КАЗАНСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

И.Л. Александрова, Д.Н. Тумаков

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C#

Учебное пособие

КАЗАНЬ

2017

УДК 681.3.06
ББК 32.973

Печатается по постановлению Редакционно-издательского совета ФГАОУВПО «Казанский (Приволжский) федеральный университет»

Научный редактор
доктор физ.-мат. наук, проф. Н. Б. Плещинский

Рецензенты
кандидат физ.-мат.наук, доцент И.Е. Плещинская
кандидат физ.-мат.наук, доцент Е.В. Рунг

Александрова И.Л.
Программирование на языке C#. Учебное пособие / И.Л. Александрова, Д.Н. Тумаков. – Издание 2-ое, исправленное и дополненное. – Казань: Казанский государственный университет, 2017. – 112 с.

ISBN

В курсе описаны основные возможности платформы .NET Framework. Показана структура программ на языке C#, возможности отладки и запуска консольных приложений. Рассмотрены основные типы данных и преобразования между ними. Изучены основные операторы языка программирования, в том числе, оператор обработки исключительных ситуаций. Рассмотрены синтаксис описания методов и способы передачи параметров между основной подпрограммой и методом. Объяснены основные концепции объектно-ориентированного программирования: инкапсуляция, наследование, полиморфизм. Показано, как работать со ссылочными типами данных, обсуждена работа сборщика мусора. Изучены предопределенные платформой классы. Объяснено, что такое индексаторы и атрибуты. Отдельные главы посвящены работе с событиями и делегатами, свойствами и перегруженными операторами.

Пособие предназначено для бакалавров и магистрантов ВУЗов, специализирующихся в области программирования.

ISBN

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

На сегодняшний момент язык программирования C#, вместе с Java и C++, один из самых мощных, быстро развивающихся и востребованных языков в ИТ-отрасли. В настоящий момент на нем пишутся самые различные приложения.

По сравнению со многими другими распространенными языками C# – достаточно молодой. Первая версия языка вышла вместе с релизом Microsoft Visual Studio .NET в феврале 2002 года. Текущей версией языка является версия C# 7.0, которая вышла в 7 марта 2017 года вместе с Visual Studio 2017.

C# является языком с Си-подобным синтаксисом и близок в этом отношении к C++ и Java. Поэтому, овладеть C# существенно легче, если есть навыки работы на C++ или Java.

C# является объектно-ориентированным и поддерживает полиморфизм, наследование, перегрузку операторов, статическую типизацию. Объектно-ориентированный подход позволяет решить задачи по построению крупных, но в тоже время гибких, масштабируемых и расширяемых приложений. C# в настоящий момент активно развивается, и с каждой новой версией появляется все больше интересных функциональностей, такие как лямбда-выражения, динамическое связывание, асинхронные методы и т.д.

В настоящем пособии описаны основные возможности платформы .NET Framework. Показана структура программ на языке C#, возможности отладки и запуска консольных приложений. Рассмотрены основные типы данных и преобразования между ними. Изучены основные операторы языка программирования, в том числе, оператор обработки исключительных ситуаций. Рассмотрены синтаксис описания методов и способы передачи параметров между основной подпрограммой и методом. Объяснены основные концепции объектно-ориентированного программирования: инкапсуляция, наследование, полиморфизм. Показано, как работать со ссылочными типами данных, обсуждена работа сборщика мусора. Изучены предопределенные платформой классы. Объяснено, что такое индексаторы и атрибуты. Отдельные главы посвящены работе с событиями и делегатами, свойствами и перегруженными операторами.

Во втором издании исправлены незначительные ошибки и добавлено рассмотрение работы с обобщенными коллекциями.

ГЛАВА 1. ОБЗОР ПЛАТФОРМЫ MS.NET

Microsoft®.NET Framework – платформа для создания windows приложений и распределенных web-приложений. .NET Framework содержит классы, интерфейсы и типы данных, которые облегчают и оптимизируют процесс разработки, а также обеспечивают доступ к функциям системы. Для упрощения взаимодействия между языками большинство типов платформы .NET Framework являются CLS-совместимыми, и поэтому их можно использовать в любом языке программирования, компилятор которого соответствует спецификации CLS.

Инструменты платформы .NET Framework представляют собой основу для создания элементов управления, компонентов и приложений .NET. .NET Framework содержит инструменты, предназначенные для следующих задач:

- представление базовых типов данных и исключений;
- инкапсуляция структур данных;
- операции ввода-вывода;
- доступ к сведениям о загруженных типах;
- вызов проверок безопасности .NET Framework;
- доступ к данным, предоставление графического пользовательского интерфейса на стороне клиента и управляемого сервером графического пользовательского интерфейса на стороне клиента.

.NET Framework предлагает расширенный набор интерфейсов, а также абстрактных и конкретных (неабстрактных) классов. Можно использовать существующие конкретные классы, кроме того, во многих случаях на их основе можно создавать собственные производные классы.

Приложению .NET через предопределенные классы в .NET Framework доступны все сервисы Windows: IIS, WMI, Component services, Message Queuing.Перечислим некоторые компоненты платформы .NET Framework:

- *Common Language Runtime (CLR) - общеязыковая среда выполнения* – упрощает разработку приложений, поддерживает различные языки, обеспечивает безопасное выполнение кода, контролирует ресурсы.
- *Библиотека классов* – содержит большое количество классов, с возможностью расширения под нужды разработчика.
- ADO.NET – обеспечивает поддержку работы с базами данных и XML.
- ASP.NET – инструмент для разработки web-приложений.

- XML Web сервисы – программируемые web компоненты, которые доступны различным приложениям.

Платформа MS.NET предоставляет следующие возможности:

- Поддержка всех web стандартов, таких как HTML, XML, SOAP, XSLT, XML Path Language и других. Создание и поддержка XML Web сервисов.
- Одинаковая среда для всех языков программирования.
- Легкое использование при разработке, все классы собраны в иерархические пространства имен, поддержка общей системы типов.
- Легко расширяемые классы – иерархия классов не скрыта от разработчика. Разработчик может разрабатывать свои собственные классы на основе имеющихся.

Общеязыковая среда выполнения

Общеязыковая среда выполнения упрощает разработку приложений, обеспечивает правильное и безопасное выполнение, поддерживает несколько языков программирования и следит за ресурсами. Эта среда также называется управляемой средой, она управляет памятью, потоками, организует удаленное взаимодействие. В среде CLR автоматически работает ряд сервисов:

- Загрузчик классов – управляет метаданными, загружает и располагает классы в памяти.
- Перевод из промежуточного языка MSIL в машинные коды во время выполнении (Just-in-time).
- Сборщик мусора – следит за неиспользуемыми объектами. При выходе из области видимости объектов и недостатке памяти сборщик очищает их.
- Ядро отладки – позволяет отлаживать и трассировать код.
- Проверка типов – не допускает опасных преобразований типов.
- Обработка исключений – структурированная система классов исключений, интегрированная с Windows.
- Поддержка COM-объектов.
- Поддержка базовых классов интегрированных в среду.
- Служба безопасности.

Библиотека классов .NET Framework увеличивает мощность среды выполнения и обеспечивает высокоуровневые сервисы, необходимые при программировании. Библиотека классов разбита на иерархию пространств имен. Представим некоторые из них:

Пространство имен System содержит фундаментальные классы и определяет наиболее часто используемые типы данных, события, интерфейсы, атрибуты и исключения. Также, классы преобразования данных, манипуляций с параметрами .

System.Collections содержит списки, хэш-таблицы и другие виды группировки данных.

ADO.NET – следующее поколение технологии ADO, обеспечивает расширенную поддержку отсоединенной программной модели БД, поддерживает работу с XML. Классы для работы с ADO.NET находятся в System.Data, с XML – в пространстве имен System.XML.

ASP.NET – программная структура, основанная на среде выполнения, работающей на сервере, которая позволяет создавать мощные Web приложения. Классы расположены в System.Web. Обеспечивается поддержка XML Web сервисов, необходимых для распределенной разработки.

Пространство имен System.Windows.Forms используется для создания windows интерфейса. В нем содержится большое количество функций, ранее доступных только в API.

Пространство имен System.Drawing обеспечивает доступ к графике GDI+.

Языки MS.NET Framework

MS.NET Framework поддерживает множество языков программирования, всего порядка двадцати, рассмотрим некоторые из них:

C# разработан для платформы .NET, это первый современный объектно-ориентированный язык из семейства C и C++. Его основные свойства – это классы, интерфейсы, делегаты, пространства имен, свойства, индексаторы и т.д. Нет необходимости в заголовочном файле.

Расширение управляемого C++ – минимальное расширение C++. Обеспечен доступ к возможностям платформы .NET Framework, включая сборщик мусора, наследование только от одного класса и т.д.

Visual Basic .NET улучшен по сравнению с предыдущей версией, добавлены наследование, конструкторы, полиморфизм, перегрузка конструкторов и т.д.

JScript.NET полностью переписан для поддержки среды .NET, включает поддержку классов, наследования, типов и компиляции.

Visual J#.NET – инструмент разработки для Java-программистов, желающих создавать приложения и сервисы в среде .NET. Имеется возможность автоматически обновлять существующие проекты Visual J++ 6.0 в решения Visual Studio .NET.

Сторонние языки – некоторые языки программирования также поддерживаются платформой .NET: APL, COBOL, Pascal, Oberon, Perl, Python и SmallTalk.

Вопросы к разделу

1. Что такое платформа MS.NET? Каковы её преимущества?
2. Перечислите основные понятия платформы .NET.
3. Охарактеризуйте компоненты MS.NET Framework.
4. Что такое общеязыковая среда выполнения? Какие ее основные сервисы?
5. Перечислите основные классы библиотеки .NET Framework.

ГЛАВА 2. ОБЗОР ЯЗЫКА C#

В главе рассмотрим основную структуру программ на языке C#. Рассмотрим класс Console для выполнения простых операций ввода и вывода. Дадим ряд советов по обработке исключений и документации кода.

Структура программы на C#

При изучении любого языка, первая программа, которую обычно пишут – «Hello, World». Рассмотрим, как она выглядит в C#.

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

При выполнении приведенного кода на экране появится надпись «Hello, World».

В C# приложение – это коллекция одного или нескольких классов, структур и других типов. Класс определяется как набор данных и методов, работающих с ними.

При рассмотрении кода приложения «Hello, World», видим, что есть единственный класс названный Hello. После имени класса открываем фигурные скобки ({). Всё, что находится до соответствующей закрывающейся скобки (}), является частью класса.

Можно распределить один класс приложения C# на несколько файлов. Также, с другой стороны, можно поместить несколько классов в один файл.

Каждое приложение должно начинаться с какого-то оператора. В C# при запуске приложения начинает выполняться метод Main. Несмотря на то, что в приложении может быть много классов, точка входа всегда одна. Может быть несколько методов Main, но запускаться будет только один, он выбирается при компиляции. Синтаксис Main важен, если проект создан в Visual Studio, то он

сгенерируется автоматически. При завершении метода Main приложение заканчивает работу.

Как часть MS.NET Framework C# содержит много различных классов, которые упорядочены в пространствах имен. Пространство имен (namespace) – это набор связанных классов. Пространство имен также может содержать вложенные пространства имен. Основное пространство имен – это System. К объектам пространств имен можно обращаться при помощи полного имени, используя префиксы. Например, пространство System, содержит класс Console, который выполняет несколько методов, включая WriteLine

System.Console.WriteLine(“Hello, World”);

Директива using позволяет обращаться к членам пространства имен напрямую без использования полного имени.

using System;

Console.WriteLine(“Hello, World”);

Основные операции ввода/вывода

В этом разделе рассмотрим класс Console и его методы ввода и вывода. Класс Console обеспечивает приложению C# доступ к стандартным потокам ввода, вывода и ошибок. Стандартный поток ввода – клавиатура. Стандартный поток вывода и ошибок – экран. Эти потоки могут быть перенаправлены из/в файлы.

