ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ
КОНСПЕКТ ЛЕКЦИЙ

Тамбов 2009

ББК
УДК
Х

Утверждена на заседании кафедры прикладной информатики Тамбовского филиала ФГОУ ВПО МГУКИ «__» августа 2009 г., протокол № __

Составители: канд. техн. наук, доцент В.Н. Точка,

Основы алгоритмизации и программирования:конспект лекций для студентов, обучающихся по специальности 080801 «Прикладная информатика (в менеджменте) (дневной формы обучения)»:/ сост. В.Н. Точка. - Тамбов, 2009.- 100 с.

СОДЕРЖАНИЕ

1. ПЕРЕМЕННЫЕ И ВЫРАЖЕНИЯ

Компьютерная программа это последовательность операций над данными. Это утверждение оказывается справедливым даже для наиболее сложных случаев, таких, например, как развернутые, многофункциональные Windows-приложения типа Microsoft Office Suite. И хотя пользователи зачастую не задумываются над структурой приложений, внутренняя сущность последних заключается именно в этом.

Когда используется приложение, например, калькулятор, вводятся данные, представленные в форме чисел, и выполняются над этими числами различные операции почти так же, как с помощью карандаша и бумаги.

Если фундаментальное свойство компьютерных программ заключается в выполнении операций над данными, то это предполагает наличие, во-первых, некоторого способа хранения данных и, во-вторых, определенных методов для выполнения манипуляций над ними. Для обеспечения этих двух функций используются соответственно переменные и выражения.

Рассмотрим базовый синтаксис, используемый при программировании на C#.

1.1. Базовый синтаксис C#

Внешне код на C# очень напоминает код на C++ или на Java. Компиляторы C# не обращает внимание на лишнее пустое пространство, которое может состоять из пробелов, возвратов каретки или символов табуляции (эти символы известны под общим названием символов пустого пространства). Отсюда можно сделать вывод, что мы обладаем большой свободой в выборе способа форматирования нашей программы, хотя следование определенным правилам помогает создавать более удобные для чтения программы.

Код на C# представляет собой последовательность операторов, каждый из которых оканчивается точкой с запятой. Поскольку пустое пространство игнорируется, то можно располагать по несколько операторов в одной строке; однако для того чтобы сделать программу более наглядной, принято размещать символ возврата каретки после каждой точки с запятой, в результате чего на строке располагается только один оператор. Хотя абсолютно приемлемо (и совершенно нормально) использовать операторы, которые занимают несколько строк кода.

C# – язык, обладающий блочной структурой; другими словами, каждый оператор является частью некоторого блока кода. Эти блоки, для обозначения начала и конца которых используются фигурные скобки, соответственно, { и }, могут содержать произвольное количество операторов или не содержать их вовсе. Фигурные скобки не сопровождаются точками с запятой.

Простой блок программы на C# может иметь следующий вид:

```
{
        <строка кода 1, оператор 1>;
        <строка кода 2, оператор 2>
        <строка кода 3, оператор 2>;
}
```

Здесь выражения <строка кода x, оператор y> не являются действительными составляющими кода на C#: этот текст используется для обозначения места, куда впоследствии будут помещены операторы C#. В данном примере вторая и третья строки кода являются частями одного и того же оператора, поэтому после второй строки точка с запятой отсутствует.

В этом простом примере кода использованы отступы, чтобы сделать запись на C# более наглядной. Такой подход является обычной практикой; в действительности VS автоматически делает такие отступы по умолчанию. Каждый блок кода имеет свою величину отступов, определяющую, насколько далеко вправо он сдвинут. Блоки могут быть вложены друг в друга (т. е. блоки могут содержать в себе другие блоки), и в таком случае величина отступа у вложенных блоков должна быть больше:

```
{
    <строка кода 1>;
    {
        <строка кода 2>;
        <строка кода 3>;
    }
    <строка кода 4>;
}
```

Кроме того, больший отступ обычно используется для тех строк кода, которые являются продолжением предшествующей строки, как в случае строки 3 в первом из приведенных примеров.

Такой стиль не является обязательным. Однако, если его не использовать, то текст программы будет трудночитаемым.

Еще одна вещь, которая может часто встречаться в коде на C#, это комментарии. Комментарии не являются частью кода на C#, однако они сосуществуют с ним. Предназначение комментариев в точности соответствует их названию: они позволяют включать в программу описательный текст на обычном английском (французском, немецком, монгольском и т. д.) языке, который игнорируется компилятором. Когда приступают к созданию достаточно длинных сегментов кода, то бывает полезно включить в программу напоминание о том, что именно разработчики собираются сделать. В C# предусмотрены два способа включения комментариев. Можно использовать либо специальные маркеры, обозначающие начало и завершение комментария, либо маркер, который означает, что "все, что находится за ним до конца строки, является комментарием".

Для того чтобы обозначить комментарии по первому способу, используются сочетания символов /* в начале комментария и */ в конце комментария. Они могут располагаться как на одной строке, так и на разных строках; в последнем случае все строки, находящиеся между ними, также являются частью комментария. Единственное, что не может входить в состав комментария,– это сочетание символов */, поскольку оно будет интерпретировано как конец комментария. Допустимы следующие комментарии:

```
/* Это комментарий */
/* Так же, как . . .
 и это! */
```

А вот приведенный ниже пример ошибочен:

```
/* Сочетание символов "*/" часто используется для завершения комментария */
```

В этом случае последняя часть комментария (символы, расположенные после */) будет интерпретирована как код C#, что приведет к возникновению ошибки.

Другой подход к оформлению комментария состоит в использовании в качестве его начала символов //. В следующем за ними комментарии можно написать все, что угодно, но не более одной строки. Данный пример является правильным:

```
// Это еще один вид комментария.
```

А вот попытка воспользоваться следующим комментарием приведет к неудаче, поскольку вторая строка будет интерпретироваться как код на C#

```
// Это тоже комментарий,
а вот это уже нет.
```

Такой вид комментариев оказывается полезным для документирования отдельных операторов, поскольку весь комментарий целиком умещается на одной строке:

```
<Некоторый оператор>; // Объяснение данного оператора
```

В C# существует еще и третий тип комментариев, который позволяет создавать документацию для программы. Этот комментарий также расположен на одной строке, но в отличие от последнего случая начинается с трех символов /, а не с двух:

```
/// Это особый вид комментария
```

В обычных условиях эти комментарии игнорируются компилятором, как и все остальные, однако можно настроить VS таким образом, что текст этих комментариев при компиляции проекта будет извлекаться и использоваться для создания специальным образом отформатированного текстового файла. На его основе в дальнейшем можно создавать документацию по проекту.

Чрезвычайно важным моментом, который необходимо учитывать при работе с C#, является чувствительность этого языка к регистру. В отличие от некоторых других языков программирования, на C# текст необходимо вводить, в точности соблюдая регистр, поскольку простая замена заглавной буквы на строчную приведет к прекращению компиляции проекта.

Базовая структура консольного приложения на C#

Простейший код консольного приложения (Console Application) имеет следующий вид:

```
using System;
namespace ConsoleApplication {
/// <summary>
/// Summary description for Classl
/// </summary>
class Classl
{
static void Main (string[] args) {
//
// TODO: Add code to start application here //
static void Main(string[] args)
Console.WriteLine("The first application"); } } }
```

Рис. 1.1 – Простейшая программа вида Console Application

Нетрудно заметить, что в этом приложении присутствуют все синтаксические элементы, обсуждавшиеся выше, а именно, точки с запятыми, фигурные скобки, комментарии и отступы.

Наиболее важной частью кода является следующее:

```
static void Main(string[] args)
{
        // TODO: Add code to start application here
        // static void Main(string[] args)
        Console.WriteLine("The first app in Beginning C# Programming!");
}
```

Это тот заключенный в фигурные скобки блок кода, который и выполняется при запуске. Единственной строкой, выполняющей какие-либо действия, является строка, которую добавили к автоматически сгенерированному коду и которая – одна из всех – не является комментарием. Этот код выводит некоторый текст в консольном окне, а понимание того, каким образом выполнение приложения доходит до точки вызова Console.WriteLine(), а также механизм его работы сейчас не важен.

1.2. Переменные

Переменные тесно связаны с хранением данных. По существу, переменные в памяти компьютера можно представить в виде коробочек, лежащих на полке. Такие коробочки могут использоваться для того, чтобы что-нибудь в них класть, а затем вынимать обратно, или же можно просто заглянуть в них и узнать, есть там что-то или нет. Все это относится и к переменным, в которые мы можем помещать данные, а затем брать их оттуда или просматривать по мере необходимости.

Хотя все данные в компьютере фактически представляют собой одно и то же (последовательность, состоящую из нулей и единиц), переменные могут отличаться друг от друга, образуя так называемые типы. Возвращаясь к аналогии с коробочками, можно представить коробочки различных размеров и форм, причем разные объекты могут помещаться только в специально предназначенные для них коробочки. Обоснование системы типов заключается в том, что для манипуляций над данными различных типов могут требоваться различные методы, и, приписав переменным конкретные типы, можно избежать серьезной путаницы. Ведь вряд ли получится что-то осмысленное, если рассматривать последовательность нулей и единиц, представляющую собой оцифрованное изображение, в качестве звукового файла.

Для того чтобы использовать переменную, необходимо ее объявить. Это означает, что ей должны быть присвоены имя и тип. Как только переменная объявлена, она может использоваться как хранилище для данных того типа, который указан в ее объявлении.

Синтаксис объявления переменных в C# состоит из указания типа переменной и ее имени:

```
<ТИП> <ИМЯ>;
```

Если попытаться использовать переменную, которая не была объявлена, то программа не будет откомпилирована и будет сгенерирована соответствующая ошибка. Однако в этом случае компилятор сможет точно указать, в чем именно заключается проблема, и поэтому такой тип ошибок не очень опасен. Попытка использовать переменную, которой не присвоено никакого значения, тоже приведет к ошибке; в этом случае компилятор также сможет точно установить причину ее возникновения.

Существует бесконечное множество типов, которые можно использовать, так как имеется возможность описывать

свои собственные типы, предназначенные для хранения любых даже самых сложных данных.

Однако существуют определенные типы данных, которые раньше или позже потребуются всем, например, переменные, в которых могут храниться числа. По этой причине существует некоторое количество простых, предварительно описанных типов данных.

1.2.1. Простые типы переменных

Простые типы – это такие типы значений, как численные или логические (true (истина) или false (ложь)), которые являются составными частями любых приложений, а также других, более сложных типов. Большинство из имеющихся типов являются численными, что на первый взгляд выглядит несколько странно: разве для хранения чисел не достаточно одного типа?

Причина такого изобилия численных типов кроется в способе хранения чисел в памяти компьютера как последовательности нулей и единиц. В случае целых значений мы используем несколько бит (отдельных цифр, которые могут принимать значение нуля или единицы) и просто представляем число в двоичном формате. Переменная, в которой может храниться N бит, позволяет представлять любые числа в диапазоне от 0 до ($N^2 - 1$). Числа, превышающие последнее значение, оказываются слишком большими и не могут быть размещены в переменной данного типа.

В качестве примера рассмотрим переменную, в которой может храниться 2 бита. Соответствие между целыми числами и их двоичным представлением в этом случае будет следующим:

0 = 00 1 = 01 2 = 10 3 = 11

Если необходимо хранить большие числа, то для этого потребуется большее количество бит (3 бита, например, позволят хранить числа, лежащие в диапазоне от 0 до 7).

Отсюда неизбежно следует вывод о том, что хранение произвольного числа требует бесконечного количества бит, что, безусловно, неприемлемо для персонального компьютера. Но даже если бы необходимое количество бит и имелось, использовать такой объем памяти для числа, изменяющегося, скажем, от 0 до 10, было бы крайне неэффективно (из-за неиспользуемой памяти). В данном случае вполне достаточно 4 бит, которые позволяют разместить в одной и той же области памяти все числа указанного диапазона.

Именно для этого существует несколько целых типов, которые могут быть использованы для хранения чисел из различных диапазонов и которые занимают различный объем памяти (вплоть до 64 битов). Эти типы целых чисел сведены в таблицу, приводимую ниже.

Каждый тип использует один из стандартных типов, определенных в .NET Framework. Применение стандартных типов – это как раз то, что делает возможным совместное использование различных языков программирования. Имена, которые используются для этих типов в C#, обозначают определенные типы, описанные в .NET Framework. В таблице 1.1 приводятся имена типов и указывается, какому типу в библиотеке .NET Framework они соответствуют.

Буква "u", встречающаяся перед некоторыми из имен, является сокращением от слова "unsigned" ("беззнаковый"). Это говорит о том, что в переменных соответствующего типа не могут храниться отрицательные значения. Этот факт отражен в столбце "Допустимые значения".

Таблица 1.1 – Типы данных C# и соответствующие им типы .NET Framework

Тип	Тип .NET Framework	Допустимые значения
sbyte	System. Byte	Целое в диапазоне от -128 до 127
byte	System. Byte	Целое в диапазоне от 0 до 255
short	System.Intl6	Целое в диапазоне от -32 768 до 32 767
ushort	System.UIntl6	Целое в диапазоне от 0 до 65 535
int	System.Int32	Целое в диапазоне от -2^{31} до $2^{31}-1$
uint	System.UInt32	Целое в диапазоне от 0 до $2^{32}-1$
long	System.Int64	Целое в диапазоне от -2^{63} до $2^{63}-1$
ulong	System.UInt64	Целое в диапазоне от 0 до $2^{64}-1$

Кроме целых чисел часто требуется хранить дробные значения (или значения с плавающей запятой), которые представляют нецелые числа. Существует три типа переменных с плавающей запятой, которые можно использовать: float, double и decimal. Первые два из них хранят числа с плавающей запятой в виде $\pm m \cdot 2^e$, причем диапазон допустимых значений m и e для каждого типа свой. Тип decimal использует альтернативную форму представления чисел $\pm m \cdot 10^e$. Эти три типа приводятся ниже вместе с допустимыми значениями m и e и диапазонами действительных чисел (таб. 1.2).

Таблица 1.2 – Дробные типы данных C# и соответствующие им типы .NET Framework

Тип	Тип .NET Framework	Минимальное значение m	Максимальное значение m	Минимальное значение e	Максимальное значение e	Приблизительно минимальное число	Приблизительно максимальное число
float	System.Single	0	224	-149	104	$1.5 \cdot 10^{-45}$	$3.4 \cdot 10^{38}$
double	System.Double	0	253	-1075	970	$5 \cdot 10^{-324}$	$1.7 \cdot 10^{307}$
decimal	System.Decimal	0	296	-26	0	$1 \cdot 10^{-28}$	$7.9 \cdot 10^{28}$

Кроме численных типов, существует еще три простых типа (таб. 1.3).

Таблица 1.3 – Символьные и логические типы данных C# и соответствующие им типы данных .NET Framework

Тип	Тип .NET Framework	Диапазон допустимых значений
char	System.Char	Отдельный символ в кодировке Unicode, хранящийся в виде целого числа в диапазоне от 0 до 65 535
bool	System.Boolean	Логическое значение (Boolean), которое может принимать значения true (истина) или false (ложь)
string	System.String	Последовательность символов

Следует обратить внимание, что верхней границы количества символов, составляющих строки типа string, не существует, поскольку они могут занимать изменяющиеся объемы памяти.

Логический тип bool – один из наиболее часто используемых в C# типов переменной; аналогичные типы часто встречаются в кодах на других языках программирования. Использование переменной, которая может принимать только

два значения – либо true, либо false,– создает очень важную возможность ветвления логики приложения. Одним из применений такого типа данных - сравнение значений переменных и проверка допустимости введенной информации.

1.2.2. Именование переменных

Нельзя использовать в качестве имени переменной произвольную последовательность символов. Однако в распоряжении разработчика имеется весьма гибкая система именования.

Основные правила при именовании переменных следующие:

- первый символ в имени переменной должен быть либо буквой, либо символом подчеркивания _, либо @;
- последующими символами могут быть буквы, символ подчеркивания или цифры.

Кроме того, существуют определенные ключевые слова, которые обладают особым смыслом с точки зрения компилятора C#, например, using или namespace.

Если по ошибке используется одно из них, компилятор сгенерирует соответствующую ошибку при компиляции.

Следующие имена переменных являются правильными:

myBigVar
VAR1
_test

А вот эти – нет:

99BottlesOfBeer
namespace
It's-All-Over

C# чувствителен к регистру: необходимо постоянно помнить о правильном использовании регистров в соответствии с тем, как они использовались при объявлении переменных. Обращение к переменной внутри программы, в котором хотя бы один символ будет набран в неверном регистре, приведет к выдаче сообщения об ошибке в процессе компиляции.

Другим следствием этой ситуации является возможность иметь множество имен переменных, которые будут отличаться друг от друга только использованием регистра; например, приведенные ниже имена различны:

myVariable MyVariable MYVARIABLE

1.2.3. Соглашения, используемые при именовании

Имена переменных – это нечто такое, что приходится использовать очень часто. Поэтому стоит посвятить немного времени тому, какие типы имен следует применять. Однако прежде чем перейти к обсуждению этого вопроса, следует отметить, что данная тема полна противоречий. На протяжении многих лет одни системы именования сменялись другими, а некоторые разработчики готовы насмерть стоять за признание только собственных систем.

До недавнего времени наиболее популярной была система, известная под именем венгерской системы записи. Она предполагает использование перед именами всех переменных префикса, набираемого в нижнем регистре и обозначающего тип переменной. Например, если переменная имеет тип int, то перед ней следует поместить символ i, например, iAge. При использовании такой системы оказывается очень просто определять тип переменных даже при беглом взгляде на программу.

Однако в более современных языках, вроде C#, реализовать венгерскую систему оказывается совсем не просто. Эта система вполне может использоваться для простых типов, описанных выше (таб. 1.1-1.3), поскольку для обозначения каждого типа вполне достаточно префиксов, состоящих из одной или двух букв. Но поскольку имеется возможность создавать свои собственные типы и, кроме того, в базовой .NET Framework таких сложных типов присутствует несколько сотен, то система быстро перестает работать. Когда над проектом трудятся несколько человек, существенно возрастает вероятность того, что с использованием различных префиксов будет возникать путаница, которая может иметь самые разрушительные последствия.

В итоге разработчики пришли к выводу, что намного лучше именовать переменные просто в соответствии с их предназначением. Если при этом возникают какие-либо сомнения, то узнать тип переменной очень просто. В VS для этого надо просто поместить курсор мыши около имени переменной, и очень быстро появится окно, в котором будет указан ее тип.

В настоящее время в пространствах имен .NET Framework используются два соглашения об именовании, известные как PascalCase и camelCase. Отличить одну систему от другой можно по использованию регистра. Обе системы применяются к именам, составленным из нескольких слов; при этом каждое слово в составном имени набирается в нижнем регистре, за исключением первой буквы слова, которая должна набираться в верхнем регистре. В системе camelCase существует еще одно дополнительное правило, которое гласит, что первое слово должно начинаться с буквы в нижнем регистре.

Следующие имена переменных соответствуют правилам camelCase:

age
firstName
timeOfDeath

А вот так имена должны записываться в соответствии с правилами PascalCase:

Age
LastName
WinterOfDiscontent

Следует отметить, что система PascalCase соответствует рекомендациям компании Microsoft.

Необходимо отметить, что многие предыдущие системы именования широко использовали символ подчеркивания, обычно в качестве разделителя слов в имени переменной: например, my_first_variable. Однако, такой подход в настоящее время не рекомендуется.

1.2.4. Литеральные значения

Таблица 1.4 – Типы переменных и соответствующие им литеральные значения

Тип(ы)	Значение	Суффикс	Пример/Допустимые значения
bool	Логическое	Отсутствует	true или false
int, uint, long, ulong	Целое	Отсутствует	100
uint, ulong	Целое	u или U	100U
long, ulong	Целое	l или L	100L
ulong	Целое	ul, uL, Ul, UL, lu, lU, Lu или LU	100UL
float	Вещественное	f или F	1,5F
double	Вещественное	Отсутствует, d или D	1,5
decimal	Вещественное	m или M	1,5M
char	Символьное	Отсутствует	'a' или escape-последовательность
string	Строковое	Отсутствует	“a. . .a", может включать escape-последовательности

В предыдущем примере были использованы два литеральных значения: целое число и строка. Переменные других типов также обладают определенными литеральными значениями, которые сведены в приведенную ниже таблицу. Многие из них предполагают использование суффиксов, т. е. добавление в конец литерального значения некоторой последовательности символов, которая позволяет определить необходимый тип. Некоторые литеральные значения могут иметь несколько типов, которые определяются в время работы компилятора по контексту.

1.2.5. Литеральные строки

Приведем таблицу всех возможных escape-последовательностей для справочных целей.

В таблице 1.5 последний столбец содержит шестнадцатеричные значения кодов символов в кодировке Unicode.

Кроме приведенных выше символов, с помощью escape-последовательности можно задать любой символ в кодировке Unicode. Такая последовательность должна состоять из стандартного символа \, за которым следует символ u и четырехзначное шестнадцатеричное значение (например, те четыре цифры, которые расположены после символа x в последнем столбце вышеприведенной таблицы).

Отсюда следует, что, например, следующие две строки эквивалентны

```
"Karli\'s string."
"Karli\u0027s string."
```

Очевидно, что возможность использовать escape-последовательности в кодировке Unicode дает дополнительную гибкость.

Также можно применять "дословные" строки (verbatim). Это означает, что в строку включаются все символы, находящиеся между двумя двойными кавычками, в том числе символы конца строки и символы, для которых в противном случае потребовалось бы использование escape-последовательностей. Единственным исключением из этого правила является escape-последовательность для символа двойной кавычки, которая должна использоваться во избежание преждевременного завершения обработки строки. Чтобы получить дословную строку, необходимо поместить в ее начало символ @:

```
@"Verbatim string literal."
```

Эта строка может быть задана и обычным способом, а вот для следующего примера такой способ оказывается единственно возможным:

```
@"A short list
 item 1
 item 2"
```

Дословные строки оказываются полезными, в частности, для имен файлов, поскольку в них обычно присутствует большое количество обратных слэшей. Использование обычных строк привело бы к необходимости использования двойных обратных слэшей во всей строке. Например:

```
"C:\\Temp\\MyDir\\MyFile.doc"
```

Таблица 1.5 – Escape-последовательности, используемые в литеральных строках

Escape-последовательность	Выводимый символ	Код символа в кодировке Unicode
\'	Одиночная кавычка	0x0027
\"	Двойная кавычка	0x0022
\\	Обратный слэш	0x005C
\0	Null	0x0000
\a	Тревога (выдает звуковой сигнал)	0x0007
\b	Удаление предыдущего символа (backspace)	0x0008
\f	Формирование строки (form feed)	0x000C
\n	Новая строка	0x000A
\r	Возврат каретки	0x000D
\t	Символ горизонтальной табуляции	0x0009
\v	Символ вертикальной табуляции	0x000B

Используя дословные литеральные строки, мы можем сделать запись более удобочитаемой. Следующая дословная строка эквивалентна вышеприведенной:

```
@"C:\Temp\MyDir\MyFile.doc"
```

Строки представляют собой ссылочные типы, в отличие от всех остальных типов, которые были приведены выше и которые являются значимыми типами. Одним из следствий этого является возможность присвоить строковой переменной значение Null: это будет означать, что данная строковая переменная не ссылается ни на какую строку. Подробнее этот вопрос будет рассматриваться ниже.

1.2.6. Объявление переменных и присваивание им значений

Для объявления переменных требуется просто указать их тип и имя, например:

```
int age;
```

Далее ей присваивается значение с помощью оператора присваивания:

```
age = 25;
```

Перед тем как начать использовать переменную, нужно ее инициализировать. В качестве такой инициализации может использоваться приведенное выше присваивание.

Существуют еще два способа объявления переменных. Первый из них – это возможность одновременного объявления сразу нескольких переменных одного типа, чего можно добиться перечислением их имен через запятую после указания типа:

```
int xSize, ySize;
```

В данном случае обе переменные – xSize и ySize – описаны как целые числа.

Второй способ заключается в одновременном объявлении переменной и присваивании ей значения, что фактически означает объединение двух строк программы

```
int age = 25;
```

Существует возможность совместного использования этих двух способов:

```
int xSize = 4, ySize = 5;
```

В данном случае соответствующие значения присваиваются и переменной xSize, и переменной ySize.

1.3. Выражения

Для выполнения действий над переменными язык C# имеет целый набор операторов, в частности, оператор присваивания =. Записывая операторы в определенном сочетании с переменными и литеральными значениями (при использовании совместно с операторами они называются операндами), можно создавать выражения, которые являются основными строительными блоками компьютерных программ.

Существуют различные операторы, начиная от самых простых и до чрезвычайно сложных, которые редко встречаются где-либо, кроме как в математических приложениях. Простые операторы включают в себя все основные математические действия (например, оператор + производит сложение двух операндов), а сложные операторы

предназначены для выполнения манипуляций над двоичным представлением содержимого переменной. Существуют также логические операторы, работающие с логическими значениями, и операторы присваивания типа =.

Все математические операторы могут быть условно разделены на три категории:

- унарные операторы, которые выполняют действие над единственным операндом;
- бинарные операторы, которые выполняют действие над двумя операндами;
- тринарные операторы, которые выполняют действие над тремя операндами.

Большинство операторов попадает в категорию бинарных; существует также несколько унарных операторов и единственный тринарный, называемый условным оператором (условный оператор представляет собой логический оператор, т. е. возвращает логическое значение.

1.3.1. Математические операторы

Существует пять простых математических операторов, причем два из них имеют как бинарную, так и унарную форму. В приведенной ниже таблице 6 перечислены все математические операторы, даны элементарные примеры их использования и показаны результаты их работы с простыми численными типами (целыми числами и числами с плавающей запятой).

Здесь приведены примеры с использованием простых численных типов, так как результаты, получаемые при действиях с другими простыми типами, могут оказаться неясными. Действительно, что можно получить при сложении двух логических переменных? Ничего, поскольку компилятор не позволит использовать оператор + (а также любые другие

Таблица 1.6.1 – Математический оператор сложения строк

Оператор	Значение	Пример использования в выражении	Результат
+	Бинарное	varl = var2 + var3;	Переменной varl присваивается значение, которое представляет собой результат объединения двух строк, хранящихся в переменных var2 и var3

Таблица 1.7 – Операторы инкремента и декремента в C#

Оператор	Значение	Пример использования в выражении	Результат
++	Унарное	varl = ++var2;	Переменной varl присваивается значение переменной var2+1. Переменная var2 увеличивается на единицу
--	Унарное	varl = --var2;	Переменной varl присваивается значение переменной var2-l. Переменная var2 уменьшается на единицу
++	Унарное	varl = var2++;	Переменной varl присваивается значение переменной var2. Переменная var2 увеличивается на единицу
--	Унарное	varl = var2--;	Переменной varl присваивается значение переменной var2. Переменная var2 уменьшается на единицу

Таблица 1.6 – Математические операторы C#

Оператор	Значение	Пример использования в выражении	Результат
+	Бинарное	varl = var2 + var3;	Переменной varl присваивается значение, которое представляет собой сумму значений переменных var2 и var3
-	Бинарное	varl = var2 - var3;	Переменной varl присваивается значение, получающееся в результате вычитания значения переменной var3 из значения переменной var2
*	Бинарное	varl = var2 * var3;	Переменной varl присваивается значение, которое представляет собой результат перемножения переменных var2 и var3
/	Бинарное	varl = var2 / var3;	Переменной varl присваивается значение, которое представляет собой результат деления переменной var2 на переменную var3
%	Бинарное	varl = var2 % var3;	Переменной varl присваивается значение, которое представляет собой остаток от деления переменной var2 на переменную var3
+	Унарное	varl = +var2;	Переменной varl присваивается значение переменной var2
	Унарное	varl = -var2;	Переменной varl присваивается значение переменной var2, умноженное на -1

математические операторы) с переменными типа bool. Сложение переменных типа char также выглядит несколько запутанным, поскольку переменные типа char на самом деле хранятся в виде чисел, поэтому результатом сложения двух переменных типа char также будет число (типа int, если быть точным). Такое действие представляет собой пример неявного преобразования типов.

Необходимо отметить, что бинарный оператор + имеет смысл и тогда, когда он используется с переменными типа string. В этом случае предыдущая таблица должна быть дополнена следующим (табл. 6.1).

Никакие другие математические операторы к строкам применяться не могут.

Два других оператора, это операторы увеличения (инкремент) и уменьшения (декремент), каждый из которых является унарным оператором и может использоваться одним из двух способов: либо непосредственно перед операндом, либо сразу после него (табл. 1.7).

Эти операторы всегда приводят к изменению значения, хранящегося в соответствующем операнде. Оператор ++ всегда увеличивает значение своего операнда на 1, а оператор – уменьшает его на 1. Отличие заключается в том

результате, который записывается в переменную varl, поскольку местоположение оператора определяет порядок выполнения действий. Когда любой из этих двух операторов помещен перед операндом, он выполняется прежде каких-либо других вычислений. Размещение любого из этих двух операторов после операнда приводит к тому, что оператор выполняется после завершения всех остальных вычислений для данного выражения.

Рассмотрим следующий код:

```
int varl, var2 = 5, var3 = 6; varl = var2++ * --var3;
```

Вопрос: какое значение будет присвоено переменной varl? Перед тем как выражение начнет вычисляться, будет выполнен оператор --, расположенный перед переменной var3, что приведет к изменению ее значения с 6 на 5. Можно проигнорировать оператор ++, расположенный за переменной var2, поскольку он не будет оказывать никакого влияния до тех пор, пока вычисление этого выражения не будет завершено, поэтому в varl будет записан результат умножения 5 на 5, т. е. 25.

Эти простые унарные операторы оказываются чрезвычайно удобными в очень многих ситуациях. На самом деле они представляют собой просто сокращенную запись выражений вроде:

```
varl = varl + 1;
```

Такие выражения имеют очень широкое распространение, в частности, при организации циклов.

1.3.2. Операторы присваивания

До настоящего момента был использован простой оператор присваивания =, но существуют и другие операторы присваивания

Все другие операторы присваивания работают схожим с оператором = образом. Так же, как и у оператора =, результат их выполнения – присваивание расположенной слева переменной значения, вычисленного посредством операторов и операндов, расположенных в правой части.

Результатом использования дополнительных операторов присваивания является вовлечение переменной varl в вычисления, т. е. код вроде.

```
varl += var2;
```

приводит к получению точно такого же результата, что и код:

```
varl = varl + var2;
```

Оператор += может использоваться и со строками, так же, как и оператор +.

Использование этих операторов, особенно при работе с длинными именами переменных, позволяет сделать программу намного более удобочитаемой.

1.3.3. Старшинство операторов

При вычислении выражения все операторы выполняются в определенной последовательности. Это не означает, что они всегда выполняются слева направо. В качестве самого простого примера рассмотрим следующий оператор

```
varl = var2 + var3;
```

Здесь оператор + выполняется перед оператором =.

Таблица 1.8 – Операторы присваивания

Оператор	Значение	Пример использования в выражении	Результат
	Бинарное	varl = var2;	Переменной varl присваивается значение переменной var2
+=	Бинарное	varl += var2;	Переменной varl присваивается значение, которое представляет собой сумму переменных varl и var2
-=	Бинарное	varl -= var2;	Переменной varl присваивается значение, которое представляет собой разность, полученную в результате вычитания переменной var2 из переменной varl
*=	Бинарное	varl *= var2;	Переменной varl присваивается значение, которое представляет собой произведение переменных varl и var2
/=	Бинарное	varl /= var2;	Переменной varl присваивается значение, полученное в результате деления переменной varl на переменную var2
%=	Бинарное	varl %= var2;	Переменной varl присваивается значение, являющееся остатком от деления переменной varl на переменную var2

Существуют и другие ситуации, когда старшинство операторов не совсем очевидно, например:

```
varl = var2 + var3 * var4;
```

В этом выражении сначала будет выполнен оператор *, затем оператор + и последним – оператор =. Это принятый в математике стандартный порядок выполнения операций, он приведет к получению того же самого результата, который можно ожидать при выполнении аналогичных арифметических вычислений на бумаге.

