поиск документов в таблице заказов по номеру документа, дате заказа или номеру клиента, и для извлечения или хранения XML-документа использовать технологии обработки CLOB-данных.

Преимуществом этого подхода является то, что его относительно просто реализовать. Он поддерживает четкое разделение между операциями SQL (такими, как обработка запросов) и операциями XML. Недостатком же его является очень низкий уровень интеграции между XML и СУБД. В простейшей реализации XML-документ совершенно прозрачен для СУБД. Последняя ничего не знает о его содержимом. Его нельзя искать по значению одного из его атрибутов или элементов, если только этот атрибут или элемент не извлечен из документа хранится в отдельном столбце таблицы. Впрочем, если вы заранее можете предположить, какие типы поиска наиболее вероятны, это ограничение становится не таким уж значительным.

Используемые для обмена данными между приложениями XML-документы, хранящиеся в файлах или базах данных в столбцах типа CLOB, всегда имеют текстовый формат. Этот формат обеспечивает максимальную переносимость содержимого, но компьютерным программам работать с ним крайне неудобно.

Синтаксический анализатор XML представляет собой элемент компьютерного программного обеспечения, выполняющий преобразование XML-документа из текстового формата в более подходящее для программной обработки внутреннее представление. Любая СУБД на основе SQL, поддерживающая XML, должна включать синтаксический анализатор для обработки XML-документов. Если СУБД поддерживает тип данных CLOB, для

усиления интеграции с XML она позволяет синтаксическому анализатору XML работать прямо с содержимым CLOB-столбцов [14].

Существует два популярных типа XML-анализаторов, поддерживающих два стиля обработки XML-документов.

– Document Object model (DOM). DOM-анализаторы преобразуют XML-документ в иерархическую древовидную структуру. После этого при помощи API DOM программа может перемещаться по дереву вверх и вниз, следуя иерархии документа.

Интерфейс API DOM облегчает программистам доступ к структуре документа и его элементам.

– Simple XPI for XML (SAX). SAX-анализаторы преобразуют XML-документ в последовательность обратных вызовов программы, которые информируют программу о каждой встреченной анализатором части документа. В ответ программа может выполнять определенные действия, например, реагировать на начало каждого раздела документа или на конкретный атрибут. Интерфейс API SAX предлагает использующим его программам более последовательный стиль обработки документа, лучше соответствующий программной структуре приложений, управляемых событиями.

Любой синтаксический анализатор XML проверит правильность оформления документа и, кроме того, сверит XML-документ со схемой.

DOM-анализатор удобен в тех случаях, когда размер XML-документа относительно мал; этот анализатор генерирует в оперативной памяти древообразное представление документа, занимающее вдвое больше места, чем исходный документ.

Анализатор типа SAX позволяет обрабатывать большие документы небольшими фрагментами. Но поскольку документ не находится в памяти целиком, программе приходится выполнять по нему несколько проходов, если она обрабатывает его фрагменты не по порядку.

Сравнение библиотек SYSTEM.XML и SYSTEM.XML. LINQ. Для начала, как и в случае с System.Xml, подключим необходимое пространство имен: using System.Xml.Linq.

Далее из множества классов выберем класс XDocument, создаем новый экземпляр этого класса и воспользуемся первым из двух подходов библиотеки Linq, заключающимся в использовании вложенных конструкторов.

```
XDocument doc = new XDocument(
        new XElement("Items",
            new XElement("item",
                new XAttribute("id", trackId++),
                new XElement("name", "One"),
                new XElement("name2", "First")
                );
```

Первый конструктор который используется это - XDocument(Params object[] content). Он создает новый Xml-документ и имеет четыре перегруженных метода, которые принимают множество различных параметров.

Воспользуемся методом, который принимает один параметр Params object[] content.

Params object[] content - это массив объектов, который будет записан в Xml-докумет при создании.

new XElement (string s,…) - записывает элемент в Xml-докумет и сразу же закрывает его, где s- имя элемента, а «…» обозначают, что конструктор может содержать в себе другие конструкторы и значения по имени элемента.

new XAttribute (string s, string value) - записывает новый атрибут в элементе, где s- имя элемента, value - значение элемента.