Write и WriteLine методы вывода информации на консоль. Они перегружаемы, то есть могут выводить как строки, так и числа.

```
Console.WriteLine(99);
Console.WriteLine("Hello, World");
Console.WriteLine("The sum of {0} and {1} is {2}", 100, 130, 100+130);
```

Можно использовать левое и правое выравнивание, задавать ширину вывода.

```
Console.WriteLine("\"Левое выравн. в поле ширины 10: {0, -10}\"", 99);
Console.WriteLine("\"Правое выравн. в поле ширины 10: {0,10}\"", 99);
```

Будет выведено

“Левое выравн. в поле ширины 10: 99 ”

“Правое выравн. в поле ширины 10: 99”

Есть настройки для вывода числовых форматов:

```
Console.WriteLine("Валюта  - {0:C}  {1:C4}", 88.8,  -888.8);
Console.WriteLine("Целое число - {0:D5}", 88);
Console.WriteLine("Экспонента - {0:E}", 888.8);
Console.WriteLine("Фиксированная точка - {0:F3}", 888.8888);
Console.WriteLine("Общий формат- {0:G}", 888.8888);
Console.WriteLine("Числовой формат - {0:N}", 8888888.8);
Console.WriteLine("Шестнадцатиричный - {0:X4}", 88);
```

Соответственно отобразят:

Валюта - $88.80 ($888.8000)
Целое число - 00088
Экспонента - 8.888000E+002
Фиксированная точка - 888.889
Общий формат - 888.8888
Числовой формат - 8,888,888.80
Шестнадцатеричный формат – 0058

Рекомендации по оформлению кода

Основная рекомендация – это комментирование кода. Комментирование программы позволяет разработчикам, не участвовавшим при написании, разобраться, как работает приложение. Нужно использовать полные и значимые имена для именования компонент. Хорошие комментарии объясняют не что написано в программе, а отвечают на вопрос, почему именно так. Если в вашей организации есть стандарты комментирования, придерживайтесь их.

C# поддерживает несколько вариантов комментирования кода: однострочные(//), многострочные комментарии(/* */) и XML-документация(///). В XML-документации можно использовать различные предопределенные теги. Для генерации XML файла с комментариями используется дополнительный параметр компиляции. Для компиляции кода из командной строки:

```
csc myprogram.cs /doc:mycomments.xml
```

Надежная программа на C# обязана обрабатывать непредвиденные ситуации. Независимо от того, сколько различных ошибок предусмотрено,

всегда существует вероятность того, что что-то может пойти не так. Когда происходит ошибка при выполнении приложения, операционная система генерирует исключение. Конструкцией try-catch можно перехватывать эти исключения. Если при выполнении программы в блоке try произошло исключение, управление передаётся блоку catch. Если в программе не обрабатывается исключение, то оно вызовет окно операционной системы с предложением отладить программу при помощи Just-in-Time Debugging.

Перед запуском приложения C# необходимо его откомпилировать. Компилятор преобразует исходный код в машинные коды. Компилятор C# можно вызывать как из командной строки, так и из Visual Studio. Пример вызова компилятора из командной строки:

csc Hello.cs /debug+ /out:Greet.exe

(Заметим, что csc – это файл, и путь к нему нужно прописывать полностью.)

Если компилятор находит ошибки, он сообщает строку ошибки и номер символа. Если ошибок нет, и приложение откомпилировалось, то его можно запускать как при помощи Visual Studio, так и в командной строке по имени файла с расширением exe.

Вопросы к разделу

1. Откуда начинается выполнение приложения C#?
2. Когда приложение заканчивает работу?
3. Сколько классов может содержать приложение C#?
4. Сколько методов Main может содержать приложение?
5. Как прочитать данные, введенные пользователем с клавиатуры?
6. В каком пространстве имен находится класс Console?
7. Что произойдёт при необработанном в приложении исключении?

Лабораторная работа

Задания на создание C# программ, компилирование и отладку, использование отладчика Visual Studio, обработку исключительных ситуаций. Время, необходимое на выполнение задания 60 мин.

Упражнение 2.1 Написать программу, которая спрашивает имя пользователя, и затем приветствует пользователя по имени. (Создать консольное приложение.)

- Откомпилировать и запустить программу с помощью Visual Studio.
- Откомпилировать и запустить программу с командной строки.
- В среде Visual Studio использовать пошаговую отладку. Посмотреть, как изменяется текущее значение переменной.

Упражнение 2.2 Написать программу, которой на вход подается два целых числа, на выходе – результат деления одного числа на другое. Предусмотреть обработку исключительной ситуации, возникающей при делении числа на ноль.

Домашнее задание 2.1 Прочитать букву с экрана и вывести на печать следующую за ней букву в алфавитном порядке.

Домашнее задание 2.2 Написать программу, которая решает квадратное уравнение. Входные данные – коэффициенты уравнения, выходные – найденные корни.

ГЛАВА 3. ИСПОЛЬЗОВАНИЕ СТРУКТУРНЫХ ПЕРЕМЕННЫХ

Все приложения работают с теми или иными данными. Разработчику C# необходимо понимать как хранятся и обрабатываются данные в приложении. Обычно данные хранятся в переменных, которые необходимо объявить до их использования. При объявлении переменной под неё резервируется некоторое количество памяти в зависимости от типа переменной и объявляется имя. После объявления переменной можно присваивать ей значения.

В этой главе рассмотрим как использовать структурные переменные в C#. Как именовать переменные в соответствии со стандартами, а также как присваивать значения и переводить существующие переменные из одного типа в другой.

Общая система типов (Common Type System)

Каждая переменная имеет тип данных, который определяет какие значения хранятся в ней. C# – язык безопасных типов, т.е. компилятор гарантирует, что значение хранящееся в переменной будет всегда соответствующего типа.

CTS (Common Type System) – интегрированная часть общеязыковой среды выполнения. Эта модель определяет правила, которыми руководствуется среда выполнения при объявлении, использовании и управления типами. CTS обеспечивает структуру, необходимую для многоязыковой интеграции, безопасности типов и высокой эффективности выполнения кода.

Рассмотрим два типа переменных: структурные и ссылочные. Структурные типы данных напрямую содержат данные. Каждая переменная содержит свою копию данных, т.е. при операции над одной переменной невозможно изменить другую. Структурные типы данных включают в себя встроенные и пользовательские типы. Разница между ними в C# минимальна, так как они используются одинаково. Все структурные типы напрямую содержат данные и не могут быть null.

Ссылочные типы данных содержат ссылки на данные. Две переменные могут указывать на один и тот же объект, т.е. при изменении одной ссылочной переменной, можно изменить другую.

Все базовые типы данных содержатся в пространстве имен System. Все типы наследуются от System.Object. Все структурные типы наследуются от System.ValueType.

Встроенные типы объявляются при помощи зарезервированных слов, кроме того, можно объявить при помощи типа структуры из пространства имен System:

sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

Правила именования переменных

При именовании переменных необходимо соблюдать следующие правила (если не соблюдать, то получим ошибку при компиляции):

- можно использовать только буквы, нижнее подчеркивание и цифры,
- имя переменной начинается только с буквы или подчеркивания,
- после первого символа можно использовать и цифры,
- нельзя использовать зарезервированные слова.

Рекомендации по именованию:

- избегать имена только из заглавных букв,
- избегать начинать имя с подчеркивания,
- избегать аббревиатур,
- именование PascalCasing – каждое слово начинается с большой буквы, используется для классов, методов, свойств, перечислений, интерфейсов, констант, пространств имен,
- именование camelCasing – каждое слово, кроме первого, начинается с большой буквы, используется для переменных, полей и параметров.

Использование встроенных типов данных

Для использования переменной необходимо выбрать ей имя, назначить тип и присвоить начальное значение.

Переменные, которые объявлены в методах, называются локальными. В C# нельзя использовать неинициализированные переменные, в этом случае выдаётся ошибка при компиляции. Переменной можно присвоить значение соответствующего типа.

Добавление значения переменной осуществляется очень просто:

```
int itemCount;
itemCount = 2;
itemCount = itemCount + 40;
```

Приведем сокращенные способы записи арифметический действий:

```
var += expression;          // var = var + expression
var -= expression;          // var = var - expression
var *= expression;          // var = var * expression
var /= expression;          // var = var / expression
var %= expression;          // var = var % expression
```

Выражения (expression) состоят из операций и операндов. Существуют следующие общие операции:

Операции присваивания – присваивают значение правого операнда левому: = *= /= %= += -= <<= >>= &= ^= |= .

Операции сравнения – сравнивают два значения: == != .

Логические – производит побитовые операции над операндами: << >> & ^ | .

Условные – выполняют одно выражение из двух в зависимости от булевого условия: && || ?: .

Инкремент и декремент – увеличивает/уменьшает значение на единицу: ++ -- .

Арифметические – выполняют стандартные арифметические операции: + - * / % .

Операции инкремент и декремент имеют два варианта префиксный (++var) и постфиксный (var++). Соответственно, сначала выполняется увеличение значения переменной var, а затем выполняется выражение и

наоборот, сначала выражение, и только после него изменяется значение операнда.

Пользовательские типы данных

В приложениях есть возможность создавать свои типы данных: структуры и перечисления.

Перечисления полезны, когда переменные могут принимать значения только из определенного набора. Каждому значению в перечислении соответствует свой номер. По умолчанию нумерация элементов начинается с нуля.

```
enum Color { Red, Green, Blue }
```

Цвет Red будет иметь значение 0, Green – 1, а Blue будет 2.

Пример использования переменной перечислимого типа:

```
Color colorPalette;           // объявление переменной
colorPalette = Color.Red;  // установка значения
```

или

```
colorPalette = (Color)0;    //явное преобразование из int
```

Структуры можно использовать для создания объектов, ведущих себя как встроенные типы. Они группируют данные различного типа.

```
public struct Employee
{
        public string firstName;
        public int age;
}
```

Для доступа к элементам структур используется следующая конструкция:

```
Employee companyEmployee;              // объявление переменной
companyEmployee.firstName = “Joe”; // присваивание
companyEmployee.age = 23;                //     значений
```

Преобразование типов

Различают два способа преобразования типов: явное и неявное. При неявном преобразовании возможно потеря точности. Для явного преобразования используется операция приведения типов.

Пример неявного преобразования:

```
using System;
class Test;
{
        static void Main()
        {
                int intValue = 123;
                long longValue = intValue;
                Console.WriteLine(“(long) {0} = {1}”, intValue, longValue)
        }
}
```

Пример явного преобразования:

```
using System;
class Test;
{
        static void Main()
        {
                long longValue = Int64.MaxValue;
                int intValue = (int) longValue;
                Console.WriteLine(“(int) {0} = {1}”, longValue, intValue)
        }
}
```

Вторая программа на экране напечатает (int) 9223372036854775807 = -1, т.к. произойдет переполнение переменной типа int. Есть возможность отслеживать такие ошибки, путем размещения данного кода внутри блока checked. Checked используется для явной проверки переполнения при выполнении арифметических операций и преобразований с данными целого типа. Пример:

```
using System;
class Test
{
static void Main()
{
        long longValue = Int64.MaxValue;
        checked{
                try{
```

```
                int intValue = (int) longValue;
                Console.WriteLine("(int) {0} = {1}", longValue, intValue);
                }
                catch (Exception e){
                Console.WriteLine("Ошибка приведения типов");
                }
        }
}
}
```

Более подробно о блоке try-catch написано в четвертой главе.

Вопросы к разделу

1. Что такое общеязыковая система типов?
2. Может ли структурная переменная иметь значение null?
3. Можно ли не инициализировать переменные в C#? Почему?
4. Можно ли потерять данные в результате неявного преобразования?

Лабораторная работа

Задания на создание пользовательских типов данных, объявление и использование переменных. Время, необходимое на выполнение задания 35 мин.

Упражнение 3.1 Создать перечислимый тип данных отображающий виды банковского счета (текущий и сберегательный). Создать переменную типа перечисления, присвоить ей значение и вывести это значение на печать.

Упражнение 3.2 Создать структуру данных, которая хранит информацию о банковском счете – его номер, тип и баланс. Создать переменную такого типа, заполнить структуру значениями и напечатать результат.

Домашнее задание 3.1 Создать перечислимый тип ВУЗ{КГУ, КАИ, КХТИ}. Создать структуру работник с двумя полями: имя, ВУЗ. Заполнить структуру данными и распечатать.

ГЛАВА 4. ОПЕРАТОРЫ И ИСКЛЮЧЕНИЯ

В этой главе рассмотрим, как использовать операторы в C#, а также опишем, как обрабатывать исключения. В частности, покажем, как генерировать и перехватывать исключения, и как правильно описывать блок try-catch, чтобы ни одно исключение не вышло за рамки приложения.