Можно также управлять порядком выполнения операторов с помощью скобок, например:

```
varl = (var2 + var3) * var4;
```

В данном случае в первую очередь будет выполняться то, что содержится в скобках, то есть оператор + будет выполнен перед оператором *.

Для всех рассматриваемых операторов порядок выполнения показан ниже; при этом операторы с одинаковым старшинством (такие как * и /) выполняются в порядке своего следования слева направо.

Таблица 1.9 – Старшинство операторов

Старшинство	Операторы
Высшее	++, -- (используемые в качестве префиксов) , +, - (унарные) *, /, % , =, *=, /=, %=, +=, - =
Низшее	++, -- (используемые в качестве суффиксов)

Для изменения порядка выполнения операторов используются скобки, как говорилось выше.

1.4. Пространства имен

Пространства имен - это способ, используемый в .NET, который позволяет создавать контейнеры для кода приложений, чтобы и код, и его составные части были однозначно идентифицированы.

Пространства имен используются также в качестве средства категоризации объектов в .NET Framework. Большинство таких объектов представляют собой определения типов, например, определения простых типов.

Код C# по умолчанию содержится в глобальном пространстве имен. Это означает, что к объектам в коде C# можно обратиться из любого другого кода в глобальном пространстве имен просто по их имени. Можно воспользоваться ключевым словом namespace для того, чтобы явно задать пространство имен для любого блока кода, заключенного в

фигурные скобки. Имена, находящиеся в таком пространстве имен, если к ним обращаются не из данного пространства имен, должны квалифицироваться.

Квалифицированным именем называется имя, в котором содержится вся информация касательно его иерархии. Это означает, что если имеется код, находящийся в одном пространстве имен, и необходимо воспользоваться именем, определенным в другом пространстве имен, то следует использовать ссылку на это пространство имен. В квалифицированных именах для разделения уровней пространств имен используется символ точки.

Например:

```
namespace LevelOne {
    // программа, находящаяся в пространстве имен LevelOne
    //в ней описывается имя "NameOne"
}
// программа, находящаяся в глобальном пространстве имен
```

В этой программе описывается единственное пространство имен – LevelOne. В данном случае в программе не содержится никакого исполняемого кода. Это сделано для того, чтобы придать обсуждению максимально общий характер; напротив, в код помещен комментарий, в котором содержится это описание. Код, содержащийся внутри пространства имен LevelOne, может просто ссылаться на имя NameOne, и никакой классификации не требуется. Однако в коде, находящемся в глобальном пространстве имен, необходимо использовать классифицированное имя LevelOne. NameOne для того, чтобы на него сослаться.

Внутри любого пространства имен можно описывать вложенные пространства имен, используя то же самое ключевое слово namespace. При обращению к вложенным пространствам имен следует указывать всю их иерархию, отделяя один уровень иерархии от другого с помощью точки. Рассмотрим следующие пространства имен:

```
namespace LevelOne {
    // программа, находящаяся в пространстве имен LevelOne
namespace LevelTwo {
    // программа, находящаяся в пространстве имен LevelOne.LevelTwo
//в ней описывается имя "NameTwo" }
}
    // программа, находящаяся в глобальном пространстве имен
```

В данном случае обращение к имени NameTwo из глобального пространства имен должно иметь вид LevelOne.LevelTwo.NameTwo, из пространства имен LevelOne – LevelTwo.NameTwo, а из пространства имен LevelOne.LevelTwo – NameTwo.

Имена идентифицируются пространствами имен уникальным образом. Можно описать одно и то же имя NameThree как в пространстве имен LevelOne, так и в пространстве имен LevelTwo:

```
namespace LevelOne {
    // здесь описывается имя "NameThree"
namespace LevelTwo {
    // здесь описывается имя "NameThree" }
}
```

В данном случае описаны два разных имени, которые могут использоваться независимо друг от друга - LevelOne.NameThree и LevelOne.LevelTwo.NameThree.

После того, как пространство имен определено, появляется возможность использовать оператор using для упрощения доступа к содержащимся в нем именам. Оператор using как бы говорит: "Да, нам действительно понадобится доступ к именам из данного пространства имен, поэтому не заставляйте меня классифицировать их каждый раз". Например, в следующей программе предполагается, что код, находящийся в пространстве имен LevelOne, должен иметь доступ к именам пространства имен LevelOne.LevelTwo без какой бы то ни было классификации:

```
namespace LevelOne
{
    using LevelTwo;
    namespace LevelTwo {
            // здесь описывается имя "NameTwo"
    }
}
```

Код, находящийся в пространстве имен LevelOne, теперь может обращаться к LevelTwo.NameTwo просто как к NameTwo.

Бывают случаи, как в примере с NameThree, когда такой подход может приводить к конфликту между именами, находящимися в различных пространствах имен (при этом код, скорее всего, компилироваться не будет). В подобных случаях можно задать для пространства имен подставное имя (alias) в операторе using:

```
namespace LevelOne {
using LT = LevelTwo;
// здесь описывается имя "NameThree"
namespace LevelTwo {
// здесь описывается имя "NameThree" } }
```

Тогда в пространстве имен LevelOne мы сможем обращаться к LevelOne.NameThree просто как к NameThree, а к LevelOne.LevelTwo.NameThree как к LT.NameThree.

Операторы using оказывают влияние на то пространство имен, в котором они находятся, а также на все вложенные пространства имен, которые могут содержаться в данном пространстве имен. В вышеприведенном коде в глобальном пространстве имен использовать LT.NameThree нельзя. Однако если оператор using расположить следующим образом:

```
using LT = LevelOne.LevelTwo;
namespace LevelOne {
    // здесь описывается имя "NameThree
    namespace LevelTwo
    {
        // здесь описывается имя 'NameThree'
    }
}
```

то использовать имя LT.NameThree можно будет и из глобального пространства имен, и из пространства имен LevelOne.

Оператор using сам по себе не обеспечивает доступа к именам, находящимся в других пространствах имен. До тех пор, пока код из пространства имен не будет каким-либо способом привязан к проекту (например, описан в исходном файле проекта или описан в каком-либо коде), привязанному к этому проекту, получить доступ к содержащимся в нем именам будет невозможно. Более того, если код, в котором содержится некое пространство имен, привязан к проекту, то

имеется доступ к содержащимся в нем именам независимо от использования оператора using. Оператор using всего лишь упрощает обращение к этим именам и позволяет сократить сильно удлиняющийся в противном случае код, делая его более понятным.

Вернувшись к программе Console Application (рис.1) можно обнаружить там следующие строки кода, относящиеся к пространствам имен:

```
using System;
namespace ConsoleApplication {
}
```

В первой строке располагается оператор using, в котором объявляется, что пространство имен System будет использоваться во всей программе C# и доступ к нему из любых пространств имен данного файла должен осуществляться без использования классификации. Пространство имен System является корневым пространством имен в .NET Framework и включает все основные функциональные возможности, для консольных приложений.

1.5. Вопросы для повторения

1. Что такое атомарные переменные?
2. Перечислите основные типы переменных, используемых в C#.
3. Приведите общий вид программы консольного приложения.
4. Какие правила используются при именовании переменных?
5. Что такое пространства имен?
6. Приведите математические операторы, используемые в языке C#.
7. Что такое старшинство операторов?

2. УПРАВЛЕНИЕ ПОРЯДКОМ ВЫПОЛНЕНИЯ ПРОГРАММЫ

У всех программ на C#, которые рассматривались до сих пор, имеется одна общая черта. Во всех случаях выполнение кода происходило последовательно сверху вниз, строка за строкой, без каких-либо пропусков. Однако если бы приложения всегда выполнялись таким образом, то возможности разработчиков были бы сильно ограничены.

Существуют два способа управления порядком выполнения программ, т. е. последовательностью выполняемых в программе на C# строк.

Ветвление – выполнение кода обусловлено результатами предшествующих вычислений, например: "выполнить этот код только в том случае, если значение переменной myVal меньше 10".

Использование циклов – повторяющееся выполнение одних и тех же операторов (определенное количество раз или до тех пор, пока не выполнится контрольное условие).

Оба способа предполагают использование булевой логики.

2.1. Булева логика

Тип bool, может принимать одно из двух значений: true (истина) или false (ложь). Этот тип очень часто применяется для записи результата выполнения какой-либо операции, с тем, чтобы мы могли выполнить какие-либо определяемые им действия. В частности, тип bool используется для хранения результатов сравнения.

В качестве примера рассмотрим ситуацию, когда выполнять код нужно только при значении переменной myVal меньше 10. Для этого необходимо владеть некоторым способом, позволяющим определять, является ли утверждение "myVal меньше 10" правдой или ложью, т. е. необходимо получить результат логического сравнения.

Логическое сравнение требует применения логических операторов сравнения (также известных под именем операторов отношения), которые приведены в таблице 2.1. В этой таблице переменная var1 во всех случаях имеет тип bool, а типы переменных var2 и var3 меняются.

Таблица 2.1 - Логические операторы сравнения

Оператор	Тип	Пример выражения	Результат
==	Бинарный	varl = var2 == var3;	Переменной varl присваивается значение true в том случае, если переменная var2 равна переменной var3, и значение false в противном случае.
!=	Бинарный	varl = var2 != var3;	Переменной varl присваивается значение true в том случае, если переменная var2 не равна переменной var3, и значение false в противном случае.
<	Бинарный	varl = var2 < var3;	Переменной varl присваивается значение true в том случае, если переменная var2 меньше переменной var3, и значение false в противном случае.
>	Бинарный	varl = var2 > var3;	Переменной varl присваивается значение true в том случае, если переменная var2 больше переменной var3, и значение false в противном случае.
<=	Бинарный	varl = var2 <= var3;	Переменной varl присваивается значение true в том случае, если переменная var2 меньше или равна переменной var3, и значение false в противном случае.
>=	Бинарный	varl = var2 >= var3;	Переменной varl присваивается значение true в том случае, если переменная var2 больше или равна переменной var3, и значение false в противном случае.

В программе можно использовать эти операторы для численных значений следующим образом:

```
bool isLessThan10;
isLessThan10 = myVal <10;
```

В результате выполнения этого кода переменной isLessThan10 будет присвоено значение true в том случае, если значение переменной myVal меньше 10, а в противном случае – значение false.

Операторы сравнения могут использоваться также и для переменных других типов, например, для строк:

```
bool isKarli;
isKarli = myString = = “Karli”
```

В данном случае переменной isKarli будет присвоено значение true только при условии, что строка, содержащаяся в переменной myString, имеет вид "Karli"

Аналогичным образом можно поступать и с логическими переменными:

```
bool isTrue;
isTrue = myBool= = true;
```

Однако в этом случае допускается использование только операторов = = и !=.

Распространенной ошибкой является безосновательное предположение, что если val1 < val2 есть ложь, то val1 > val2 – истина. Если val1 = = val2, оба предыдущих выражения будут ложными.

Существуют и некоторые другие операторы, специально предназначенные для работы с логическими значениями (см. таб. 2.2).

Теперь можно записать последний пример следующим образом:

```
bool isTrue;
isTrue = myBool & true;
```

Для операторов & и | существуют аналоги (табл. 2.3).

Таблица 2.2 – Операторы для работы с логическими значениями

Оператор	Тип	Пример выражения	Результат
!	Унарный	varl = ! var2;	Переменной varl присваивается значение true в том случае, если переменная var2 имеет значение false, и значение false, если значение var2 – true (операция "логическое не" (NOT)).
&	Бинарный	varl = var2 & var3;	Переменной varl присваивается значение true в том случае, если обе переменные var2 и var3 имеют значение true, и значение false в противном случае (операция "логическое и" (AND)).
|	Бинарный	varl = var2 | var3;	Переменной varl присваивается значение true в том случае, если хотя бы одна из переменных – var2 или var3 (или обе) – имеет значение true, и значение false в противном случае (операция "логическое или" (OR)).
^	Бинарный	varl = var2 ^ var3;	Переменной varl присваивается значение true в том случае, если одна из переменных – var2 или var3, но не обе одновременно – имеет значение true, и значение false в противном случае (операция "исключающее или" (XOR)).

Таблица 2.3 - Аналоги логических операторов & и |

Оператор	Тип	Пример выражения	Результат
&&	Бинарный	varl = var2 && var3;	Переменной varl присваивается значение true в том случае, если обе переменные var2 и var3 имеют значение true, и значение false в противном случае (операция "логическое и" (AND)).
||	Бинарный	varl = var2 || var3;	Переменной varl присваивается значение true в том случае, если одна из переменных, var2 или var3 (или обе), имеет значение true, и значение false в противном случае (операция "логическое или" (OR)).

Результат выполнения этих операций в точности соответствует результату выполнения операций & и |, однако существует одно важное отличие; оно заключается в способе достижения результата, позволяющем в некоторых случаях ускорить операцию. Оба эти оператора рассматривают значение своего первого операнда (в приведенной выше таблице это var2) и в определенных случаях могут вообще обходиться без обработки второго операнда (в нашем примере var3).

Если значение первого операнда оператора && false, то нет необходимости рассматривать значение второго операнда, поскольку результатом все равно будет значение false. Аналогично оператор || вернет значение true, если его первый операнд имеет значение true, независимо от значения второго операнда. При выполнении операторов & и | всегда происходит вычисление обоих операндов.

С учетом этого при использовании операторов && и || вместо & и | можно ожидать небольшого увеличения быстродействия. Выгода будет особенно ощутима в тех приложениях, где задействовано много таких операторов. Существует эмпирическое правило, согласно которому при наличии возможности всегда следует использовать операторы && и ||.

2.2. Операторы работы с битами

По прочтении предыдущего раздела может возникнуть вопрос, для чего вообще существуют операторы & и |. Ответ заключается в том, что эти операторы могут использоваться для выполнения операций над численными значениями. Фактически они имеют дело с хранящейся в переменной последовательностью битов, а не со значением переменной как таковым.

Таблица 2.4- Битовая операция &

Бит операнда 1	Бит операнда 2	Бит результата выполнения операции &
1	1	1
1	0	0
0	1	0
0	0	0

Рассмотрим работу этих операторов, начав с оператора &. Он действует следующим образом: каждый бит первого операнда сравнивается с находящимся в такой же позиции битом второго операнда, а бит, записывающийся в аналогичной позиции результата, определяется следующим образом (см. табл. 2.4).

Аналогичные действия выполняются и в случае применения оператора |, только результат вычисляется несколько иначе (см. табл. 2.5).

В качестве примера рассмотрим операцию, выполняемую следующим кодом:

```
int result, op1, op2;
op1 = 4;
op2 = 5;
result = opl & op2;
```

Здесь необходимо принять во внимание двоичное представление операндов op1 и op2, соответственно 100 и 101. Результат этой операции получается путем сравнения двоичных битов, находящихся в одинаковых позициях в каждом из этих логических представлений:

Таблица 2.6 - Битовая операция &

Бит операнда 1	Бит операнда 2	Бит результата выполнения операции
1	1	0
1	0	1
0	1	1
0	0	0

- первому слева биту результата присваивается значение 1, если оба крайних левых бита op1 и op2 равны 1, и значение 0 в противном случае;
- следующему биту результата присваивается значение 1, если оба следующих бита op1 и op2 равны 1, и значение

Таблица 2.7 – Логический оператор ~

Бит операнда	Бит результата выполнения операции
1	0
0	1

Таблица 2.8 – Трехбитовое представление цвета

Биты	Десятичное представление	Значение
000	0	Черный
100	4	Красный
010	2	Зеленый
001	1	Синий
101	5	Пурпурный
по	6	Желтый
011	3	Голубой
111	7	Белый

– так продолжается для всех остальных битов.

В данном примере крайние биты слева у обоих операндов op1 и op2 равны 1, поэтому крайний левый бит результата также будет равен 1. В следующей позиции оба бита равны 0, а в третьей позиции биты равны соответственно 1 и 0; таким образом, второй и третий биты результата будут равны 0. В качестве окончательного результата в двоичном представлении будет получено число 100, т. е. результат выполнения этой операции будет равен 4.

Аналогичный процесс происходит при использовании оператора |, с тем лишь отличием, что получающийся в результате бит равняется 1 в том случае, если хотя бы один из соответствующих битов операндов равен 1.

Точно таким же способом можно использовать и оператор ^. В этом случае в результате появляется 1, если только у одного из операндов, но не у обоих одновременно, в соответствующей позиции находится единица.

В языке C# допускается использование унарного оператора работы с битами ~, воздействие которого на операнд состоит в инвертировании (изменении) всех его битов таким образом, что все биты, которые были равны 0, получают значение 1, и наоборот (см. табл. 2.7).

Такие побитовые операции оказываются весьма полезными в самых разных ситуациях, поскольку они предлагают простой метод использования отдельных битов переменной для хранения информации. Рассмотрим простое представление цвета, в котором используется три бита для указания наличия красной, зеленой и синей составляющих. У нас появляется возможность задавать каждый из этих битов независимо и изменить цвет, определяемый всеми тремя битами, на один из следующих вариантов (см. табл. 2.8).

Предположим, что мы храним эти значения в переменной типа int. Начиная с черного цвета или, что то же самое, со значения переменной типа int, равного 0, можно выполнить различные операции. Например:

```
int myColor = 0;
bool containsRed;
myColor = myColor | 2;
// Добавлен бит зеленого цвета, значение переменной myColor стало 010
myColor = myColor | 4;
// Добавлен бит красного цвета, значение переменной myColor стало 110
containsRed = (myColor & 4) = = 4;
// Выполняется проверка на наличие бита, отвечающего за красный цвет
```

В результате выполнения последней строки кода значение переменной containsRed будет true, поскольку "красный" бит в переменной myColor равен 1.

Этот способ может оказаться весьма полезным в том случае, если выполняемые операции используются для одновременной проверки нескольких битов (например, для 32 в случае значений типа int). Однако существуют более эффективные способы хранения дополнительной информации в отдельных переменных, для чего используются переменные сложных типов.

Помимо описанных выше четырех операторов для работы с битами следует рассмотреть еще два оператора (табл. 2.9).

Таблица 2.9 – Операторы побитового сдвига

Оператор	Тип	Пример выражения	Результат
>>	Бинарный	varl = var2 >> var3;	Переменной varl присваивается значение, которое получается при сдвиге двоичного содержимого переменной var2 вправо на число битов, равное значению переменной var3.
<<	Бинарный	varl = var2 << var3;	Переменной varl присваивается значение, которое получается при сдвиге двоичного содержимого переменной var2 влево на число битов, равное значению переменной var3.

Работу этих операторов, обычно называемых операторами побитового сдвига, лучше всего проиллюстрировать с помощью небольшого примера:

```
int varl, var2 =10, var3 = 2;
var1 = var2 << var3;
```

В этом примере переменной varl будет присвоено значение 40. Действительно, двоичное представление числа 10 – "1010"; после сдвига на две позиции влево оно превратится в "101000" – двоичное представление числа 40. То, что мы сделали, можно изобразить как операцию умножения. Каждый бит при сдвиге на одну позицию влево увеличивает свое значение в два раза, соответственно сдвиг на две позиции эквивалентен умножению на 4. И наоборот, при сдвиге всех битов вправо возникает эффект деления операнда на два, при этом любой целый остаток теряется:

```
int varl, var2 =10;
varl = var2 >> 1;
```

В данном случае varl будет содержать значение 5, а в результате выполнения следующего примера ее значение станет равным 2:

```
int varl, var2 = 10;
varl = var2 >>2;
```

Маловероятно, что в настоящее время активно пользуются этими операторами, однако знать об их существовании полезно. Их применяют главным образом для получения предельно оптимизированного кода, когда использование других математических операций из-за их большей затратности оказывается неприемлемым. По этой причине рассмотренные здесь операции часто задействуют, например, в драйверах устройств или системных кодах.

2.2.1. Логические операторы присваивания

Последние операторы - это комбинация рассмотренных выше операторов с оператором присваивания. Они во многом сходны с математическими операторами присваивания, рассматривавшимися в предыдущей лекции (+ = , *= и т.д.). Логические операторы приведены в таблице 2.10.

Таблица 2.10 – Логические операторы присваивания

Оператор	Тип	Пример выражения	Результат
&=	Бинарный	varl &= var2;	Переменной varl присваивается значение, являющееся результатом выполнения операции varl & var2.
|=	Бинарный	var1 |= var2;	Переменной varl присваивается значение, являющееся результатом выполнения операции varl I var2.
^=	Бинарный	varl ^= var2;	Переменной varl присваивается значение, являющееся результатом выполнения операции varl * var2.

Эти операторы используются как с логическими, так и с численными значениями, точно так же, как и операторы &, | и ^.

При выполнении операторов &= и |= применяются операторы & и |, а не && и ||, что приводит к издержкам, характерным для этих более простых операторов.

Для операторов побитового сдвига также существуют соответствующие операторы присваивания (табл. 2.11).

Таблица 2.11 – Операторы побитового присваивания

Оператор	Тип	Пример выражения	Результат
>>=	Унарный	varl >> var2;	Переменной varl присваивается значение, которое получается в результате сдвига двоичного содержимого переменной varl вправо на число битов, равное значению переменной var2.
<<=	Унарный	varl << var2;	Переменной varl присваивается значение, которое получается в результате сдвига двоичного содержимого переменной varl влево на число битов, равное значению переменной var2.

2.2.2. Старшинство операторов с дополнениями

Старшинство операторов приведено в таблице 2.12.

Таблица 2.21 – Старшинство операторов

Старшинство	Операторы
Высшее	++, - - (используемые в качестве префиксов), (), + , - (унарные), !, ~ *, /, % +, -, <<, >> <, >, <=, >=, ==, !=, &, ^, |, &&, ||, =, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=
Низшее	++, - - (используемые в качестве суффиксов)

Теперь она точно определяет, каким образом следует вычислять выражения, например:

```
varl = var2 <= 4 && var2 >= 2;
```

в котором оператор && выполняется после операторов <= и >=.

Здесь необходимо отметить тот факт, что совершенно не возбраняется использовать скобки с целью сделать выражения более понятными. Компилятор в любом случае разберется, в каком порядке следует выполнять операторы, а вот людям свойственно забывать о таких вещах. Записав вышеприведенное выражение следующим образом:

```
varl = (var2 <= 4) && (var2 >= 2);
```

можно решить эту проблему, явно указав последовательность выполнения операций.

2.2.3. Оператор goto

C# позволяет помечать строки кода, а затем непосредственно переходить к их выполнению с помощью оператора goto. У этого оператора имеются свои преимущества и недостатки Основное преимущество заключается в том, что это очень простой способ управлять очередностью выполнения кодов. Однако чрезмерное его использование может привести к трудному для понимания "лоскутному" коду, что является его главным недостатком.

Для лучшего понимания рассмотрим, каким образом может использоваться этот оператор.

Оператор goto имеет вид:

```
goto <имяМетки>;
```

Метки задаются таким образом:

```
<имяМетки>:
```

В качестве примера разберем следующий код:

```
int myInteger = 5;
goto myLabel;
mylnteger += 10;
myLabel:
Console.WriteLine("myInteger = {0}", myInteger);
```

Его выполнение осуществляется следующим образом:

- переменная myInteger объявляется как имеющая тип int и ей присваивается значение 5;
- оператор goto прерывает нормальный ход выполнения кода и передает управление на строку с меткой myLabel:;
- значение переменной myInteger выводится на консоль.

Строка кода, выделенная в приведенной выше примере программы курсивом, никогда не будет выполняться.

Если попытаться использовать этот код в приложении, то при попытке его откомпилировать появится предупреждение "Unreachable code detected" ("Обнаружен невыполняемый код") с номером соответствующей строки.

Операторы goto используются на практике, но они могут приводить к очень большой путанице.

В качестве примера "лоскутного" кода, получающегося при использовании оператора goto, рассмотрим следующую программу

```
start:
int myInteger = 5;
goto addVal;
writeResult: Console.WriteLine("myInteger = {0}", myInteger);
goto start;
addVal: myInteger +=10;
goto writeResult;
```

Это вполне допустимый код, однако в нем чрезвычайно трудно разобраться.

Оператор goto предназначен для совместного использования с некоторыми другими конструкциями, рассматриваемыми в настоящей лекции.

2.3. Ветвление

Ветвление – это способ управлять тем, какая строка кода должна выполняться следующей. Строка, на которую следует передать управление, определяется с помощью некоторого условного оператора. Работа такого условного оператора основана на выполнении сравнения контрольного значения и одного или нескольких возможных значений с использованием булевой логики.

Существует три способа ветвления, имеющихся в C#:

- тринарный оператор;
- оператор if;
- оператор switch.

2.3.1. Тринарный оператор

Наиболее простой способ выполнения проверки – использование тринарного (или условного) оператора. Синтаксис тринарного оператора имеет следующий вид:

<проверка> ? <результатЕслиИстина> : <результатЕслиЛожь>

В данном случае условие <проверка> вычисляется для получения логического значения, и в зависимости от него результатом выполнения этого оператора будет <результатЕслиИстина> или <результатЕслиЛожь>.

Можно использовать этот оператор, как показано на рис. 2.1.

```
string resultString = (myInteger < 10) ? “Меньше 10”: “Больше или равно 10”;
```

Рис. 2.1 – Пример использования тринарного оператора

Результатом выполнения тринарного оператора будет одна из двух строк, каждая из которых может быть присвоена переменной resultString. Выбор того, какую именно из этих двух строк следует присвоить, производится путем сравнения значения переменной myInteger с 10, причем если значение переменной меньше 10, присваивается первая строка, а если больше или равно 10 – вторая. Например, в том случае, если значение переменной myInteger равняется 4, переменной resultString будет присвоена строка “Меньше 10”.

Описанный оператор вполне подходит для выполнения простых присваиваний, как в приведенном примере, однако для выполнения больших объемов кода он не приспособлен. Здесь лучше использовать оператор if.

2.3.2. Оператор if

Оператор if является гораздо более гибким способом принятия решений. В отличие от операторов ?:, операторы if не имеют результата (поэтому их нельзя использовать в операторах присваивания), зато появляется возможность применять их для выполнения каких-либо других операторов при соблюдении определенного условия.

Простейшей формой оператора if является следующая:

```
if (<условие>)
    <код, который выполняется, если вычисление <условия> дает результат true>;
```

Когда происходит вычисление <условие> (а в результате этих вычислений должно получаться логическое значение, иначе код не пройдет компиляцию), строка кода, расположенная под оператором, будет выполнятся только в случае значения true. После того как эта строка кода будет выполнена, а также если эта строка не будет выполняться (из-за того, что получено значение false), работа продолжится со следующей за условным оператором строки кода.

Можно задать дополнительный код, воспользовавшись оператором else в сочетании с оператором if. Этот оператор будет выполняться только в том случае, если в результате вычисления <условия> будет получено значение false.

```
if (<условие>)
    <код, который выполняется, если вычисление <условия> дает результат true>;
else
    <код, который выполняется, если вычисление <условия> дает результат false>;
```

Каждый раздел кода может занимать несколько строк, для чего используются фигурные скобки:

```
if (<условие>)
{
    <код, который выполняется, если вычисление <условия> дает результат true>;
}
else
{
    <код, который выполняется, если вычисление <условия> дает результат false>;
}
```

В качестве простого примера перепишем код (рис.2), в котором использовался тринарный оператор, с помощью оператора if:

```
string resultString = (myInteger < 10) ? “Меньше 10” : “Больше или равно 10”;.
```

Поскольку результат выполнения оператора if нельзя присваивать переменным, то потребуется отдельный шаг

```
string resultString;
if (myInteger < 10)
    resultString = “Меньше 10”;
else
    resultString = “Больше или равно 10”;
```

Такой код, хотя он и многословнее, гораздо проще читать и понимать, чем тот, в котором использовался тринарный оператор, к тому же такой код оказывается более гибким.

Давайте обратимся к примеру.

1. Создайте новое консольное приложение.
2. Добавьте следующий код в Class1.cs:

```
static void Main(string[] args) {
    string comparison;
    Console.WriteLine("Enter a number:");
    double varl = Convert.ToDouble(Console.ReadLine()) ;
    Console.WriteLine(“Enter another number:”) ;
    double var2 = Convert.ToDouble(Console.ReadLine());
    if (varl < var2)
```

```
        comparison = "less than";
    else {
        if (varl == var2)
            comparison = "equal to";
        else
            comparison = "greater than";
        }
    Console.WriteLine("The first number is {0} the second number.", comparison);
}
```

3. Запустите код и введите два числа в ответ на приглашения.

Как это работает

Первая часть кода вам уже знакома здесь происходит получение двух значений типа double из строк, введенных пользователем.

```
string comparison;
Console.WriteLine("Enter a number:");
double varl = Convert.ToDouble (Console. ReadLine());
Console.WriteLine("Enter another number:");
double var2 = Convert.ToDouble(Console.ReadLine());
```

Далее переменной comparison типа string, в зависимости от значений, полученных для varl и var2, присваивается строка:

```
if (varl < var2)
    comparison = "less than";
```

Если данное условие не выполняется, то это значит, что varl либо больше, либо равно var2. Тогда в разделе else первого сравнения делается еще одно, вложенное сравнение:

```
else {
    if (varl == var2)
    comparison = "equal to";
```

Раздел else второго сравнения будет достигнут только в том случае, если var1 больше, чем var2:

```
else
    comparison = "greater than";
```

В заключение происходит вывод значения переменной comparison на консоль:

```
Console.WriteLine("The first number is {0} the second number.", comparison);
```

Использование вложенных операторов представляет собой лишь один из возможных путей решения этой задачи. С тем же успехом можно было бы написать следующее:

```
if (varl < var2)
    comparison = "less than";
if (varl == var2)
    comparison = "equal to";
if (varl > var2)
    comparison = "greater than";
```

Недостатком такого способа является необходимость выполнять все три сравнения независимо от значений переменных varl и var2. В первом же случае выполняется только одно сравнение, если varl < var2 есть true, и два сравнения в противном случае (мы также выполняем сравнение varl == var2), т. е. выполняется меньшее число строк кода. Разница в быстродействии несущественна, однако она может иметь значение для тех приложений, для которых скорость выполнения является критичной.

2.3.3. Проверка большего количества условий с помощью оператора if

В предыдущем примере проверялись три условия для значения переменной var1, которые охватывали все возможные для нее значения. Однако в некоторых случаях может потребоваться выполнять проверку на какие-то конкретные значения, например, 1,2, 3, 4 и т. д. Используя тот же подход, получим неудобоваримый вложенный код вроде:

```
if (varl == 1) {
    // выполнение каких-либо действий }
else {
    if (varl == 2) {
    // выполнение каких-либо других действий >
        else {
        if (varl ==31 || varl == 4) {
            // выполнение каких-либо других действий >
                else {
                // выполнение каких-либо других действий
            }
    }
}
```

Существует очень распространенная ошибка, когда условие в третьем сравнении записывается как if (var1 == 3 || 4). В таком случае в полном соответствии со старшинством операторов сначала будет выполнен оператор = =, в результате чего оператор || будет выполняться с операндами логического и численного типов. Это приведет к ошибке.

В такой ситуации разумнее применить несколько иную систему отступов и объединения элементов кода для блоков else (т. е. использовать после блока else одну строку кода, а не целый блок кода). Если поступить таким образом, то получим следующую структуру, содержащую операторы else if:

```
if (varl == 1) {
    // выполнение каких-либо действий }
else if (varl == 2) {
    // выполнение каких-либо других действий }
else if (varl == 3 || varl == 4) {
    // выполнение каких-либо других действий }
else {
    // выполнение каких-либо других действий }
```

На самом деле операторы else if представляют собой два отдельных оператора, и вышеприведенный код идентичен первоначальному варианту. Однако такой код является более удобочитаемым.