После всех предыдущих действий необходимо сохранить документ, воспользовавшись функцией Save.

doc.Save(Name+".xml").

В итоге получим новый Xml-докумет с записанными в него данными:

```
<?xml version="1.0"?>
<Items>
  <Item id="1">
    <Name>One</Name>
    <Name2>First</Name2>
  </Item>
</Items>
```

Воспользуемся вторым способом создания Xml-докумета.

```
XDocument docx = new XDocument();
        XElement Items = new XElement("Items");
        docx.Add(Items);
        XElement Item = new XElement("Item");
        Item.Add(new XAttribute("id", 1));
        XElement name = new XElement("name");
        name.Value = "One";
        Item.Add(name);
        XElement name2 = new XElement("name2");
        name2.Value = "First";
        Item.Add(name2);
```

С помощью конструктора XDocument() создадим элемент класса XDocument. По аналогии, с помощью всех необходимых конструкторов создадим экземпляры соответствующих классов и с помощью метода Add() запишем их в файл. В итоге получим такой же Xml-докумет с записанными в него данными:

```
<?xml version="1.0"?>
<Items>
  <Item id="1">
    <Name>One</Name>
    <Name2>First</Name2>
  </Item>
</Items>
```

Чтоб изменить свойства элементов в Xml-документе, создадим метод WriteXml(), который будет находить необходимый элемент по заданному атрибуту id и изменять соответствующие ему дочерние элементы и их атрибуты.

Load() - считывает данные из документа:

XDocument doc = XDocument.Load(NameFile + ".xml").

Теперь с помощью цикла найдем необходимый объект по его id, определим список всех элементов находящихся по указанному id и изменим интересующие нас элементы и их свойства. И в конце, не забудем сохранить.

```
foreach (XElement el in doc.Root.Elements("item"))
{
int idm = Int32.Parse(el.Attribute("id").Value);
if (idm == id)
{
   el.SetElementValue("name"," Two ");
   el.SetElementValue("name2","First");
}
}
doc.Save(NameFile + ".xml");
```

После выполнения описанных выше действий, получим:

```
<?xml version="1.0"?>
```

```
<Items>
  <Item id="1">
    <Name> Two</Name>
    <Name2>First</Name2>
  </Item>
```

Так как в пространстве имен System.Xml определение новых элементов и их свойств, производится путем вставки каждого нового элемента и его атрибутов (свойств) в корневой элемент, что уже описано выше, то второй раз описывать это не будем. Рассмотрим уже полученный документ, в котором вставлен элемент со значением атрибута «2» с дочерними элементами имеющими значения Two и Second.

```
<?xml version="1.0"?>
<Items>
  <Item id="1">
    <Name>One</Name>
    <Name2>First</Name2>
  </Item>
  <Item id="2">
    <Name>Two</Name>
    <Name2>Second</Name2>
  </Item>
```

Рассмотрим добавление новых элементов и их атрибутов с использованием пространства имен System.Xml.Linq.

Создается новый элемент:

```
XElement Item = new XElement("item").
```

Далее добавим в него необходимые атрибуты с помощью уже описанных методов Add() , XAttribute () и XElement().

```
Item.Add(new XAttribute("id", id));
XElement name = new XElement("name");
```

```
name.Value = "Two ";
Item.Add(name);
XElement name2 = new XElement("name2");
name2.Value = " Second ";
Item.Add(name2);
docx.Root.Add(Item);
```

Сохранить изменения:

```
docx.Save(NameFile + ".xml");
```

Документ примет вид:

```
<?xml version="1.0"?>
<Items>
  <Item id="1">
    <Name>One</Name>
    <Name2>First</Name2>
  </Item>
  <Item id="2">
    <Name>Two</Name>
    <Name2>Second</Name2>
  </Item>
```

Различия между способами работы с Xml-документами в пространствах имен System.Xml и System.Xml.Linq. В библиотеке Linq возможно создание документа и запись в него данных несколькими способами, в отличие от библиотеки Xml, что ограничивает разработчика и осложняет ему работу с библиотекой Xml. Также необходимо выделить способы создания, которые очень осложнены и запутаны в библиотеке Xml, в отличие от Linq. Заметим, что при создании новых объектов Xml-документа, библиотека Linq, в отличие от Xml, предусматривает автоматическое открытие и закрытие создаваемого объекта.