Операторы в C#

Программа состоит из последовательности операторов. При выполнении они запускаются по очереди. При разработке приложения необходимо группировать операторы, в C# они группируются в блоки при помощи фигурных скобок. Каждый блок задает свою область видимости, в которой существуют локальные переменные

```
{
        // блок операторов
}
```

Операторы, используемые в программах, можно разделить на: операторы присваивания, арифметические операторы, логические операторы, условные операторы, оператор инкремента и декремента, операторы цикла и операторы перехода. Операторы присваивания, арифметические, логические, инкремента и декремента были рассмотрены в предыдущей главе. Рассмотрим остальные операторы.

Условные операторы

Операторы if и switch – операторы выбора, они вычисляют значение условия и на основе этого выбора выполняют операторы.

Неявное преобразование из int в bool содержит потенциальный путь к ошибкам, поэтому в C# такое действие запрещено. В условии обязательно должно быть булево выражение.

```
int x; ...
if (x) ...    // Должно быть x != 0 в C#
if (x = 0) ... // Должно быть x == 0 в C#
```

Можно использовать каскадную последовательность if, для чего предусмотрена конструкцию else if.

Оператор switch – механизм для обработки сложных условий. Он состоит из нескольких блоков case, каждый из которых определяется своей константой. Для группировки констант, пишем каждую из них в собственном case, а выполняющийся оператор – после всей группы. Операторы выполняются до оператора break.

```
enum MonthName { January, February, ..., December }
MonthName current;
int monthDays; ...
switch (current) {
case MonthName.February :
        monthDays = 28;
        break;
case MonthName.April :
case MonthName.June :
case MonthName.September :
case MonthName.November :
        monthDays = 30;
        break;
default :
        monthDays = 31;
        break;
}
```

Если константы повторяются, то получим ошибку при компиляции:

```
switch (trumps) {
        case Suit.Clubs :
        case Suit.Clubs : // Ошибка: повтор метки
                ...
        default :
        default : // Ошибка: повтор метки
                ...
}
```

Операторы цикла

Операторы цикла выполняют операции до тех пор, пока условие верно. Перечислим операторы цикла: while, do - while, for и foreach.

Цикл while – простейший оператор цикла. Заметим, что C# не поддерживает неявное преобразование в bool, поэтому обязательно надо использовать условие.

Отличие цикла do – while, в том что условие проверяется после выполнения операции.

Цикл for включает в себя первоначальную инициализацию счетчика, условие выполнения и обновление счетчика. Однако в нем может отсутствовать любая часть for – инициализация, условие и обновление. Например,

```
for (;;) {
        Console.WriteLine("Help ");
        ...
}
```

Цикл foreach – цикл для выборки всех элементов коллекции. Необходимо выбрать имя и тип переменной, которая будет идти по коллекции. Это переменная в теле цикла будет доступна только для чтения.

```
foreach(int number in numbers)
        Console.WriteLine(number);
```

Операторы перехода

Операторы goto, break и continue – операторы перехода. Они используются для передачи управления из одной части программы в другую.

goto – простейший оператор, управление при его вызове передается в точку программы, отмеченную меткой. Метка определяется двоеточием после идентификатора.

```
if(number % 2 == 0) goto Even;
Console.WriteLine(“odd”);
goto End;
Even:
Console.WiteLine(“even”);
End:;
```

Операторы break и continue используются в циклах, первый из них предназначен для выхода из цикла, второй – для перехода к следующей итерации.

Обработка исключений

В этом разделе изучим, как перехватывать и обрабатывать исключительные ситуации.

Традиционно обработка ошибок производится в теле программы, что затрудняет понимание логики самого приложения и запутывает код. Сообщения об ошибках представляют собой числа, которые не несут никакой смысловой нагрузки. Более того, один и тот же код ошибки может использоваться в разных функциях и означать совершенно разные ошибки.

В C# предусмотрен специальный механизм для генерирования и обработки исключительных ситуаций. Все исключения – это объекты, унаследованные от класса Exception. Названия классов раскрывают вид исключения. Объекты могут содержать дополнительную информацию об ошибке, например, FileNotFoundException может хранить имя файла.

Для обработки исключительной ситуации, необходимо потенциально опасный код заключить в блок **try-catch-finally.** В блоке try содержится код, который может сгенерировать исключительную ситуацию, например деление на ноль или отсутствие запрашиваемого файла на диске. Если возникла исключительная ситуация, выполнение кода в блоке try приостанавливается и начинается поиск блока catch, соответствующего по типу исключительной ситуации. Блок finally выполняется обязательно, вне зависимости от того, произошло исключение или нет. Он используется в двух случаях: для избегания повтора операторов и для освобождения ресурсов.

Есть несколько вариантов блока catch:

- общий catch { ... }, что равносильно catch (System.Exception) { ... } – обработка ошибки любого типа;
- по виду исключения, например, catch (OutOfMemoryException caught) { ... }, где caught – объект класса исключения OutOfMemoryException.

После блока try может присутствовать несколько блоков catch. В этом случае у каждого блока catch необходимо явно указывать, ошибку какого типа он обрабатывает. Нельзя чтобы в следующем блоке catch был указан тип

данных, наследник от класса указанном в предыдущем блоке catch. Нельзя чтобы было два блока catch, предназначенных для обработки ошибки одного и того же класса. Блок catch для ошибки типа System.Exception может быть только последним.

Приведем несколько примеров.

```
try {
    Console.WriteLine("Enter first number");
    int i = int.Parse(Console.ReadLine());
    Console.WriteLine("Enter second number");
    int j = int.Parse(Console.ReadLine());
    int k = i / j;
}
catch (OverflowException caught)
{
    Console.WriteLine(caught);
}
catch(DivideByZeroException caught)
{
    Console.WriteLine(caught);
}
finally
{
    Console.WriteLine(“Блок try-catch завершен”);
}
```

Приведенный выше код не содержал никаких ошибок.

```
try {
...
}
catch { ... }
```

catch (OutOfMemoryException caught) { ... } // ошибка – общий блок catch следует перед блоком с конкретной ошибкой.

В следующем примере один и тот же класс используется два раза:

```
catch (OutOfMemoryException caught) { ... }
catch (OutOfMemoryException caught) { ... } // Error
```

В следующем примере сначала идет блок catch, который обрабатывает ошибку класса Exception, в следующем блоке – ошибку класса OutOfMemoryException, который является наследником от класса Exception.

```
catch (Exception caught) { ... }
catch (OutOfMemoryException caught) { ... } // Error
```

Оператор throw генерирует исключение вручную. Можно генерировать только объекты класса исключения, то есть для этого надо создать свой собственный класс, наследник от класса Exception.

```
if (minute<1|| minute>=60){
        string fault=minute + “is not a valid minute”;
        throw new InvalidTimeException(fault);
        …
}
```

В приведенном примере предполагается, что класс InvalidTimeException описан где-то раньше, этот класс является наследником от класса Exception, его конструктор содержит один строковый параметр.

Можно использовать оператор throw внутри блока catch:

```
catch (IOException caught) {
        ...
        throw new FileNotFoundException(filename);
}
```

По умолчанию проверка переполнения стека арифметическими операциями в C# выключена. Для включения проверки переполнения можно создать блок checked, или при компиляции указать опцию: csc /checked+ example.cs, а для отключения:
csc /checked- example.cs. Блок unchecked явно выключает проверку. В конце главы 3 был приведен пример для обработки исключительной ситуации, которая возникает при переполнении арифметическими операциями.

Рекомендации по обработке исключений:

· Включать строки описания

```
string description = String.Format("{0}({1}): newline in string constant", filename, linenumber);
throw new System.Exception(description);
```

· Генерировать исключения более специфичного вида, например класс FileNotFoundException, лучше чем более общий класс IOException.

· Не позволять исключениям выходить из метода Main, то есть всегда должна существовать такая конструкция

```
static void Main( )
{
```

```
        try {
                ...
        }
        catch (Exception caught) {
        ...
        }
}
```

Вопросы к разделу

1. Перечислите основные группы операторов, выделив их характерные особенности.
2. Что такое исключение? Какие классы для обработки исключительных ситуаций вы знаете?
3. Как организовать обработку исключительной ситуации?

Лабораторная работа

Задания на операторы if, switch, for, while, foreach, обработку исключений. Время, необходимое на выполнение задания 45 мин.

Упражнение 4.1 Написать программу, которая читает с экрана число от 1 до 365 (номер дня в году), переводит этот число в месяц и день месяца. Например, число 40 соответствует 9 февраля (високосный год не учитывать).

Упражнение 4.2 Добавить к задаче из предыдущего упражнения проверку числа введенного пользователем. Если число меньше 1 или больше 365, программа должна вырабатывать исключение, и выдавать на экран сообщение.

Домашнее задание 4.1 Изменить программу из упражнений 4.1 и 4.2 так, чтобы она учитывала год (високосный или нет). Год вводится с экрана. (Год високосный, если он делится на четыре без остатка, но если он делится на 100 без остатка, это не високосный год. Однако, если он делится без остатка на 400, это високосный год.)

ГЛАВА 5. МЕТОДЫ И ПАРАМЕТРЫ

При разработке большинства приложений их разделяют на функциональные модули, так как маленькие разделы кода удобнее для понимания, разработки и отладки. Кроме того, это позволяет несколько раз использовать одни и те же участки кода для других приложений.

В C# приложение состоит из классов, которые содержат именованные блоки кода, называемые методами. Метод – это член класса, который может осуществлять действия или вычислять значения.

Использование методов

В C# программа состоит из классов, содержащих методы. Метод может выполнять действие или вычислять значение.

Метод – это набор операторов, которые выполняются вместе. В C# все методы принадлежат какому-либо классу. Для создания метода необходимо задать его имя, определить список параметров и тело метода.

Для вызова метода используется имя метода, если у метода есть параметры, то необходимо их определить. Для вызова метода другого класса необходимо, чтобы он был объявлен как public, вызов осуществляется по имени класса и метода.

```
using System;
class NestExample
{
        static void Method1
        {
                Console.WriteLine(“Method1”);
        }
        static void Method2
        {
                Method1();
                Console.WriteLine(“Method2”);
                Method1();
        }
```

```
        static void Main()
        {
        Method2();
        Method1();
        }
}
```

В результате получим:

```
Method1
Method2
Method1
Method1
```

Оператор return останавливает выполнение метода и передаёт управление вызвавшему данный метод оператору. Если метод не void, то необходимо вернуть значение соответствующего типа.

Каждый метод имеет набор своих локальных переменных, они видны только в нем и при завершении работы метода уничтожаются. Для того чтобы переменные были видны в нескольких методах класса, необходимо объявить их полями вне метода, но внутри класса.

Для не void методов необходимо возвращать значение, каждый «путь выполнения» метода должен заканчиваться оператором return. Для void методов оператор return не обязателен.

Использование параметров

Параметры позволяют передавать информацию из одного метода в другой. При объявлении метода можно задать список его параметров, если список пустой, то это означает, что метод не имеет параметров.

```
static void MethodWithParameters(int n, string y)
{
        // ...
}
```

При вызове метода необходимо задать значения его параметрам.

```
MethodWithParameters(2, "Hello, world");
```

или

```
int p = 7;
string s = "Test message";
 MethodWithParameters(p, s);
```

В C# существуют три варианта передачи параметров: по значению, по ссылке и выходные параметры. При передаче параметров по значению, изменение значения параметра в методе не влияет на значение в вызвавшем методе.

```
static void AddOne(int x)
{
        x++;
}
static void Main()
{
        int k = 6;
        AddOne(k);
        Console.WriteLine(k); // Выведет на экран 6, не 7
}
```

При передаче по ссылке передается ссылка на переменную, поэтому все действия, производимые с параметром, оказывают влияние на значение переменной в вызывающем методе. Для каждого параметра по ссылке необходимо указывать ключевое слово ref. До вызова метода необходимо обязательно инициализировать переменную.

```
static void AddOne(ref int x)
{
        x++;
}
static void Main()
{
        int k = 6;
        AddOne(ref k);
        Console.WriteLine(k); // Выведет на экран 7
}
```

Выходные параметры похожи на параметры по ссылке, единственное отличие их состоит в том, что их можно не инициализировать до вызова

метода. При передаче выходного параметра перед ним указывается ключевое слово out.

```
static void OutDemo(out int p)
{
        // …
}
static void Main()
{
        int k;
        OutDemo(out  k);
        Console.WriteLine(k);
}
```

C# позволяет использовать механизм передачи списка параметров изменяемой длины. Для этого используется ключевое слово params. Правило использования: допуст***и***м только один список параметров, который должен быть массивом конкретного типа и размещаться последним в общем списке параметров.

```
static long AddList(int k, params long[] v)
{
        long total;
        long i;
        for(i = 0, total = 0; i < v.Length; i++)
                total += v[i];
        return total*k;
}
```

При вызове можно использовать два пути: вызывать как список отдельных элементов или как массив.

```
static void Main( )
{
        long x;
        x = AddList(63, 21, 84); // Список
        x = AddList(new long[ ]{ 63, 21, 84 }); // Массив
}
```

Для выбора вида параметров необходимо учитывать два аспекта: механизм передачи и эффективность. По эффективности передача по значению лучше, чем передачи по ссылке.