Когда необходимо выполнять такое большое количество сравнений, как в этом примере, стоит рассмотреть альтернативную возможность организации ветвления.

2.3.4. Оператор switch

Оператор switch очень похож на оператор if в том смысле, что он также обеспечивает условное выполнение кода в зависимости от результата выполненной проверки. Однако оператор switch позволяет производить проверку для многих значений переменной за один проход вместо того, чтобы проверять всего лишь одно условие. Эти проверки ограничены отдельными значениями и не допускают конструкций типа "больше, чем X", поэтому их использование имеет некоторые отличия; однако такой способ является очень эффективным.

Принципиальная структура оператора switch имеет следующий вид.

```
switch (<контрольнаяПеременная>)
{
case <значение1>:
    <код, который должен выполняться в том случае, если <контрольнаяПеременная> == <значение1> >
    break;
case <значение2>:
    <код, который должен выполняться в том случае, если <контрольнаяПеременная> == <значение2> >
    break;
case <значениеN>:
    <код, который должен выполняться в том случае, если <контрольнаяПеременная> == <значениеN> >
    break;
default:
    <код, который должен выполняться в том случае, если <контрольнаяПеременная> != ни одному из значений>>
    break;
}
```

Значение, которым обладает <контрольнаяПеременная>, сравнивается со всеми значениями <значениеX>, каждое из которых имеет свой собственный оператор case; в случае совпадения значений выполняется код, предусмотренный для данного условия. Если ни одно из значений не совпало, будет выполняться код из раздела default (при условии, что этот блок не пуст). После завершения выполнения кода в некотором разделе выполняется еще одна команда - break. В случае выполнения кода в одном из блоков case переход к выполнению другого оператора case не происходит.

Такая логика выполнения программы является одним из отличий языка C# от языка C++, в котором при обработке операторов case разрешается переходить от выполнения одного оператора к выполнению другого.

Оператор break в данном случае просто прерывает выполнение оператора switch, и работа программы продолжается со следующего оператора

В программе на C# имеются и другие способы избежать передачи управления от одного оператора case другому. Мы можем использовать либо оператор return, который приведет к прерыванию выполнения текущей функции, а не просто оператора switch, либо оператор goto. Оператор goto (он описан выше) в данном случае будет работать, поскольку операторы case в программе на C# фактически выполняют роль меток.

Имеется единственное исключение из общего правила, гласящего, что после выполнения одного оператора case не может выполняться другой оператор case. Если разместить несколько операторов case подряд (объединим их) перед каким-либо блоком кода, фактически организуем одновременное выполнение проверки нескольких условий. Если хотя бы одно из этих условий окажется выполненным, то управление будет передано на этот блок кода.

Эти условия также относятся и к оператору default. Не существует никакого правила, требующего, чтобы этот оператор обязательно находился в конце всего перечня сравнений, и при желании мы можем объединять его с операторами case. Включение оператора, прерывающего нормальное выполнение программы - break, goto или return,- гарантирует правильную последовательность выполняемых действий в любом случае.

Все переменные <значениех>, использующиеся для сравнений, должны иметь постоянное значение. Один из способом добиться этого - применение литеральных значений:

```
switch (myInteger) {
case 1:
    <код, который должен выполняться в том случае, если myInteger == 1 > break;
case -1:
    <код, который должен выполняться в том случае, если myInteger == -1 > break;
default:
    <код, который должен выполняться в том случае, если myInteger != ни одному из значений, использовавшихся
    для сравнения> break; }
```

Другой способ получить постоянные значения переменных – это использование переменных-констант. Переменные-константы являются точно такими же переменными, как и все остальные, за исключением одного ключевого момента: значение, которое в них содержится, не может быть изменено ни при каких условиях. После того как присвоить переменной-константе какое-либо значение, оно будет оставаться неизменным на протяжении выполнения всей программы. Переменные-константы могут оказаться удобными в данном случае, поскольку зачастую код легче понимать, когда используемые для сравнения значения не видны в момент выполнения сравнения.

Можно объявить переменную-константу, добавив ключевое слово const к описанию типа переменной; при этом требуется сразу определить ее значение. Например:

```
const int intTwo =2; '
```

Этот код является вполне допустимым, однако если попытаться выполнить следующий код:

```
const int intTwo;
intTwo =2;
```

то получим сообщение об ошибке на этапе компиляции. Также приведет к ошибке и попытка каким-либо образом изменить содержимое переменной-константы после первоначального присваивания.

2.4. Организация циклов

Цикл – это участок кода, в котором операторы выполняются по несколько раз. Такой способ в ряде случаев бывает чрезвычайно удобным, поскольку позволяет повторять необходимые действия столько раз, сколько требуется (тысячи и даже миллионы), без необходимости выписывать соответствующий код такое же количество раз.

В качестве простого примера рассмотрим программу, вычисляющую количество денег, которое окажется на счете в банке по истечении десятилетнего срока, исходя из предположения, что проценты начисляются ежегодно и никакие суммы не снимаются и не добавляются на счет:

```
double balance = 1000;
```

```
double interestRate =1.05; //5% годовых
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
```

Выписывать один и тот же код 10 раз весьма расточительно, и потом, как быть, если потребуется изменить срок с 10 лет на другое значение? Придется вручную копировать строку с кодом необходимое число раз, что добавит немало головной боли!

К счастью, это не придется делать. Достаточно выполнить необходимые инструкции в цикле требуемое число раз.

Другой тип цикла – это цикл, вычисления в котором продолжаются до тех пор, пока не будет выполнено определенное условие. Такие циклы несколько проще (хотя не менее полезны), чем описанные выше, поэтому мы с них и начнем.

2.4.1. Цикл do

Цикл do работает следующим образом. Код, который помечен как выполняющийся в цикле, выполняется первый раз, затем проверяется некоторое логическое условие, и если результат равен true, то этот код выполняется еще раз, и т. д. Цикл заканчивается, когда условие перестает выполняться.

Цикл do имеет следующую структуру:

```
do {
    <код, выполняющийся в цикле> }
while (<условие>);
```

где в результате вычисления <условия> получается логическое значение.

Точка с запятой после оператора while является обязательной; весьма распространена ошибка, когда ее забывают.

Мы можем использовать этот цикл, например, для того, чтобы вывести в столбец числа от 1 до 10:

```
int i = 1;
do
{
    Console.WriteLine("{0}", i++); }
while (i <= 10);
```

В данном случае мы использовали оператор ++ в виде суффикса, который увеличивает значение i на единицу после его вывода на экран; поэтому необходимо выполнять проверку i <= 10, чтобы число 10 попало в набор выводимых чисел.

2.4.2. Цикл while

Цикл while очень похож на цикл do, однако у него имеется одно важное отличие: вычисление логического условия в цикле while происходит в начале цикла, а не в конце. Если в результате проверки условия будет получено значение false, то цикл while вообще не будет выполняться и управление передается коду, расположенному после этого цикла.

Цикл while оформляется следующим образом:

```
while (<условие>) {
    <код, выполняющийся в цикле> }
```

Он может использоваться практически так же, как и цикл do. Например:

```
int i = 1; while (i <= 10) {
Console.WriteLine("{0}", i++); }
```

Выполнение этого кода приведет к тому же результату: числа от 1 до 10 будут выведены в столбец.

2.4.3. Цикл for

Последний тип цикла - это цикл for. Он относится к тому типу циклов, которые выполняются заранее заданное количество раз и сами отвечают за организацию счетчика цикла. Для организации цикла for требуется следующая информация:

- начальное значение для инициализации переменной цикла;
- условие для продолжения выполнения цикла, зависящее от переменной цикла;
- операция, которая будет выполняться над переменной цикла по завершении очередного прохода цикла.

Например, если необходимо организовать цикл с переменной цикла, изменяющейся от 1 до 10 с шагом, равным единице, то в этом случае начальное значение равно 1, условием для продолжения цикла будет "переменная цикла меньше или равна 10", а операцией, выполняющейся по окончание каждого прохода цикла, будет прибавление значению переменной цикла единицы. Эта информация должна быть размещена в структуре цикла for следующим образом:

```
for (<инициализация>; <условие>; <операция>)
    <код, выполняющийся в цикле>
```

Этот код работает так же, как и следующий цикл while

```
<инициализация>; while (<условие>)
    <код, выполняющийся в цикле> <операция>
```

Однако формат цикла for оказывается более понятным, поскольку все необходимые параметры, определяющие цикл, собраны воедино, а не распределены по разным операторам

При рассмотрении циклов do и while приводился пример с выводом чисел от 1 до 10. Приведем код, позволяющий сделать то же самое с помощью цикла for

```
int i;
for (i = 1; i <= 10; ++ i)
Console. WriteLine ("{0}", 1);
```

Целой переменной цикла с именем i присваивается начальное значение 1, после чего она увеличивается на 1 в конце каждого прохода. В теле цикла происходит вывод значения i на консоль.

Обратите внимание, что когда начинает выполняться код, расположенный за циклом, переменная i обладает значением, равным 11. Дело в том, что в конце прохода цикла, в котором переменная i равняется 10, ее значение увеличивается на единицу до проверки условия i <= 10. Очевидно, что после проверки выполнение цикла прекращается.

Как и циклы while, циклы for выполняются только в том случае, если проверка условия дает true перед первым проходом цикла, т. е. код, находящийся внутри цикла, может вообще ни разу не выполняться

В заключение отметим, что у нас имеется возможность объявить переменную цикла в качестве составной части оператора for, для чего вышеприведенный пример следует переписать следующим образом

```
for (int i = 1; i <= 10; ++i)
    Console.WriteLine("{0}", i) ; }
```

В этом случае переменная i будет недоступна коду, находящемуся вне цикла.

2.4.4. Прекращение выполнения цикла

Иногда возникают ситуации, когда требуется более тонкий контроль за выполнением кода в цикле. В C# для этих целей предусмотрено четыре команды:

- break – приводит к немедленному завершению цикла.
- continue – приводит к немедленному завершению текущего прохода цикла (вычисления продолжаются со следующего прохода).
- goto – позволяет выйти из цикла в помеченную точку кода (этот метод не рекомендуется использовать, если вы хотите получить легко читаемый и понимаемый код).
- return – выводит из цикла и из функции, в которой он находится.

Команда break позволяет просто выйти из цикла, при этом управление передается на первую строку кода после цикла. Например:

```
int i = 1;
while (i <= 10) {
    if (i == 6)
            break;
    Console.WriteLine("{0}", i++) ; }
```

Этот код выведет числа от 1 до 5, поскольку когда значение переменной i достигнет 6, команда break приведет к выходу из цикла.

Команда continue прекращает выполнение текущего прохода цикла, а не всего цикла. Например:

```
int i;
for (i = 1; i <= 10; i++)
{
    if ((i % 2) == 0)
    continue;
    Console.WriteLine(i);}
```

Третий способ предполагает использование оператора goto. Например:

```
int i = 1;
while (i <= 10)
{
    if (i == 6)
            goto exitPoint;
    Console.WriteLine("{0}", i++) ; }
    Console.WriteLine("'Этот код никогда не будет выполняться!");
    exitPoint:
    Console.WriteLine("Этот код будет выполнен после того, как произойдет выход из цикла посредством goto.");
```

Выход из цикла посредством команды goto допустим (разве что несколько неряшлив), а вот использование goto для входа в цикл извне запрещено.

2.4.5. Бесконечные циклы

Возможно – в результате ошибки или преднамеренно – создать циклы, которые никогда не завершатся: так называемые бесконечные циклы. В качестве самого простого примера можно привести следующий цикл'

```
while (true) {
// код, выполняющийся в цикле
}
```

Иногда такой цикл бывает полезен, тем более что из него всегда можно выйти, например, с помощью оператора break.

Однако в тех случаях, когда создание бесконечного цикла происходит по недосмотру, последствия могут оказаться очень неприятными. Рассмотрим следующий цикл, который очень напоминает цикл for из предыдущего раздела:

```
int i = 1;
while (i <= 10)
{
   if ((i % 2) == 0)
    continue;
 Console.WriteLine("{0}", i++); }
```

Здесь увеличение переменной цикла i происходит в последней строке цикла, т. е. после оператора continue. Поэтому если управление передается на оператор continue (а это произойдет, когда переменная i станет равной 2), следующая итерация будет исполняться с тем же значением переменной i, опять пройдет проверка текущего значения 1, начнет выполняться итерация и т. д. Это приведет к зависанию приложения. Обратите внимание, что в этом случае существует возможность выйти из зависшего приложения обычным способом, поэтому не придется перезагружать компьютер.

2.5. Вопросы для повторения

1. Что такое ветвление?
2. Что такое булева логика?
3. Логические операторы сравнения.
4. Опереторы, используемые для работы с логическими переменными.
5. Операторы работы с битами.
6. Старшинство операторов.
7. Оператор if-else.
8. Оператор switch.
9. Организация циклов.
10. Цикл while, while-do, do-while.
11. Цикл for.

3. ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ О ПЕРЕМЕННЫХ

В языке C# существуют еще несколько типов переменных:
– перечислимый тип – тип переменных, которые принимают дискретные значения из множества, определяемого пользователем, и которые могут нести дополнительную смысловую нагрузку.
– структуры – сложные типы переменных, которые создаются
– из определяемого пользователем набора других типов переменных.
– массивы – это тип, который содержит большое количество переменных одного и того же типа и который позволяет осуществлять индексный доступ к отдельным значениям.
Преобразование типов описывает преобразование значений одного типа в значения другого типа.

3.1. Преобразованиепеременных из одного типа в другой

Любые данные, независимо от своего типа, представляют собой последовательность двоичных битов, т. е. последовательность, состоящую из нулей и единиц. Собственно значение переменной зависит от того, каким образом эти биты интерпретируются. Простейшим примером такого подхода может служить тип char, который содержит номер символа из множества символов Unicode. Этот номер хранится абсолютно так же, как и число типа ushort: в обоих случаях это число, лежащее в диапазоне от 0 до 65 535.

Однако в большинстве случаев оказывается, что для различных типов переменных используются разные способы представления данных. Отсюда следует, что даже в тех ситуациях, когда имеется физическая возможность перенести последовательность битов из одной переменной в переменную какого-либо другого типа (например, обеим переменным для хранения требуется одинаковая область памяти, или у той переменной, в которую осуществляется запись, оказывается достаточно места для размещения всех исходных битов), результаты могут совершенно не совпадать с теми, которые ожидали.

Вместо такой переписи битов из одной переменной в другую приходится использовать преобразования типов данных.

Существует два вида преобразования типов.

Неявное преобразование – когда преобразование типа A в тип B возможно при любых обстоятельствах, а правила проведения такого преобразования достаточно просты, чтобы доверить его компилятору.

Явное преобразование – когда преобразование типа A в тип B возможно только при определенных обстоятельствах либо правила преобразования настолько сложны, что возникает необходимость выполнения каких-либо дополнительных действий.

3.1.1. Неявные преобразования

Неявное преобразование не требует со стороны программиста никакой дополнительной работы и никакого дополнительного кода. Рассмотрим пример кода:

```
varl = var2;
```

Такое присваивание может включать в себя проведение неявного преобразования типов в том случае, если тип переменной var2 допускает проведение неявного преобразования в тип переменной varl; однако с тем же успехом в этом присваивании могут быть задействованы две переменные одного и того же типа, и тогда никакого неявного преобразования типов вообще не требуется.

Рассмотрим пример.

Значения типа ushort и char являются фактически взаимозаменяемыми, поскольку оба они предполагают хранение чисел в диапазоне между 0 и 65 535. Можно осуществлять преобразования этих типов друг в друга неявно, что иллюстрируется следующим кодом:

```
ushort destinationVar;
char sourceVar = 'a';
destinationVar = sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar};
```

Таблица 3.1 – Возможность неявных преобразований для простых типов

Тип	Допускает безопасное преобразование в
byte	short, ushort, int, uint, long, ulong, float, double, decimal
sbyte	short, int, long, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

В этом коде значение, хранящееся в переменной sourceVar, переносится в переменную destinationVar. Затем две команды Console.WriteLine() дают следующий выходной поток:

```
sourceVar val: a
destinationVar val: 97
```

Хотя в обеих переменных хранится одна и та же информация, она интерпретируется совершенно по-разному в зависимости от типа переменной.

Для простых типов существует большое количество неявных преобразований; для типов bool и string неявных преобразований не существует, зато они имеются у численных типов. В таблице слева, приведенной для справки, показаны преобразования численных типов, которые могут выполняться компилятором неявно (значения типа char хранятся в виде чисел, поэтому рассматривается как численный тип).

Существует простой способ определять, какие именно преобразования компилятор может выполнять неявно. Если вернуться к лекции 1 (таб. 1.1 – 1.3), то там можно найти диапазоны допустимых значений для каждого простого численного типа. Правило неявных преобразований для рассматриваемых типов можно сформулировать следующим образом, любой тип A, диапазон допустимых значений которого целиком находится внутри диапазона допустимых значений типа B, может быть преобразован неявным образом в этот тип.

Это правило имеет очень простое обоснование. Если попытаться присвоить переменной некоторое значение, а это значение окажется выходящим за пределы тех значений, которые она может принимать, то возникнут определенные проблемы. Так, например, в переменной типа short могут храниться значения до 32 767, а максимально допустимое значение типа byte ограничено 255, поэтому если попытаться преобразовать значение типа short в значение типа byte, это может создать проблему. Если значение типа short лежит в диапазоне между 256 и 32 767, то оно просто не уместится в одном байте.

Однако в том случае, если точно известно, что значение типа short меньше 255, должна существовать какая-то возможность выполнить такое преобразование.

Такая возможность существует, но для этого необходимо воспользоваться явным преобразованием. При выполнении явного преобразования вы как бы говорите: "Хорошо, я предупрежден об опасности подобных действий, и я принимаю всю ответственность за возможные последствия на себя".

3.1.2. Явные преобразования

Как и предполагает их название, явные преобразования выполняются в тех случаях, когда явным образом обращаются к компилятору с просьбой преобразовать значение одного типа в другой. Именно поэтому для их выполнения требуется наличие дополнительного кода, причем формат этого кода может варьироваться в зависимости от конкретного метода выполнения преобразования. Прежде чем приступить к рассмотрению кода, применяемого для выполнения явных преобразований, давайте узнаем, что произойдет в том случае, если для явного преобразования не будет использоваться никакого кода.

В качестве примера, несколько модифицировав код из предыдущего раздела, попытаемся преобразовать значение типа short в значение типа byte.

```
byte destinationVar;
short sourceVar = 7;
destinationVar = sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

При попытке выполнить компиляцию этого кода будет выведено следующее сообщение об ошибке:

Cannot implicitly convert type 'short' to 'byte' (Невозможно неявно преобразовать тип 'short' в 'byte').

Компилятор C# умеет обнаруживать отсутствие необходимых явных преобразований!

Для того чтобы такой код можно было откомпилировать, нужно добавить код, который выполнял бы это преобразование явно. Наиболее простой способ добиться этого в данном контексте – это привести переменную типа short к переменной типа byte. Приведение типа, как правило, означает принудительный перевод данных из одного типа в другой и имеет следующий синтаксис:

```
(destinationType) sourceVar
```

Эта запись приведет к преобразованию значения переменной sourceVar к типу destinationType.

Такое возможно только в определенных ситуациях. Для типов, мало похожих друг на друга или вообще не имеющих ничего общего, преобразование типов с помощью приведения, вероятнее всего, не будет определено.

Теперь можно модифицировать наш пример таким образом, чтобы добиться преобразования из типа short в тип byte:

```
byte destinationVar;
short sourceVar = 7;
destinationVar = (byte)sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

Результатом будет:

```
sourceVar val: 7 destinationVar val: 7
```

А что произойдет в тех случаях, когда пытаются загнать в переменную значение, которое там не помещается? Давайте модифицируем нашу программу:

```
byte destinationVar;
short sourceVar = 281;
destinationVar = (byte)sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLinel"destinationVar val: {0}", destinationVar);
```

В этом случае мы получим следующий результат:

```
sourceVar val: 281 destinationVar val: 25
```

Итак, что же произошло? Посмотрим на двоичное представление этих двух чисел, а также максимально допустимого значения для типа byte, равного 255:

```
281 = 100011001
25 = 000011001
255 = 011111111
```

Легко заметить, что потерян самый левый бит исходного значения. Немедленно возникает вопрос: каким образом это можно предсказать? Совершенно очевидно, что будут встречаться ситуации, когда придется явно приводить один тип к другому, и было бы неплохо знать заранее, какие данные при этом будут утрачиваться. Не умея этого делать, можно столкнуться с серьезными ошибками, например, в приложениях для работы с банковскими счетами.

Один из способов предвидеть потерю битов – проверка исходного значения и его сравнение с известными предельными значениями новой переменной. Каким образом это можно выполнить, будет рассмотрено ниже при объяснении логики работы приложений и управления порядком выполнения. Но имеется и другой способ, который заставляет систему во время работы уделять особое внимание данному преобразованию. Попытка присвоить переменной слишком большое значение приводит к переполнению, и это как раз та ситуация, которую требуется контролировать

Контекст проверки на переполнение задается с помощью двух ключевых слов: checked (проверяемое) и unchecked (непроверяемое), которые используются следующим образом:

```
checked(выражение) unchecked(выражение)
```

Потребуем осуществить проверку на переполнение в предыдущем примере:

```
byte destinationVar;
short sourceVar = 281;
destinationVar = checked((byte)sourceVar);
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

Процесс выполнения кода будет прерван появлением сообщения об ошибке (OverflowCheck).

Однако в том случае, если заменить ключевое слово checked на unchecked, получится тот же результат и никакой ошибки не возникнет. Такой вариант идентичен поведению по умолчанию, который рассмотрен выше. Наряду с этими

ключевыми словами имеется возможность поменять конфигурацию приложения, чтобы оно работало так, как если бы в любом выражении данного типа использовалось ключевое слово checked, кроме тех выражений, в которых явно использовано ключевое слово unchecked (другими словами, можно менять установки, которые используются по умолчанию для выполнения проверок на переполнение). Для этого потребуется изменить свойства приложения в VS. (Properties-> Configuration Properties-> Build -> Check for Arithmetic Overflow/Underflow)/ По умолчанию эта установка имеет значение false, однако при замене ее на true приложение будет работать в режиме checked, подробно описанном выше.

3.1.3. Выполнение явных преобразований с помощью команд преобразования

Способ явного преобразования, использовавшийся во многих предыдущих примерах, несколько отличается от рассмотренного в данной лекции. Прежде осуществляли преобразования строковых значений в числа посредством таких команд, как Convert.ToDouble(), которые совершенно очевидно не смогут работать с любой строкой. Если, например, попытаться преобразовать строку "Number" в число типа double с помощью Convert. ToDouble (), то в процессе выполнения кода будет открыто диалоговое окно с сообщением:

"Необрабатываемая исключительная ситуация типа 'System.FormatException' в mscorlib.dll. Дополнительная информация: Входная строка имеет неверный формат.".

Операция не выполнена. Для того чтобы подобный тип преобразования мог осуществиться, передаваемая строка должна быть допустимым представлением числа и это число не должно приводить к переполнению. Допустимым называется такое представление числа, которое содержит необязательный знак (плюс или минус), ноль или несколько цифр, необязательную десятичную точку, за которой следует одна или несколько цифр, а также необязательный символ "e" или "E", за которым следуют необязательный знак и одна или несколько цифр, и ничего более, кроме пробелов (располагающихся перед или после этой последовательности).

Используя все эти дополнительные опции, можно распознавать строки типа -1.245e-24 как число.

Существует много явных преобразований, которые могут быть определены таким же образом (табл. 3.2).

Здесь val может быть переменной почти любого типа (а если она не может быть обработана данными командами, то компилятор выдаст соответствующее сообщение).

К сожалению, как можно заметить из приведенной таблицы, имена команд преобразования несколько отличаются от названий типов, принятых в C#; например, для того чтобы произвести преобразование в тип int, следует использовать Команду Convert.ToInt32(). Это объясняется тем, что имена в командах взяты из пространства имен System .NET Framework и не являются родными для C#. Зато благодаря этому их можно использовать из любых других языков, совместимых с NET.

Таблица 3.2 – Команды явных преобразований

Команда	Результат
Convert.ToBoolean(val)	val преобразовано в bool
Convert.ToByte(val)	val преобразовано в byte
Convert.ToChar(val)	val преобразовано в char
Convert.ToDecimal(val)	val преобразовано в decimal
Convert. ToDouble I. val)	val преобразовано в double
Convert.Tolntl6(val)	val преобразовано в short
Convert.Tolnt32(val)	val преобразовано в int
Convert.Tolnt64(val)	val преобразовано в long
Convert. ToSBy te (val)	val преобразовано в sbyte
Convert.ToSingle(val)	val преобразовано в float
Convert.ToStnng(val)	val преобразовано в string
Convert.ToUlnt16(val)	val преобразовано в ushort
Convert.ToUlnt32(val)	val преобразовано в uint
Convert.ToUlnt64(val)	val преобразовано в ulong

Необходимо отметить один важный момент: в процессе таких преобразовании всегда осуществляется контроль за переполнением, при этом ключевые слова checked и unchecked, а также установки, заданные для всего проекта, влияния не имеют.

3.2. Сложные типы переменных

Итак, к настоящему моменту были рассмотрены все простые типы переменных, которые используются в C#. Кроме них, в C# существует три более сложных (но очень полезных) типа переменных:

- перечислимый тип;
- структуры;
- массивы.

3.2.1. Перечислимый тип

Все типы, которые рассматривались до сих пор (за исключением типа string), имеют четко определенное множество допустимых значений. Это множество может быть настолько большим (как, например, у типа double), что его можно рассматривать в качестве континуума, однако все равно это фиксированное множество. В качестве простейшего примера можно привести тип bool, который может принимать только одно из двух значений: true или false.

Существует большое количество ситуаций, когда требуется переменная, принимающая значение из фиксированного множества. Например, может возникнуть необходимость в использовании переменной типа orientation (ориентирование), которая принимает одно из значений: north (север), south (юг), east (восток) или west (запад).

В подобных ситуациях очень полезным может оказаться перечислимый тип. Он позволяет сделать как раз то, что требуется для переменной orientation: определить тип, который принимает одно значение из конечного множества задаваемых нами значений.

Все, что для этого требуется сделать,– это создать свой собственный перечислимый тип, который будет называться orientation и для которого будут существовать четыре возможных значения, перечисленных выше. Обратите внимание, что это является дополнительным шагом: не просто объявляется переменная некоторого типа, сначала объявляется и подробно описывается создаваемый пользователем тип и только после этого имеется возможность объявить переменную данного типа.

3.2.2. Определение перечислимых типов

Перечислимый тип описывается с помощью ключевого слова enum следующим образом

```
enum имяТипа {
значение1,
значение2,
значение3,
значениеN }
```

Затем объявляются переменные этого типа <имяТипа имяПеременной>. Им присваиваются конкретные значения <имяПеременной> = <имяТипа.значение>;.

Перечислимый тип обладает базовым типом (underlying type), который используется для хранения. Любое из значений, которые этот тип может принимать, будет храниться в памяти как значение базового типа (по умолчанию это тип int) Однако существует возможность задать в качестве базового другой тип, добавив к описанию типа имя его базового типа

```
enum имяТипа : базовыйТип {
значение1,
значение2,
значение3,
```

значениеN }

Перечислимые типы могут использовать в качестве базовых следующие типы byte, sbyte, short, ushort, int, uint, long и ulong.

По умолчанию каждому значению перечислимого типа автоматически присваивается соответствующее значение базового типа, начиная с нуля, в том порядке, в котором они описаны. Другими словами, значение 1 получит базовое значение 0, значение 2 – 1, значение 3 – 2 и т. д. Для того чтобы переопределить такой порядок, следует использовать оператор = и фактические базовые значения для каждого перечислимого значения.

```
enum имяТипа : базовыйТип {
значение1 = фактическоеЗначение1,
значение2 = фактическоеЗначение2,
значение3 = фактическоеЗначение3,
значениеN = фактическоеЗначениеN}
```

Кроме того, существует возможность задавать идентичные базовые значения для нескольких перечислимых значений, используя одно значение как базовое значение другого

```
enum имяТипа : базовыйТип {
значение1 = фактическоеЗначение1,
значение2 = значение1,
значение3,
...
значениеN= фактическоеЗначениеN }
```

Всем значениям, оставшимся неуказанными, будут автоматически присвоены базовые значения, для этого используется последовательность, начинающаяся со значения, на единицу большего последнего явно заданного значения. В вышеприведенном объявлении значение3 получит базовое значение, равное значение1 + 1.

Обратите внимание, что это может привести к возникновению непредвиденных проблем, когда какие-либо из базовых значений, заданных после определения вроде значение2 = значение1, окажутся идентичными другим базовым значениям. Так, например, в следующей программе базовое значение для значение4 и для значение2 будет одним и тем же

```
enum имяТипа : базовыйТип {
значение1 = фактическоеЗначение1,
значение2,
значение3 = значение1,
значенне4,
...
значениеN = фактическоеЗначениеN }
```

Это совершенно нормально, если, конечно, требуется добиться именно такого эффекта.

Обратите также внимание, что присваивание значений циклическим образом приведет к возникновению ошибки:

```
enum имяТипа.базовыйТип {
значение1 = значение2,
значение2 = значение1 }
```

3.2.3. Структуры

Еще одним типом переменной, который требуется рассмотреть, является struct (сокращение от слова structure – "структура"). Структуры вполне соответствуют своему названию: это структуры данных, которые составлены из информации различного характера, возможно, из данных различных типов. Они позволяют программистам описывать свои собственные типы переменных, для которых данная структура является базовой. В качестве примера предположим, что необходимо хранить информацию о маршруте из одной точки в другую, включающую направление и расстояние в милях. Для простоты в качестве направления будет использоваться только одна из четырех сторон света (так что для описания направления вполне подойдет перечислимый тип orientation), а длина маршрута будет представлена типом double.

В этом случае можно было бы использовать две разные переменные, написав уже знакомый нам код:

```
orientation myDirection;
double myDistance;
```

В таком подходе с использованием двух переменных нет ничего плохого, однако намного удобней (особенно если маршрутов много) хранить всю эту информацию в одном месте.

Описание структур

Для описания структур используется ключевое слово struct:

```
struct <имяТипа> {
<объявлениеЧлена> }
```

В разделе <объявлениеЧлена> содержатся объявления переменных (они называются полями) практически в обычном формате. Объявление каждого члена имеет следующий вид:

```
<доступ> <тип> <имя>;
```

Для того чтобы код, обращающийся к структуре, имел доступ к ее полям, в разделе <доступ> следует использовать ключевое слово public (общий). Например:

```
struct route {
public orientation direction;
public double distance; }
```

После того как структура описана, можно использовать ее для описания переменных соответствующего типа:

```
route myRoute;
```

Для получения доступа к отдельным полям этой составной переменной следует использовать точку:

```
myRoute.direction = orientation.north;
myRoute.distance =2.5;
```

3.2.4. Массивы

Все типы переменных, которые были рассматренны до сих пор, обладают одним общим свойством: в них хранится единственное значение (или – в случае структур – единственный набор значений). Но в ситуациях, когда необходимо хранить большое количество однородных данных, такой подход может оказаться не совсем удобным. В таких случаях более приемлемым было бы одновременное хранение нескольких значений одного типа без использования отдельной переменной для каждого значения.

Предположим, к примеру, что вам необходимо выполнить вычисления, в которых задействованы имена всех ваших друзей. Для этого можно воспользоваться обычными переменными типа string, например, так:

```
string friendNamel = "Robert Barwell";
string friendName2 = "Mike Parry";
string friendName3 = "Jeremy Beacock" ;
```

Однако в этом случае потребуется затратить много усилий, особенно по мере написания кода для обработки каждой переменной в отдельности. Отсутствует и возможность пройти по всему этому списку строк в цикле.

Альтернативной возможностью является использование массивов. *Массив представляет собой список индексированных переменных.*

Допустим, что имеется массив, в котором хранятся все три имени, приведенные выше, и которое называется friendNames. Тогда для получения доступа к отдельным элементам этого массива, просто указывая их индекс в квадратных скобках, как показано в следующем примере:

```
friendNames [<индекс>]
```

Индекс – это целое число, имеющее значение 0 для первого вхождения, 1 – для второго и т. д. Это означает, что у нас появляется возможность пройти все вхождения в цикле, например:

```
int i;
for (i = 0; i < 3; i++)
{
    Console.WriteLine("Имя с индексом {0}: {1}", i, friendNames[i]); }
```

Каждый массив обладает единым базовым типом, т. е. все отдельные вхождения в массив имеют один и тот же тип. В данном случае массив friendNames имеет базовый тип string, поскольку он предназначается для хранения переменных типа string.

Вхождения массива часто называются элементами.

Объявление массивов

Массивы объявляются следующим образом:

```
<базовыйТип> [ ] <имя>;
```

В данном случае <базовыйТип> может быть произвольным типом, в том числе перечислимым или структурой.

Прежде чем получить доступ к массиву, его необходимо инициализировать. Но получить доступ или присвоить значение элементу массива следующим образом нельзя:

```
int[] myIntArray;
myIntArray[10] = 5;
```

Массивы могут быть инициализированы двумя способами. Мы можем либо задать полное содержимое массива в литеральной форме, либо указать размер массива и использовать для инициализации всех его элементов ключевое слово new (новый).

Определение значений элементов массива в литеральной форме предполагает задание списка значений, разделенных запятыми и заключенных в фигурные скобки. Например:

```
int[] myIntArray = {5, 9, 10, 2, 99};
```

В данном случае описывается массив, состоящий их пяти элементов, и всем им присваивается целое значение.

Второй способ предполагает использование следующего синтаксиса:

```
int[] myIntArray = new int[5];
```

Здесь используется ключевое слово new для явной инициализации массива и константа для определения его размера. При этом методе всем вхождениям массива будет присвоено значение по умолчанию, которое для численных типов равно 0. Для инициализации массивов вместо констант с тем же успехом можно использовать переменные, например.