В библиотеке Xml при создании документа, необходимо создавать экземпляр класса XmlTextWriter, который является, по своей сути, лишь

способом создания. Нельзя создать документ с помощью конструктора класса, а Linq, в свою очередь, позволяет это сделать, как с помощью конструктора класса, так и с помощью специального метода для создания документов.

Между методами замены элементов и их атрибутов есть различия в реализации, а конкретнее в Xml изменение элемента и его атрибутов (свойств) производится путем обращения к переменной описанной в классе Xml, что усложняет понимание, но упрощает работу с элементами и их атрибутами. В свою очередь в Linq изменение элемента и его атрибутов (свойств) производится путем обращения к методу описанному в классе Linq, что усложняет работу, но не требует досконального знания всего списка переменных в классе Linq.

Пространство имен Linq позволяет работать с Xml-документами, используя при этом меньшее количество строчек кода, а также расширено в своих возможностях (создание, удаление, добавление, изменение, вставка) работы с Xml-документами. Также Linq может выполнять запросы подобные по синтаксису с языком SQL, благодаря этому упрощаются способы и скорость доступа к данным записанным в Xml-документ.

Пространство имен Xml обладает классом TextWriter, с помощью которого может осуществляться создание Xml-документа и запись данных в него.

4.3. Вопросы для самоконтроля

1) С какими СУБД работает API Linq To Sql ?

а) Microsoft SQL Server

б) Oracle

в) MySQL

г) все вышеперечисленные

2) Как Linq to Sql абстрагирует физическую структуру базы данных?

а) в виде бизнес-объектов

б) в виде таблиц

в) в виде иерархической структуры

г) в виде реляционных отношений

3) Классом, устанавливающий соединение с базой данных, является

а) DataContext

б) SQLConnect

в) DataEntities

г) DataMember

4) Обновление базы данных осуществляется вызовом метода

а) SubmitChanges( )

б) ExecuteQuery( )

в) ExecuteTransaction( )

г) Close( )

5) Сущностные классы LinqToSql создаются

а) непосредственно программистом, либо средствами инструмента командной строки SQLMetal или Linq To Sql Designer

б) только программистом

в) утилитой командной строки SQL Metal

г) инструментом Object Relational Designer, входящим в состав Visual Studio 2010

6) Приведен код:

Northwind db = new Northwind(@”DataSource=.\SQLEXPRESS;InitialCatalog=Northwind”);

Customer cust = (from c in db.Customers where c.CustomerID == “LONEP” select c).Single<Customer>();

Вызовет ли его запуск немедленное выполнение запроса?

а) да, вызов операции Single() вызовет немедленное выполнение запроса

б) да, запросы Linq To Sql всегда выполняются немедленно

в) нет, запросы Linq To Sql всегда выполняются отложенно

г) нет, запросы с условием всегда выполняются во время чтения данных

7) Запросы Linq to Sql возвращают объект типа

а) IQueryable<T>

б) IEnumerable<T>

в) опеределенный самим программистом

г) Table<T>

8) Модель классов Linq To Sql содержит

а) модели таблиц, и ассоциации между ними

б) таблицы и ограничения

в) только таблицы

г) таблицы, триггеры, и хранимые процедуры

9) Приведен код:

```
Customer customer = (from c in db.Customers
where c.CompanyName == “Alfreds Futterkiste”
select c).Single<Customer>();
//db.OrderDetails.DeleteAllOnSubmit(
//        customer.Orders.SelectMany(o => o.OrderDetails));
//db.Orders.DeleteAllOnSubmit(customer.Orders);
db.Customers.DeleteOnSubmit(customer);
db.SubmitChanges();
```

Таблица Customers связа с таблицей Orders, а та в свою очередь с таблицей OrderDetails, вызовет ли выполнение этого кода удаление объектов, зависимых от родительского ?