Когда метод вызывает себя, то это называется рекурсией. Если это происходит при участии другого метода, то это будет неявной рекурсией. Рекурсия часто используется для упрощения логики программы.

Перегрузка методов

Имя метода не может совпадать с именем любого другого элемента класса, но может совпадать с другим методом – это называется перегрузка метода. Перегружаемые методы – это методы с одинаковыми именами в одном классе.

```
class OverloadingExample
{
        static int Add(int a, intb)
        {
                return a + b;
        }
        static int Add(int a, intb, int c)
        {
                return a + b + c;
        }
        static void Main()
        {
                Console.WriteLine(Add(1,2) + Add(1,2,3));
        }
}
```

Нельзя использовать одно имя для метода и переменной, константы или перечисления.

```
class BadMethodNames
{
        static int k;
        static void k( )
        {
                // ...
        }
}
```

Компилятор использует сигнатуру метода для различия методов в классе. В сигнатуру входят следующие элементы: имя метода, типы параметров, модификаторы параметров. В сигнатуру не входят имена параметров и возвращаемый тип метода.

Следующие три метода имеют различную сигнатуру

```
static int LastErrorCode( ) {  }
static int LastErrorCode(int n) {  }
static int LastErrorCode(int n, int p) {  }
```

Следующие три метода имеют одинаковую сигнатуру

```
static int LastErrorCode(int n) { }
static string LastErrorCode(int n) { }
static int LastErrorCode(int x) { }
```

Перегружаемые методы полезны, если есть одинаковые методы, различающиеся только списком параметров.

```
class GreetDemo
{
        static void Greet( )
        {
                Console.WriteLine("Hello");
        }
        static void Greet(string Name)
        {
                Console.WriteLine("Hello " + Name);
        }
        static void Main( )
        {
                Greet( );
                Greet("Alex");
        }
}
```

Кроме того, перегрузки методов полезны при добавлении новой функциональности. Допустим в предыдущем классе, захотим приветствовать пользователя в соответствии со временем дня.

```
class GreetDemo
{
	enum TimeOfDay { Morning, Afternoon, Evening }

	static void Greet( )
	{
		Console.WriteLine("Hello");
	}
	static void Greet(string Name)
	{
		Console.WriteLine("Hello " + Name);
	}
	static void Greet(string Name, TimeOfDay td)
	{
		string Message = "";
		switch(td)
		{
		case TimeOfDay.Morning:
			Message="Good morning";
			break;
		case TimeOfDay.Afternoon:
			Message="Good afternoon";
			break;
		case TimeOfDay.Evening:
			Message="Good evening";
			break;
		}
		Console.WriteLine(Message + " " + Name);
	}
	static void Main( )
	{
		Greet( );
		Greet("Alex");
		Greet("Sandra", TimeOfDay.Morning);
	}
}
```

Вопросы к разделу

1. Объясните что такое методы и почему они важны?
2. Опишите три возможных пути передачи параметров и соответствующие ключевые слова C#.
3. Когда создаются и уничтожаются локальные переменные?
4. Что входит в сигнатуру метода?

Лабораторная работа

Задания на методы с параметрами и без, различные механизмы передачи параметров. Время, необходимое на выполнение задания 60 мин.

Упражнение 5.1 Написать метод, возвращающий наибольшее из двух чисел. Входные параметры метода – два целых числа. Протестировать метод.

Упражнение 5.2 Написать метод, который меняет местами значения двух передаваемых параметров. Параметры передавать по ссылке. Протестировать метод.

Упражнение 5.3 Написать метод вычисления факториала числа, результат вычислений передавать в выходном параметре. Если метод отработал успешно, то вернуть значение true; если в процессе вычисления возникло переполнение, то вернуть значение false. Для отслеживания переполнения значения использовать блок checked.

Упражнение 5.4 Написать рекурсивный метод вычисления факториала числа.

Домашнее задание 5.1 Написать метод, который вычисляет НОД двух натуральных чисел (алгоритм Евклида). Написать метод с тем же именем, который вычисляет НОД трех натуральных чисел.

Домашнее задание 5.2 Написать рекурсивный метод, вычисляющий значение n-го числа ряда Фибоначчи. Ряд Фибоначчи – последовательность натуральных чисел 1, 1, 2, 3, 5, 8, 13… Для таких чисел верно соотношение $F_k = F_{k-1} + F_{k-2}$.

ГЛАВА 6. МАССИВЫ И КОЛЛЕКЦИИ

Массивы хороший инструмент группировки данных. Однако, массивы хранят фиксированное количество объектов, а иногда заранее не неизвестно, сколько потребуется объектов. И в этом случае намного удобнее применять коллекции. Еще один плюс коллекций состоит в том, что некоторые из них реализует стандартные структуры данных, например, стек, очередь, словарь, которые могут пригодиться для решения различных специальных задач.

Большая часть классов коллекций содержится в пространствах имен System.Collections (простые необобщенные классы коллекций), System.Collections.Generic (обобщенные или типизированные классы коллекций) и System.Collections.Specialized (специальные классы коллекций). Также для обеспечения параллельного выполнения задач и многопоточного доступа применяются классы коллекций из пространства имен System.Collections.Concurrent.

В этой главе рассмотрим использование массивов и обобщенных коллекций.

Массивы

Синтаксис массивов в C#

```
type[ ]  name;    // правильно
type   name[ ];   // неправильно в C#
type[4]    name;   // также неправильно C#
```

Для объявления двумерных массивов, используются пустые индексы через запятую.

```
int[,] grid;
```

Для доступа к элементам массива используются индексы в квадратных скобках. Нумерация начинается с нуля. Если размерность массива больше одного, то индексы перечисляются через запятую.

```
long[] row;                    int[,] grid;
…                              …
```

```
row[3];                                grid[1,2];
```

В C# индексы массива автоматически проверяются на удовлетворение размерности, при выходе за интервал выдаётся исключение IndexOutOfRangeException. Для проверки размерности можно использовать свойство массива Length и метод GetLength.

Сравним массивы с коллекциями. Коллекции гибче массивов, могут хранить различные элементы и имеют переменную длину. Но в связи с этим работа с коллекциями медленнее.

```
ArrayList flexible = new ArrayList( );
flexible.Add("one"); // Добавили строку...
flexible.Add(99);    // Добавили int
```

Создание коллекции только для чтения

```
ArrayList flexible = new ArrayList( ); ...
ArrayList noWrite = ArrayList.ReadOnly(flexible);
noWrite[0] = 42;    // Исключение при выполнении
```

Объявление массива не создает его, так как массив – это ссылочный тип. При объявлении массива можно не знать его размерность, но при создании знать размер необходимо. Память для массива выделяется последовательно.

```
long[] row = new long[4];
int[,] grid = new int[2,3];
```

Можно использовать список инициализации при создании массива. Необходимо объявить все элементы, в качестве элементов можно использовать выражения. Те же правила справедливы для многомерных массивов: необходимо определить все строки и в каждой строке все элементы.

```
int[,] data = new int[2,3]{
        {2,3,4},
        {5,6,7}  }
```

Размерность массива можно задавать как константами, так и вычисляемыми значениями. Единственное ограничение при вычисляемой длине, нельзя использовать список инициализации.

```
string s = Console.ReadLine();
```

```
int size = int.Parse(s);
long[] row = new long[size];
```

При копировании переменной массива, не создается новый массив, создается новая ссылка на тот же массив.

```
long [] row = new long[4];
long [] copy = row;
row[0]++; // изменит copy[0]
```

Рассмотрим некоторые свойства и методы класса System.Array.

- Свойство Rank – размерность массива.
- Свойство Length – общая длина массива, для многомерных массивов количество всех ячеек.

System.Array – класс, от которого неявно наследуют все массивы в C#.

- Sort – сортировка массива, поддержка интерфейса IComparable.
- Clear – очистка массива, все элементы устанавливаются в NULL.
- Clone – создаёт копию массива, копирует все элементы. Этот метод не следит за значениями в элементах. Если там ссылки на объекты, то они просто скопируются, новых объектов создано не будет.
- GetLength – по номеру размерности получаем длину массива в этой размерности.
- IndexOf – ищет значение в массиве, если нашел возвращает индекс первого вхождения, иначе -1.

При возвращении массива из метода его размеры не задаются, скобки остаются пустыми. Если задать, получим ошибку при компиляции. Также с многомерными массивами.

```
static int[,] CreateArray( ) {
        string s1 = System.Console.ReadLine( );
        int rows = int.Parse(s1);
        string s2 = System.Console.ReadLine( );
        int cols = int.Parse(s2);
        return new int[rows,cols];
}
```

При передаче массива в метод в качестве параметра, новый массив не создается, передается ссылка на тот же массив. Все действия с массивом в

методе сохранятся для вызывающего метода. Если нужно избежать этого, необходимо передавать копию массива при помощи метода Array.Copy

При вызове приложения из командной строки можно использовать строку параметров, которые разделяются пробелом. Например:

C:\> pkzip –add –rec –path=relative c:\code *.cs

Тогда если pkzip написан на C#, получим массив

```
string[ ] args = {
 "-add",
 "-rec",
 "-path=relative",
 "c:\\code",
 "*.cs"
};
```

Этот массив получим при запуске метода Main.

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

Для прохода по массиву можно использовать цикл foreach. Тогда не нужен счетчик, проверка длины массива и обращение к элементу.

Две следующие конструкции эквивалентны

```
for (int i = 0; i < args.Length; i++) {
        System.Console.WriteLine(args[i]);
}
```

и

```
foreach (string arg in args) {
        System.Console.WriteLine(arg);
}
```

Также foreach можно использовать и для многомерных массивов

```
int[,] numbers = { {0,1,2}, {3,4,5} };
foreach (int number in numbers) {
        System.Console.WriteLine(number);
}
```

Списки – List<T>

Класс List<T> является самым простым из классов коллекций. Его можно использовать практически так же, как массив, ссылаясь на существующий в коллекции List<T> элемент с использованием обычной для массивов системы записи с квадратными скобками и индексом элемента. Можно добавить элемент к концу коллекции List<T>, воспользовавшись имеющимся в ее классе методом Add, которому предоставляется добавляемый элемент. Размер коллекции увеличивается List<T> автоматически.

```
List<int> list = new List<int>();
list.Add(3);
list.Add(1);
list[0]=list[1]+3;
```

Указывать размер коллекции List<T> при ее создании не обязательно. Коллекция может изменять свои размеры по мере добавления (или удаления) элементов. Но надо иметь в виду, что на физическое добавление элементов уходит время процессора, и при необходимости нужно указать начальный размер. Но если он будет превышен, то в силу необходимости коллекция List<T> просто расширится. Если первую строку примера изменить на

```
List<int> list = new List<int>(2);
```

то результат будет один и тот же, но последний вариант отработает быстрее. В этом случае команда Add – это просто инициализация очередного элемента в конце списка.

Для удаления из коллекции List<T> указанного элемента можно воспользоваться методом Remove. Элементы коллекции List<T> автоматически перестроятся, закрывая образовавшееся пустое место. С помощью метода RemoveAt можно также удалить элемент, указав его позицию в коллекции List<T>. Можно вставить элемент в середину коллекции List<T>, воспользовавшись для этого методом Insert. При этом размер коллекции List<T> также изменится автоматически. Размер коллекции, т.е. количество инициализированных элементов, можно получить через свойство (только на чтение) Count, а емкость коллекции – через свойство (чтение и запись) Capacity.