```
int [ ] myIntArray = new int[размерМассива];
```

При желании можно одновременно использовать оба метода

```
int[] myIntArray = new int[5] {5, 9, 10, 2, 99};
```

В этом случае размеры массива должны совпадать с количеством элементов в списке. Следующая запись является недопустимой:

```
int[] myIntArray = new int[10] {5, 9, 10, 2, 99};
```

Здесь описывается массив, состоящий из 10 элементов, а определяются только 5 из них; компиляция этой записи не пройдет. Побочным эффектом такого положения вещей является то, что если мы описываем размерность массива с помощью переменной, то она должна являться константой. Например:

```
const int arraySize = 5;
int[] myIntArray = new int [arraySize] {5, 9, 10, 2, 99};
```

Если ключевое слово const будет пропущено, программа работать не будет.

Как и в случае переменных других типов, нет никакой необходимости инициализировать массив в той же строке, в которой он объявляется Вполне допустимой является следующая запись:

```
int[] myIntArray;
myIntArray = new int[5];
```

Теперь вы уже знаете достаточно для того, чтобы попробовать написать некоторую программу

1. Создайте новое консольное приложение
2. Добавьте следующий код в Classl.cs.

```
static void Main(string[] args) {
string[] fnendNames = {"Robert Barwell", "Mike Parry", "Jeremy Beacock"};
int i;
Console.WriteLine("Here are {0} of my friends:", friendNames.Length);
for (i=0; i < friendNames.Length; i++) {
    Console.WriteLine(friendNames[i] ) ; } }
```

3. Запустите программу (см. рис. справа).

Как это работает

В этом коде объявляется массив типа string, состоящий из трех значений, которые затем выводятся на консоль с помощью цикла for. Обратите внимание, что существует возможность узнать общее количество элементов массива посредством параметра friendNames.Length:

```
Console.WriteLme("Here are {0} of my friends:"; friendNames.Length) ;
```

Это очень удобный способ определения размера массива.

При выводе значений массива в цикле for легко допустить ошибку. Например, попытаемся вместо оператора < использовать оператор < =, как показано ниже.

```
for (i = 0; i <= friendNames.Length; i++) {
```

```
    Console.WriteLine(friendNames[i]); }
```

Компиляция с последующим выполнением программы приведет к появлению окна с сообщением об ошибке: "Необрабатываемая исключительная ситуация типа 'выход индекса за пределы диапазона' в …." (см. рис. слева).

В приведенном случае мы попытались получить доступ к friendNames[3]. Как вы помните, значения индексов массива начинаются с нуля, поэтому последним элементом данного массива является friendNames[2]. Обращение к элементам массива, выходящим за его границу, приведет к ошибке. На самом деле существует более удобный способ доступа ке всем элементам массива с помощью цикла foreach.

3.2.5. Циклы foreach

Цикл foreach позволяет обращаться ко всем элементам массива с помощью очень простого синтаксиса

```
foreach (<базовыйТип> <имя> in <массив>) {
// можно использовать <имя> для доступа к очередному элементу }
```

Этот цикл пройдет по всем элементам массива, последовательно помещая каждый из них в переменную <имя>; при этом опасность выхода за границы массива отсутствует. Нам не придется беспокоиться о том, сколько элементов имеется в массиве, и мы сможем воспользоваться в цикле каждым из них. Для того чтобы применить этот способ, изменим код последнего примера:

```
static void Main(string [] args) {
string[] friendNames = {"Robert Barwell", "Mike Parry", "Jeremy Beacock"};
Console.WriteLine("Here are {0} of my friends:", friendNames.Length) ;
foreach (string friendName in friendNames)
{
    Console.WriteLine(friendName) } }
```

Выходной поток этой программы будет абсолютно таким же, как и в предыдущем примере.

Основное отличие этого метода от обычного цикла for заключается в том, что он позволяет использовать содержимое массива в режиме "только чтение", т. е. отсутствует возможность изменять значения элементов массива. Например, написать такую программу нельзя:

```
foreach (string friendName in friendNames) {
    friendName = "Rupert the bear"; }
```

При попытке откомпилировать этот код будет выдано сообщение об ошибке.

3.2.6. Многомерные массивы

Многомерный массив – это объект, в котором для осуществления доступа к его элементам используется несколько индексов.

В качестве примера представим случай, когда требуется отразить высоту возвышенности по измеряемым точкам на местности. Мы можем определить такие точки двумя координатами x и y и использовать эти координаты в качестве индексов так, чтобы в массиве с названием hillHeight хранилась высота холма, соответствующая точке с данной парой координат. Для этого потребуется использовать многомерный массив.

Двумерный массив такого вида объявляется следующим образом:

```
<базовыйТип> [,] <имя>;
```

Описания массивов с большим числом размерностей требуют больше запятых. Например:

```
<базовыйТип>[,,,] <имя>;
```

Так объявляется четырехмерный массив.

Для присваивания значений используется аналогичный синтаксис, причем размерности отделяются одна от другой запятыми. Для того чтобы объявить и инициализировать обсуждавшийся выше двумерный массив hillHeight с базовым типом double, размерностью x, равной 3, и размерностью y, равной 4, необходима следующая строка.

```
double[,] hillHeight = new double[3,4];
```

В качестве альтернативы при начальном присваивании значений могут использоваться литеральные значения. Для этого воспользуемся вложенными блоками, взятыми в фигурные скобки и разделенными запятыми. Например:

```
double[,] hillHeight = new {{1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6}};
```

Этот массив имеет точно такие же описания размерностей, что и предыдущий, однако в данном случае значения элементов массива задаются явно.

Для того чтобы получить доступ к отдельным элементам многомерного массива, следует указать индексы этого элемента, разделенные запятыми. Например:

```
hillHeight[2,I]
```

Теперь над этим элементом можно выполнять действия так же, как и над другими элементами.

Это выражение позволяет получить доступ ко второму элементу третьего вложенного массива в соответствии с вышеприведенным описанием (в данном случае значение равняется 4). Отсчет начинается с 0 и что первая цифра относится к вложенному массиву, т.е. первая цифра определяет номер пары фигурных скобок, а вторая цифра определяет соответствующий элемент внутри этой пары фигурных скобок.

Цикл foreach позволяет осуществлять доступ ко всем элементам многомерных массивов так же, как и в случае одномерных. Например:

```
double [,] hillHeight = {{1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6}};
foreach (double height in hillHeight)
{
Console.WriteLine("{0}", height); )
```

Элементы этого массива будут выводиться в том же порядке, в котором происходило присваивание им литеральных значений:

```
hillHeight[0,0], hillHeight[0,1], hillHeight[0,2], hillHeight[0,3], hillHeight[1,0], hillHeight[1,1], hillHeight[1,2]
```

и т д.

3.2.7. Массивы массивов

Многомерные массивы, обсуждавшиеся в предшествующем разделе, обычно называются прямоугольными, поскольку у них каждая "строка" имеет один и тот же размер. Если использовать предыдущий пример, то координата y может изменяться от 0 до 2 для любой координаты x.

Однако существует возможность использовать ступенчатых (jagged) массивов, у которых "строки" могут быть неодинакового размера. Для этого требуется массив, каждым элементом которого также является массив. При желании можно использовать массивы массивов массивов и даже более сложные конструкции. Однако это оказывается возможным только в том случае, если все массивы имеют один и тот же базовый тип.

Синтаксис, применяемый для объявления массивов массивов, подразумевает использование в объявлении массива нескольких пар квадратных скобок, например

```
int[ ] [ ] jaggedIntArray;
```

К сожалению, инициализация подобных массивов оказывается не таким простым делом, как инициализация многомерных массивов Так, например, мы не имеем возможности использовать вслед за этим объявлением следующую запись

```
jaggedIntArray = new int[3] [4];
```

Но даже в том случае, если бы мы могли применять запись такого вида, вряд ли от этого была бы какая-нибудь польза, поскольку в случае многомерных массивов точно такого же эффекта можно достичь с меньшими затратами. Следующий код также является недопустимым

```
jaggedIntArray = {{1, 2, 3}, {1}, {1, 2}};
```

На самом деле существуют две возможности. Можно сначала инициализировать массив, содержащий другие массивы (чтобы избежать путаницы, мы будем называть вложенные массивы субмассивами), а затем, в свою очередь, инициализировать эти субмассивы:

```
jaggedIntArray = new int[2] [];
jaggedIntArray [0] = new int[3];
jaggedIntArray [1] = new int[4];
```

Можно также воспользоваться видоизмененной формой приведенного выше литерального присваивания

```
jaggedIntArray = {new int[] {1, 2, 3}, new int[] {1}, new int[] {1, 2}};
```

Для таких неоднородных массивов можно использовать циклы foreach, однако чаще всего вложенные циклы применяются для того, чтобы получить доступ к реальным данным. Например, допустим, что имеется следующий неоднородный массив, который включает в себя десять других массивов, каждый из которых содержит целые значения, являющиеся делителями целых чисел из диапазона от 1 до 10

```
int[][] divisors1Tol0 = {new int[] {1},
new int [] {1, 2},
new int[] {1, 3},
new int [] {1, 2, 4},
new int[] {1, 5},
new int[] {1, 2, 3, 6},
new int[] {1, 7},
new int [ ] {1, 2, 4, 8},
new int[] {1, 3, 9},
new int[] {1, 2, 5, 10}};
```

Использование следующего кода

```
foreach (int divisor in divisors1To10) {
Console.WriteLine(divisor); }
```

является недопустимым, поскольку divisorslTol0 содержит в себе не элементы int, а элементы int [ ] Поэтому нам придется проходить не только по самому массиву, но и непосредственно по каждому субмассиву:

```
foreach (int[] divisors1To10 in divisors1Tol0) {
    foreach (int divisor in divisors1To10)
    {
        Console.WriteLine(divisor);
} }
```

При использовании неоднородных массивов синтаксис очень быстро становится сложным. В большинстве случаев оказывается проще использовать либо прямоугольные массивы, либо какой-нибудь другой, более простой способ хранения данных.

3.2.8. Действия над строками

До сих пор все наши операции над строками сводились к выводу строк на консоль, чтению строк с консоли и объединению строк посредством оператора +. Переменная типа string – это всего лишь массив переменных типа char, доступных в режиме "только чтение". Иными словами, можно получить доступ к отдельным символам строки следующим образом:

```
string myString = "A string";
cnar myChar = myString[l] ;
```

Однако присваивать отдельные символы таким способом нельзя.

Для получения массива, доступного для записи, следует воспользоваться приведенным ниже кодом. В нем используется команда ToCharArray () переменной типа массива:

```
string myString = "A string";
char [ ] myChars = myStrrig.ToCharArray();
```

Теперь можно выполнять манипуляции с массивом типа char обычным путем. Строки могут использоваться и в циклах foreach:

```
foreach (char character m myString) {
    Console WriteLine("{0}" , character);
}
```

Как и в случае других массивов имеется возможность узнать число элементов с помощью myString.Length. Таким же образом можно определить общее количество символов в строке:

```
string myString = Console.ReadLine() ;
Console.WriteLine("You typed {0} characters.", myString.Length);
```

Другим основополагающим способом работы со строками является использование команд в формате, аналогичном формату команды <string>ToCharArray(). Существуют две простые, но очень полезные команды <stnng>.ToLower() и <string>.ToUpper(). Они позволяют переводить всю строку целиком в нижний и верхний регистр соответственно. Они могут быть использованы, например, в поступлении от пользователя какого-то ответа, например, строки "yes". Если перевести всю введенную строку в нижний регистр, то можно воспринять и такие строки, как "YES", "Yes", "yeS" и т д.

```
string userResponse = Console.ReadLine() ,
if (userResponse.ToLower() == "yes") {
    // выполнение действий в случае получения ответа }
```

Эта команда, как и другие рассматриваемые в данной лекции команды, на самом деле не изменяет ту строку, к которой применяется. Напротив, использование этой команды для некоторой строки приводит к созданию новой строки, которую можно сравнить с какой-либо другой строкой (как показано выше) или присвоить другой переменной. В роли этой переменной может выступать переменная, над которой выполняется данная операция:

```
userResponse = userResponse.ToLower();
```

Что еще можно сделать для облегчения интерпретации ответов пользователя. Что произойдет в том случае, если пользователь случайно включил в начало или в конец своего ответа пробел. В таком варианте вышеприведенный код не

сработает. Необходимо убрать из введенной строки все пробелы, чего можно добиться посредством команды <string>.Trim().

```
string userResponse = Console ReadLine();
userResponse = userResponse Trim();
if (userResponse ToLower() = = “yes”) {
    // выполнение действий в случае получения ответа. }
```

В этом случае можно определить строки, подобные следующим: "YES" "Yes".

Такая же команда может быть использована для удаления любых других символов, которые задаются с помощью массива типа char, например:

```
char[] trimChars = {' ', 'e', 's'};
string userResponse = Console.ReadLine ();
userResponse = userResponse.ToLower();
userResponse = userResponse Trim(trimChars);
if {userResponse = = "y") {
    // выполнение действий в случае получения ответа }
```

Это позволяет удалить все пробелы, символы "e" и символы "s", находящиеся в начале или в конце строки. Если предположить, что в строке отсутствуют какие-либо другие символы, то появится возможность определять строки: "Yeeeees" " У" и им подобные.

Существует также возможность использовать команды <string>.TrimStart() и <string>.TrimEnd(), которые будут удалять пробелы только из начала или только из конца строки соответственно. В этих командах также имеется возможность задания массива типа char удаляемых символов.

Имеются еще две команды работы со строками, которые выполняют манипуляции с пробелами внутри строк <string>.PadLeft() и <string>.PadRight () эти команды позволяют дополнять строки пробелами слева или справа до заданной длины строки. Они могут использоваться следующим образом:

```
 <string>. PadX (<требуемаяДлина>);
```

Например:

```
myString = "Aligned";
myString = myString. PadLeft (10);
```

В результате к слову "Aligned", содержащемуся в переменной myString, будут добавлены три пробела слева. Этот метод может оказаться полезным для выравнивания строк, располагаемых одна над другой, в частности, при расположении строк с номерами

Так же, как и в случае команд, предназначенных для удаления символов, описываемые команды допускают вариант использования, позволяющий задавать символ, которым будет дополняться строка. В данном случае, однако, следует задавать отдельный символ, а не массив символов. Например:

```
myString = “Aligned”;
inyString = myString. PadLeft (10, '-') ;
```

В итоге в начало строки, хранящейся в переменной myString, будет добавлено три символа тире.

3.3. Вопросы для повторения

1. Преобразование переменных из одного типа в другой.
2. Явные и неявные преобразования типов.
3. Сложные типы переменных.
4. Перечислимый тип, его определение и использование.
5. Структуры, их определение и использование.
6. Массивы: одномерные и многомерные.
7. Цикл foreach.
8. Строки, действия над строками.

4. ФУНКЦИИ

Все программы до этого момента имели форму единого блока, в ряде случаев включавшем циклы, позволяющие многократно выполнять отдельные строки кода, и ветвление, позволяющее выполнять некоторые операторы по условию. Если возникала необходимость выполнить какую-либо операцию над данными, то нужный код должен был находиться там, где он будет работать.

Однако такая структура кода весьма ограниченна. Довольно часто возникают ситуации, когда выполнение определенных задач – например, поиск максимального элемента массива – необходимо в различных местах программы. Конечно, можно размещать идентичные (или почти идентичные) участки кода в разных частях приложения каждый раз, когда в этом возникает необходимость, однако такой подход имеет свои недостатки. Изменение даже одной небольшой детали, касающейся выполнения общей задачи (например, исправление некоторой ошибки), потребует внесения изменений во многих местах, возможно, разбросанных по всему приложению, а размер таких приложений бывает очень большим. Пропуск одного места может привести к непоправимым последствиям и неверному использованию всего приложения.

Решением такого рода проблем является применение функций. Функции в C# – это средство, позволяющее выполнять некоторые участки кода в произвольном месте приложения.

Функции особого типа, которые мы будем рассматривать в настоящей лекции, известны под названием методов. Однако при программировании в среде .NET этот термин имеет особое значение, которое станет понятным по мере изучения, поэтому на данный момент мы будем избегать его использования.

Например, можно написать функцию, которая осуществляет поиск максимального элемента массива. В результате появится возможность использовать эту функцию из произвольной точки программы, причем в каждом случае будут выполняться одни и те же строки кода. Поскольку мы должны написать этот код только один раз, то изменения, которые потребуется в него внести, повлияют на все вычисления, где бы этот код ни использовался. Такую функцию можно представить себе как повторно используемый код.

Функции также обладают тем преимуществом, что они позволяют делать программу более удобочитаемой, и мы получаем возможность группировать вместе логически связанные между собой части программ. Поступая таким образом, можно сделать тело самого приложения небольшим, поскольку решение внутренних задач приложения будет осуществляться отдельно. Это напоминает способ, посредством которого в VS можно соединять различные участки кода, используя режим схематического просмотра программ (outline view), что позволяет придать приложению более логичную структуру.

Функции могут также использоваться для создания многоцелевых программ, которые выполняют одни и те же операции над различными данными. Мы имеем возможность передавать функциям информацию, с которой они должны работать, в виде параметров и получать результаты работы функции в виде возвращаемых значений. В приведенном выше примере можно передать функции в качестве параметра массив, в котором осуществляется поиск, и получить элемент массива с максимальным значением в качестве возвращаемого значения. Отсюда следует, что мы можем каждый раз использовать одну и ту же функцию для работы с различными массивами. Параметры и возвращаемое значение функции вместе называются сигнатурой функции.

4.1. Описание и использование функций

В данном разделе рассказывается, каким образом можно включать функции в состав приложений, а затем использовать (вызывать) их из кода. Сначала будут рассмотрены простые функции, не обменивающиеся данными с вызывающим их кодом, а затем мы перейдем к более сложному использованию функций.

Рассмотрим следующий пример.

```
class Classl {
static void Write()
{
      Console.WriteLine("Text output from function.");
}
static void Main(string[] args) {
      Write(); } }
```

Как это работает

Следующие две строки кода описывают простую функцию с именем Write ():

```
static void Write () {
      Console.WriteLine("Text output from function."); }
```

Код, содержащийся в данной функции, просто выводит некоторый текст в консольном окне. Однако это не столь важно, поскольку в настоящий момент нас в большей степени интересуют механизмы, лежащие в основе описания и использования функции.

В данном случае описание функции состоит из:

- двух ключевых слов: static и void;
- имени функции, за которым располагаются параметры: Write();
- участка выполняемого кода, заключенного в фигурные скобки;

Код, который используется для описания функции Write(), выглядит почти так же, как и код самого приложения:

```
static void Main(string[] args) {}
```

Это объясняется тем, что весь код, который мы создавали до сих пор (не считая описания типов), представляет собой часть некоторой функции. Эта функция – Main() – выполняет (как упоминалось в первой лекции) функцию точки входа в консольное приложение. Когда запускается приложение, написанное на C#, то происходит вызов содержащейся в нем функции точки входа, а когда эта функция заканчивает свою работу, выполнение приложения прерывается. Любой выполняемый код, написанный на C#, должен иметь точку входа.

Единственное отличие между функцией Main() и функцией Write() (не считая тех строк кода, которые в них содержатся) заключается в том, что в круглых скобках, расположенных за именем функции Main, находится некоторый код. Этот код служит для задания параметров; к более подробному его обсуждению мы вернемся несколько позже.

Как уже упоминалось ранее, обе функции – и Main(), и Write() – описываются с использованием ключевых слов static (статический) и void (отсутствует). Ключевое слово static имеет отношение к понятиям объектно-ориентированного программирования и его рассмотрение отложим. На данном этапе от требуется запомнить только то, что все функции, которые будут задействованы в приложениях данного раздела, обязательно должны использовать это ключевое слово.

Ключевое слово void, напротив, объяснить очень просто. Оно указывает, что функция не возвращает никакого

значения. Далее в этой лекции будет рассказано, что необходимо писать в тех случаях, когда у функции имеется возвращаемое значение.

Продолжая рассматривать наше приложение, мы обнаруживаем код, который осуществляет вызов функции Write().

Он состоит из имени функции, за которым помещаются пустые круглые скобки. Когда выполнение программы достигнет этой точки, начнет выполняться код, содержащийся в функции Write().

Следует обратить внимание, что использование круглых скобок как при описании функции, так и при ее вызове является обязательным.

4.2. Возвращаемые значения

Самый простой способ обмена данными с функциями – использование возвращаемого значения. Функции, в которых применяются возвращаемые значения, точно так же обладают численным значением, как и любые переменные, используемые при вычислении выражений. Аналогично переменным возвращаемые значения обладают типом.

Например, можно описать функцию с именем getString(), возвращаемое значение которой будет иметь тип string, и использовать ее в своей программе:

```
string myString;
myString = getString();
```

С другой стороны, можно описать функцию с именем getVal(), которая будет возвращать значение типа double, и использовать ее в математическом выражении:

```
double myVal;
double multiplier =5.3;
myVal = getVal() * multiplier;
```

Если функция должна обладать возвращаемым значением, то необходимо внести два изменения:

- в описании функции вместо ключевого слова void указать тип возвращаемого значения;
- по завершении всех вычислений в функции использовать ключевое слово return и передать возвращаемое значение вызывающему коду.

Синтаксис кода для рассматриваемого типа функций консольного приложения будет выглядеть следующим образом:

```
static <возвращаемыйТип> <имяФуккции>() {
    return <возвращаемоеЗначение>; }
```

Единственным ограничением в данном случае является требование, гласящее, что <возвращаемоеЗначение> должно иметь тип <возвращаемыйТип> или же должна существовать возможность его неявного преобразования в этот тип. Вообще говоря, <возвращаемыйТип> может быть любым, включая самые сложные типы из числа рассмотренных ранее.

В простейшем случае это может выглядеть следующим образом:

```
static double getVal() {
    return 3.2; }
```

Однако в реальной жизни возвращаемые значения обычно являются продуктом выполняемых функцией некоторых вычислений, поскольку того же результата можно достигнуть простым использованием переменной типа const.

Когда при выполнении программы достигается оператор return, управление немедленно передается обратно в вызывающий код. Никакие строки кода после этого оператора выполняться не будут. Отсюда, однако, совершенно не следует, что в теле функции оператор return обязательно должен быть последним. Он может быть использован и раньше, например, при ветвлении по какому-либо условию. Включение оператора return в цикл for, в блок if или в какую-нибудь другую структуру приведет к немедленному окончанию выполнения как этой структуры, так и всей функции в целом. Например:

```
static double getVal() {
      double checkVal;
      // присваивание переменной checkVal некоторого значения,
      // полученного в результате некоторых вычислений.
      if (checkVal < 5) return 4.7;
            return 3.2; }
```

В данном случае будет возвращено одно из двух значений – в зависимости от значения переменной checkVal.

Имеется единственное ограничение: оператор return должен выполняться до того, как будет достигнута закрывающая фигурная скобка } данной функции. Следующий код не является допустимым:

```
static double getVal() {
double checkVal;
// присваивание переменной checkVal значения,
// полученного в результате некоторых вычислений.
if (checkVal < 5) return 4.7;
```

Если checkVal >= 5, то не встретится ни одного оператора return, а это запрещено. Все ветви должны оканчиваться этим оператором.

И последнее замечание: оператор return может применяться в функциях, объявленных с использованием ключевого слова void (у них отсутствует какое-либо возвращаемое значение). В таких случаях функция просто прекращает работу. Поэтому при использовании оператора return будет ошибкой размещать возвращаемое значение между ключевым словом return и следующей за ним точкой с запятой.

4.3. Параметры

Если функция должна получать параметры, то необходимо задать:

- список принимаемых функцией параметров в ее описании, а также типы этих параметров;
- совпадающий список параметров при каждом вызове функции.

Это предполагает использование следующего кода:

```
static <возвращаемыйТип> <имяФункции>(<типПараметра> <имяПараметра>, . . .) {
return <возвращаемоеЗначение>; }
```

Здесь может быть произвольное число параметров, для каждого из которых указываются тип и имя. В качестве разделителя между параметрами ставятся запятые. Каждый из параметров доступен внутри данной функции в качестве переменной.

Например, следующая простая функция принимает два параметра типа double и возвращает их произведение:

```
static double product(double param1, double param2) {
    return param1 * param2; )
```

4.3.1. Соответствие параметров

При вызове функции ее параметры должны в точности соответствовать ее описанию. Необходимо совпадение типов параметров, их количества и порядка их следования. Это означает, что, например, функция

```
static void myFunction(string myString, double myDouble)
```

не может быть вызвана с использованием строки myFunction(2.6, "Hello") ;

В данном случае пытаются передать значение типа double в качестве первого параметра, а значение типа string – в качестве второго, что не соответствует порядку, в котором эти параметры содержатся в описании функции. Нельзя использовать и такую строку:

```
myFunction("Hello");
```

поскольку в ней передается только один параметр типа string вместо двух обязательных параметров.

Попытка воспользоваться любой из этих двух строк для вызова функции приведет к ошибке при компиляции, так как компилятор налагает требование точного соответствия сигнатурам вызываемых функций.

Возвращаясь к предыдущему примеру, такое требование означает, что функция MaxValue() может использоваться только для получения максимального целого из массива целых чисел. Если мы изменим код в Main()следующим образом:

```
static void Main(string[] args) {
double[] myArray = {1.3, 8.9, 3.3, 6.5, 2.7, 5.3};
double maxVal = MaxValue(myArray};
Console.WriteLine("The maximum value in myArray is {0}",maxVal);
```

то такой код не пройдет процедуру компиляции из-за неверного типа параметра.

Ниже, в разделе, посвященном перегрузке операторов, описан способ, позволяющий решить эту проблему.

4.3.2. Массивы параметров

В C# предусмотрена возможность задания одного (и только одного) специального параметра функции. Этот параметр, который обязательно должен располагаться последним, известен под названием массива параметров. Он позволяет при обращении к функциям использовать переменное количество параметров. Массив параметров описывается посредством ключевого слова params.

Массивы параметров служат одним из способов упрощения кода, поскольку благодаря им не приходится передавать массивы из вызывающего кода. Напротив, можно передать несколько параметров одного и того же типа, которые помещаются в массив для последующего использования внутри функции.

При описании функции с массивом параметров применяется следующий код:

```
static <возвращаемьйТип <имяФункции>(<п1тип> <п1Имя>, . . . ,
params <тип>[] <имя>)
return <возвращаемоеЗначение>; }
```

Для того чтобы вызвать эту функцию, потребуется следующий код:

```
... , <значение1>, <значение2>, ...);
```

В данном случае <значение1>, <значение2> и т.д.– это значения типа <тип>. Они используются для инициализации массива с именем <имя>. Никаких ограничений на количество параметров, которые могут быть здесь заданы, не существует; единственное предъявляемое к ним требование – они все должны быть одного типа <тип>. Можно вообще не задавать ни одного параметра.

Эта последняя особенность делает массивы параметров полезными, в частности, при задании некоторой дополнительной информации, которая будет использоваться функцией. Например, допустим, что имеется функция getWord(), которая принимает значение типа string в качестве первого параметра и возвращает первое слово из этой строки:

```
string firstWord = getWord("This is a sentence.");
```

Переменной firstord будет присвоена строка "This".

Мы можем добавить в функцию getWord() параметр params, который позволит выбирать и возвращать другое слово по его индексу:

```
string firstWord = getWord("This is a sentence.", 2);
```

Если исходить из предположения, что нумерация слов начинается с 1, то в результате такого вызова переменной firstWord будет присвоена строка "is".

Можно ограничить количество возвращаемых символов, введя третий параметр, что совершенно допустимо для параметра params:

```
string firstWord = getWord("This is a sentence.", 4, 3);
```

В этом случае переменной firstWord будет присвоена строка "sen". Давайте разберем пример полностью.

4.3.3. Передача параметров по ссылке и по значению

Во всех предыдущих функциях применялись параметры, передаваемые по значению. Другими словами, мы передавали значение в переменную, которая затем использовалась внутри функции Любые изменения, которые эта переменная претерпевала внутри функции, не оказывали никакого влияния на параметр, использованный при вызове функции В качестве примера рассмотрим функцию, которая удваивает передаваемый ей параметр и выводит результат на экран:

```
static void showDouble(int val) {
val *= 2;
Console.WriteLine("val doubled = {0}", val); }
```

В этой функции происходит удвоение передаваемого параметра. Если же вызвать эту функцию следующим образом:

```
int myNumber = 5;
Console.WriteLine("rayNumber = {0}", myNumber);
showDouble(myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

то выходной поток, выведенный на консоль, будет таким:

```
myNumber = 5;
val double = 10;
myNumber = 5;
```

Вызов функции showDouble() с переменной myNumber в качестве параметра не оказывает никакого влияния на переменную myNumber, описанную в Main (), несмотря на то, что параметр, которому присваивается ее значение, удваивается.

Однако, чтобы значение переменной myNumber было изменено, возникнет проблема. Конечно, можно воспользоваться функцией, которая возвращает новое значение переменной myNumber, и вызвать ее следующим образом:

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
myNumber = showDouble(myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

Однако в таком коде довольно трудно разобраться; кроме того, такой способ не позволяет отразить изменения значений переменных, используемых в качестве параметров (поскольку функции всегда возвращают только одно значение).

Вместо этого следует передавать параметр по ссылке. Это означает, что функция будет работать именно с той переменной, которая использовалась при вызове функции, а не с переменной, которая имеет то же значение. Следовательно, любые изменения переменной, использовавшейся функцией, отразятся на значении переменной, использовавшейся в качестве параметра. Для этого при описании параметра необходимо воспользоваться ключевым словом ref:

```
static void showDouble(ref int val) {
        val *= 2;
        Console.WriteLine("val doubled= {0)", val); }
```

Вторично ключевое слово ref необходимо использовать при вызове функции (это обязательное требование, поскольку тот факт, что параметр описан как ref, является неотъемлемой частью сигнатуры функции).