а) нет, сначала нужно удалить все связанные данные из таблицы OrderDetailes, затем из Orders, и только потом можно удалить запись из таблицы Customers

б) да, все ассоциации и связанные данные удаляются автоматически – за этим следит механизм Linq To Sql

в) нет, после удаления данных из таблицы Customers нужно удалить данные из таблицы Orders, а затем из OrderDetails

г) да, об удалении соответствующих связанных данных позаботится СУБД

10) Переопределение методов вставки, обновления и удаления производится

а) программистом с помощью реализации частичных методов, либо с помощью Object Relational Designer

б) только программистом и только при определении классов Linq To Sql

в) только программистом, использую механизм частичных методов

г) утилитой командной строки SQL Metal.

СПИСОК ЛИТЕРАТУРЫ

Основная

1. Биллиг В. А. Основы программирования на C# : учеб. пособие / В.А. Биллиг. - М. : Интернет-ун-т информ. технологий : Бином. Лаб. знаний, 2006.- 483 с. АУЛ(6), АНЛ(1), Чз1(1)

2. Кариев Ч. А. Разработка Windows-приложений на основе Visual C# : учеб. пособие / Ч.А. Кариев. - М.: Интернет-Ун-т информ. технологий: БИНОМ. Лаб. знаний, 2007. - 767 с. + электрон. опт. диск (CD-ROM). АНЛ(1), Чз1(1)

3. Кариев Ч.А. Технология Microsoft ADO. NET: учеб. пособие / Ч.А. Кариев. - М. : Интернет-Ун-т информ. технологий : БИНОМ. Лаб. знаний, 2007.- 543 с. АУЛ(1), АНЛ(1), Чз1(1)

4. Клаус М. Язык программирования C#: Лекции и упражнения / М.Клаус.- СПб: ООО "ДиаСофтЮП", 2002.- 636 с.

5. Кузнецов С. Д. Базы данных: модели и языки : учеб. пособ. для студентов вузов, обучающихся по специальности " Прикладная математика и информатика" и "Информационные технологии" / С. Д. Кузнецов. - М. : Бином, 2008. - 720 с. Всего 50: АУЛ(49), АНЛ+Чз1(1)

6. Кулямин В. В. Технологии программирования. Компонентный подход: учеб. пособие / В. В. Кулямин. - М. : Интернет-ун-т информ. технологий : Бином. Лаб. знаний, 2007. - 463 с. АУЛ (3), АНЛ (1), Чз1 (1)

7. Мак-Дональд М. Microsoft ASP.NET 3.5 с примерами на C# 2008 для профессионалов / Мэтью Мак-Дональд, Марио Шпушта ; [пер. с англ. Я.П. Волковой и др.]. - 2-е изд. - Москва [и др.] : Вильямс, 2008. - 1420 с. + 1 электрон. опт. диск (CD-ROM). АНЛ (1), Чз1(1)

8. Марченко А. Л. Основы программирования на C# 2.0 : учеб. пособие / А.Л. Марченко.- Москва: Интернет-Ун-т информ. технологий: БИНОМ. Лаб. знаний, 2007.- 551 с. АУЛ(1), АНЛ(1), Чз1(1)

9. Прайс Д. Visual C# .NET: полное руководство / Д. Прайс, М. Гандэрлой. - Киев: Век+, 2011. - 957 с. Чз1(1)

10. Раттц-мл. Дж. С. LINQ: язык интегрированых запросов C#2008 для профессионалов / Джозеф Раттц-мл.- Москва: Вильямс, 2008. - 549 с. Чз1 (1).

11. Троелсен Э. C# и платформа .NET: Пер. с англ. / Э. Троелсен.- М. и др. : Питер, 2004. - 796 с. Чз1(1)

12. Фленов М.Е. Библия C#. - 3- е изд., перераб. и доп. / М.Е. Фленов.- СПб.: БХВ-Петербург, 2016.- 544 с.