Приведем ряд методов класса List<T>:

- Add(T) – добавляет объект в конец списка List<T>.
- AddRange(IEnumerable<T>) – добавляет элементы указанной коллекции в конец списка List<T>.
- Clear()– удаляет все элементы из коллекции List<T>.
- Contains(T) – определяет, входит ли элемент в коллекцию List<T>.
- ConvertAll<TOutput>(Converter<T, TOutput>) – преобразует элементы текущего списка List<T> в другой тип и возвращает список преобразованных элементов.
- Exists(Predicate<T>) – определяет, содержит ли List<T> элементы, удовлетворяющие условиям указанного предиката.
- Find(Predicate<T>) – выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает первое найденное вхождение в пределах всего списка List<T>.
- FindAll(Predicate<T>) – извлекает все элементы, удовлетворяющие условиям указанного предиката.
- FindIndex(Predicate<T>) – выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает отсчитываемый от нуля индекс первого найденного вхождения в пределах всего списка List<T>.
- IndexOf(T) – осуществляет поиск указанного объекта и возвращает отсчитываемый от нуля индекс первого вхождения, найденного в пределах всего списка List<T>.
- Insert(Int32, T) – вставляет элемент в коллекцию List<T> по указанному индексу.
- Remove(T) – удаляет первое вхождение указанного объекта из коллекции List<T>.
- RemoveAll(Predicate<T>) – удаляет все элементы, удовлетворяющие условиям указанного предиката.
- RemoveAt(Int32) – удаляет элемент списка List<T> с указанным индексом.
- Sort() – сортирует элементы во всем списке List<T> с помощью функции сравнения по умолчанию.
- ToArray() – копирует элементы списка List<T> в новый массив.

Многие методы коллекции List<T> содержатся и в других коллекциях.

Двухсвязные списки – LinkedList<T>

Класс коллекций LinkedList<T> реализует двусвязный список. В каждом элементе списка содержится значение элемента со ссылкой на следующий элемент списка (свойство Next) и его предыдущий элемент (свойство Previous).

В классе LinkedList<T> записи, присущие массивам, не поддерживаются. Вставка элементов осуществляется отличным от List<T> способом. Можно воспользоваться методом AddFirst для вставки элемента в начало списка с перемещением предыдущего первого элемента дальше по списку и установки в качестве значения его свойства Previous ссылки на новый элемент. Аналогично этому для вставки элемента в конец списка можно воспользоваться методом AddLast. Для вставки элемента перед указанным элементом списка или после него можно воспользоваться методами AddBefore и AddAfter.

Первый элемент коллекции LinkedList<T> можно найти, запросив значение свойства First, а свойство Last даст ссылку на последний элемент списка. Для последовательного обхода элементов связного списка можно приступить к этой операции с одного конца и пошагово применять ссылки из свойства Next или Previous, пока не будет найден элемент, у которого это свойство имеет значение null. Конечно же, лучше воспользоваться инструкцией foreach, которая выполнит последовательный обход элементов вперед по списку LinkedList<T>-объекта, автоматически остановившись в конце.

Удаление элемента из коллекции LinkedList<T> осуществляется с помощью методов Remove, RemoveFirst и RemoveLast.

Преимущество связного списка проявляется в том, что операция вставки элемента в середину выполняется очень быстро. Это происходит за счет того, что только ссылки Next (следующий) предыдущего элемента и Previous (предыдущий) следующего элемента должны быть изменены так, чтобы указывать на вставляемый элемент. В классе List<T> при вставке нового элемента все последующие должны быть сдвинуты.

Естественно, у связных списков есть и свои недостатки. Так, например, все элементы таких списков доступны лишь друг за другом. Поэтому для нахождения элемента, находящегося в середине или конце списка, требуется довольно много времени. Связный список не может просто хранить элементы внутри себя. Вместе с каждым из них ему необходимо иметь информацию о следующем и предыдущем элементах. Поэтому LinkedList<T> содержит элементы типа LinkedListNode<T>. С помощью класса LinkedListNode<T> появляется возможность обратиться к предыдущему и последующему

элементам списка. Класс LinkedListNode<T> определяет свойства List, Next, Previous и Value. Свойство List возвращает объект LinkedList<T>, ассоциированный с узлом. Свойства Next и Previous предназначены для итераций по списку и для доступа к следующему и предыдущему элементам. Свойство Value типа T возвращает элемент, соответствующий узлу.

Если в простом списке List<T> каждый элемент представляет объект типа T, то в LinkedList<T> каждый узел помимо T содержит также объект класса LinkedListNode<T>. Этот класс имеет следующие свойства:

- Value –значение узла, представленное типом T.
- Next – ссылка на следующий элемент типа LinkedListNode<T> в списке. Если следующий элемент отсутствует, то имеет значение null.
- Previous – ссылка на предыдущий элемент типа LinkedListNode<T> в списке. Если предыдущий элемент отсутствует, то имеет значение null.

Используя методы класса LinkedList<T>, можно обращаться к различным элементам, как в конце, так и в начале списка:

- AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode): вставляет узел newNode в список после узла node.
- AddAfter(LinkedListNode<T> node, T value): вставляет в список новый узел со значением value после узла node.
- AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode): вставляет в список узел newNode перед узлом node.
- AddBefore(LinkedListNode<T> node, T value): вставляет в список новый узел со значением value перед узлом node.
- AddFirst(LinkedListNode<T> node): вставляет новый узел в начало списка.
- AddFirst(T value): вставляет новый узел со значением value в начало списка.
- AddLast(LinkedListNode<T> node): вставляет новый узел в конец списка.
- AddLast(T value): вставляет новый узел со значением value в конец списка.
- RemoveFirst(): удаляет первый узел из списка. После этого новым первым узлом становится узел, следующий за удаленным.
- RemoveLast(): удаляет последний узел из списка.

Словари – Dictionary<TKey, TValue>

Массив и объекты типа List<T> предоставляют способ отображения на элемент целочисленного индекса. Целочисленный индекс указывается с помощью квадратных скобок (например, [4]), и извлекается элемент по индексу 4, будучи фактически пятым. Но иногда может понадобиться реализация отображения, при котором используется другой, нецелочисленный тип, например string, double или DateTime. В других языках программирования такая организация хранения данных часто называется ассоциативным массивом. Эта функциональная возможность реализуется в классе Dictionary<TKey, TValue> путем внутреннего обслуживания двух массивов, один из которых предназначен для ключей, от которых выполняется отображение на одно из отображаемых значений. Когда в коллекцию Dictionary<TKey, TValue> вставляется пара «ключ–значение», класс автоматически отслеживает принадлежность ключа к значению, позволяя быстро и легко извлекать значение, связанное с указанным ключом. В конструкции класса Dictionary<TKey, TValue> имеется ряд важных особенностей.

В коллекции Dictionary<TKey, TValue> не могут содержаться продублированные ключи. Если для добавления уже имеющегося в массиве ключа вызывается метод Add, выдается исключение. Но для добавления пары «ключ–значение» можно воспользоваться системой записи с использованием квадратных скобок, не опасаясь при этом выдачи исключения, даже если ключ уже был добавлен: любое значение с таким же самым ключом будет переписано новым значением. Протестировать наличие в коллекции Dictionary<TKey, TValue> конкретного ключа можно с помощью метода ContainsKey.

По внутреннему устройству коллекция Dictionary<TKey, TValue> является разряженной структурой данных, работающей наиболее эффективно, когда в ее распоряжении имеется довольно большой объем памяти. По мере вставки элементов размер коллекции Dictionary<TKey, TValue> в памяти может очень быстро увеличиваться.

Когда для последовательного обхода элементов коллекции Dictionary<TKey, TValue> используется инструкция foreach, возвращается элемент KeyValue-Pair<TKey, TValue>. Это структура, содержащая копию элементов ключа и значения, находящихся в коллекции Dictionary<TKey, TValue>, и доступ к каждому элементу можно получить через свойства Key и Value. Эти элементы доступны только для чтения, и их нельзя использовать для

изменения данных в коллекции Dictionary<TKey, TValue>.

Отметим, что есть схожая необобщенная коллекция – Hashtable, имеющая такой же функционал. Однако эта коллекция проигрывает в скорости при работе с однотипными объектами и используется, как и все необобщенные коллекции, для группировки различных объектов.

Относительно производительности рассмотренных здесь коллекций заметим, что добавление нового объекта (Add) быстрее делает List<T>, медленнее – Dictionary<TKey, TValue>. На поиск элемента уходит примерно одинаковое время, однако поиск по ключу существенно быстрее в Dictionary<TKey, TValue>. Удаление объекта медленнее делается в классе List<T>.

Вопросы к разделу

1. В чем отличия коллекций от массивов?
2. Перечислите основные свойства и методы класса System.Array.
3. Приведите примеры описания массивов и коллекций.
4. Как передавать и возвращать массивы и коллекции из методов.
5. Объясните принцип работы цикла foreach.
6. Скажите о плюсах и минусах использования двусвязных списков.
7. Приведите практические примеры эффективного использования рассмотренных коллекций.

Лабораторная работа

Задания на массивы, передачу массива в качестве аргумента методу Main. Время, необходимое на выполнение задания 80 мин.

Упражнение 6.1 Написать программу, которая вычисляет число гласных и согласных букв в файле. Имя файла передавать как аргумент в функцию Main. Содержимое текстового файла заносится в массив символов. Количество гласных и согласных букв определяется проходом по массиву. Предусмотреть метод, входным параметром которого является массив символов. Метод вычисляет количество гласных и согласных букв.

Упражнение 6.2 Написать программу, реализующую умножению двух матриц, заданных в виде двумерного массива. В программе предусмотреть два метода: метод печати матрицы, метод умножения матриц (на вход две матрицы, возвращаемое значение – матрица).

Упражнение 6.3 Написать программу, вычисляющую среднюю температуру за год. Создать двумерный рандомный массив temperature[12,30], в котором будет храниться температура для каждого дня месяца (предполагается, что в каждом месяце 30 дней). Сгенерировать значения температур случайным образом. Для каждого месяца распечатать среднюю температуру. Для этого написать метод, который по массиву temperature [12,30] для каждого месяца вычисляет среднюю температуру в нем, и в качестве результата возвращает массив средних температур. Полученный массив средних температур отсортировать по возрастанию.

Домашнее задание 6.1 Упражнение 6.1 выполнить с помощью коллекции List<T>.

Домашнее задание 6.2 Упражнение 6.2 выполнить с помощью коллекций LinkedList<LinkedList<T>>.

Домашнее задание 6.3 Написать программу для упражнения 6.3, использовав класс Dictionary<TKey, TValue>. В качестве ключей выбрать строки – названия месяцев, а в качестве значений – массив значений температур по дням.

ГЛАВА 7. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

C# – объектно-ориентированный язык. В этой главе изучим терминологию и концепцию ООП.

Классы и объекты

Ключевым словом в ООП является класс. Все языки программирования могут обращаться с общими данными и методами. Эта возможность помогает избежать дублирования. Главная концепция программирования – не писать один и тот же код дважды. Программы без дублирования лучше и понятнее, так как содержат меньше кода. ООП переводит эту концепцию на новый уровень, он позволяет описывать классы (множества объектов), которые делают общими и структуру, и поведение. Классы не ограничиваются описанием конкретных объектов, они также могут описывать и абстрактные вещи.

Объект – это конкретный представитель класса. Его определяет три характеристики: уникальность, поведение и состояние. *Уникальность* – это характеристика, определяющая отличие одного объекта от другого. *Поведение* определяет то, чем объект может быть полезен, что он может делать. Поведение объекта классифицирует его. Объекты разных классов отличаются своим поведением. *Состояние* описывает внутреннюю работу объекта то, что обеспечивает его поведение. Хорошо спроектированный объект оставляет своё состояние недоступным. Нас не интересует, как он это делает, нам важно то, что он умеет это делать.

Сравним структуры и классы. В C# структуры могут иметь методы, но желательно избегать этого. Однако в некоторых структурах необходимы операторы. Операторы – стилизованные методы, но они не добавляют поведение, а обеспечивают более краткий синтаксис.