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
showDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

В этом случае на консоль будет выведен следующий текст:

```
myNumber = 5;
val double d = 10;
myNumber = 10;
```

На этот раз переменная myNumber была изменена в функции showDouble ().

Для переменных, используемых в качестве параметров типа ref, имеются два ограничения. Во-первых, поскольку существует вероятность, что в результате вызова функции значение вызываемого по ссылке параметра будет изменено, то при вызове функции запрещается использовать переменные типа const. Отсюда следует, что следующий вызов является недопустимым:

```
const int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
showDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

Во-вторых, необходимо использовать заранее инициализированную переменную. C# не гарантирует, что параметр типа ref будет инициализирован той функцией, в которой он используется. Следующий код также является недопустимым:

```
int myNumber;
showDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

4.3.4. Выходные параметры

В дополнение к возможности передавать параметры по ссылке мы можем указать, что данный параметр является выходным: в описание такого параметра включается ключевое слово out, используемое так же, как и ключевое слово ref (в качестве модификатора параметра в описании функции и при вызове функции). Фактически этот способ предоставляет почти такие же возможности, что и передача параметров по ссылке, в том смысле, что значение параметра после выполнения функции попадает в переменную, использовавшуюся при вызове этой функции. Имеются, однако, и существенные отличия. Хотя использовать переменную, которой не присвоено начальное значение, в качестве параметра типа ref недопустимо, она может применяться в качестве параметра типа out. Более того, параметр типа out будет рассматриваться как не имеющий начального значения самой функцией, в которой он используется. Это означает, что хотя передача переменной, которой присвоено некоторое значение, в качестве параметра типа out является допустимой, однако в процессе выполнения функции хранящееся в этой переменной значение будет утрачено.

В качестве примера рассмотрим расширение функции maxVa1ue(), которая возвращает элемент массива с максимальным значением. Модифицируем эту функцию так, чтобы получать индекс элемента массива, содержащего наибольшее значение; в случаях, когда максимальное значение содержится в нескольких элементах, мы будем получать индекс первого максимального элемента. Для этого добавим выходной параметр:

```
static int MaxValue(int[] intArray, out int maxIndex)
{
        int maxVal = intArray[0];
        maxIndex = 0;
        for (int i = 1; i < intArray.Length; i++)
        {
            if (mtArray[i] >maxVal) {
                    maxVal = intArray[i];
                    maxIndex = i;
            }
    }
return maxVal; }
```

Эта функция может быть использована таким образом:

```
int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
int maxIndex;
Console.WriteLine("The maximum value in myArray is {0}",
maxValue(myArray, out maxIndex));
Console.WriteLine("The first occurrence of this value is at element {0}", maxIndex + 1) ;
```

Результатом ее выполнения будет следующее:

```
The maximum value in myArray is 9.
```

(Максимальное значение, содержащееся в массиве myArray,– 9).

```
The first occurrence of this value is at element 7.
```

(Первое вхождение с таким значением найдено в элементе 7).

Важным моментом, на который следует обратить внимание, является необходимость использовать ключевое слово

out при вызове функции (так же, как и ключевое слово ref).

Следует обратить внимание, что в тексте программы к значению переменной maxIndex при выводе ее на экран прибавляется единица. Это сделано для того, чтобы, придать индексу более удобную для восприятия форму, как если бы первый элемент массива считался элементом 1, а не элементом 0.

4.4. Область действия переменных

Возможно, читая предыдущую лекцию, вы задавались вопросом, зачем вообще нужно обмениваться данными с функциями. Ответ заключается в том, что в C# доступ к переменным может осуществляться только из определенных участков кода. Принято говорить, что переменная имеет определенную область действия, в рамках которой она является доступной.

Область действия переменной представляет собой очень важное понятие.

Перед тем как пойти дальше, следует обратить внимание на то, что важность заявлений, сделанных в предшествующем разделе, далеко выходит за рамки определения области действия переменных в разных функциях. Утверждалось, что область действия переменных распространяется на блок кода, в котором они описаны, и на все блоки, непосредственно в него вложенные. Это также применимо и к другим типам блоков, например, к программным конструкциям, использующим ветвление и циклы. Рассмотрим следующую программу:

```
int i;
for (i = 0; i < 10; i++)
{
    string text = "Line" + Convert.ToString(i);
    Console.WriteLine("{0}", text); };
Console.WriteLine("Last text output in loop: {0}", text);
```

В данной программе строковая переменная text является локальной для цикла for. Такой код не пройдет компиляцию, поскольку в обращении к Console.WriteLine(), которое происходит вне этого цикла, делается попытка использовать переменную text, область действия которой не распространяется за пределы цикла. Изменим код следующим образом:

```
int i;
string text;
for (i = 0; i < 10; i++)
{
    text = "Line " + Convert.ToString(i);
    Console.WriteLine("{0}", text); };
Console.WriteLine("Last text output in loop: {0}", text);
```

Этот код также недопустим. Причина кроется в том, что переменные должны и описываться, и инициализироваться до того, как они будут использоваться, а переменная text инициализируется только в цикле for. Значение, присвоенное переменной text, при выходе из цикла будет утрачено. Однако мы можем модифицировать код еще раз:

```
int i;
string text = " ";
for (i = 0; i < 10; i++)
{
    text = "Line" + Convert.ToString(i);
    Console.WriteLine("{0}", text); }
Console.WriteLine("Last text output in loop: {0}", text);
```

На этот раз переменная text инициализирована вне цикла, и мы имеем доступ к ее значению. Результат выполнения этой простой программы показан на рисунке слева.

В данном случае значение, присвоенное переменной text внутри цикла, оказывается доступным и вне его.

Как вы, вероятно, заметили, эта тема требует определенных усилий для понимания. В первый момент представляется не вполне очевидным, почему – в свете предыдущего примера – переменная text не сохраняет в качестве своего значения пустую строку, присвоенную ей перед началом цикла, для кода, расположенного после его окончания.

Причина кроется в способе выделения памяти для переменной text, да и для всех остальных переменных тоже. Простое объявление переменной некоторого простого типа не влечет за собой выполнения каких-либо существенных действий. Только тогда, когда переменным присваиваются значения, для этих значений выделяется память, в которой они будут храниться. Когда такое выделение памяти происходит внутри цикла, это значение определяется как локальное и область его действия не выходит за пределы цикла. И хотя переменная не является локализованной внутри данного цикла, к ее значению это не относится. Напротив, присваивание переменной значения вне цикла дает гарантию того, что это значение локальное для всего основного кода и что область действия переменной распространяется в том числе и на цикл. Другими словами, мы будем оставаться внутри области действия переменной до тех пор, пока не покинем блок основного кода, поэтому возможность доступа к переменной имеется и за пределами цикла.

Компилятор C# самостоятельно обнаруживает проблемы, связанные с областью действия переменных, и генерирует сообщения об ошибках. В качестве заключительного замечания необходимо упомянуть о "наилучшей практике". Вообще говоря, лучше всего объявлять и инициализировать переменные перед теми блоками кода, в которых они используются. Исключением из этого правила могут быть переменные циклов, объявление которых является составной частью самого цикла. Например:

```
for (int i = 0; i < 10; i++)
{
}
```

В данном случае, переменная i является локализованной в блоке кода, представляющем собой цикл, но это и хорошо, поскольку редко когда требуется доступ к значению счетчика цикла из внешнего кода.

Параметры и возвращаемые значения по сравнению с глобальными данными.

Этот раздел посвящен более детальному рассмотрению обмена данными с функциями через механизм глобальных переменных и через механизм параметров и возвращаемых значений. Напомним, что речь идет о различиях между двумя программами. Первая из них имеет вид:

```
class Classl {
    static void showDouble(ref int val) {
    val *= 2;
    Console.WriteLine("val doubled = {0}", val); }
    static void Main(string[] args) (
```

```
    int val = 5;
    Console.WriteLine("val = {0}", val);
    showDouble(ref val) ;
    Console.WriteLine("val = {0}", val); } }
```

Этот код несколько отличается от того кода, где в функции Main() использовалась переменная с именем myNumber. Это иллюстрация того факта, что локальные переменные могут обладать идентичными именами и не мешать друг другу. Это также означает, что две программы, которые приводятся здесь, оказываются очень похожими друг на друга, позволяя нам сосредоточить внимание на принципиальных отличиях и не беспокоиться об именах переменных.

Вторая программа имеет следующий вид:

```
class Classl {
    static int val;
    static void showDouble() {
    val *= 2;
    Console.WriteLine("val doubled = {0}", val); }
    staticvoidMain(string[] args) {
    val = 5;
    Console.WriteLine("val = {0}", val);
    showDouble ();
    Console.WriteLine("val = {0}", val); } }
```

Результаты выполнения обеих программ совершенно идентичны.

Следует заметить, что не существует жестких правил, определяющих предпочтительность использования одного способа вместо другого: оба эти способа являются вполне допустимыми. Тем не менее существуют некоторые соображения, которые желательно учитывать при выборе одного из них.

Для начала обратите внимание на то, что версия функции showDouble(), которая использует глобальное значение, сможет использовать только глобальное значение переменной val. Применение такой версии функции принуждает нас использовать именно эту глобальную переменную, что несколько ограничивает гибкость в применении данной функции и требует постоянно копировать значение глобальной переменной в другие переменные в том случае, если необходимо сохранить результат. Кроме того, глобальные данные могут быть изменены где-нибудь в другом месте приложения, что приведет к получению непредсказуемых результатов (мы можем вспомнить о том, что значения изменены, слишком поздно).

Однако такая потеря гибкости зачастую является преимуществом. Существуют ситуации, когда применение некоторой функции планируется исключительно для одной конкретной цели и использование глобально описанных данных уменьшает вероятность того, что мы неправильно обратимся к функции, например, передав неверные параметры.

Конечно, такая простота в действительности делает программу менее понятной. Явное указание параметров позволяет увидеть, что именно подвергается изменениям, даже мельком взглянув на функцию. Если встречается вызов функции, имеющий вид myFunction(val1, out va12), то сразу становится ясно, что следует обратить самое пристальное внимание на переменные val1 и val2 и что переменной val2 по окончании выполнения функции будет присвоено новое значение. И наоборот, если у этой функции не имеется никаких параметров, то мы не сможем сделать предположения относительно того, над какими данными она выполняет манипуляции.

Наконец, не следует забывать о том, что использование глобальных данных не всегда оказывается возможным. Далее нам встретятся примеры программ, хранящихся в разных файлах и/или принадлежащих различным пространствам имен, которые взаимодействуют между собой посредством использования функций. В таких случаях код зачастую оказывается до такой степени разделенным, что непонятно, где именно следует хранить глобальные переменные.

Итак, оба способа обмена данными можно использовать с равным успехом. В большинстве случаев мы будем настаивать на применении параметров вместо глобальных данных, однако существуют отдельные ситуации, когда более подходящим может оказаться использование именно глобальных данных; и уж, во всяком случае, такой способ не является ошибкой.

4.5. Функция Main()

Теперь, когда вы познакомились почти со всеми простыми способами, применяемыми для создания и использования функций, давайте вернемся назад и более подробно рассмотрим функцию Main ().

Как было сказано ранее, функция Main() является точкой входа в приложение на C# и выполнение этой функции охватывает выполнение приложения. У этой функции имеется параметр – string[] args, однако до сих пор мы его не описывали. В данном разделе вы познакомитесь с этим параметром и узнаете, каким образом его можно применять.

Существуют четыре различные сигнатуры, которые можно использовать для функции Main():

- static void Main();
- static void Main(string[] args);
- static int Main();
- static int Main(string[] args).

Естественно, следует обратить внимание, что никакие знаки препинания после функции Main() не ставятся, здесь они стоят исключительно вследствие требований грамматики русского языка.

При желании аргумент args можно опускать. Причина, по которой мы до сих пор пользовались вариантом с этим аргументом,– в том, что такая версия генерируется автоматически при создании консольного приложения в VS.

Третья и четвертая версии сигнатуры, приведенные выше, возвращают значение типа int, которое может быть использовано для указания на то, каким образом завершилось выполнение приложения, и которое часто используется для определения ошибки (хотя это в любом случае является обязательным). Обычно, возвращаемое значение равно 0, что означает "нормальное" окончание (т. е. приложение закончило свою работу и может быть завершено в безопасном режиме).

Параметр args функции Main() представляет собой метод, позволяющий приложению получать извне информацию, задаваемую в процессе его выполнения. Эта информация принимает форму параметров командной строки.

Параметры командной строки вам уже наверняка встречались. Когда производится запуск приложения из командной строки, то при этом очень часто имеется возможность непосредственно задать некоторую информацию, например, имя файла, загрузка которого требуется для выполнения приложения. В качестве примера давайте рассмотрим Windows-приложение Notepad. Запустить его можно, просто набрав notepad – либо в окне командного приглашения, либо в окне, которое появляется при выборе опции Run (Выполнить) из меню Start (Пуск). Однако в этих же окнах мы можем набрать нечто вроде: notepad myfile.txt. В результате Notepad либо загрузит файл myfile.txt, либо, если такого файла не существует, предложит его создать. В данном случае "myfile.txt" является аргументом командной строки. Мы можем сами написать консольное приложение, которое будет работать аналогичным образом, использовав параметр args.

Когда консольное приложение запускается, то все заданные параметры командной строки помещаются в массив

args. В дальнейшем эти параметры могут использоваться по мере необходимости.

4.6. Функции структур

В предшествующей лекции рассматривались типы структур, предназначенных для хранения в одном месте нескольких элементов информации. На самом деле возможностей у структур гораздо больше, и одной из них является способность содержать не только данные, но и функции. На первый взгляд подобное может показаться немного странным, однако на практике это приносит много пользы.

В качестве простого примера рассмотрим следующую структуру.

```
struct customerName {
    public string firstName, lastName; }
```

Если у нас имеются переменные типа customerName и нам необходимо вывести на консоль полное имя, мы будем вынуждены конструировать это имя из его составных частей, например, для переменной типа customerName с именем customer можно использовать следующий синтаксис:

```
customerName myCustomer;
myCustomer.firstName = "Jobn";
myCustomer.lastName = "Franklin";
Console.WriteLine("{0} {1}", customer.firstName, customer.lastName);
```

Имея возможность включать в структуры функции, мы можем упростить эту процедуру за счет централизованного выполнения часто встречающихся задач. В данном случае это можно сделать следующим образом:

```
struct customerName {
    public string firstName, lastName;
public string Name () {
    return firstName + lastName; } }
```

Эта функция очень похожа на другие функции, рассматриваемые в этой лекции, за исключением того, что в ней отсутствует модификатор static. Причины этого отсутствия станут понятны позже, а пока достаточно просто запомнить, что это ключевое слово для функций структур не требуется. Мы можем воспользоваться этой функцией следующим образом:

```
customerName myCustomer;
myCustomer.firstName = "John";
myCustomer.lastName = "Franklin";
Console.WriteLine(customer.Name());
```

Этот синтаксис гораздо проще и понятнее, чем предшествующий.

Необходимо отметить, что функция Name() имеет непосредственный доступ к полям структуры firstName и lastName. В рамках структуры customerName они могут считаться глобальными.

4.7. Перегрузка функций

В начале этой лекции описывалось как сравнивать сигнатуру функции при ее вызове. При этом необходимо иметь отдельные функции для работы с переменными различных типов. Наличие оператора перегрузки подтверждает, что последнее действительно является проблемой. Этот оператор предоставляет возможность создавать одновременно несколько функций, имеющих одинаковое имя, но работающих с параметрами различных типов.

Например, ранее мы использовали следующую программу, в которой содержалась функция с именем MaxValue ():

```
class Classl {
    static intMaxValue(int[] intArray) {
        int maxVal = intArray[0];
        for (int i = 1; i < intArray.Length; i++)
        {
            if (intArray[i] > maxVal)
            maxVal = intArray[i];
        }
        return maxVal; }
    static void Main(string[] args)
    {
        int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
        int maxVal = MaxValue(myArray);
        Console.WriteLine("The maximum value in myArray is {0}", maxVal);
    }
    }
```

Эта функция может использоваться только для массивов значений типа int. Мы могли бы создать функции с другими именами, предназначенные для работы с параметрами других типов, переименовав вышеприведенную функцию как-нибудь вроде IntArrayMaxValue() и добавив функции наподобие DoubleArrayMaxValue() для работы с другими типами. В качестве альтернативы мы можем просто включить в нашу программу следующую функцию:

```
static double MaxValue(double[] doubleArray) {
    double maxVal = doubleArray[0];
    for (int i = 1; i < doubleArray.Length; i++)
    {
        if (doubleArray[i] > maxVal)
            maxVal = doubleArray[i];
    }
    return maxVal;
}
```

Разница между двумя функциями заключается в том, что эта функция работает со значениями типа double. Имя функции – MaxValue () – оказывается тем же самым, однако сигнатура (это принципиально) отличается. Было бы ошибкой описать две функции с одинаковым именем и одинаковой сигнатурой, однако поскольку в данном случае сигнатуры различны, то все нормально.

Теперь у нас имеются две версии функции MaxValue (), которые принимают массивы типа int и массивы типа double и возвращают максимальное значение типа int или типа double соответственно.

Красота такой формы программы в том, что не требуется явно указывать, которую из этих двух функций мы

собираемся использовать. Мы просто задаем массив-параметр, и это приводит к выполнению того варианта, который соответствует типу используемого параметра.

На данном этапе стоит отметить еще одну высокоинтеллектуальную черту VS. Если в нашем приложении имеются две одноименные функции, описанные выше, и мы наберем это имя, например, в Main(), то VS выведет доступные варианты перегрузки данной функции. Если будет набрано следующее:

```
double result = MaxValue(
```

то VS выведет информацию по обеим версиям функции MaxValue (), которые мы можем просмотреть с помощью кнопок "стрелка вверх" и "стрелка вниз". При перегрузке функций учитываются все аспекты, касающиеся их сигнатур. Существует возможность, например, описать две различные функции, одна из которых принимает параметры по значению, а другая, соответственно, по ссылке:

```
static void showDouble(int val) {
}
static void showDouble(ref int val) {
}
```

Выбор используемой версии осуществляется исключительно на основании того, имеется ли в обращении к функции ключевое слово ref. При следующем вызове будет использован вариант, в котором параметр передается по ссылке:

```
showDouble(ref val);
```

А такой вызов позволит передать параметр по значению:

```
showDouble(val);
```

Аналогичным образом можно описывать функции, отличающиеся числом требующихся им параметров и т. п.

4.8. Вопросы для повторения

1. Описание функций.
2. Параметры функций.
3. Возвращаемые значения.
4. Сигнатура функций.
5. Передача параметров по ссылке и по значению.
6. Область действия переменных.
7. Функция Main.
8. Перегрузка функций.

5. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

В предыдущих лекциях мы рассмотрели синтаксис языка C# и основы программирования на нем. Кроме того, мы научились собирать работоспособные консольные приложения. Однако для того, чтобы получить доступ ко всему потенциалу C# и .NET Framework, необходимо научиться использовать методы объектно-ориентированного программирования (ООП).

5.1. Объектно-ориентированное программирование

Объектно-ориентированное программирование – это относительно новый подход к созданию компьютерных приложений, в котором предпринимается попытка решить многие проблемы, возникающие при применении "традиционного" программирования. Тип программирования, с которым вы были до сих пор знакомы, известен под названием функционального (или процедурного) программирования. Он очень часто приводит к созданию так называемых монолитных приложений, в которых все функциональные возможности содержатся в небольшом числе модулей (зачастую в одном). Применяя методы ООП, мы очень часто будем использовать большое количество программных модулей, каждый из которых предоставляет только одну функциональную возможность, причем каждый модуль может существовать отдельно от остальных и даже быть совершенно независимым. Такой модульный метод написания программ оказывается более гибким и увеличивает возможности для повторного использования кода.

В качестве более наглядной иллюстрации ООП представим себе сложное приложение в виде высококлассного гоночного автомобиля. Если бы использовались традиционные способы программирования, то этот автомобиль представлял бы собой единое целое. И пожелай мы его усовершенствовать, нам бы пришлось заменить его целиком, отправив на завод-изготовитель для доводки заводскими специалистами (или купить новый). Если же применяются методы ООП, то можно просто приобрести новый двигатель и, следуя прилагающимся к нему инструкциям, заменить его самостоятельно.

В "традиционном" приложении порядок выполнения прост и прямолинеен. Приложения загружаются в память, их выполнение начинается в точке A и завершается в точке B, после чего они выгружаются из памяти. В процессе выполнения такого приложения могут использоваться самые разнообразные объекты, например, файлы на различных носителях или возможности, предоставляемые видеокартой, однако основное тело вычислений расположено в едином месте. Весь код, как правило, предназначается для манипуляций над данными с помощью различных математических и логических средств. Способы такого манипулирования обычно очень просты благодаря применению базовых типов (таких как целый и логический) для более сложного представления данных.

При использовании ООП такая прямолинейность встречается редко. И хотя результат достигается тот же самый, способ его получения зачастую оказывается совершенно отличным. Методы ООП в большей степени основываются на структуре и значении данных, а также на взаимодействии между различными данными. Это обычно требует больших усилий при разработке проекта, но дает преимущество расширяемости. После того как соглашение о способе представления конкретного типа данных достигнуто, оно может использоваться и в более поздних версиях приложения, и в совершенно новых приложениях. Тот факт, что такое соглашение уже существует, позволяет радикально сократить время, затрачиваемое на разработку. Это позволяет объяснить вышеприведенный пример с гоночным автомобилем. В этом случае в качестве соглашения применяется такая структура кода для "двигателя", которая позволяет использовать новый код (новый двигатель) взамен старого без всяких проблем и не требует отправки автомобиля на завод.

Кроме соглашений о представлении данных, ООП часто позволяет упростить жизнь за счет соглашений о структуре и об использовании более абстрактных сущностей. Например, соглашение может быть достигнуто не только относительно формата данных, который должен применяться при отправке выходного потока на какое-либо устройство вроде принтера, но и относительно способов обмена данны-, ми с этим устройством. Это будет касаться инструкций, которые оно может воспринимать и в дальнейшей работе.

Как можно предположить по названию, все это достигается за счет использования некоторых объектов. Так что же собой представляет объект?

5.2. Объект

Объект – это строительный блок ООП-приложения. В этом блоке инкапсулируется некоторая часть приложения, которая может представлять собой процесс, группу данных или какую-либо более сложную сущность.

В простейшем смысле объект очень напоминает тип структуры, с которым вы познакомились раньше и в котором содержатся члены в виде переменных и функций различных типов. Совокупность переменных представляет данные, хранящиеся в этом объекте, а функции обеспечивают доступ к функциональным возможностям объекта. Несколько более сложные объекты могут вообще не содержать никаких данных; вместо это они могут представлять некоторый процесс и состоять исключительно из функций. Например, можно использовать объект, представляющий принтер, которой состоит из функций, обеспечивающих контроль за работой этого устройства (распечатать документ, распечатать тестовую страницу и т. д.).

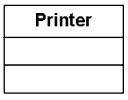

Рис. 6.1 UML - Представление класса Printer

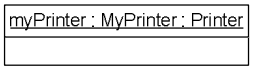

Рис. 6.2. UML – представление экземпляра класса Printer

Объекты в C# создаются на основе типов точно так же, как и переменные. Тип объекта известен в ООП под специальным названием "класс объекта". Мы можем использовать определения классов для создания экземпляров объектов, что означает создание реального поименованного экземпляра данного класса. Словосочетание "экземпляр данного класса" и термин "объект" в данном случае означают одно и то же, однако уверяем вас, что, вообще говоря, "класс" и "объект" обозначают фундаментально отличающиеся понятия.

В данной главе мы будем описывать классы и объекты с использованием синтаксиса Unified Modeling Language (UML – унифицированный язык моделирования). UML – это язык, разработанный для моделирования приложений с использованием тех объектов, из которых они строятся, тех операций, которые они выполняют, и тех вариантов их применения, для которых они предназначаются. В данном случае мы будем использовать только основы этого языка, объясняя требующиеся нам понятия по ходу дела и не отвлекаясь на более сложные аспекты.

Диаграммы, использованные в данной главе, создавались с помощью программы Microsoft Visio, которая поставляется с коммерческой версией VS.

На рисунке 6.1 приводится UML-представление класса, предназначенного для описания принтера, названного

Printer. Имя класса изображено в верхней секции этого квадрата (к двум его нижним секциям мы обратимся немного позднее).

На рисунке 6.2 изображено UML-представление экземпляра класса Printer с именем myPrinter. Здесь в верхней секции располагается имя экземпляра, за ним следует имя соответствующего класса. Имена разделяются двоеточием.

5.2.1. Свойства и поля

Свойства и поля обеспечивают доступ к данным, хранящимся в объекте. Данные, содержащиеся в объектах,– это то, что отличает один объект от другого, поскольку в различных объектах одного и того же класса могут храниться различные значения свойств и полей.

Теперь нам потребуется ввести еще один термин. Различные части информации, хранящиеся в объекте, в совокупности определяют состояние объекта.

Представьте себе класс объектов с именем CupOfCoffee (чашка кофе), описывающий чашку кофе. При создании экземпляра этого класса (т. е. при создании объекта данного класса) мы должны описать его состояние, для того чтобы объект был осмысленным. В этом случае мы могли бы использовать такие свойства и поля, которые позволят программе, использующей этот объект, задавать марку кофе, определять, добавлены ли в кофе сахар и молоко, является ли этот кофе растворимым и т. д. В таком случае объект, представляющий чашку кофе, будет обладать определенным состоянием, например, "колумбийский фильтрованный кофе с молоком и двумя кусочками сахара".

Как поля, так и свойства являются типизированными, поэтому информацию, содержащуюся в них, можно хранить в переменных типа string, int и т. п. Однако свойства отличаются от полей тем, что они не обеспечивают непосредственного доступа к данным. Объекты обладают возможностью изолировать пользователей, которым не требуется точное представление о структуре существующих свойств, от реального устройства своих данных. Если для описания числа кусочков сахара в экземпляре CupOfCoffee мы используем поле, то пользователи смогут занести в это поле любое значение, какое им заблагорассудится; если же мы используем свойство, то мы сможем ограничить это значение диапазоном от 0 до 2.

Вообще говоря, для организации доступа к состоянию лучше использовать свойства, а не поля, поскольку в этом случае мы обладаем большим контролем за происходящим, хотя синтаксис и в том, и в другом случае применяется один и тот же. Режим доступа к свойствам также может быть четко определен для данного объекта. Некоторые свойства могут использоваться в режиме "только чтение", что дает возможность просматривать их, но не изменять (по крайней мере, непосредственно). Очень часто полезными оказываются способы, позволяющие одновременно считывать несколько различных частей информации. Например, мы можем описать свойство класса CupOfCoffee с именем Description (описание), использующееся в режиме "только чтение", которое при обращении будет возвращать строку, описывающую состояние некоторого экземпляра данного класса (например, строку, приведенную выше). Эти же самые данные можно было бы получить, обращаясь к нескольким различным свойствам, однако использование одного свойства позволяет экономить время и усилия. Аналогичным образом можно использовать свойства, работающие в режиме "только запись".

Подобно тому как мы имеем возможность установить режим чтения/записи свойств, мы можем задать тип доступа совместно для полей и свойств. Это позволяет определять, какие программы имеют доступ к этим членам: доступны ли они для всего кода (public – общие), только для кода в рамках данного класса (private – частные), или же доступ к ним определяется по более сложной схеме (этот вопрос будет рассматриваться в настоящей главе по мере необходимости). Чрезвычайно распространенной является практика, когда поля описываются как частные, а доступ к ним организовывается посредством свойств, описанных как общие. Это означает, что код в рамках данного класса обладает непосредственным доступом к данным, хранящимся в поле, а общее свойство скрывает эти данные от внешних пользователей и не позволяет им записывать в эти данные недопустимые значения. Принято говорить, что общие члены предоставляются данным классом. Для большей наглядности давайте отождествим это с областью действия переменной. Частные поля и свойства в таком случае можно представить как локальные по отношению к объекту, которому они принадлежат, в то время как область действия общих полей и свойств также распространяется и на код, внешний по отношению к данному объекту.

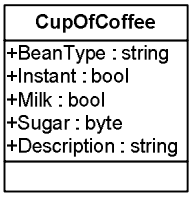

Рис. 6.3. UML – представление класса CupOfCoffee

При представлении класса посредством UML вторая секция используется для изображения свойств и полей.

На рисунке 6.3 можно видеть представление нашего класса CupOfCoffee, в котором описано пять членов (неважно, свойств или полей – в UML между ними не существует никаких отличий), обсуждавшихся нами ранее. Каждое вхождение содержит следующую информацию.

– Тип доступа: символ + используется для общего члена, символ - используется для частного члена. В дальнейшем мы не будем изображать частные члены на диаграммах данной главы, поскольку эта информация является внутренней по отношению к классу. Не будет приводится также и информация, касающаяся режима доступа (чтение/запись).
– Имя члена.
– Тип члена.

В качестве разделителя между именем и типом члена используется двоеточие.

5.2.2. Методы

Метод – это термин, используемый для обозначения функций, предоставляемых данным объектом. Методы могут вызываться точно так же, как и любые другие функции, и в них так же, как и в функциях, могут использоваться возвращаемые значения и параметры (более подробно эта тема описывается в главе 6).

Методы применяются для обеспечения доступа к функциональным возможностям объектов. Подобно полям и свойствам, они могут быть общими или частными, ограничивая по мере необходимости доступ к ним из внешнего кода. Методы часто учитывают состояние объекта при своей работе и обладают доступом к частным членам, например, к частным полям. Так, в классе CupOfCoffee мы можем описать метод с именем AddSugar() (добавить сахар), который обеспечит использование более удобного синтаксиса для увеличения значения свойства сладости кофе, чем присвоение соответствующего значения свойству Sugar (сахар).

В UML при описании объектов для изображения функций используется третья секция прямоугольника (см. рис. 6.4).

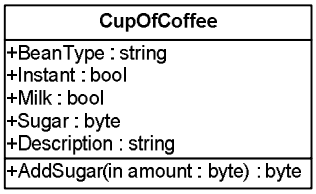

Рис. 6.4. UML – представление класса CupOfCoffee, содержащего метод AddSugar

Используемый в этом случае синтаксис аналогичен применяемому для полей и свойств, за исключением того, что в конце строки указывается тип возвращаемого значения, а также приводятся параметры. В UML каждый параметр изображается с одним из следующих идентификаторов: in, out или inout. Эти идентификаторы используются для обозначения направления потока данных, при этом out и inout в первом приближении соответствуют применяемым в C# ключевым словам out и ref, описанным в главе 6. Идентификатор in примерно соответствует такому поведению C#, когда отсутствуют оба ключевых слова.

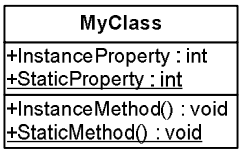

Рис. 6.5. Класс MyClass, содержащий статические поле и метод

Настал момент, когда необходимо, наконец, объясниться. Мы использовали объекты, свойства и методы на всем протяжении этой книги. В действительности в C# и .NET Framework объектом может быть все, что угодно. Функция Main() в консольном приложении является методом класса. Все типы переменных, рассмотренных нами, являются классами. Каждая из использовавшихся команд представляет собой свойство или метод, например, <Строка>.Length, <Строка>.ToUpper() и т. д. В данном случае точка отделяет имя экземпляра объекта от свойства или от имени метода.

Объекты существуют буквально повсеместно, и синтаксис, который требуется для их использования, зачастую оказывается очень простым. Действительно, он настолько прост, что до настоящего времени позволял нам сосредоточиться на более фундаментальных аспектах C#.

С этого момента мы приступаем к более детальному рассмотрению объектов. Имейте в виду, что понятия, которые будут вводиться, обладают весьма широкой областью использования. Они применимы даже к той простой маленькой переменной типа int, с которой вы так любили играться. К счастью, мы можем с успехом использовать их и для более сложных понятий.