13. Шарп Дж. Microsoft Visual C#. Подробное руководство. 8-е изд. / Джон Шарп.- СПб.: Питер, 2017. — 848 с.

14. Шилдт Г. C# 2.0 : полное руководство : классическое справочное руководство для версии языка C# 2.0, обновлен. и доп.: [пер. с англ.] / Г. Шилдт. - М. : ЭКОМ, 2007. - 961 с. Чз1(1)

15. Шилдт Г C# 4.0: полное руководство. – М.: ООО "И.Д. Вильяме", 2011. – 1056 с.

16. Эвери Дж. Microsoft ASP.NET : конфигурирование и настройка : [пер. с англ.] / Джеймс Эвери. - М. : БИНОМ. Лаб. знаний : СП ЭКОМ, 2005. - 269 с. АНЛ(1), Чз1(2)

Дополнительная

17. Евсеева О.Н., Шамшев А.Б. Основы языка C# 2005: Учебное пособие. - Ульяновск: УлГТУ, 2008. - 132 с. http://window.edu.ru/resource/863/58863

18. Евсеева О.Н., Шамшев А.Б. Работа с базами данных на языке C#. Технология ADO .NET: Учебное пособие. - Ульяновск: УлГТУ, 2009. - 170 с. http://window.edu.ru/resource/225/65225

19. Мак-Дональд М. Microsoft ASP.NET 4 с примерами на C# 2010 для профессионалов / Мак-Дональд М., Фримен Ад., Шпушта М. – М. : ООО "И.Д. Вильяме", 2011. – 1424 с.

20. Оньон Ф. Основы ASP.NET с примерами на C# / Оньон Ф.– М., 2003.– 304 с.

21. Петцольд Ч. Программирование для Microsoft Windows на C#: В 2-х т. / Ч.Петцольд.- М.: Издательско-торговый дом "Русская редакция", 2002.- 624 с.

22. Практикум по курсу "Объектно-ориентированное программирование" на языке C#: Учебное пособие / А.А. Андрианова, Л.Н. Исмагилов, Т.М. Мухтарова. - Казань: Казанский (Приволжский) федеральный университет, 2012. - 112 с. http://window.edu.ru/resource/949/79949

23. C# 4.0 и платформа .NET 4 для профессионалов / Кристиан Нейгел, Билл Ивьен, Джей Глинн, Карли Уотсон, Морган Скиннер.: Пер. с англ.- М. : ООО "И.Д. Вильяме", 2011. — 1440 с.

24. Скит Дж. C# для профессионалов. Тонкости программирования / Дж.Скит.- СПб.: Вильямс, 2014.- 608 с.

Информационные ресурсы

1. https://studfiles.net/preview/6211355/page:37/

2. http://nullpro.info/2013/samouchitel-po-c-dlya-nachinayushhix-01-osnovy-yazyka-peremennye-logika-cikly/

3. https://msdn.microsoft.com/ru-ru/library/bb399349(v=vs.110).aspx

4. http://codingcraft.ru/c_sharp_coding/auxiliary/linq.php

5. https://metanit.com/sharp/tutorial/15.1.php

УЧЕБНОЕ ИЗДАНИЕ

Рекомендовано Ученым советом
ГОУ ВПО «Донецкий национальный университет»
(протокол № 1 25.01.2019 г.)

СОВРЕМЕННЫЕ КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ

УЧЕБНОЕ ПОСОБИЕ

Издание второе

для студентов направления подготовки

01.04.02 Прикладная математика и информатика

Авторская верстка
Компьютерный дизайн: Е.В. Авдюшина

Адрес издательства:
ГОУ ВПО «Донецкий национальный университет»,
ул. Университетская, 24. г. Донецк, 283055

Подписано в печать 20.02.2019 г.
Формат 60×84/16. Бумага офисная.
Печать – цифровая. Усл.-печ. л. 4,5.
Тираж 100 экз. Заказ № 27 – март.19.
Донецкий национальный университет
283001, г. Донецк, ул. Университетская, 24.
Свидетельство про внесение субъекта
издательской деятельности в Государственный реестр
серия ДК № 1854 от 24.06.2004г.