Структурные типы – нижний уровень программы, это элементы, из которых строятся более сложные элементы. Переменные структурных типов свободно копируются и используются как поля и атрибуты объектов.

Ссылочные типы – верхний уровень программы, они состоят из мелких элементов. Ссылочные типы в основном не могут быть скопированы.

Абстракция – это тактика очистки объекта от всех несущественных деталей, оставляя только существенную минимальную форму. Абстракция – важный принцип программирования. Хорошо спроектированный класс содержит минимальный набор методов, полностью описывающий его поведение.

Инкапсуляция данных

Традиционное процедурное программирование содержит много данных и много процедур. Любая функция имеет доступ к любым данным. Когда программы становятся большими, это создаёт много проблем – при небольших изменениях в коде необходимо следить за всей программой. Другая проблема – это хранение данных отдельно от функций. В ООП эта проблема решается за счет объединения данных и методов, которые работают с этими данными как одно целое.

Данные и функции объединены в одну сущность, эта сущность ограничивает капсулу. Получаем две области: снаружи этой капсулы и внутри неё. Элементы, которые доступны извне, называются public, те, которые доступны только внутри класса – private. C# не ограничивает области видимости, любой элемент может быть как public, так и private.

Две причины использования инкапсуляции – это контроль использования и минимизация воздействий при изменениях. Можно использовать инкапсуляцию данных и определить поведение для того, чтобы с объектом работали по заданным правилам. Вторая причина вытекает из первой, если данные закрыты от внешнего использования, то их изменение не влияет на использование объекта извне.

Большинство данных внутри объектов описывают информацию об индивидуальности объекта. Данные внутри объекта обычно private и доступны только из методов класса.

Иногда необязательно хранить информацию внутри каждого объекта. Т.е. может быть информация, одинаковая для всех объектов данного класса. Для этого используются статичные поля, которые принадлежат не конкретному объекту, а всему классу.

Статичные методы инкапсулируют статичные данные. Статичные методы существуют на уровне класса, в них нельзя использовать оператор this, но можно обращаться к полям класса, если получить объект класса как параметр.

```
class Time
{
	public static void Reset(Time t)
	{
		t.hour = 0;    // ОК
		t.minute = 0;  // ОК
		hour = 0;       // ошибка при компиляции
		minute = 0;      // ошибка при компиляции
	}
	private int hour, minute;
}
```

Теперь вернемся к программе Hello world, рассмотрим её со стороны объектно-ориентированного программирования. Ответим на два вопроса: как при выполнении вызывается класс и почему метод Main статичный?

Если в файле два класса с методом Main, то точка входа определяется при компиляции.

```
// TwoEntries.cs
using System;
class EntranceOne
{
	public static void Main( )
	{
		Console.Write("EntranceOne.Main( )");
	}
}
class EntranceTwo
{
	public static void Main( )
	{
		Console.Write("EntranceTwo.Main( )");
	}
}
// Конец файла
```

```
c:\> csc /main:EntranceOne TwoEntries.cs
c:\> twoentries.exe
EntranceOne.Main( )
c:\> csc /main:EntranceTwo TwoEntries.cs
c:\> twoentries.exe
EntranceTwo.Main( )
c:\>
```

Если в файле нет классов с методом Main, то из него нельзя скомпилировать запускаемый файл, только библиотеку dll.

```
// NoEntrance.cs
using System;
class NoEntrance
{
        public static void NotMain( )
        {
                Console.Write("NoEntrance.NotMain( )");
        }
}
// Конец файла

c:\> csc /target:library NoEntrance.cs
c:\> dir
...
NoEntrance.dll
...
```

Почему метод Main должен быть статичным? Так как для вызова нестатичного метода необходимо создать объект, а при вызове Main программа только начинает работу, и никаких объектов ещё нет.

Для определения простых классов необходимо выполнить следующую последовательность действий: обозначить ключевым словом class начало класса, определить поля как в структурах, определить методы внутри класса, установить модификаторы доступа для всех полей и методов класса. Модификатор public означает, что “доступ неограничен”, private – “доступ ограничен типом, которому принадлежит”. Если пропустить модификатор, то по умолчанию поле или метод будут private.

```
class BankAccount
{
	public void Deposit(decimal amount)
	{
		balance += amount;
	}
	private decimal balance;
}
```

При объявлении объекта класса, объект не создаётся, необходимо использовать оператор new, при этом все поля инициализируются нулями. При использовании объекта без создания будет ошибка при компиляции.

```
Time now = new Time();    // пример объявления объекта класса
```

Ключевое слово this неявно указывает на объект вызвавший метод. Например, в следующем примере полю name нельзя присвоить значение, компилятор будет считать его параметром

```
class BankAccount
{
	public void SetName(string name)
	{
		name = name;
	}
	private string name;
}
```

Здесь необходимо было использовать this.name = name;

В C# можно выделить пять различных видов типов:

class struct interface enum delegate

Любой из них можно использовать в классе. Т.е. в классе могут содержаться другие классы. Вложенные классы должны помечаться модификатором доступа, использование вложенных классов переводит имена из глобальной области видимости и пространства имен.

Public вложенные классы не имеют ограничений по использованию, полное имя класса можно использовать в любом месте программы. Private вложенный класс виден только из класса, содержащего его. Класс без модификатора по умолчанию является private.

Наследование и полиморфизм

Наследование – это связи на уровне классов. Новый класс может наследовать существующий класс. Наследование – это мощная связь, так как наследуемый класс наследует все (не private) элементы базового класса. От базового класса может наследоваться любое количество классов. Изменение базового класса, автоматически изменяет все классы-потомки.

Классы потомки могут быть одновременно и базовыми классами для других классов. Группа классов, связанных наследованием, формирует структуру, называемую иерархией классов. При движении по иерархии вверх переходим к более общим классам. При движении вниз – к более специализированным классам.

Простое наследование – это случай, когда у класса есть только один прямой базовый класс.

Множественное наследование, когда у класса есть несколько прямых базовых классов. Множественное наследование создаёт предпосылки к ошибочному использованию наследования. Поэтому C#, как и большинство современных языков программирования, запрещает множественное наследование. Напомним, что наследование, особенно множественное, позволяет рассматривать один объект с разных точек зрения.

Полиморфизм с литературной точки зрения означает много форм или много обликов. Это концепция, по которой один и тот же метод, определенный в базовом классе, может быть по-разному определен в разных классах потомках. Появляется новая проблема, как работать методу у объекта базового класса? Есть возможность не определять метод в базовом классе, т.е. оставить тело метода пустым. Такие методы называют *операциями.*

В типичной иерархии классов операции объявляются в базовом классе, а определяются различными путями в различных классах потомках. Базовый класс представляет имя метода в иерархии. В случае, когда метод не определен в базовом классе, то нельзя создавать объекты этого класса, так для этого объекта будет не определен этот метод. Такие классы называются *абстрактными.*

Абстрактные классы и интерфейсы похожи, так как не могут иметь объектов. Отличие между ними в том, что абстрактный класс может содержать

определения методов, *интерфейсы* содержат только операции (имена методов). То есть интерфейсы абстрактнее абстрактных классов.

Когда вызываем метод напрямую из объекта, а не из операции базового класса, то метод связывается с вызовом при компиляции – *раннее связывание*. При вызове метода не напрямую через объект, а через операцию базового типа, он вызывается при выполнении программы – *позднее связывание.* Гибкость позднего связывания обеспечивается физической и логической ценой. Позднее связывание выполняется дольше, чем раннее. При позднем связывании классы-потомки могут заменять базовые классы. Операции могут быть вызваны из интерфейса, а классы-потомки обеспечат правильное выполнение.

Вопросы к разделу

1. Объясните концепцию абстракции, и почему она важна для программной инженерии?
2. Назовите два принципа инкапсуляции.
3. Опишите наследование в контексте ООП.
4. Что такое полиморфизм? Как он связан с ранним и поздним связыванием?
5. Опишите разницу между интерфейсами, абстрактными классами и конкретными классами.

Лабораторная работа

Задания на классы. Время, необходимое на выполнение задания 45 мин.

Упражнение 7.1 Создать класс счет в банке с закрытыми полями: номер счета, баланс, тип банковского счета (использовать перечислимый тип из упр. 3.1). Предусмотреть методы для доступа к данным – заполнения и чтения. Создать объект класса, заполнить его поля и вывести информацию об объекте класса на печать.

Упражнение 7.2 Изменить класс счет в банке из упражнения 7.1 таким образом, чтобы номер счета генерировался сам и был уникальным. Для этого надо создать в классе статическую переменную и метод, который увеличивает значение этого переменной.

Упражнение 7.3 Добавить в класс счет в банке два метода: снять со счета и положить на счет. Метод снять со счета проверяет, возможно ли снять запрашиваемую сумму, и в случае положительного результата изменяет баланс.

Домашнее задание 7.1 Реализовать класс для описания здания (уникальный номер здания, высота, этажность, количество квартир, подъездов). Поля сделать закрытыми, предусмотреть методы для заполнения полей и получения значений полей для печати. Добавить методы вычисления высоты этажа, количества квартир в подъезде, количества квартир на этаже и т.д. Предусмотреть возможность, чтобы уникальный номер здания генерировался программно. Для этого в классе предусмотреть статическое поле, которое бы хранило последний использованный номер здания, и предусмотреть метод, который увеличивал бы значение этого поля.

ГЛАВА 8. ИСПОЛЬЗОВАНИЕ ССЫЛОЧНЫХ ТИПОВ ДАННЫХ

C# поддерживает типы данных, такие как int, long и bool, эти типы данных называются значимыми (value type). Переменные такого типа непосредственно содержат значение в самой переменной. Кроме этого, есть также ссылочные типы данных (reference type). Переменные такого типа содержат ссылку на некоторую область памяти с данными. К ссылочным типам данных относятся встроенные в платформу .NET такие типы, как: Array, string, классы.

Для того чтобы объявить переменную ссылочного типа данных, используется следующий код:

coordinate c1;

Данная строка только объявляет переменную c1, которая может содержать ссылку на объект типа coordinate. Строка

c1=new coordinate();

создает новый объект и возвращает ссылку на него, которую сохраняет в переменной c1. После создания объекта c1 можно получить доступ к полям этого объекта, используя оператор .(точку), например:

c1.x=10; c1.y=5;

Чтобы освободить ссылочную переменную, ей явно надо присвоить значение null.

При попытке доступа к полю не проинициализированной ссылочной переменной может возникнуть ошибка на этапе компиляции или будет сгенерировано исключение во время выполнения программы. Например, если сначала проинициализировали переменную, потом присвоили ей null, а потом попытались получить доступ к полю этого объекта, то в этом случае возникнет исключительная ситуация типа NullReferenceException во время выполнения программы.

Операторы проверки равенства (==) и (!=) для ссылочных переменных будут проверять, ссылаются ли две переменные на один и тот же объект – на одну и ту же область памяти. Для типа данных string оператор (==) можно использовать для проверки равенства значений (строк) в двух строковых переменных. Для ссылочных переменных нельзя использовать операторы отношения (>,<.>=,>=).

Две ссылочных переменных могут ссылаться на одну и ту же область памяти. Соответственно изменение значения объекта через одну ссылку изменит значение объекта через другую ссылку.

Ссылочные переменные можно передавать в качестве параметров в методы. В случае передачи параметра по значению будет создана копия ссылки на тот же объект внутри метода. При передаче по ссылке (ключевое слово ref) используется одна ссылка внутри и снаружи метода. При передаче out параметра используется одна ссылка снаружи и внутри метода, но в отличие от передачи по ссылке в данном случае ссылочная переменная должна быть инициализирована внутри метода. При передаче ссылочной переменной в качестве параметра в метод любым из трех способов, изменение значения объекта внутри метода приведет к изменению значения объекта вне метода.

Рассмотрим несколько определенных в .NET ссылочных типов данных.

1. Класс **Exception**. Объекты этого класса и только они используются в операторе генерирования исключительной ситуации throw и операторе обработки исключительной ситуации catch. Кроме того, в C# есть множество классов, наследников от класса Exception, для разного рода исключений.