Жизненный цикл объекта

У каждого объекта имеется четко определенный жизненный цикл, который длится с момента использования определения класса до уничтожения объекта. Кроме обычного состояния "используется", в жизненном цикле объекта присутствуют еще два важных этапа:

- создание объекта – состояние, когда происходит первоначальное создание экземпляра объекта. Такая инициализация известна под названием создания объекта и осуществляется конструктором объекта;
- уничтожение объекта – состояние, когда происходит уничтожение объекта, при котором очень часто возникает необходимость проведения различных восстановительных мероприятий, например освобождения памяти. Эта работа входит в функции деструктора объекта.

5.2.3. Конструкторы

Инициализация объекта происходит автоматически. Нам не приходится беспокоиться, допустим, о поиске свободной памяти, в которой будет размещен новый объект. Однако иногда возникают ситуации, когда на этапе инициализации может потребоваться выполнение каких-либо дополнительных действий. Например, очень часто бывает необходимо инициализировать данные, хранящиеся в объекте. Как раз это и входит в функции конструктора.

У каждого объекта имеется конструктор, используемый по умолчанию, который представляет собой метод без параметров. Его имя совпадает с именем самого класса. Определение класса может включать в себя несколько конструкторов. Они могут иметь отличающиеся сигнатуры, которые используются для создания экземпляра объекта. Для присваивания начальных значений данным, хранящимся внутри объекта, часто применяются конструкторы с параметрами.

В C# конструктор можно вызвать, введя ключевое слово new. Например, мы можем создать объект типа string следующим образом:

```
string myString = new string();
```

Объекты также могут создаваться с помощью конструктора, используемого не по умолчанию. Как и имя конструктора по умолчанию, имена этих конструкторов совпадают с именем класса, но у них имеются еще и параметры. Используются они аналогичным образом:

```
string myString = new string('a', 10);
```

В результате применения этого конструктора будет создана новая строка, которой в качестве начального значения будет присвоено "аааааааааа", т. е. символ "а", повторенный десятикратно.

Конструкторы, подобно полям, свойствам и методам, могут быть общими или частными. Код, внешний по отношению к данному классу, не может использовать для создания экземпляра объекта частный конструктор; для этого необходимо воспользоваться общим конструктором. Таким способом мы можем, например, заставить пользователей нашего класса применять конструктор, использующийся не по умолчанию.

В некоторых классах вообще отсутствуют общедоступные конструкторы, что означает невозможность для внешнего кода создавать экземпляры данного класса. Однако, как вы вскоре сможете убедиться, такие классы вовсе не являются совершенно бесполезными.

5.2.4. Деструкторы

Деструкторы используются в .NET Framework для того, чтобы выполнять уборку за объектами. В общем случае нам не требуется писать какой бы то ни было код для метода деструктора; напротив, за нас будет работать операция, выполняющаяся по умолчанию. Однако если необходимо выполнить какие-либо важные действия перед уничтожением экземпляра объекта, то можно задать определенные инструкции.

Очень важно запомнить, что метод деструктора объекта не вызывается сразу, как только данный объект перестает использоваться.

Когда, например, мы выходим за область действия переменной, она становится недоступной для нашего кода, но при этом она может по-прежнему существовать где-нибудь в памяти компьютера. Только после того как сборщик мусора среды исполнения .NET выполнит свою работу, она будет окончательно уничтожена.

Это означает, что не следует полагаться на деструктор в плане освобождения ресурсов, которые использовались экземпляром объекта, поскольку объект может оставаться неиспользуемым на протяжении длительного времени. Если используемые ресурсы критичны, то это может привести к возникновению проблем. Однако здесь имеется решение, и оно рассматривается в разделе "Удаляемые объекты" этой главы.

5.2.5. Статические члены класса и члены класса экземпляра

Наряду с такими членами, как свойства, методы и поля, относящимися к конкретным экземплярам объектов, существует возможность использовать статические (static) члены (известные также под названием членов с общим доступом (shared)), среди которых могут быть методы, свойства и поля Доступ к статическим членам имеют все экземпляры данного класса, поэтому их можно представить себе в качестве глобальных для всех объектов данного класса. Статические свойства и поля позволяют осуществлять доступ к данным независимо от экземпляров объектов, а статические методы позволяют выполнять команды, которые относятся к данному типу объектов, но не к конкретным экземплярам объектов. Для того чтобы использовать статические члены нам, по существу, даже не требуется создавать экземпляры объектов с типом данного класса.

Использовавшиеся методы Console.WriteLine() и Convert.ToString() являются статическими. Ни в какой момент времени от нас не требуется создавать экземпляры классов Console или Convert (более того, даже если попытаться это сделать, то все равно ничего бы не получилось, поскольку к конструкторам этих классов не существует общего доступа,

что обсуждалось ранее).

Существуют различные ситуации, при которых статические свойства и методы могут быть использованы с большой пользой.

В синтаксисе UML статические члены классов выделяются подчеркиванием (см. рис. 6.5).

5.3. Ссылочные и значимые типы данных

Все данные в C# хранятся в переменных двумя способами, которые определяются типом переменной. Этот тип может относиться к одной из двух категорий: либо ссылочный тип, либо значимый. Вот в чем состоит различие между ними:

- значимые типы хранятся сами и хранят свое содержимое в одном месте памяти (называемом стеком);
- ссылочные типы содержат ссылку на какую-либо другую область памяти (она называется динамической), в которой хранится содержимое такой переменной.

На самом деле при использовании C# это не должно нас особенно беспокоить. До настоящего момента мы использовали переменные типа string (которые относятся к ссылочному типу) и другие простые переменные (большинство из которых относится к значимому типу, например, int) практически одинаково.

Среди простых типов ссылочными являются только string и object, хотя массивы также неявно представляют собой ссылочный тип. Каждый создаваемый нами класс будет относиться к ссылочному типу – именно поэтому мы и остановились на этом моменте.

5.3.1. Структуры

На данном этапе необходимо обратить внимание на один важный момент. Ключевым отличием структурных типов от классов является то, что первые представляют собой значимые типы.

Тот факт, что структурные типы и классы похожи друг на друга, возможно, стал вам ясен давно, особенно после того, как в главе 6 было сказано, что в структурных типах могут использоваться функции. Этот вопрос более подробно будет рассматриваться в следующей главе.

5.3.2. ООП в приложениях Windows

ПРАКТИКУМ: объекты в действии

1. Создайте новое приложение Windows.
2. Добавьте в приложение новое средство управления Button с помощью панели Toolbox и разместите его в центре Form1, как показано на рисунке слева.
3. Дважды щелкните по кнопке мышью для того, чтобы добавить код, предназначенный для обработки нажатия кнопки мыши. Внесите изменения в появившийся код, как показано ниже:

```
/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main()
{
    Application.Run(new Form1()); }
    private void button1_Click(object sender, System.EventArgs e) {
    ( (Button)sender).Text = "Clicked!";
    Button newButton = new Button();
    newButton. Text = "NewButton!";
    newButton. Click += new EventHandler(newButton_Click);
    Controls.Add(newButton); }
    private void newButton_Click(object sender, System.EventArgs e) {
    ((Button)sender).Text = "Clicked!!"; } } }
```

4. Запустите приложение.
5. Щелкните мышью по кнопке с надписью button1.
6. Щелкните мышью по кнопке с надписью New Button!.

Как это работает

Написав всего несколько строк кода, мы создали Windows-приложение, которое выполняет некоторые действия, одновременно иллюстрируя использование определенных методов ООП в C#. Фраза "объектом может быть все, что угодно" становится еще более справедливой, когда дело касается Windows-приложений. Начиная от формы, в которой это выполняется, и заканчивая средствами управления в этой форме, нам постоянно приходится пользоваться методами ООП. В процессе описания этого примера автор выделил некоторые понятия; это сделано для того, чтобы уяснить их соотношение друг с другом.

Первое, что мы сделали в нашем приложении,– добавили в форму Form1 новую кнопку. Она представляет собой объект, называемый Button. Далее, щелкнув два раза мышью, мы добавили обработчик событий, который ожидает наступления события click (нажатие), генерируемого объектом Button. Этот обработчик событий добавляется в код объекта Form, где инкапсулировано приложение, в качестве частного метода:

```
private void button1_Click(object sender, System.EventArgs e) {…}
```

В качестве описателя используется ключевое слово private.

Первая строка кода из числа добавленных нами изменяет текст на нажимаемой кнопке. В этом случае используется полиморфизм, с которым мы уже сталкивались. Объект Button, представляющий нажатую нами кнопку, передается обработчику событий как параметр типа object, тип которого мы изменяем на тип Button (это оказывается возможным, поскольку объект Button наследуется от класса .NET System.Object, для которого object является синонимом). Далее мы изменяем свойство объекта Text, чтобы изменить выводящийся текст:

```
((Button)sender).Text = "Clicked!";
```

Затем мы создаем новый объект Button с помощью ключевого слова new (обратите внимание, что в этом примере пространства имен заданы так, что позволяют использовать простой синтаксис, в противном случае нам бы пришлось вводить полностью квалифицированное имя объекта – System.Windows.Forms.Button):

```
Button newButton = new Button();
newButton.Text = "NewButton!";
```

Затем в произвольное место программы добавляется обработчик событий, который будет использоваться для того, чтобы отреагировать на событие click, генерируемое новой кнопкой:

```
private void newButton_Click(object sender, System.EventArgs e) {
```

```
((Button)sender).Text = "Clicked!!"; }
```

Затем мы регистрируем данный обработчик событий как ожидающий наступления события Click с помощью некоторого синтаксиса перегрузки оператора. Одновременно мы создаем новый объект EventHandler, используя для этого конструктор не по умолчанию, с именем новой функции обработчика событий:

```
newButton.Click += new EventHandler(newButton_Click);
```

В заключение мы используем объект из семейства controls данной формы для того, чтобы добавить в форму новую кнопку с помощью метода Add():

```
Controls.Add(newButton);
```

Этот простой пример включает в себя почти все методы, рассмотренные в данной лекции. Как вы можете заметить, в ООП нет ничего сложного, просто оно требует несколько иного угла зрения.

5.4. Вопросы для повторения

1. Принципы ООП.
2. Понятие объекта.
3. Понятие свойств и полей.
4. Понятие методов.
5. Жизненный цикл объекта.
6. Конструкторы и деструкторы.
7. Статические члены класса.
8. Ссылочные и значимые типы данных.

6. ОПРЕДЕЛЕНИЕ КЛАССОВ

6.1. Определение классов в C#

В языке C# для описания классов используется ключевое слово class При этом необходимо задействовать следующую основную конструкцию

```
class myClass {// члены класса }
```

Этот код определяет класс с именем myClass. После того как мы определили класс, мы вольны создавать экземпляры этого класса в нашем проекте везде, где имеется доступ к этому определению. По умолчанию классы определяются как internal (внутренние), что означает, что доступ к ним будет иметь только код текущего проекта. Мы можем указать это явно, воспользовавшись ключевым словом определения доступа internal (хотя и не обязаны этого делать):

```
internal class MyClass { // члены класса }
```

При желании можно указать, что класс является общим и должен быть доступен из других проектов Для этого используется ключевое слово public.

```
public class MyClass { // члены класса }
```

Обратите внимание, что классы, которые объявляются самостоятельно, не могут быть частными или защищенными. Соответствующие модификаторы – private и protected – можно использовать только для описания классов, являющихся членами других классов. Их мы рассмотрим в следующей главе.

Кроме ключевых слов этих двух модификаторов доступа, для описания класса можно использовать ключевое слово abstract (абстрактный; создавать экземпляры такого класса запрещено, он может только наследоваться, в нем допускается наличие абстрактных членов) или sealed (изолированный, такой класс не может наследоваться). Эти ключевые слова являются взаимоисключающими Таким образом, абстрактный класс должен определяться следующим образом.

```
public abstract class MyClass {
    // члены класса, среди которых могут быть абстрактные }
```

В данном случае MyClass является общим абстрактным классом, хотя также возможны и внутренние абстрактные классы.

Изолированные классы определяются следующим образом

```
public sealed class MyClass {
    // члены класса }
```

Так же, как абстрактные классы, изолированные классы могут быть общими или внутренними.

В определении класса может также задаваться и наследование. Для этого следует после имени класса поместить двоеточие, за которым должно следовать имя базового класса. Например.

```
public class MyClass : MyBase {
    // члены класса }
```

Заметьте, что в определении класса в C# допускается наличие только одного базового класса, и если данный класс наследуется от некоторого абстрактного класса, то в нем должны быть реализованы все наследуемые абстрактные члены (если, конечно, этот класс сам не является абстрактным).

Компилятор не допустит ситуации, при которой производный класс будет более доступным, чем его базовый класс. Отсюда следует, что внутренний класс может наследоваться от общего базового класса, но общий класс не может наследоваться от внутреннего базового класса. Следующий код является допустимым:

```
public class MyBase {
    // члены класса }
internal class MyClass : MyBase {
    // члены класса }
```

А такой код не пройдет компиляцию:

```
interna1 class MyBase {
    // члены класса }
public class MyClass : MyBase {
    // члены класса
}
```

Если базовый класс не задан, то класс наследуется только от базового класса System.Object (для которого в C# используется синоним object). System.Object безусловно является корневым в иерархии наследования для всех классов. Мы будем изучать этот фундаментальный класс ниже.

Помимо указания базовых классов, мы можем таким же способом – через двоеточие – указать и поддерживаемые интерфейсы. Если одновременно требуется задавать базовый класс, то он должен располагаться первым после двоеточия, а интерфейсы – за ним. Если базовый класс не задается, то интерфейсы указываются непосредственно за двоеточием. Для отделения имени базового класса (если таковое присутствует) и для отделения одних имен интерфейсов от других следует использовать запятую.

Например, мы можем добавить в класс MyClass интерфейс:

```
public class MyClass : IMyInterface {
    // члены класса }
```

Все члены, являющиеся интерфейсами, должны иметь реализацию в любом классе, поддерживающем данный интерфейс, хотя, конечно, всегда существует возможность реализовать "пустой" интерфейс (без какого бы то ни было функционального кода), если от данного члена не требуется выполнения каких-либо действий.

Следующее объявление неверно, поскольку базовый класс MyBase не является первым вхождением списка наследования:

```
public class MyClass: IMyInterface, MyBase {
    // члены класса }
```

Правильный способ задать и базовый класс, и интерфейс следующий:

```
public class MyClass : MyBase, IMyInterface {
    // члены класса }
```

Не забывайте, что допускается наличие нескольких интерфейсов, поэтому следующий код также является допустимым:

```
public class MyClass: MyBase, IMyInterface, IMySecondInterface {
    // члены класса }
```

Ниже приводится таблица допустимых в определениях классов комбинаций модификаторов доступа.

Модификатор	Значение
Отсутствует либо internal	Класс доступен только в рамках текущего проекта
public	Класс доступен отовсюду
abstract или internal abstract	Класс доступен только в рамках текущего проекта, не допускает создания экземпляров, может только наследоваться
public abstract	Класс доступен отовсюду, не допускает создания экземпляров, может только наследоваться
sealed или internal sealed	Класс доступен только в рамках текущего проекта, не может наследоваться, допускает только создание экземпляров
public sealed	Класс доступен отовсюду, не может наследоваться, допускает только создание экземпляров

System.Object

Поскольку все классы наследуются от System.Object, то они обладают доступом ко всем его защищенным и общим членам. Поэтому имеет смысл ознакомиться с тем, что нам доступно System.Object содержит следующие методы:

Метод	Возвращаемый тип	Виртуальный	Статический	Описание
Object()	Отсутствует	Нет	Нет	Конструктор типа System.Object. Автоматически вызывается конструкторамипроизводных типов
-Object() (также известен под именем Finalize())	Отсутствует	Нет	Нет	Деструктор для типа System.Object Автоматически вызывается деструкторамипроизводных типов, сам по себе вызванбыть не может
Equals (object)	bool	Да	Нет	Сравнивает объект, для котороговызывается, с другим объектом и возвращает значение true, если они равны. Реализация, выполняющаяся по умолчанию, проверяет, ссылается ли переданный в качестве параметра объект на тот же самый объект (поскольку объекты представляют собой ссылочные типы). Если необходимо сравнивать объекты каким-либо иным образом, например, на предмет одинакового значения, этот метод может быть переопределен.
Equals (object, object)	bool	Нет	Да	Сравнивает два объекта, передаваемых ему в качестве параметров, на предмет того, равны ли они. Эта проверка выполняется. С помощью метода Equals (object). Заметьте, что, если оба объекта обладают нулевыми ссылками, возвращается значение true.
ReferenceEquals(object, object)	bool	Нет	Да	Сравнивает два переданных ему объекта, определяя, являются ли они ссылками на один и тот же экземпляр.
ToString()	string	Да	Нет	Возвращает строку, соответствующую экземпляру объекта. По умолчанию это квалифицированное имя класса (см предыдущий пример), однако метод можно переопределить для выполнения действий, подходящих для типа данного класса.
MemberwiseClone()	object	Нет	Нет	Копирует объект посредством создания нового экземпляра объекта и копирования всех членов. Обратите внимание, что такое копирование членов не приводит к созданию новых экземпляров этих членов. Любые члены ссылочного типа в новом объекте будут ссылаться на те же объекты, на которые они ссылаются в исходном классе. Рассматриваемый метод является защищенным, поэтому его можно использовать внутри класса или внутри производных классов.
GetType()	System.Type	Нет	Нет	Возвращает тип объекта в виде объекта System.Type.
GetHashCode()	int	Да	Нет	Используется как функция хеширования для объектов. Хеш-функция – это функция, возвращающая значение, которое позволяет идентифицировать объект в некоторой сжатой форме.

Эти методы являются основными, они должны поддерживаться всеми типами обектов в .NET Framework, хотя, возможно, некоторые из них вам никогда не придется использовать (или только в определенных обстоятельствах, как, например, GetHashCode()).

Метод GetType () полезен при использовании полиморфизма, поскольку он позволяет выполнять разные коды для объектов разных типов, а не один и тот же код для всех объектов, как это часто бывает. Например, если у нас имеется функция, которой передается параметр типа object (это означает, что мы можем передавать ей практически все, что угодно), то можно предусмотреть выполнение в ней специальных работ в случае поступления объектов конкретного типа. Воспользовавшись сочетанием Getrype() с typeof() (оператор C#, который преобразовывает имя класса в объект System.Type), мы получаем возможность выполнять сравнения примерно следующим образом:

```
if (typeof(myObj) = = typeof(MyComplexClass)) {
    // Объект myObj является экземпляром класса MyComplexClass }
```

Возвращаемый объект system.type обладает гораздо более широкими возможностями, но здесь мы не будем на нем останавливаться.

Очень часто оказывается весьма полезным переопределить метод ToString (), особенно в тех ситуациях, когда содержимое объекта может быть с легкостью представлено в виде одной удобочитаемой строки.

6.2. Конструкторы и деструкторы

Когда мы описываем какой-либо класс в C#, то в большинстве случаев нет необходимости описывать соответствующие ему конструкторы и деструкторы, поскольку объект базового класса System.Object обеспечивает их реализацию по умолчанию. Однако иногда стоит описать их самостоятельно, что позволит инициализировать и уничтожать объекты.

Для того чтобы добавить в класс простой конструктор, необходимо воспользоваться следующим несложным синтаксисом:

```
class MyClass {
public MyClass()
{
// Код конструктора
}}
```

Такой конструктор обладает тем же именем, что и класс, в котором он содержится, не имеет параметров (что превращает его в конструктор, использующийся по умолчанию) и является общим, что позволяет создавать экземпляры объектов данного класса (более подробная информация по этому вопросу содержится в предыдущей лекции).

Существует также возможность использовать частный конструктор по умолчанию, что означает, что экземпляры объектов данного класса не могут создаваться с помощью этого конструктора (еще раз отсылаем читателей к предыдущей лекции).

```
class MyClass {
        private MyClass()
        {
                // Код конструктора
        }
}
```

Кроме того, у нас имеется возможность аналогичным образом определить конструкторы данного класса, использующиеся не по умолчанию; для этого достаточно просто задать параметры. Например:

```
class MyClass
{
        public MyClass () {
                // Код конструктора, использующегося по умолчанию }
                public MyClass (int myInt) {
                // Код конструктора, использующегося не по
                // умолчанию (используется параметр myInt) } }
```

Количество задаваемых конструкторов не ограничено.

Деструкторы определяются с помощью несколько иного синтаксиса. Деструктор, используемый в .NET (и предоставляемый классом System.Object), называется Finalize(), однако для объявления деструктора используется другое имя. Вместо того чтобы переопределять Finalize(), мы используем следующий код:

```
class MyClass {
        ~MyClass()
        {
        // тело деструктора
        } }
```

Код, заключенный в деструкторе, будет выполняться при сборке мусора, позволяя освобождать удерживаемые ресурсы. После вызова деструктора происходят явные вызовы деструкторов базовых классов, включая вызов Finalize() в корневом классе System.Object. Такой способ позволяет .NET Framework гарантировать выполнение, поскольку переопределение Finaiize() означало бы, что необходимы явные вызовы базовых классов, а это таит в себе потенциальную опасность (мы познакомимся с вызовом методов базовых классов в следующей лекции).

Последовательность выполнения конструкторов

Если нам требуется выполнить несколько действий в конструкторах класса, то удобнее поместить весь этот код в одном месте, что дает те же преимущества, что и разбиение кода на функции. Для этого можно использовать отдельный метод (см следующую лекцию), однако в C# имеется замечательная альтернативная возможность. Каждый конструктор может быть настроен таким образом, что перед выполнением собственного кода он будет вызывать некоторый другой конструктор.

Однако прежде чем перейти к рассмотрению этого вопроса, давайте поближе познакомимся с тем, что по умолчанию происходит при создании нового экземпляра класса.

Для того чтобы создать экземпляр производного класса, необходимо создать экземпляр его базового класса В свою очередь, чтобы создать экземпляр этого базового класса, требуется создать экземпляр базового класса этого базового класса – и так далее до System.Object. В результате, какой бы конструктор ни использовался для создания класса, System.Object.Object() всегда будет вызываться первым.

Если используется конструктор класса не по умолчанию, то в таком случае по умолчанию будет использоваться конструктор базового класса, сигнатура которого совпадает с сигнатурой данного конструктора. Если таковой обнаружить не удается, то используется конструктор базового класса по умолчанию (это происходит всегда, за исключением корневого класса System.Object, поскольку у него отсутствуют конструкторы, использующиеся не по умолчанию) Давайте рассмотрим некоторый пример, который поможет проиллюстрировать последовательность событий. Рассмотрим следующую иерархию объектов:

```
public class MyBaseClass
 {
     public MyBaseClass ()
     {
     }
     public MyBaseClass (int i) { }
 }
public class MyDerivedClass : MyBaseClass
{
     public MyDerivedClass()
     {
```

```
    }
    public MyDerivedClass(int i)
    {
    }
    public MyDerivedClass(int i, int j) { }
}
```

Если мы попытаемся создать экземпляр класса MyDerivedClass следующим образом:

```
MyDerivedClass myObj = new MyDerivedClass () ;
```

то это приведет к такой последовательности событий:

- выполнится конструктор System.Object.Object ();
- выполнится конструктор MyBaseClass.MyBaseClass();
- выполнится конструктор MyDerivedClass.MyDerivedClass ().

Если же мы попытаемся создать экземпляр класса таким образом:

```
MyDerivedClass myObj = new MyDerivedClass(4);
```

то соответствующая последовательность будет иметь следующий вид:

- выполнится конструктор System.Object.Object ();
- выполнится конструктор MyBaseClass.MyBaseClass(int i);
- выполнится конструктор MyDerivedClass.MyDerivedClass (int i).

Наконец, если мы воспользуемся следующим вариантом:

```
MyDerivedClass myObj = new MyDerivedClass(4, 8);
```

то произойдет вот что:

- выполнится конструктор System. Object. Object();
- выполнится конструктор MyBaseClass.MyBaseClass();
- выполнится конструктор MyDerivedClass.MyDerivedClass(int i, int j).

При таком подходе мы получаем возможность поместить код, ответственный за обработку параметра int i, в MyBaseClass(int i), подразумевая, что конструктору MyDerivedClass(int i, int j) достанется меньше работы – ему придется обрабатывать только параметр int j (подобные рассуждения строятся на основе предположения, что параметр int i в обоих случаях имеет один и тот же смысл, что может и не выполняться, хотя на практике при таком способе оформления обычно выполняется). C# при необходимости позволяет нам задать именно такой тип поведения.

Для этого требуется просто указать конструктор базового класса в определении конструктора нашего производного класса следующим образом:

```
public class MyDerivedClass : MyBaseClass
{
    public MyDerivedClass (int i, int j) : base(i)
    { }
}
```

Ключевое слово base указывает NET, что в процессе создания экземпляра следует использовать конструктор базового класса с сигнатурой, совпадающей с заданной. В данном случае мы задействуем единственный параметр int, поэтому будет вызван MyBaseCiass(int i) конструктор MyBaseClass () не будет вызываться, т. е. последовательность событий будет такой же, как и в последнем примере,– что, собственно, в данном случае и требовалось.

Существует возможность с помощью этого же ключевого слова задавать литеральные значения для конструкторов базового класса, допустим, применяя конструктор класса MyDerivedClass, использующийся по умолчанию, для вызова конструктора класса MyBaseClass, использующегося не по умолчанию:

```
public class MyDerivedClass : MyBaseClass {
     public MyDerivedClass() : base(5)
     { }
     }
```

В этом случае последовательность событий будет иметь такой вид

- выполнится конструктор System Object.Object();
- выполнится конструктор MyBaseClass MyBaseClass(int i);
- выполнится конструктор MyDerivedClass MyDerivedClass().

Кроме ключевого слова base, в этом контексте может использоваться еще одно ключевое слово this. Оно указывает процессу создания экземпляра в NET на необходимость использовать конструктор не по умолчанию для текущего класса, прежде чем будет вызван указанный конструктор. Например:

```
public class MyDerivedClass : MyBaseClass
{
    public MyDerivedClass() : this (5, 6)
    { }
    public MyDerivedClass(int i, int j) : based) { }
}
```

Это приведет к такой последовательности событий:

- выполнится конструктор System Object.Object();
- выполнится конструктор MyBaseClass.MyBaseClass(int i);
- выполнится конструктор MyDerivedClass.MyDerivedClass (int i, int j)
- выполнится конструктор MyDerivedClass.MyDerivedClass ().

Единственным ограничением в данном случае является задание только одного конструктора, использующего ключевые слова base или this. На самом деле, как было продемонстрировано в предыдущем примере, это не такое уж серьезное ограничение, поскольку все равно остается возможность конструирования чрезвычайно изощренных последовательностей выполняемых действий.

Мы увидим этот способ в действии немного позже

6.3. Типы структур

В предыдущей главе мы обращали внимание на то, что структуры и классы очень похожи, хотя структуры представляют собой значимые типы, а классы – ссылочные типы. Что же это может означать на самом деле? Самый простой способ разобраться в этом – рассмотреть некоторый пример.

ПРАКТИКУМ: сравнение классов и структур

1. Создайте новое консольное приложение.
2. Измените код следующим образом:

```
namespace C
{
class MyClass
    {
    public int val;
    }
struct myStruct {
    public int val; }
/// <summary>
/// Summary description for Classl.
/// </summary>
class Classl
{
    static void Main(string[] args) {
        MyClass objectA = new MyClass ();MyClass objectB = objectA;
        objectA. val = 10;
        objectB.val = 20;
        myStruct structA = new myStruct();
        myStruct structB = structA;
        structA.val = 30;
        structB.val = 40;
        Console.WriteLine("objectA.val = {0}", objectA.val);
        Console.WriteLine("objects.val = {0}", objectB.val);
        Console.WriteLine("structA.val = {0}", structA.val);
        Console.WriteLine("structB.val = {0}", structB.val);
        Console.ReadLine(); }
}
}
```

3. Запустите приложение:

Как это работает

В этом приложении содержатся два определения типов, одно из которых описывает структуру с именем myStruct, обладающую единственным общим полем типа int с именем val, а второе описывает класс с именем MyClass, который содержит аналогичное поле (мы будем рассматривать члены классов, такие как поля, в следующей **главе**, а пока вам достаточно знать, что синтаксис в данном случае используется точно такой же). Далее мы выполняем одинаковые операции над экземплярами обоих типов:

- объявляем переменную данного типа;
- создаем новый экземпляр переменной данного типа;
- объявляем вторую переменную данного типа;
- присваиваем первую переменную второй переменной;
- присваиваем значение полю val первого экземпляра переменной;
- присваиваем значение полю val второго экземпляра переменной;
- выводим значение поля val обеих переменных.

Хотя мы выполняем над переменными разных типов одни и те же операции, окончательные результаты оказываются различными. При выводе значений поля val обнаруживается, что обе переменные типа object обладают одинаковым значением, в то время как переменные типа структуры имеют различающиеся значения.

Итак, что же произошло?

Объекты – это ссылочные типы. Когда мы присваиваем объект некоторой переменной, мы на самом деле присваиваем этой переменной указатель, который ссылается на этот объект. С точки зрения программы указатель – это некий адрес в памяти. В данном случае это адрес, по которому располагается объект. Когда мы присваиваем ссылку на первый объект второй переменной типа Myciass, то мы, по сути, копируем этот адрес. Это означает, что обе переменных содержат ссылки на один и тот же объект.

Структуры – это значимые типы. В переменных не хранится никаких ссылок на структуру, вместо этого в них хранится сама структура. Поэтому, когда одна структура присваивается другой переменной типа mystruct, происходит копирование всей информации из одной структуры в другую. Это поведение идентично тому, которое мы наблюдали ранее для переменных простых типов, таких как тип int. Поэтому в качестве конечного результата мы получаем две переменные, содержащие разные структуры.

Весь механизм использования указателей скрыт от нас в управляемом коде, что позволяет этому коду быть существенно проще. На самом деле в C# существует возможность получить доступ к операциям более низкого уровня, таким как манипуляции над указателями, воспользовавшись "небезопасным" кодом, однако эту сложную тему мы здесь рассматривать не будем.

6.4. Неглубокое и глубокое копирование

Копирование объектов из одной переменной в другую по значению, а не по ссылке (т. е. копирование тем же способом, который используется при копировании структур) иногда оказывается делом весьма сложным. Поскольку конкретный объект может содержать большое количество ссылок на другие объекты – например, в виде своих полей, этот процесс может потребовать выполнения очень большого количества действий. Простое копирование каждого члена одного объекта в другой может не сработать, потому что некоторые из этих членов сами по себе могут иметь ссылочные типы.

В .NET Framework все это учитывается. Почленное копирование простых объектов достигается за счет использования метода MemberwiseClone(), который наследуется от System.Object. Данный метод является защищенным, однако не составляет никакого труда определить общий метод на объекте, который его использует. Копирование, которое обеспечивается этим методом, известно под названием неглубокого копирования – в том смысле, что при его осуществлении не принимаются во внимание те члены, которые представляют собой ссылочные типы. Это означает, что ссылочные члены во вновь создаваемом объекте будут ссылаться на те же самые объекты, на которые ссылаются соответствующие члены в исходном объекте: во многих случаях это не совсем идеальный вариант. Если нам требуется создать новые экземпляры рассматриваемых полей и скопировать туда значения вместо ссылок, то мы должны выполнить так называемое глубокое копирование.

Существует интерфейс, который мы можем реализовать и который позволяет выполнить эту процедуру стандартным образом: ICloneable. Если мы собираемся его применить, нам необходимо реализовать единственный метод,

содержащийся в нем,– Clone(). Этот метод возвращает значение типа System.Object. Мы можем использовать совершенно произвольные способы обработки для получения этого объекта, реализуя тело этого метода так, как нам требуется. Другими словами, мы можем реализовывать процедуру глубокого копирования в зависимости от нашего желания (например, мы при необходимости можем реализовать и процедуру неглубокого копирования).

6.5. Вопросы для повторения

1. Определение классов в C#.
2. Модификаторы доступа к членам класса.
3. Базовый класс System.Object.
4. Последовательность выполнения конструкторов.
5. Неглубокое и глубокое копирование.

7. ЧЛЕНЫ КЛАССОВ И ИХ СВОЙСТВА

В данной главе мы продолжим обсуждение способов определения классов в C#. Теперь мы переходим к рассмотрению того, что необходимо для определения членов классов, а именно полей, свойств и методов.

Мы начнем с того, что познакомимся с кодом, требующимся для определения каждого из этих членов, а также рассмотрим создание необходимой структуры кода с помощью соответствующего мастера (wizard) VS. Мы также узнаем, каким образом можно быстро изменять члены, редактируя их свойства.

После того как вы познакомитесь с основами определения членов, мы перейдем к рассмотрению более сложных способов их использования: перевода членов базового класса в скрытое состояние, обращения к переопределенным членам базового класса и вложенных определений типов.

В заключение мы применим эту теорию на практике и создадим библиотеку классов, которую можно будет постепенно достраивать и использовать в последующих главах.

7.1. Члены классов

7.1.1. Определение членов

В рамках определения класса должны находиться определения всех членов данного класса, включая поля, методы и свойства. У каждого члена должен быть свой уровень доступа, который во всех случаях описывается одним из следующих ключевых слов:

- public (общий) – член, доступный из любого кода;
- private (частный) – член, доступный только из того кода, который является составной частью данного класса (значение по умолчанию, которое используется в случае отсутствия ключевого слова);
- internal (внутренний) – член, доступный только из кода, находящегося внутри проекта (модуля), в котором он определен;
- protected (защищенный) – член, доступный только из кода, являющегося составной частью данного класса или класса, производного от него.

Последние два ключевых слова могут использоваться совместно, т е. член можно описать как protected internal. Такие члены доступны только из производных классов, описанных в рамках проекта (или, точнее, модуля; модули будут рассматриваться в главе 21).

Поля, методы и свойства могут быть описаны также посредством ключевого слова static (статические): в этом случае они будут статическими членами данного класса, а не экземпляров объектов, что подробно обсуждалось в главе 8.

7.1.2. Определение полей

Поля определяются с помощью обычного формата объявления переменной (с дополнительной возможностью инициализации) и с использованием модификаторов, обсуждавшихся выше. Например:

```
class MyClass { public int MyInt; }
```

Для имен общих полей в .NET Framework применяется система регистров PascalCasing, а не camelCasing, и в данной книге будет использоваться такой же подход. Именно поэтому поле, описанное выше, названо MyInt, а не myInt. Это всего лишь рекомендуемая схема использования регистров, но она кажется автору наиболее осмысленной.

Для частных полей не существует никаких рекомендаций, такие поля обычно именуются с использованием системы camelCasing.

Для полей также может использоваться ключевое слово readonly (только чтение), что означает, что значение этого поля может быть задано только в процессе выполнения конструктора или при начальном присваивании. Например:

```
class MyClass { public readonly int MyInt = 17; }
```

Как было отмечено во введении к этой главе, поля могут объявляться статическими с помощью ключевого слова static. Например:

```
class MyClass { public static int MyInt; }
```

Доступ к статическим полям может осуществляться через класс, в котором они описаны (для вышеприведенного примера –MyClass.MyInt), но не через экземпляры объектов данного класса.

Кроме всего этого, существует возможность использования ключевого слова const (константа). Члены, описанные как const, являются статическими по определению, поэтому в данном случае модификатор static не требуется (более того, его использование является ошибочным).

7.1.3. Определение методов

Для описания методов используется стандартный формат функций и необязательный модификатор static. Например:

```
class MyClass {
public string GetString()
    { return "Это строка."; } }
```

При именовании методов в .NET Framework, как и при именовании полей, используется система PascalCasing, а не camelCasing.

Заметьте, что если при описании метода использовано ключевое слово static, то этот метод будет доступен только через класс, но не через экземпляр объекта. При определении метода также можно использовать следующие ключевые слова:

- virtual (виртуальный) – метод может быть переопределен;

– abstract (абстрактный) – метод должен быть переопределен (допускается только в абстрактных классах);
– override (переопределить) – метод переопределяет метод базового класса (это ключевое слово должно использоваться в том случае, если метод переопределяется);
– extern (внешний) – определение метода находится где-либо в другом месте.

Следующий пример демонстрирует переопределение метода.

```
public class MyBaseClass {
public virtual void DoSomething()
{ // Базовая реализация } }
public class MyDenvedClass : MyBaseClass {
public override void DoSomething() {
// Реализация метода в производном классе,
// она переопределяет базовую реализацию } }
```

При использовании модификатора override может применяться также модификатор sealed, который указывает на то, что никаких дальнейших модификаций этого метода в производных классах не допускается, т. е метод не может переопределяться в производных классах. Например:

```
public class MyDerivedClass : MyBaseClass {
 public override sealed void DoSomething() {
// Реализация метода в производном классе,
// она переопределяет базовую реализацию } }
```

Использование модификатора extern позволяет использовать реализацию метода, внешнюю по отношению к данному проекту. Это сложная тема, и мы здесь не будем в нее углубляться.

7.1.4. Определение свойств

Свойства определяются аналогично полям, но с ними все оказывается не так просто. Как обсуждалось ранее, свойства сложнее полей, поскольку они допускают дополнительную обработку перед изменением своего состояния. Это достигается за счет того, что они обладают двумя блоками, напоминающими функции, один из которых предназначается для получения значения свойства, а второй – для задания значения свойства.

Оба этих блока, определяемых посредством ключевых слов get (получи) и set (установи), могут использоваться для управления уровнем доступа к данному свойству. Существует возможность опустить тот или иной блок для создания свойства, которое будет использоваться в режиме "только чтение" или "только запись" (причем отсутствие блока get определяет "только запись", а отсутствие блока set определяет режим доступа "только чтение"). Это все, естественно, применимо только ко внешнему коду, поскольку любой код, находящийся внутри класса, будет иметь доступ ко всем тем данным, к которым имеет доступ код, находящийся в этих блоках. Для того чтобы получить корректное свойство, необходимо включить в его описание хотя бы один из этих двух блоков (свойство, которое нельзя ни считать, ни изменить, едва ли может принести хоть какую-нибудь пользу).

Основная структура свойства включает стандартное ключевое слово – модификатор доступа (public, private и т. д.), за которым следует имя типа, имя свойства и один или оба блока set и get, в которых содержится код обработки свойства. Например:

```
public int MyIntProp {
get
{ // Код для получения значения свойства }
set
{ // Код для задания значения свойства } }
```

Имена общих свойств в .NET Framework также строятся на основе регистров PascalCasing, а не camelCasing, и в этой книге мы будем использовать для них – так же, как для полей и методов – именно эту систему.

Первая строка этого определения очень напоминает определение поля. Отличие заключается в том, что в конце строки отсутствует точка с запятой, вместо нее там размещаются вложенные блоки set и get.

Блоки get должны обладать возвращаемым значением, имеющим тот же тип, что и само свойство. Простые свойства очень часто привязывают к единственному частному полю, которое управляет доступом к этому свойству; в этом случае блок get может возвращать значение этого поля напрямую. Например:

```
// Поле, используемое свойством private int myInt;
// Свойство
public int MyIntProp
{
get { return myInt; }
set { // Код для задания свойства }
}
```

Заметьте, что внешний код не будет иметь непосредственного доступа к полю MyInt из-за его уровня доступа (оно описано как частное). Напротив, такой код должен использовать свойство для того, чтобы получить доступ к полю.

Функция set присваивает полю значение аналогичным образом. В этом случае можно воспользоваться ключевым словом value, чтобы сослаться на значение, полученное от пользователя.

```
// Поле, используемое свойством private int myInt;
// Свойство
public int MyIntProp
{
get { return myInt; }
set { myInt = value; } }
```

Значение value имеет такой же тип, что и свойство, поэтому, если используется тот же тип, что и тип поля, не приходится беспокоиться о приведении типов.

Это простое свойство не только защищает поле myint от непосредственного доступа. Истинная мощь свойств проявляется, когда требуется несколько больший контроль над происходящим. Например, мы можем реализовать блок set следующим образом:

```
set {
if (value >= 0 && value <= 10)
myInt = value; }
```

В данном случае мы изменяем myInt, только если значение свойства находится между 0 и 10. В подобных ситуациях необходимо решить: что предпринять, если использовано недопустимое значение? Здесь имеются четыре возможности:

– ничего не предпринимать (как в коде, приведенном выше);
– присвоить полю значение по умолчанию;
– продолжать как ни в чем ни бывало, но зарегистрировать событие для последующего анализа;

– сгенерировать исключительную ситуацию.

Вообще говоря, предпочтительными являются две последние возможности, а выбор между ними зависит от того, каким образом будет использоваться данный класс и в какой степени следует передать контроль пользователям класса. Генерирование исключительной ситуации дает пользователям очень большие возможности контроля, позволяет получать информацию о том, что происходит, и соответственно реагировать. Для этого можно воспользоваться стандартной исключительной ситуацией System, например

```
set {
if (value >= 0 && value <= 10) myInt = value;
else
throw (new ArgumentOutOfRangeException("MyIntProp", value, "Значение свойства MyIntProp должно лежать в диапазоне между 0 и 10.")); }
```

Эту ситуацию можно обработать, применив конструкцию try. . .catch. . .finally в коде, который использует данное свойство, как это было описано в главе 7.

Регистрация данных, возможно, в текстовом файле, может оказаться полезной для производственных программ, в которых никаких проблем возникать не должно. Эти данные позволяют разработчикам проверять работоспособность программ и, при необходимости, исправлять обнаруженные ошибки.

Свойства, так же как и методы, могут использовать ключевые слова virtual, override и abstract, т. е. те ключевые слова, использование которых для полей оказывается недопустимым.

ПРАКТИКА: использование полей, методов и свойств

1. Создайте новый проект консольного приложения.
2. С помощью VS добавьте новый класс с именем MyClass в файл MyClass.cs.
3. Измените код в MyClass.cs следующим образом:

```
public class MyClass {
public readonly string Name;
private int intVal;
public intVal {
get {
return intVal; }
set {
if (value >= 0 && value <= 10)
intVal = value; else
throw (new ArgumentOutOfRangeException("Val" , value, "Val must be assigned a value between 0 and 10. ")) ; // Значение, присваиваемое Val, должно лежать в диапазоне между 0 и 10. / '
} }
public override string ToStrmg() {
return "Name: " + Name + "\nVal: " + Val; }
private MyClass () : this("Default Name") { }
public MyClass(string newName)
Name = newName; intVal = 0; } }
```

4. Измените код в Class1.cs следующим образом:

```
static void Main(string[] args) {
Console.WriteLine("Creating object myObj...");
MyClass myObj = new MyClass("My Object") ;
Console.WriteLine("myObj created.");
for (int i = -1; i <= 0; i++) {
try {
Console.WriteLine ("\nAttempting to assign {0} to myObj .Val. . . " , i) ;
myObj.Val = i;
Console.WriteLine("Value {0} assigned to myObj.Val.", myObj.Val); }
catch (Exception e) {
Console.WriteLine("Exception {0} thrown.", e.GetTvpe().FuilName);
Console.WriteLine("Message:\n\"{0}\"", e.Message); ) )
Console.WriteLine("\nOutputting myObj .ToString() ...");
Console.WriteLine(myObj.ToString());
Console.WriteLine("myObj .ToString() Output.") ; }
```

5. Запустите приложение (см. рис. слева).

Как это работает

Код в Main () создает и использует экземпляр класса MyClass, который определен в MyClass.cs. Создание экземпляра данного класса должно осуществляться с помощью конструктора не по умолчанию, поскольку конструктор по умолчанию класса MyClass является частным:

```
private MyClass () : this("Default Name") { }
```

Обратите внимание, что используется this ('Default Name') –для гарантированного присвоения Name какого-либо значения при обращении к конструктору. Последнее возможно, если данный класс используется для создания нового класса. Отсутствие значения у поля Name в дальнейшем может привести к возникновению ошибок.

Используемый не по умолчанию конструктор присваивает значения описанному как readonly полю паше (это присваивание можно осуществить либо при объявлении поля, либо с помощью конструктора) и частному полю intVal.

Далее Main() пытается выполнить два присваивания свойству val объекта myObj (он является экземпляром класса MyClass). Для присваивания значений -1 и 0 используются два прохода цикла for, а для обнаружения возможных исключительных ситуаций применяется конструкция try...catch. Когда свойству присваивается -1, возникает исключительная ситуация System.ArgumentOutOfRangeException и код, находящийся в блоке catch, выводит информацию о ней в окно консоли. При следующем проходе цикла свойству Val присваивается значение 0, а затем через это свойство значение присваивается частному полю intVal.

В заключение для вывода отформатированной строки, представляющей содержимое объекта, используется переопределенный метод ToString ():

```
public override string ToString() {
return "Name: " + Name + "\nVal:" + Val; }
```

Этот метод должен быть объявлен с использованием ключевого слова override, поскольку названный метод переопределяет виртуальный метод ToString() базового класса System.Object. Код в данном случае использует непосредственно свойство val, а не частное поле intVal. Нет никаких причин, по которым мы не могли бы использовать свойства внутри класса подобным образом, хотя в этом случае может возникать небольшое замедление работы программы (настолько не-

большое, что мы вряд ли сумеем его обнаружить). Использование свойства, кроме того, позволяет осуществлять контроль за допустимостью значений, присущий использованию свойств, что также может оказаться полезным для кода, находящегося внутри класса.

7.2. Свойства членов

Последняя из основных тем, которую необходимо рассмотреть,– изменение свойств членов с помощью окна Properties. После того как мы выбрали член в окне Class View, мы получаем возможность просматривать его свойства в окне Properties (см рис слева).

Из этого окна мы можем непосредственно изменять многие свойства, например, уровень доступности, посредством свойства Access (доступ) (оно выделено на рисунке) В этом случае код будет модифицирован автоматически – и не придется ничего набирать на клавиатуре.

Обратите внимание, что свойство CanOverride (может переопределяться) позволяет определить, является ли член виртуальным, а IsShared (с разделением доступа) – является ли он статическим.

7.2.1. Дополнительные действия с членами класса

Теперь, когда вы познакомились с основами определения членов, самое время перейти к рассмотрению более сложных тем. Здесь будут рассмотрены следующие из них:

- сокрытие методов базового класса;
- вызов переопределенных или скрытых методов базового класса;
- вложенные определения типов;
- сокрытие методов базового класса.

При наследовании некоторого (не абстрактного) члена базового класса наследуется также и его реализация. Если наследуемый член является виртуальным, то имеется возможность переопределить его реализацию, воспользовавшись ключевым словом override. Независимо от того, является ли наследуемый член виртуальным, при необходимости можно скрыть его реализацию. Это полезно, например, если наследуемый общий член работает не так, как требуется.

Сокрытие достигается использованием следующего кода:

```
public class MyBaseClass {
public void DoSomething ()
{
// Базовая реализация
} }
public class MyDerivedClass : MyBaseClass {
public void DoSomething ()
{
// Реализация в производном классе, скрывающая базовую реализацию
} }
```

Хотя такой код будет работать нормально, он сгенерирует предупреждение о том, что в нем скрывается член базового класса. Это дает возможность исправить положение дел, если мы случайно скрыли член, который на самом деле желаем использовать. Если же этот член действительно требуется скрыть, то об этом можно сказать явно, воспользовавшись ключевым словом new:

```
public class MyDerivedClass : MyBaseClass {
new public void DoSomething()
{ // Реализация в производном классе, скрывающая базовую реализацию
} }
```

Этот код будет работать точно так же, только без выдачи предупреждения.

На данном этапе нам представляется важным отметить, в чем заключается разница между скрытыми и переопределенными членами базового класса. Рассмотрим следующий код:

```
public class MyBaseClass {
public virtual void DoSomething()
{
Console.WriteLine("Базовая реализация") ;
} }
public class MyDerivedClass : MyBaseClass {
public override void DoSomething()
{
Console.WriteLine("Производная реализация") ;
} }
```

В данном случае переопределенный метод заменяет собой реализацию базового класса, так что в следующем коде будет использоваться новая версия, несмотря на то, что обращение к методу происходит через тип базового класса (с использованием полиморфизма)

```
MyDerivedClass myObj = new MyDenvedClass ();
MyBaseClass myBaseObj;
myBaseObj = myObj;
myBaseObj.DoSomething();
```

В данном случае выходной поток будет следующим:

Производная реализация

В качестве альтернативы мы можем скрыть метод базового класса с помощью следующего кода:

```
public class MyBaseClass {
public virtual void DoSomething()
{
        Console.WriteLine("Базовая реализация") ;
} }
        public class MyDerivedClass : MyBaseClass {
new public void DoSomething ()
{
        Console.WriteLine("Производная реализация") ;
} }
```

Метод базового класса при этом совершенно не обязательно должен быть виртуальным, однако от этого ничего не изменится, и приведенный здесь код отличается от предыдущего варианта только одной строкой Результат выполнения этого кода как при виртуальном методе, так и в противном случае будет следующим:

Базовая реализация

Несмотря на то, что базовая реализация является скрытой, к ней, как и раньше, имеется доступ через базовый класс.

Вызов переопределенных или скрытых методов базового класса.

Независимо от того, переопределяем ли мы некоторый член или делаем его скрытым, по-прежнему остается возможность доступа к члену базового класса внутри данного класса. Существует множество различных ситуаций, когда это оказывается полезным, например:

Если требуется скрыть наследуемый общий член от пользователей производного класса, но необходимо иметь доступ к его функциональным возможностям внутри класса.

Если требуется что-либо добавить к реализации наследуемого виртуального члена, а не просто заменить ее на новую переопределенную реализацию.

Для достижения подобного эффекта можно воспользоваться ключевым словом base, которое указывает на реализацию базового класса, содержащуюся внутри производного класса (аналогично тому, как оно используется в управляющих конструкторах, с которыми вы познакомились в предыдущей главе). Например:

```
public class MyBaseClass
{
public virtual void DoSomethmg ()
{
        // Базовая реализация
} }
public class MyDerivedClass MyBaseClass {
public override void DoSomething () {
        // Реализация в производном классе, расширяющая реализацию в базовом классе
        base.DoSomething(),
        // Дополнительная реализация в производном классе
} }
```

В представленном коде выполняется версия DoSomethmg (), содержащаяся в классе MyBaseClass (базовом по отношению к MyDerivedClass), из которой вызывается версия DoSomethmg(), содержащаяся внутри класса MyDerivedClass.

Поскольку ключевое слово base работает с экземплярами объектов, то будет ошибкой использовать его внутри статического члена.

Ключевое слово this

Кроме ключевого слова base, в предыдущей главе мы также использовали ключевое слово this. Так же, как и base, this может применяться внутри членов класса, и так же, как и base, оно относится к экземпляру объекта. Экземпляр объекта, на который указывает this, является текущим экземпляром объекта (отсюда следует, что это ключевое слово нельзя использовать в статических членах, поскольку они не являются составной частью экземпляра объекта).

Наиболее ценной функцией ключевого слова this является возможность передавать некоторому методу ссылку на текущий экземпляр объекта, например:

```
public void doSomethmg () {
MyTargetClass inyObj = new MyTargetClass ();
myObj.doSomethmgWith(this); }
```

В данном примере класс MyTargetClass, экземпляр которого создается, обладает методом DoSomethmgWith(). Ему передается единственный параметр, имеющий тип, совместимый с тем классом, в котором содержится этот метод. Тип параметра может совпадать с типом данного класса, с типом класса, наследником которого является данный класс, может являться интерфейсом, реализованном в данном классе, или же (естественно) быть типом System.Object.

Вложенные определения типов

Помимо описания типов как классов в пространствах имен их можно описывать внутри других классов. В таком случае появляется возможность использовать в определениях весь спектр модификаторов доступности, а не только public и internal, а также применять ключевое слово new для перевода определения типа, наследуемого от базового класса, в скрытое состояние.

В следующем коде, кроме класса MyClass, описывается также вложенный класс С именем myNestedClass:

```
public class MyClass {
public class myNestedClass
{
public int nestedClassField,
} }
```

Если потребуется создать экземпляр класса myNestedClass откуда-нибудь извне класса MyClass, то его имя должно квалифицироваться

```
MyClass.myNestedClass myObj = new MyClass.myNestedClass();
```

Однако вполне возможно, что мы окажемся не в состоянии выполнить такие действия, если вложенный класс объявлен как частный или обладает каким-либо другим уровнем доступности, несовместимым с кодом в той точке, в которой осуществляется попытка создания соответствующего экземпляра

Основной причиной этого является необходимость описания классов, которые являются частными по отношению к содержащему их классу, так что никакой другой код в данном пространстве имен не будет иметь к ним доступа

7.3. Вопросы для повторения

1. Определение членов класса.
2. Определение полей, методов и свойств.
3. Назначение ключевого слова this.
4. Вложенные определения типов.

8. РАБОТА С ФАЙЛАМИ

8.1. Потоки

Любой ввод и вывод информации в .NET Framework включает в себя использование потоков – абстрактных представлений последовательного устройства. Последовательное устройство – это нечто, обеспечивающее как последовательное хранение данных, так и последовательный доступ к ним – по одному байту в каждый конкретный момент времени. В качестве такого устройства могут выступать расположенный на диске файл, принтер, область памяти, а также любой другой объект, который допускает последовательное считывание и запись информации. Рассматривая такое устройство как абстрактное, мы можем скрыть назначение/ источник потока. Такой уровень абстракции делает возможным многократное использование кода и позволяет создавать подпрограммы более общего назначения. Таким образом, аналогичный код может передаваться и повторно использоваться, когда приложение считывает данные из файлового входного потока и когда оно считывает информацию из сетевого входного потока. Кроме того, используя понятие потока, мы получаем возможность абстрагироваться от физических особенностей конкретного устройства. Поэтому при считывании информации из файлового потока нам не приходится волноваться по поводу работы головок дискового устройства или заниматься вопросами распределения памяти.

Выходной поток используется, когда данные записываются на некое внешнее устройство. Это может быть файл на физическом диске, сетевой сервер, принтер или какая-либо другая программа. Понимание программирования потоков открывает множество дополнительных возможностей. В этой главе мы ограничим обсуждение рамками записи в файлы на диске.

Входной поток используется для считывания данных в память или в переменные, к которым наша программа имеет доступ. Наиболее распространенной формой входного потока, с которой нам приходилось работать, является клавиатура. Входной поток может поступать практически от любого устройства, но мы сосредоточимся на чтении дисковых файлов. Понятия, применимые для чтения/записи дисковых файлов, оказываются применимыми практически для всех остальных устройств, что позволяет получить общее представление о потоках и познакомиться с фундаментальным подходом, который может быть использован в самых различных ситуациях.

8.1.1. Классы для ввода и вывода

Пространство имен System.IO содержит в себе все классы, которые будут рассматриваться в этой главе. В пространстве имен System.IO хранятся классы, предназначенные для считывания данных из файлов и записи данных в файлы; для того чтобы получить в программе доступ к этим классам, необходимо сослаться на это пространство имен. Как можно увидеть на следующей диаграмме, в пространстве имен содержится не так уж много классов, но мы будем рассматривать только классы, являющиеся основными при вводе и выводе файлов:

File – широко используемый на практике класс, который обладает большим количеством статических методов, позволяющих переносить, копировать и удалять файлы.

Directory – класс, который обладает большим количеством статических методов, позволяющих переносить, копировать и удалять директории.

Path – класс, позволяющий выполнять манипуляции над именами путей.

FileInfo – представляет физический файл, расположенный на диске, и обладает методами, позволяющими выполнять манипуляции над этим файлом. Для любых операций чтения/записи, выполняемых над файлом, необходимо создавать объект Stream.

DirectoryInfo – представляет физическую директорию, расположенную на диске, и обладает методами, позволяющими выполнять манипуляции над этой директорией.

FileStream – представляет файл, который допускает либо считывание, либо запись, либо и то, и другое одновременно. Этот файл может считываться и записываться как асинхронно, так и синхронно.

StreamReader – считывает символьную информацию из потока и может создаваться на базе класса Fiiestream.

StreamWriter – записывает символьную информацию в поток и может создаваться на базе класса Fiiestream.

FileSystemWatcher – это один из наиболее сложных классов, который используется для слежения за состоянием файлов и директорий и генерирует события в моменты, когда изменяется их местоположение, что позволяет перехватывать их внутри приложения. Такая функциональная возможность отсутствовала при программировании в среде Windows, но теперь, с появлением .NET Framework, реагировать на события, происходящие в файловой системе, становится намного проще.

8.1.2. Классы File и Directory

Оба класса предоставляют большое количество методов для осуществления манипуляций над самой файловой системой, а также над файлами и директориями, которые хранятся в этой файловой системе. Эти методы являются статическими и позволяют перемещать файлы, запрашивать и вносить изменения в атрибуты, также создавать объекты FileStream. Как мы узнали в главе 8, статические методы могут вызываться классами без создания их экземпляров.

Некоторые из наиболее полезных статических методов класса File приведены в таблице 9.1.

В таблице 9.2 представлены некоторые полезные статические методы класса Directory:

Таблица 9.1 – Основные методы класса File

Метод	Описание
Copy ()	Копирует файл в указанное место.
Create ()	Создает файл в соответствии с заданным путем.
Delete()	Уничтожает файл.
Open ()	Возвращает объект FileStream по заданному пути.
Move ()	Перемещает файл в новое место. При этом имеется возможность задать для нового файла другое имя.

8.1.3. Класс FileInfo

В отличие от класса File, класс FileInfo не обладает статическими методами и может использоваться только посредством создания экземпляров объектов. Объект FileInfo представляет файл, расположенный на диске или по некоторому сетевому адресу. Заметьте, что этот класс не является потоком. Для того чтобы осуществить запись в файл или чтение из файла, необходимо создать объект Stream. Объект FileInfo помогает решить эту задачу, предоставляя несколько методов, которые возвращают созданные экземпляры объектов Stream. Однако, прежде всего, для того чтобы создать объект FileInfo, необходимо указать путь к файлу или директории:

```
FileInfo aFile = new FileInfo(@"C:\Год.txt");
```

Многие методы, предоставляемые классом FileInfo, аналогичны соответствующим методам класса File, однако, поскольку класс File является статическим классом, он требует задания строкового параметра с местоположением файла при каждом вызове метода. Отсюда следует, что следующие вызовы выполняют одни и те же действия:

```
FileInfo aFile = new FileInfo("Data.txt");
if (aFile.Exists)
Console.WriteLine("File Exists"); / Файл существует /
if (File.Exists("Data.txt"))
Console.WriteLine("File Exists"); / Файл существует /
```

Большинство методов объекта FileInfo представляют в этом смысле зеркальное отражение методов объекта File. Имеет прямой смысл использовать статический класс File, когда требуется осуществить единственный вызов метода на объект. В этом случае вызов будет выполнен быстрее, поскольку .NET Framework не придется проходить через процедуру создания экземпляра нового объекта с последующим вызовом метода. Однако если приложение осуществляет несколько операций над файлом, то более разумным представляется создать экземпляр объекта FileInfo и использовать его методы. Это позволит сэкономить определенное время, поскольку объект будет заранее настроен на нужный файл в файловой системе, в го время как статическому классу придется каждый раз осуществлять его поиск заново.

Класс FileInfo обладает перечисленными в таблице 9.3 свойствами, относящимися к данному файлу, осуществление манипуляций над которыми позволяет вносить изменения в файл.

8.1.4. Класс DirectoryInfo

Таблица 9.2 – Основные методы класса Directory

Метод	Описание
CreateDirectory()	Создает директорию в соответствии с заданным путем.
Delete()	Уничтожает указанную директорию вместе со всеми находящимися в ней файлами.
GetDirectories()	Возвращает массив объектов Directory, в котором представлены все директории, находящиеся в данной директории.
GetFiles()	Возвращает массив объектов File для данной директории.
Move()	Переносит указанную директорию на новое место, где этой папке может быть присвоено новое имя.

Таблица 9.3 – Свойства класса FileInfo

Свойство	Описание
Attributes	Позволяет получать или задавать значения атрибутов текущего файла.
CreationTime	Позволяет получать дату и время создания текущего файла.
DirectoryName	Возвращает путь к директории, в которой находится файл.
Exists	Определяет, существует ли данный файл.
FullName	Позволяет получать полный путь к данному файлу.
Length	Позволяет получать размер файла.
Name	Возвращает только имя самого файла – без полного пути, ведущего к его местоположению.

Класс DirectoryInfo работает в точности так же, как и класс FileInfo. Он требует создания экземпляра объекта, который представляет собой одну из директорий, имеющихся в компьютере. Как и для класса FileInfo, многие методы дублируются в классах Directory и DirectoryInfo. Аналогичны и указания относительно того, когда какой из них следует использовать: при единственном вызове следует использовать статический класс Directory. При осуществлении последовательности вызовов нужно использовать экземпляр объекта DirectoryInfo.

8.1.5. Имена пути и относительные пути

При задании имени пути в программе для .NET имеется возможность использовать либо абсолютное, либо относительное имя пути. Абсолютное имя пути явным образом указывает на расположение файла или директории в каком-либо известном месте, например, на диске C:. Вот пример такого пути: C:\work\LogFile.txt. Обратите внимание на то, что в данном случае местоположение описывается точно, без какой-либо двусмысленности.

Относительные пути указывают на местоположение относительно того места в файловой системе, в котором выполняется приложение. При использовании относительного пути не требуется указывать ни имени диска, ни какого-либо известного места; в качестве отправной точки используется текущая директория.

Например, если приложение выполняется. В директории C:\Development\FileDemo, а в качестве относительного пути используется "LogFile.txt", то это будет файл C:\Development\FileDemo\LogFile.txt. Для перехода. К директории более высокого уровня используется две точки (..). Следовательно, для того же самого приложения путь ". .\Log.txt" будет указывать на файл с именем Log.txt, расположенный в директории C:\Development.

Наиболее коварным при использовании относительных путей в процессе разработки является именно их относительность, зависящая от того, где именно выполняется данное приложение. Если разработка ведется с использованием Visual Studio.NET, то это означает, что приложение находится на несколько уровней ниже создаваемой вами папки проекта. Обычно приложения располагаются в директории ProjectName\bin\Debug.

8.1.6. Объект FileStream

Объект FileStream представляет поток, указывающий на какой-либо файл на диске или на местоположение в сети. Хотя этот класс и обладает методами для чтения из файлов и записи в файлы байтов, чаще для выполнения этих функций вам придется обращаться к классам StreamReader и StreamWriter. Это происходит потому, что класс FileStream оперирует байтами и массивами байтов, в то время как классы stream оперируют символьными данными. Иметь дело с символьными данными оказывается проще, однако нам придется столкнуться с некоторыми операциями, например, с произвольным доступом к файлам, которые могут осуществляться только посредством объекта FileStream, к изучению которого мы приступим несколько позже.

Существует несколько способов создания объекта FileStream. Его конструктор обладает большим количеством перегрузок/версий, однако в самом простом случае он требует передачи двух аргументов – имени файла и значения перечислимого типа FileMode:

```
FileStream aFile = new FileStream("Log.txt", FileMode.OpenOrCreate);
```

У перечислимого типа FileMode имеется несколько членов, которые позволяют задавать то, каким образом файл открывается или создается. Допускается их использование в различных сочетаниях.