2. Класс **String** – последовательность неизменяемых символов в кодировке Unicode. Все методы, которые работают со строкой и, казалось бы, изменяют ее, на самом деле создают новый объект класса String и возвращают его в качестве результата. Некоторые методы, определенные в классе:

   string str=”alphabet”;
   char c=str[5]; //возвращает шестой символ строки;
   str[3]=’z’;//приведет к ошибке, т.к. свойство взятие индекса только для чтения;
   int n=str.Length; //возвращает длину строки;
   string str1=str.Insert(2,”ABC”); //добавляет во вторую позицию строку ABC и результат возвращает в str1. Значение строки str1 будет равно “aABCalphabet”, значение строки str останется без изменения (str=”alphabet”);
   string str2=String.Copy(str); //создается новая строка str2 и в нее копируется значение строки str;

string s=String.Concat(“a”,”b”,”c”); // операция конкатенации эквивалента операции:
string s=”a”+”b”+”c”;
string s=str.Trim(); //создает новый объект String на основе строки str, из которой удалены специальные символы и пробелы;
string s=str.ToUpper(); // переводит строку к верхнему регистру
string s=str.ToLower(); //переводит строку к нижнему регистру
Для сравнения значения двух строковых переменных между собой можно использовать оператор == и !=. Также можно использовать метод **Equals**
s1.Equals(s2); //вернет true, если строки совпадают;
String.Equals(s1,s2); //вернет true, если строки совпадают.
Метод **Compare** сравнивает две строки в соответствии с их лексикографическим порядком. Метод возвращает отрицательное значение, если первая строка меньше второй; число ноль – если две строки совпадают; положительное число – если первая строка больше второй.
String.Compare(s1,s2);
Метод Compare можно также использовать с тремя параметрами. Третий параметр типа bool, если он равен true, то метод не чувствителен к регистру.

3. Класс **StringBuilder** – класс для хранения строк. В отличии от класса String, любой метод этого класса изменяет саму строку, не создавая новый объект класса.

Все классы в C# явным или неявным образом наследуются от класса System.Object (часто вместо полного имени класса используется псевдоним object). У класса Object есть общие методы, которые наследуются всеми классами:

- **ToString** – возвращает строковое представление объекта. Реализация по умолчанию вернет имя типа класса. У каждого класса наследника этот метод можно переопределить.

- **Equals** – определяет, указывают ли ссылки на один и тот же объект. Метод можно перегружать, например, для проверки на равенство значений полей объектов.

- **Finalize** – метод вызывается системой во время выполнения, когда объект становится недоступным.

- **GetType** – позволяет во время выполнения получить информацию о типе.

Reflection (рефлексия)

Процесс получения информации о типе во время выполнения называется рефлексией. В пространстве имен System.Reflection содержатся классы и интерфейсы, которые позволяют получить информацию о типах, методах и полях. Класс System.Type содержит методы для получения информации об объявлении типа: о конструкторах, полях, методах, событиях и свойствах класса. Для получения информации о типе можно использовать оператор typeof:

```
using System;
using System.Reflection;
Type t = typeof(byte);
Console.WriteLine("Type: {0}", t);
```

Чтобы получить более подробную информацию о классе можно использовать метод GetMethods у объекта класса Type:

```
using System;
using System.Reflection;
Type t = typeof(string); // Get type information
MethodInfo[ ] mi = t.GetMethods( );
foreach (MethodInfo m in mi) {
        Console.WriteLine("Method: {0}", m);
}
```

Оператор typeof можно применять только тогда, когда объект существует на этапе компиляции. Если необходимо информацию о типе получить во время выполнения программы, следует использовать метод GetType класса Object.

Пространства имен

В платформе .NET Framework реализована большая коллекция классов, которые предоставляют интерфейс к общеязыковой среде исполнения, к операционной системе и сети. Все классы сгруппированы в пространства имен.

Пространство имен System.IO содержит классы для работы с операциями ввода/вывода, для работы с файловой системой.

Классы File и Directory предоставляют методы для создания, удаления и управления директориями и файлами в файловой системе.

Классы StreamReader и StreamWriter позволяют программе получать доступ к содержимому файлов как к потоку битов или символов.

Класс FileStream содержит операции чтения и записи в файл, открытия и закрытия файлов в файловой системе, а также методы для изменения других дескрипторов операционной системы для обработки файлов, позволяет задать синхронное или асинхронное выполнение операций чтения и записи.

Классы BinaryReader и BinaryWriter позволяет читать и записывать простые типы данных как двоичные значения в заданной кодировке.

Пространство имен System.Data содержит классы для работы с данными из различных источников. Класс DataSet предназначен для работы с данными из разных источников. Класс DataSet состоит из таблиц (класс DataTable). Пространство имен System.Data.SqlClient содержит классы, которые предоставляют прямой доступ к базе данных SQL Server. Для доступа к другим реляционным БД используются классы из System.Data.OleDb.

Пространство имен System содержит классы для работы со значимыми типами данных и ссылочными типами данных, событиями, делегатами, интерфейсами, атрибутами. Предоставляет методы для преобразования типов данных, управления программой.

Пространство имен System.Net представляет простой программный интерфейс для работы с сетью с помощью различных протоколов.

Пространство имен System.Windows.Forms – классы для создания графического интерфейса в Windows приложениях.

Приведение типов данных

Для значимых типов данных различают явное и неявное преобразование. Неявное преобразование происходит тогда, когда переменную одного типа данных пытаются присвоить в переменную другого типа. В C# неявное преобразование разрешается обычно тогда, когда значение может быть преобразовано к другому типу без потери значимости. Например, от int к long:

```
int a=4;
long b;
```

```
b=a;
```

При явном преобразовании типов данных используется оператор cast. Если в процессе преобразования возникли проблемы, генерируется исключительная ситуация:

```
try{
        a=checked((int)b);
}
catch (Exception e){
        Console.WriteLint(“Problem in cast”);
}
```

Все преобразования между различными базовыми типами данных осуществляются классом System.Convert.

Можно преобразовать объект класса потомка к объекту родительского класса и наоборот. Преобразование к родительскому классу возможно всегда:

```
Animal a;
Bird b=new Bird(); //класс Animal – класс предок для класса Bird
a=b;
```

Можно использовать оператор приведения типов:

```
a=(Animal) b;
```

Преобразование к классу потомка необходимо осуществлять явно:

```
Bird b=(Bird) a;
```

В данном случае в процессе работы программы будет выполнена проверка, действительно ли объект *a* типа Bird. Если это не так, будет сгенерирована исключительная ситуация типа InvalidCastException. Код, осуществляющий явное приведение типов данных, лучше размещать в блоке try-catch.

В C# есть возможность проверить, можно ли преобразовать объект одного типа данных к другому типу данных. Для этого используется оператор is, который возвращает true или false в зависимости от того, возможно ли приведение типов:

```
if (a is Bird)
        b=(Bird)a; //Приведение типов возможно
else
        Console.WriteLine(“a не является птицей”);
```

Оператор as преобразует объект одного типа данных к другому, если преобразование возможно, и возвращает null, если преобразование не возможно:

```
Bird b=a as Bird;
if (b==null)
        Console.WriteLine(“a не является птицей”);
```

В .NET все ссылочные типы данных наследуют от класса object, поэтому объект любого ссылочного типа данных можно привести к типу данных object.

Преобразования типов можно выполнять, работая с интерфейсами:

```
IHashCodeProvider hcp;
hcp = (IHashCodeProvider) x;
```

При помощи оператора is можно узнать определяет ли класс интерфейс

```
if (x is IHashCodeProvider) ...
```

Также можно использовать оператор as, вместо оператора преобразования

```
IHashCodeProvider hcp;
hcp = x as IHashCodeProvider;
```

C# может переводить структурные типы в ссылочные и наоборот (boxing и unboxing).

```
static void Show(object o)
{
        Console.WriteLine(o.ToString( ));
}
Show(42);
```

В случае явного преобразования типов:

```
object o = (object) 42; // Box
Console.WriteLine(o.ToString( ));
```

Вопросы к разделу

1. Как распределяется память для переменных ссылочного типа?
2. Какое значение присваивают ссылочной переменной, чтобы показать, что она не указывает на объект? Что произойдет, если обратиться к ней как к объекту?
3. Какой класс является базовым для всех классов C#?
4. Объясните разницу между операцией преобразования типа (cast) и оператором as.

Лабораторная работа

Задания на использование переменных ссылочного типа, передачу их в качестве параметров методам, преобразование типов данных. Время, необходимое на выполнение задания 75 мин.

Упражнение 8.1 В класс банковский счет, созданный в упражнениях 7.1-7.3 добавить метод, который переводит деньги с одного счета на другой. У метода два параметра: ссылка на объект класса банковский счет откуда снимаются деньги, второй параметр – сумма.

Упражнение 8.2 Реализовать метод, который в качестве входного параметра принимает строку string, возвращает строку типа string, буквы в которой идут в обратном порядке. Протестировать метод.

Упражнение 8.3 Написать программу, которая спрашивает у пользователя имя файла. Если такого файла не существует, то программа выдает пользователю сообщение и заканчивает работу, иначе в выходной файл записывается содержимое исходного файла, но заглавными буквами.

Упражнение 8.4 Реализовать метод, который проверяет реализует ли входной параметр метода интерфейс System.IFormattable. Использовать оператор is и as. (Интерфейс IFormattable обеспечивает функциональные возможности форматирования значения объекта в строковое представление.)

Домашнее задание 8.1 Работа со строками. Дан текстовый файл, содержащий ФИО и e-mail адрес. Разделителем между ФИО и адресом электронной почты является символ #:

Иванов Иван Иванович # iviviv@mail.ru

Петров Петр Петрович # petr@mail.ru

Сформировать новый файл, содержащий список адресов электронной почты. Предусмотреть метод, выделяющий из строки адрес почты. Методу в качестве параметра передается символьная строка s, e-mail возвращается в той же строке s:

public void SearchMail (ref string s).

Домашнее задание 8.2 Список песен. В методе Main создать список из четырех песен. В цикле вывести информацию о каждой песне. Сравнить между собой первую и вторую песню в списке.

Песня представляет собой класс с методами для заполнения каждого из полей, методом вывода данных о песне на печать, методом, который сравнивает между собой два объекта:

```
class Song{
        string name; //название песни
```

```
        string author; //автор песни
        Song prev; //связь с предыдущей песней в списке
        //метод для заполнения поля name
        //метод для заполнения поля author
        //метод для заполнения поля prev
        //метод для печати названия песни и ее исполнителя
        public string Title(){… /*возвращ название+исполнитель*/ …}
        //метод, который сравнивает между собой два объекта-песни:
        public bool override Equals(object d){…}
}
```

ГЛАВА 9. СОЗДАНИЕ И УДАЛЕНИЕ ОБЪЕКТОВ

В этой главе изучим, что происходит при создании объекта, как конструкторы инициализируют объекты и как используются деструкторы для уничтожения объектов. Узнаем, что происходит с объектом после уничтожения и как работает сборщик мусора.

Использование конструкторов

Конструкторы – это специальные методы, которые используются для инициализации объектов при создании. Если конструктор не описан в классе, то запускается конструктор по умолчанию.

Процесс создания объекта проходит в два этапа, но записывается всё в одно выражение. Первый шаг – это выделение памяти, за это отвечает оператор **new**. Второй шаг – инициализация объекта при помощи конструктора, в этот момент полям класса присваиваются начальные значения либо те, которые указаны в конструкторе, либо значения по умолчанию (для чисел – это ноль, для ссылочных типов данных – это null).

При создании объекта создаётся конструктор по умолчанию, если не создан свой. Иногда конструктор по умолчанию не подходит для инициализации объекта, тогда можно определить свой конструктор. Есть несколько причин, когда не подходит конструктор по умолчанию: не подходит доступ public, инициализация нулями неправильна, невидимый код тяжело понимать.

Все поля, которые не определены в конструкторе, инициализируются нулями. Если конструктор отработал, то с объектом можно работать, если произошла ошибка в конструкторе, то объект не создался.

Конструкторы, как и любые другие методы, можно перегрузить.

```
class Overload
{
        public Overload( ) { this.data = -1; }
```

```
        public Overload(int x) { this.data = x; }
        private int data;
}
class Use
{
        static void Main( )
        {
                Overload o1 = new Overload( );
                Overload o2 = new Overload(42);
        }
}
```

Если определен свой конструктор, то компилятор не создаст конструктор по умолчанию, и его придется прописать самостоятельно.