Элементы перечислимого типа FileMode	Описание
Append	Открывает файл (если он существует), переводит указатель файла в его конец или создает новый файл. FileMode.Append может использоваться только совместно с членом перечислимого типа FileAccess.Write.
Create	Создает новый файл; если файл с таким именем уже существует, то он уничтожается.
CreateNew	Создает новый файл; если файл с таким именем уже существует, то будет сгенерирована исключительная ситуация.
Open	Открывает существующий файл. Если указанного файла не существует, генерируется исключительная ситуация.
OpenOrCreate	Определяет, что если такой файл существует, то он должен быть открыт, в противном случае создается новый файл. Если такой файл существует, вся информация в нем сохраняется.
Truncate	Открывает существующий файл, а вся информация, которая в нем хранилась, уничтожается. После этого мы получаем возможность записать в файл абсолютно новую информацию, однако при этом дата создания файла остается прежней. Этот файл должен существовать, иначе будет сгенерирована исключительная ситуация.

Приведенный выше пример конструктора по умолчанию открывает файл в режиме чтение/запись. Для того чтобы задать другой уровень доступа, следует использовать дополнительный параметр – FileAccess:

```
FileStream aFile = new FileStream("Log.txt", FileMode.OpenOrCreate, FileAccess.Write) ;
```

Эта строка кода откроет файл с разрешением доступа на запись. Любая попытка прочитать содержимое этого файла приведет к возникновению исключительной ситуации. Перечислимый тип FileAccess обладает всего тремя Типами; Read, Write и ReadWrite. Следовательно, есть возможность открыть файл только на чтение, только на запись и на чтение и запись одновременно. Это свойство используется в качестве одного из способов варьирования уровня доступа пользователя к файлу в зависимости от уровня его прав.

Оба класса File и FileInfo предоставляют методы OpenRead() и OpenWrite(), которые облегчают создание объектов FileStream. Первый открывает метод с режимом доступа "только чтение", а второй, кроме того, разрешает осуществлять запись. Это позволяет упростить обращение к файлу, так что вам теперь не придется указывать всю информацию, как в предыдущих примерах. Например, следующая строка открывает файл Data.txt в режиме "только чтение":

```
FileStream aFile = File.OpenRead("Data.txt");
```

Обратите внимание на то, что следующий код выполняет те же самые действия:

```
FileInfo aFile = new FileInfo("Data.txt") ;
FileStream aFile = aFile OpenRead() ;
```

8.1.7. Позиция внутри файла

Класс FileStream поддерживает внутренний указатель файла, ссылающийся на то место в файле, в котором будет производиться очередная операция чтения или записи. В большинстве случаев при открытии файла указатель устанавливается в начало файла, однако такое поведение указателя может быть изменено, что позволяет приложению осуществлять чтение или запись в любой точке файла. Можно также организовать произвольный доступ к файлу и осуществлять поиск конкретной позиции в файле. Такой подход позволяет сэкономить массу времени при работе с очень большими файлами, поскольку можно постоянно отыскивать необходимую позицию.

Метод, который реализует данную функциональную возможность, носит название Seek ():

```
public long Seek(long offset, SeekOrigin origin);
```

Первый параметр определяет, насколько далеко вперед в байтах должен быть передвинут указатель. Второй параметр определяет, с какой точки вести отсчет. Перечислимый тип SeekOrigin состоит из трех значений: Begin (начало), current (текущая позиция) и End (конец). Следующая строка передвигает указатель файла на восьмой байт, считая с самого первого байта данного файла:

```
aFile.Seek(8, SeekOrigin.Begin);
```

Следующая строка кода приведет к перемещению указателя файла на два байта вперед относительно его текущей позиции. Если эта строка будет выполнена непосредственно сразу после предыдущей строки, то указатель будет ссылаться на десятый байт файла:

```
aFile.Seek(2, SeekOrigin.Current);
```

Следует обратить внимание на то, что как при чтении из файла, так и при записи в файл происходит изменение позиции указателя файла. В случае считывания, например, десяти байтов, указатель файла будет ссылаться на байт, расположенный после десятого считанного байта.

Существует также возможность указывать отрицательные значения для определения требуемой позиции, что в сочетании с использованием значения перечислимого типа SeekOrigin.End может быть использовано для поиска необходимой позиции вблизи конца файла. Следующая строка позволит перейти к пятому с конца байту файла:

```
aFile.Seek(-5, SeekOrigin.End);
```

Файлы, доступ к которым может осуществляться подобным образом, зачастую называют файлами с произвольным доступом, поскольку у приложения имеется возможность получить доступ к любой позиции в файле. Классы Stream позволяют осуществлять последовательный доступ к файлам, и в них не предусмотрена возможность работы с указателем.

8.1.8. Чтение данных

Чтение данных с использованием класса FileStream оказывается не таким простым, как с использованием классов StreamReader и StreamWriter, к рассмотрению которых мы вернемся ниже в данной главе. Это происходит потому, что класс FileStream имеет дело исключительно с байтами, что делает класс полезным при работе с файлами произвольного

типа, а не только с текстовыми файлами. Побайтное считывание данных позволяет использовать этот класс при работе с графическими и звуковыми файлами. Однако платой за подобную гибкость является невозможность с помощью класса FileStream считывать данные непосредственно в переменные типа строки. Это позволяют осуществлять классы StreamWriter и StreamReader. Существует, однако, несколько конверсионных классов, которые существенно облегчают выполнение преобразования массива байтов в символьные массивы и обратно.

Метод FileStream.Read() является основным средством, позволяющим осуществлять доступ к данным, содержащимся в файле, на который ссылается объект FiieStream:

```
public int Read(byte[] array, int offset, int count);
```

Первый параметр представляет собой массив байтов, передаваемый для размещения в нем данных, получаемых от объекта FileStream. Второй параметр определяет ту позицию в массиве байтов, начиная с которой должна осуществляться операция записи данных. Обычно этот параметр имеет значение, равное нулю, в результате чего данные из файла заносятся в самое начало массива. Последний параметр определяет, какое количество байтов должно быть считано из файла.

Следующий пример демонстрирует считывание данных из файла с произвольным доступом. В качестве файла, из которого будет производиться чтение, используется файл, специально созданный для данного примера.

ПРАКТИКУМ. Чтение данных из файла с произвольным доступом

1. Откройте Visual Studio.NET.
2. Откройте новый проект, выбрав File | New j Project. Выберите пункт C# Console Application и назовите его ReadFile.
3. Добавьте два следующих объявления пространств имен в самое начало файла Class1.cs. Пространство имен System.IO необходимо для класса FiieStream, а пространство Имен System.Text потребуется для осуществления преобразования, которое нам придется выполнять над байтами, считанными из файла.

```
using System;
using System.IO;
using System.Text;
```

4. Добавьте следующий код в метод Main():

```
static void Main(string[] args)
{
byte[] byData = new byte[100]; char[] charData = new Char[100]; try {
    FileStream aFile = new FileStream("../../Classl.cs",FileMode.Open);
    aFile. Seek (55, SeekOngin.Begin) ;
    aFile.Read(byData,0,100); }
catch(IOException e) {
    Console.WriteLine("An IO exception has been thrown!");
    Console.WriteLine{e.ToString ()) ;
    Console.ReadLine();
    return; }
Decoder d = Encoding. UTF8.GetDecoder();
d.GetChars(byData, 0, byData.Length, charData, 0);
Console.WriteLine(charData);
Console.ReadLine();
return;
}
```

5. Запустите приложение. Нажмите клавишу Enter для закрытия консоли

Как это работает

Чтение данных из файла с произвольным доступом

Данное приложение открывает свой собственный файл с расширением .cs, из которого затем осуществляется чтение. Для этого оно осуществляет поиск по файловой структуре с использованием двух точек в следующей строке:

```
FileStream aFile = new FileStream(" ./. ./Classl.cs", FileMode.Open) ;
```

Две строки кода, которые в действительности осуществляют поиск по файлу и считывание из него, имеют следующий вид:

```
try {
    aFile.Seek(55,SeekOrigin.Begin);
    aFile. Read (byData, 0,100); }
catch(IOException e) {
    Console.WriteLine("An 10 exception has been thrown!");
    Console.WriteLine(e.ToString());
    Console.ReadLine();
    return; }
```

В результате выполнения первой из них указатель файла перемещается на 55-й байт файла. Это символ 'n' из слова namespace; 54 предыдущих символа находятся в строках с директивами using. В результате выполнения второй строки следующие 100 байтов считываются в байтовый массив byData.

Обратите внимание на то, что нами используется конструкция для обработки исключительных ситуаций try-catch. Практически любая операция, связанная с вводом/выводом, может привести к возникновению исключительной ситуации типа IOException. Любой исполнимый код должен включать в себя обработку исключительных ситуаций, особенно если в нем ведется работа с файловой системой.

После того как информация считана из файла в байтовый массив, возникает необходимость преобразовать ее в символьный массив с тем, чтобы получить возможность выводить ее на консоль. С этой целью воспользуемся классом Decoder из пространства имен System.Text. Этот класс предназначен для того, чтобы преобразовывать исходные байты в какие-либо более полезные объекты, например, в символы:

```
Decoder d = Encoding.UTF8.GetDecoder();
d.GetChars(byData, 0, byData.Length, charData, 0) ;
```

В этих строках осуществляется создание объекта Decoder, который основывается на кодировке UTF8. Это схема кодирования Unicode. После этого вызывается метод GetChars (), который принимает массив байтов и преобразовывает его в массив символов. После завершения этой процедуры массив символов оказывается готовым к выводу на консоль.

8.1.9. Запись данных

Процесс записи данных в файл с произвольным доступом совершенно аналогичен. Сначала необходимо создать байтовый массив; наиболее простой способ достигнуть этого – создать сначала символьный массив, который мы собираемся записывать в файл. Затем следует воспользоваться объектом Encoder, позволяющим преобразовать его в массив байтов; использование этого объекта очень напоминает использование объекта Decoder. В завершение необходимо вызывать метод Write (), который и запишет информацию в файл.

Давайте создадим простой пример, который позволит продемонстрировать, каким образом это выполняется.

ПРАКТИКУМ. Запись данных в файл с произвольным доступом

1. Создайте новый проект. Выберите пункт меню C# Console Application и назовите его WriteFile.
2. Как и в предыдущем случае, вставьте в начало файла Class1.cs следующие два описания пространств имен:

```
using System;
using System.IO;
using System.Text;
```

3. Добавьте в метод Main() следующий код:

```
static void Main(string[] args) {
byte[] byData = new byte[100];
char[] charData = new Char [100] ;
try {
    FileStream aFile = new FileStream("Temp. txt" . FileMode.OpenOrCreate) ;
    charData = "Hello World".ToCharArray();
    Encoder e = Encoding.UTF8.GetEncoder();
    e.GetBytes (charData, 0,charData.Length, byData, 0 , true);
    // Перемещение указателя файла в самое начало файла
    aFile.Seek(0,SeekOrigin.Begin);
    aFile.Write(byData,0,byData.Length); }
catch(IOException ex) {
    Console.WriteLine("An 10 exception has been thrown!") ;
    Console.WriteLine(ex.ToString());
    Console.ReadLine();
    return; }
return; }
```

4. Запустите приложение. Оно будет выполняться некоторое время, а затем завершится.
5. Перейдите в директорию приложения – файл будет сохранен именно в ней, поскольку мы используем относительный путь. Файл будет помещен в папку WriteFile\bin\Debug. Откройте файл Temp.txt.

Как это работает

Запись данных в файл с произвольным доступом

Данное приложение открывает файл в своей собственной директории и записывает в него несложную строку. Все это происходит аналогично предшествующему примеру.

Следующая строка создает символьный массив с помощью статического метода ToCharArray () класса String. Поскольку в C# все можно рассматривать в качестве объекта, а текст "Hello world" на самом деле представляет собой объект string, то эти методы могут вызываться даже для обычной строки символов:

```
charData = "Hello World" .ToCharArray() ;
```

Из следующих строк становится понятным, каким образом можно преобразовать массив символов в соответствующий массив байтов, который требуется объекту FileStream.

```
Encoder e = Encoding. UTF8.GetEncoder();
e.GetBytes(charData,0,charData.Length, byDate, 0,true;
```

На этот раз мы создаем объект Encoder, который также основывается на кодировке UTF8. В данном случае для выполнения перекодировки также используется кодировка Unicode, однако на этот раз нам необходимо перекодировать символьные данные в корректный байтовый формат, прежде чем у нас появится возможность записать информацию в поток. Это совершается в методе GetBytes (). В нем происходит преобразование символьного массива в байтовый. Этому методу в качестве первого параметра передается символьный массив, а в качестве второго – индекс элемента, с которого следует начинать преобразование. В качестве третьего параметра этому методу передается количество символов, подлежащих преобразованию; в качестве четвертого – байтовый массив, в котором должна размещаться информация после преобразования, и в качестве пятого – индекс элемента, с которого начинается заполнение байтового массива. Последний параметр определяет, должен ли объект Encoder обновлять свое состояние после окончания работы. Это имеет отношение к тому факту, что объект Encoder сохраняет в памяти то место, на котором была прекращена запись в байтовый массив Это оказывается весьма полезным при осуществлении последовательных обращений к объекту Encoder, но оказывается совершенно бессмысленным в случае единственного обращения к нему. При завершающем обращении к объекту Encoder этому параметру должно быть присвоено значение true, означающее, что память должна быть очищена, а объект освобожден для процедуры сборки мусора.

После выполнения всех этих действий запись байтового массива в объект FileStream посредством метода Write() не представляет никаких трудностей:

```
aFile.Seek(0,SeekOrigin.Begin);
aFile.WritetbyData,0,byData.Length);
```

8.2. Объект StreamWriter

Работа с объектом FileStream достаточна сложна, поэтому может возникнуть вопрос, нет ли какого-либо более простого способа. Обычно вся работа с объектом FileStream оказывается скрытой от пользователя за счет работы с объектами StreamWriter и StreamReader и использования их методов для выполнения манипуляций над файлами. Если перед вами не стоит задача перемещать указатель файла на произвольную позицию, то здесь эти классы позволяют значительно упростить работу с файлами. Класс StreamWriter позволяет осуществлять запись в файл символов и строк, он самостоятельно выполняет все необходимые преобразования и производит запись в объект FileStream.

Существуют разнообразные способы создания экземпляров класса StreamWriter. Если уже имеется объект FileStream, то можно воспользоваться им для создания объекта StreamWriter:

```
FileStream aFile = new FileStream("Log. txt" ,FileMode.CreateNew) ;
StreamWriter sw = new StreamWriter(aFile);
```

Объект StreamWriter может также создаваться непосредственно из файла:

```
StreamWriter sw = new StreamWriter("Log.txt",true);
```

Этот конструктор принимает в качестве параметров имя файла и логическое значение, которое указывает на то, следует ли добавлять информацию к уже существующему файлу или необходимо создать новый. Если это значение равно false, то либо происходит создание нового файла, либо из существующего файла сначала удаляется все хранящаяся в нем информация, а затем он открывается на запись. Если же это значение равно true, то файл просто открывается, а все хранящаяся в нем информация сохраняется. Если такого файла не существует, то создается новый файл.

Обратите внимание на то, что здесь нет такого широкого спектра возможностей, как при создании объекта FileStream. Кроме логического значения, определяющего, следует ли добавлять информацию к уже существующему файлу или создавать новый файл, нег никаких других опций, например, отсутствует возможность задания свойства FileMode, которое мы уже задавали при работе с классом FileStream. Точно так же отсутствует возможность задания свойства FileAccess, поэтому вы всегда будете обладать доступом к файлу на чтение/запись. Для того чтобы воспользоваться какой-либо из этих возможностей, следует сначала обратиться к конструктору класса FileStream, а затем создать объект StreamWriter на основе объекта FileStream.

ПРАКТИКУМ. Выходной поток

1. Создайте новый проект. Выберите пункт меню C# Console Application и назовите его StreamWrite.

2. В данном случае опять придется воспользоваться пространством имен System.IO, поэтому добавьте соответствующую строку в начало файла Classl.cs.

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

3. Добавьте следующий код в метод Main():

```
static void Main(string[] args) {
try
{
FileStream aFile = new FileStream("Log.txt",FileMode.OpenOrCreate) ;
StreamWriter sw = new StreamWriter(aFile) ;
// Запись данных в файл
sw.WriteLinet("Hello World");
sw.Write("This is a") ;
sw.Write("string of characters.");
sw.Closed ; }
catch(IOException e) {
Console. WriteLine("An IO exception has been thrown!");
Console.WriteLine(e.ToString());
Console.ReadLine() ;
return; }
return; }
```

4. Постройте проект и запустите его. Если в нем не будет обнаружено никаких ошибок, то он очень быстро выполнится и закроется. Поскольку мы не предусмотрели вывода на консоль, то наблюдать за работой этой программы оказывается не очень интересно.

5. Перейдите в директорию приложения и найдите там файл Log.txt. Он должен располагаться в папке StreamWriter\bin\Debug, поскольку используется относительный путь.

Как это работает – Выходной поток

Данное простое приложение демонстрирует работу двух наиболее важных методов класса StreamWriter – Write () и WriteLine(). У каждого из них имеется большое количество перегружаемых версий, позволяющих использовать различные сложные возможности при выводе в файл, однако в настоящем примере мы воспользовались основной возможностью вывода строки.

Метод WriteLine () осуществляет запись переданной строки, за которой немедленно следует символ новой строки. Как мы можем видеть в настоящем примере, это приводит к тому, что следующая операция записи выполняется с новой строки. Метод Write() просто записывает передаваемую ему строку в файл, не добавляя символа перехода к новой строке. Это позволяет осуществлять запись целого предложения или абзаца с помощью нескольких обращений к методу Write ().

8.3. Форматирование данных

Точно так же, как существует возможность вывода данных в отформатированном виде на консоль, так и в файл мы можем записывать отформатированные данные. Для этого применяются те же методы – Write () и WriteLine(), но в этом случае им могут передаваться параметры, определяющие характер форматирования. Например:

```
sw.WriteLine("The Date is {0}", System.DateTime.Now.ToShortDateString());
sw.Write("{0}, {1}, {2}", "Kay", "Dan", "Rob");
```

В данном примере вы можете увидеть, каким образом осуществляется запись файлов, разделенных запятыми. В более сложных примерах данные могут поступать из объекта DataSet и других источников.

8.4. Объект StreamReader

Входные потоки используются для считывания данных, хранящихся во внешних источниках. В большинстве случаев таким источником является дисковый файл или сетевой адрес. Однако не следует забывать, что в качестве такого источника может использоваться любой объект, отправляющий данные, например, сетевое приложение, web-служба и даже консоль.

Для чтения данных мы будем использовать класс StreamReader. Так же, как и класс StreamWriter, класс StreamReader является общим классом, который может использоваться для любого потока. Мы снова будем создавать его на базе объекта FileStream, поэтому он будет ссылаться на нужный нам файл.

Объекты StreamReader создаются способом, напоминающим способ создания объектов StreamWriter. Наиболее распространенный способ создания таких объектов заключается в использовании заранее созданного объекта FileStream:

```
FileStream aFile = new FileStream("Log. txt".FileMode.Open);
StreamReader sr = new StreamReader(aFile);
```

Так же, как и класс StreamWriter, StreamReader может создаваться непосредственно на основе строки, в которой содержится путь к конкретному файлу:

```
StreamReader sr = new StreamReader("Log.txt") ;
```

ПРАКТИКУМ. Выходной поток

1. Создайте новый проект. Выберите пункт меню C# Console Application и назовите его StreamRead.

2. В данном случае нам опять потребуется импортировать пространство имен System.IO, поэтому добавьте соответствующую строку в начало файла Class1.cs.

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

3. Добавьте следующий код в метод Main ():

```
static void Main (string[] args) {
string strLine,
try
{
FileStream aFiie = new FileStream("bog. txt",FileMode.Open);
StreanReader sr = new StreamReader(aFile);
strLine = sr ReadLine () ;
//Построчное считывание данных
while(strLine !=null)
{
Console.WriteLine(strLine) ;
strLine = sr. ReadLine ();
}
sr.Close() ;
Console.ReadLine(); }
catch(IOException e) {
Console. WriteLme ("An IO exception has been thrown!");
Console.WriteLine(e.ToString());
Console.ReadLine();
return; }
return; }
```

4. Создайте в директории StreamReader\bm\Debug файл Log. txt.

Это может быть совершенно новый файл или скопированный регистрационный файл, созданный в предыдущем примере. Убедитесь, что файл находится в той же директории, что и приложение.

В противном случае приложение завершится неудачей, поскольку конструктор класса FileStream сгенерирует исключительную ситуацию, если ему не удастся обнаружить требуемый файл.

5. Запустите приложение – текст, хранящийся в файле, будет выведен на экран.

Как это работает – Входной поток

Данное приложение очень похоже на предыдущее. Единственное очевидное отличие заключается в том, что в нем происходит считывание информации из файла, а не запись. В этом случае также необходимо импортировать пространство имен System.IO, для того чтобы осуществлять доступ к необходимым классам.

Для чтения текста из файла использован метод ReadLine(). Этот метод осуществляет считывание текста до того момента, когда будет обнаружен символ "возврат каретки", и возвращает считанный текст в виде строки. В случае, если будет достигнут конец файла, то данный метод возвращает значение null, что используется при выполнении проверки достижения конца файла:

```
strLine = sr.ReadLine ();
while(strLine != null) {
    Console.WriteLine(strLine) ;
    strLine = sr.ReadLine () ; }
```

Здесь используется метод Console.ReadLine (), чтобы консоль не завершила свою работу, до того как мы успели прочитать выведенную на дисплей итоговую информацию.

8.4.1. Чтение данных

Метод ReadLine() не является единственным способом доступа к данным в файле. Класс StreamReader имеет несколько способов для чтения данных.

Простейший из них – Read(). Он возвращает следующий символ в виде положительного целого числа или значения -1, если достигнут конец. С помощью переводного класса Convert это значение может быть преобразовано в символ. В приведенном выше примере основная часть программы может быть переписана следующим образом.

```
int nChar; nChar = sr. Read () ;
while(nChar != -1) {
Console.Write(Convert.ToChar(nChar));
nChar = sr.Read(); }
```

Удобным методом для чтения небольших файлов является метод ReadToEnd(), осуществляющий считывание всего файла целиком и возвращающий его в виде строки. В этом случае приведенное выше приложение упрощается следующим образом:

```
strLine = sr.ReadToEnd();
Console.WriteLine(strLine);
```

Хотя такой подход может показаться весьма простым и удобным, при использовании его необходимо соблюдать известную осторожность. Производя считывание всех данных в строковый объект, вы форсируете перевод данных, хранящихся в файле, в память компьютера. В зависимости от размера файла с данными, это может оказаться невыполнимым. Если файл данных достаточно велик, то надежнее будет оставить данные в файле и осуществлять доступ к ним с помощью методов объекта StreamReader.

8.4.2. Файлы с ограничителем

Распространенным способом хранения данных в файле является разделение данных запятыми. Мы уже сталкивались с использованием класса StreamWriter для записи файлов с использованием такого подхода. Не менее просто оказывается читать файлы, использующие запятые в качестве разделителя. В класс String включен метод с названием Split(), который используется для преобразования строки в массив с использованием заданного символа в качестве разделителя. Если мы зададим режим использования запятой в качестве разделителя, то это приведет к созданию строкового массива требуемой размерности, в котором будет содержаться вся информация из исходной строки.

ПРАКТИКУМ. Файлы с использованием запятой в качестве разделителя

1. Создайте новый проект. Выберите пункт меню C# Console Application и назовите его CommaValues.

2. Добавьте соответствующую строку в начало файла Class1.cs:

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

3. Добавьте следующий код в метод Main():

```
static void Main(string [] args) {
string strLine;
string[] strArray;
```

```
char [ ] charArray = new Char[] {','};
try {
    FileStream aFile = new FileStream("Log.txt",FileMode.Open);
    StreamReader sr = new StreamReader(aFile) ;
    strLine = sr.ReadLme(); while(strLine !=null) {
    // Преобразование строк данных в массив строк
    strArray = strLine.Split(charArray);
    // Проход в цикле по массиву строк и вывод их на печать
    for(int x=0;x<=strArray.GetUpperBound(0);x++)
    {
    Console. WriteLine (strArray [x] .Trim()) ; }
    strLine = sr. ReadLine (); }
    sr.Close() ;
    Console.ReadLine () ; }
catch(IOException e) {
    Console.WriteLine("An IO exception has been thrown!");
    Console.WriteLine(e.ToString ());
    Console.ReadLine();
    return; }
return; }
```

4. Откройте Notepad и введите в него следующий текст:

Amanda, Kay, Sandy, Dan, Rob Natalie, Kate, David, Zoe, Andrew

5. Сохраните файл под именем log.txt в директории CommaValues\bin\Debug или директории, где будет запускаться выполняемый файл вашего приложения.

6. Запустите приложение – вы увидите выведенный на консоль текст файла.

Как это работает – Файлы с использованием запятой в качестве разделителя

Как и в предыдущем примере, данное приложение осуществляет построчное считывание данных в объект String. Однако, поскольку заранее известно, что в этом файле в качестве разделителей текстовых значений применяются запятые, то нами используется другой подход для его обработки. В следующей строке программы происходит создание строкового массива для каждой строки файла:

```
strArray = strLine.Split(charArray);
```

Данный метод принимает строковый массив, созданный ранее. Обратите внимание, что в настоящем примере в массиве содержится единственный символ – запятая. Мы могли бы записать в массив несколько символов; в таком случае каждый раз, когда в файле будет встречен любой из указанных символов, он будет рассматриваться в качестве допустимого ограничителя поля. Например, в следующем коде создается символьный массив, в который помещается сразу несколько символов. Теперь, в случае если метод Split будет вызван для строки "Please split?this|string up", то в строковом массиве будут возвращены все слова по отдельности, поскольку все символы, расположенные между словами, будут рассматриваться методом Split в качестве разделителей (пробел, вопросительный знак и вертикальная черта):

```
char[] charArray = new Char[] {'?',' ','|'};
strArray = "Please split?this | String up".Split(charArray);
```

После получения строкового массива он просматривается в цикле, а его содержимое выводится на консоль. Это достигается за счет использования обычного цикла for. Обратите внимание на то, что мы вызываем метод Trim() класса String. Этот метод удаляет все пустое пространство (пробелы и символы табуляции) в начале и в конце строк. Мы делаем это, поскольку нас не интересуют пробелы в начале и конце данных – нас интересуют только сами имена. Такой подход не всегда подойдет для ваших приложений, поскольку пустое пространство может оказаться значимым:

```
for(int x=0;x<=strArray.GetUpperBound(0);x++) {
Console.WriteLine(strArray[x].Trim())); }
```

Как легко заметить, извлечение данных из файлов, в которых в качестве разделителя переменных используется запятая (comma-separated variable – CSV), в .NET Framework не представляет никакого труда. Хотя в данном случае приведенный пример оказывается очень простым, вы столкнетесь с тем, что эти принципы используются в бизнес-приложениях уровня целого предприятия. Несмотря на то, что XML является намного превосходящим по всем параметрам методом хранения и передачи данных, использование CSV-файлов по-прежнему очень распространено и будет распространено еще в течение некоторого времени. Преимуществом файлов, в которых используются ограничители, заключается еще и в том, что они являются очень компактными и, следовательно, занимают меньше места, чем их XML-аналоги

Файлы с ограничителями используются многими системами, и если вашему приложению придется взаимодействовать с такой системой, то вы обнаружите, что формат данных с использованием ограничителей применяется достаточно часто. Обратите внимание на то, что, хотя мы и уделяли основное внимание файлам, в которых в качестве разделителя используется запятая, в этом качестве вполне может использоваться и символ табуляции – такой формат также является весьма распространенным. Следует также помнить, что CSV-файлы могут экспортироваться и импортироваться такими системами, как Excel, Access и многими реляционными базами данных.

8.5. Вопросы для повторения

1. Понятие потоков.
2. Кляссы ввода и вывода.
3. Классы File, FileInfo.
4. Классы Directory, DirectoryInfo.
5. Класс FileStream.
6. Объекты StreamReader, StreamWriter.
7. Форматирование данных.

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Ввести **N** чисел. Напечатать из них **M** самых больших.
2. Напечатать строку матрицы с максимальной суммой элементов.
3. Отсортировать строки матрицы в порядке убывания их максимального элемента.
4. Заданы координаты **N** - точек плоскости (X_i, Y_i), i=1,N. Найти точки расстояние между которыми максимально, и точки расстояние между которыми минимально.
5. Найти длину замкнутой ломаной линии, заданной координатами точек излома (X_i, Y_i), i=1,N. Определить пересекает она сама себя или нет.
6. В матрице найти наименьший из положительных элементов.
7. Отсортировать строки матрицы в порядке убывания суммы их элементов.
8. Напечатать фрагмент матрицы (строку и столбец), содержащие ее максимальный элемент.
9. Напечатать фрагмент матрицы в противоположных углах которого стоят ее максимальный и минимальный элементы.
10. Заполнить и вывести на печать треугольник Паскаля заданного размера.
11. Вектор B(N) заполнить случайными числами от -9 до 9. Напечатать его фрагмент расположенный между максимальным и минимальным элементом и вычислить сумму квадратов элементов этого фрагмента.
12. Заполнить матрицу случайными числами от 0 до 9. Сформировать два вектора. Первый содержит элементы побочной диагонали и выше. Второй - элементы ниже побочной диагонали. Отсортировать первый по убыванию, второй по возрастанию элементов.
13. Матрица содержит лишь нули и единицы. При этом единицы расположены группами, образующими прямоугольники. Найти количество этих прямоугольников.
14. В матрице найти прямоугольник максимальной площади в углах которого стоят одинаковые числа.
15. Заполнить и напечатать массив первых N простых чисел.
16. Отнормировать вектор X. $(X(i)=(X(i) - X_{min})/ (X_{max} - X_{min})$. В матрице поменять местами ее максимальный и минимальный элементы.

СПИСОК ЛИТЕРАТУРЫ

1. .Net. Сетевое программрование для проффесионалов \ Эндрю Кровчик, Винод Кумар, Номан Лагари и др. – М.: WROX, 2006. – 416с.
2. C# для профессионалов \ Нейгел Кристиан, Ивьен Билл, Глинн Ждей. – М.: Издательский дом "Вильямс", 2006. – 1376 с.
3. C# для профессионалов \ Симон Робинсон, Олди Корнес, Джей Глинн и др.-М.: WROX, 2005. – 512 с.
4. C# для профессионалов \ Симон Робинсон, Олди Корнес, Джей Глинн и др. – М.: WROX, 2005.- 544 с.
5. Visual Studio .NET: разработка приложений баз данных. \ А.В. Постолит. – СПб.: БХВ-Петербург, 2003. – 544 с.
6. Байдачный С.С. .Net Framework 2.0. Секреты создания Windows-приложений \ С.С. Байдачный. – М.: СОЛОН-Пресс, 2006. – 519 с.
7. Лабор В. В. Си Шарп: Создание приложений для Windows \ В.В. Лабор. – Мн.: Харвест, 2003. – 384с.
8. Основы Microsoft Visual Studio .NET 2003 \ Microsoft Corporation. - М.: "Русская редакция", 2003. – 464 с.
9. Петцольд Ч. Программирование в тональности C# \ Ч. Петцольд. – М.: "Русская редакция", 2004. – 512 с.
10. Разработка Web-сервисов XML и серверных компонентов на Microsoft Visual Basic .NET и Microsoft Visual C# NET: учебный курс MCAD/MCSD. – М.: "Русская редакция", 2004. – 576 с.
11. Фролов А.В., Фролов Г.В. \ Визуальное проектирование приложений C# \ А.В. Фролов, Г.В. Фролов – М.: КУДИЦ-ОБРАЗ, 2003. – 512 с.

Учебно- методическое издание

Составители:

Точка Владимир Николаевич, канд. техн. наук, доцент

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ
КОНСПЕКТ ЛЕКЦИЙ

Подписано в печать 03.11.2006 Формат 60х84/16. Объем 6,25 п.л. Бумага для ксерокса.
Тираж 500 экз. Цена договорная.

Россия, 392680, г. Тамбов, ул. Монтажников, 3
Тел. (4752) 564024. E-mail: tvn0752@mail.ru