При определении конструкторов можно использовать конструкцию, называемую списком инициализации. Определение одного конструктора при помощи вызова другого перегруженного конструктора.

```
class Date
{
        public Date() : this(1970, 1, 1) {   }
        public Date(int year, int month, int day) {   }
}
```

Ограничения – нельзя вызывать конструктор со списком инициализации из другого метода

```
class Point
{
        public Point(int x, int y) { ... }
        public void Init( ) : this(0, 0) { }   // Ошибка при компиляции
}
```

нельзя вызывать самого себя

```
class Point
{               // Ошибка при компиляции
        public Point(int x, int y) : this(x, y) { }
}
```

нельзя использовать ключевое слово this в списке инициализации.

```
class Point
{    // Ошибка при компиляции
        public Point( ) : this(X(this), Y(this)) { }
        public Point(int x, int y) { ... }
        private static int X(Point p) { ... }
        private static int Y(Point p) { ... }
}
```

При определении конструкторов необходимо определить константы и поля только для чтения. Поля, которые не могут быть переприсвоены, называются полями только для чтения. Есть три варианта инициализации полей только для чтения:

- нулём неявно;
- присвоением в конструкторе, что разрешено;
- присвоением при объявлении поля в классе.

```
class SourceFile
{
        public SourceFile( ) {   }
        private readonly ArrayList lines = new ArrayList( );
}
```

Синтаксис конструкторов одинаков и для структур. Отличие в том, что для структур обязательно необходимо инициализировать все поля.

Закрытый (private) конструктор – это особый конструктор экземпляров. Обычно он используется в классах, содержащих только статические элементы.

Если в классе один или несколько закрытых конструкторов и ни одного открытого конструктора, то прочие классы (за исключением вложенных классов) не смогут создавать экземпляры этого класса. Объявление пустого конструктора запрещает автоматическое создание конструктора по умолчанию. Стоит заметить, что если не использовать с конструктором модификатор доступа, то по умолчанию он все равно будет закрытым. Однако обычно используется модификатор private, чтобы ясно обозначить невозможность создания экземпляров данного класса.

Закрытые конструкторы используются, чтобы не допустить создание экземпляров класса при отсутствии полей или методов экземпляра, например для класса Math, или когда осуществляется вызов метода для получения экземпляра класса.

```
class Math
{
        public static double Cos(double x) { ... }
        public static double Sin(double x) { ... }
        private Math( ) { ... }
}
class LessCumbersome
{
        static void Main( )
        {
                double answer = Math.Cos(42.0);
        }
}
```

Достоинства статичных методов – простота и быстрота, не надо создавать объект. Статичный конструктор вызывается при загрузке класса в память.

```
class Example
{
   static Example( ) { ... }
}
```

Уничтожение объектов

В приложении необходимо знать, что случается, когда объект выходит из области видимости или уничтожается. Процесс уничтожения объекта в C# состоит из двух шагов: деинициализация объекта и возвращение памяти в управляемую кучу.

Отметим различия во времени жизни переменных структурных типов и объектов. Переменные структурного типа обычно имеют короткое время жизни, ограниченное блоком в котором они объявлены. Также их отличает детерминированное создание и уничтожение. Для объектов время жизни дольше и время уничтожения недетерминированное.

В C# нельзя вручную удалить объект, так как эта возможность вызывала много различных ошибок в других языках: забывание удаления объектов, попытка уничтожения объекта дважды, удаление активного объекта.

Сборщик мусора удаляет объекты за программиста. Сборщик гарантирует, что объекты будут удалены, и, причем, только один раз. Удаляются только недостижимые объекты.

```
class Example
{
    void Method(int limit)
    {
        for(int i = 0; i < limit; i++){
            Example eg = new Example();
            ….
        }
        // eg за пределами блока, существует ли он ещё?
    }
}
```

Как было указано выше, удаление объекта – двухшаговый процесс. На

первом шаге память очищается, на втором возвращается в кучу. Второй шаг одинаков для всех классов, а вот первый индивидуален для каждого класса. Для описания первого шага определяется деструктор или метод Finalize. Можно описать деструктор для определения очистки объекта. В C# нельзя вручную вызвать деструктор или метод Finalize.

В C# программист не знает, когда будет уничтожен объект, только известно, что это произойдёт после того, как он станет недостижимым. Порядок и время вызова деструкторов неопределенны. Желательно избегать деструкторов, так как он использует ресурсы сборщика мусора. Если в классе нет неуправляемых частей, то сборщик мусора сам уничтожит объекты и деструктор не нужен.

Память от удаленного объекта освободится, когда сборщик мусора уничтожит объект, однако, кроме памяти объект может содержать другие ресурсы, которые лучше освободить быстрее: подключение к БД, открытые файловые потоки. Для этого используется метод Dispose, для его использования необходимо, чтобы класс определял интерфейс IDisposable, и описав метод Dispose, убедиться что метод не вызывается дважды.

Оператор using определяет область видимости объект, в конце этого блока для объекта вызывается метод Dispose.

```
Resource r1 = new Resource( );
try {
        r1.Test( );
}
finally {
        if (r1 != null) ((IDisposable)r1).Dispose( );
}
```

Рассмотрим более подробно как происходит работа с памятью в языке C#.

При создании объекта память размещается в управляемой куче, и переменная хранит только ссылку на расположение объекта. Для типов в управляемой куче требуются служебные данные и при их размещении, и при их удалении функциональной возможностью автоматического управления памятью среды CLR, также известной как сборка мусора. Сборка мусора в

высокой степени оптимизирована, и в большинстве сценариев она не создает проблем с производительностью.

Автоматическое управление памятью является одной из служб, которые предоставляет среда CLR во время управляемого выполнения. Сборщик мусора среды CLR управляет освобождением и выделением памяти для приложения. Для разработчиков это означает, что при разработке управляемого приложения не нужно писать код для управления памятью. Автоматическое управление памятью позволяет устранить распространенные проблемы, такие как не освобожденный по забывчивости объект, вызывающий утечку памяти, или попытки доступа к памяти для уже удаленного объекта.

Выделение памяти. При инициализации нового процесса среда выполнения резервирует для него непрерывную область адресного пространства. Это зарезервированное адресное пространство называется управляемой кучей. Эта управляемая куча содержит указатель адреса, с которого будет выделена память для следующего объекта в куче. Изначально этот указатель устанавливается в базовый адрес управляемой кучи. Все ссылочные типы размещаются в управляемой куче. Когда приложение создает первый ссылочный тип, память для него выделяется, начиная с базового адреса управляемой кучи. При создании приложением следующего объекта сборщик мусора выделяет для него память в адресном пространстве, непосредственно следующем за первым объектом. Пока имеется доступное адресное пространство, сборщик мусора продолжает выделять пространство для новых объектов по этой схеме.

Выделение памяти из управляемой кучи происходит быстрее, чем неуправляемое выделение памяти. Поскольку среда выполнения выделяет память для объекта путем добавления значения к указателю, это осуществляется почти так же быстро, как выделение памяти из стека. Кроме того, поскольку выделяемые последовательно новые объекты и располагаются последовательно в управляемой куче, приложение может получать доступ к объектам очень быстро.

Освобождение памяти. Механизм оптимизации сборщика мусора определяет наилучшее время для выполнения сбора, основываясь на произведенных выделениях памяти. Когда сборщик мусора выполняет очистку, он освобождает память, выделенную для объектов, которые больше не используются приложением. Он определяет, какие объекты больше не используются, основываясь на корнях приложения. Каждое приложение имеет набор корней. Каждый корень либо ссылается на объект, находящийся в

управляемой куче, либо имеет значение NULL. Корни приложения содержат указатели глобальных и статических объектов, локальные переменные и параметры ссылочных объектов в стеке потока, а также регистры процессора. Сборщик мусора имеет доступ к списку активных корней, которые поддерживаются JIT-компилятором и средой выполнения. С помощью этого списка он проверяет корни приложения и в процессе проверки создает граф, содержащий все объекты, к которым можно получить доступ из этих корней.

Объекты, не входящие в этот граф, являются недостижимыми из данных корней приложения. Сборщик мусора считает недостижимые объекты мусором и будет освобождать выделенную для них память. В процессе очистки сборщик мусора проверяет управляемую кучу, отыскивая блоки адресного пространства, занятые недостижимыми объектами. При обнаружении недостижимого объекта он использует функцию копирования памяти для уплотнения достижимых объектов в памяти, освобождая блоки адресного пространства, выделенные под недостижимые объекты. После уплотнения памяти, занимаемой достижимыми объектами, сборщик мусора вносит необходимые поправки в указатель, чтобы корни приложения указывали на новые расположения объектов. Он также устанавливает указатель управляемой кучи в положение после последнего достижимого объекта. Память уплотняется, только если при очистке обнаруживается значительное число недостижимых объектов. Если после сборки мусора все объекты в управляемой куче остаются на месте, то уплотнение памяти не требуется.

Для повышения производительности среда выполнения выделяет память для больших объектов в отдельной куче. Сборщик мусора автоматически освобождает память, выделенную для больших объектов. Однако для устранения перемещений в памяти больших объектов эта память не сжимается.

Поколения и производительность. С целью оптимизации производительности сборщика мусора управляемая куча подразделяется на три поколения: 0, 1 и 2. Сборщик мусора среды выполнения хранит новые объекты в поколении 0. Для созданных ранее объектов, оставшихся после сборок мусора, их уровень повышается, и они переводятся в поколения 1 и 2. Поскольку быстрее сжать часть управляемой кучи, чем всю кучу, эта схема позволяет сборщику мусора освобождать память в определенном поколении, а не освобождать память для всей кучи каждый раз при сборке мусора.

В действительности сборщик мусора выполняет очистку при заполнении поколения 0. Если приложение пытается создать новый объект, когда поколение 0 заполнено, сборщик мусора обнаруживает, что в поколении 0 не

осталось свободного адресного пространства для объекта. Сборщик мусора выполняет сборку, пытаясь освободить для этого объекта адресное пространство в поколении 0. Сборщик мусора начинает проверять объекты в поколении 0, а не все объекты в управляемой куче. Это наиболее эффективный подход, поскольку, как правило, новые объекты имеют меньшее время жизни, и можно ожидать, что многие из объектов в поколении 0 к моменту проведения сборки мусора уже не используются приложением. Кроме того, сборка мусора только в поколении 0 зачастую освобождает достаточно памяти для того, чтобы приложение могло продолжить создавать новые объекты.

После того, как сборщик мусора выполнит освобождение для поколения 0, он уплотняет память для достижимых объектов. Затем сборщик мусора повышает уровень этих объектов и считает эту часть управляемой кучи поколением 1. Поскольку объекты, оставшиеся после сборок мусора, как правило, имеют большее время жизни, имеет смысл повысить их уровень до более старшего поколения. В результате сборщику мусора не обязательно выполнять повторную проверку объектов поколений 1 и 2 при каждой сборке мусора в поколении 0.

После того, как сборщик мусора выполнил свою первую очистку для поколения 0 и повысил уровень достижимых объектов до поколения 1, он считает оставшуюся часть управляемой кучи поколением 0. Он продолжает выделять память для новых объектов в поколении 0 до тех пор, пока оно не заполнится и не появится необходимость в следующей сборке мусора. В этот момент оптимизатор сборщика мусора определяет, есть ли необходимость проверки объектов в более старых поколениях. Например, если при очистке поколения 0 не освободится достаточно памяти для того, чтобы приложение смогло успешно завершить свою попытку создания нового объекта, сборщик мусора может выполнить очистку поколения 1, а затем - поколения 0. Если и при этом не освобождается достаточно памяти, он может выполнить очистку поколений 2, 1 и 0. После каждой сборки мусора сборщик уплотняет объекты в поколении 0 и продвигает их в поколение 1. Объекты поколения 1, оставшиеся после сборки мусора, продвигаются в поколение 2. Поскольку сборщик мусора поддерживает только три поколения, объекты поколения 2, оставшиеся после сборки мусора, остаются в этом поколении до тех пор, пока при очередной очистке они не будут определены, как недостижимые.

Освобождение памяти для неуправляемых ресурсов. Для большинства объектов, созданных приложением, сборщик мусора автоматически выполнит

необходимые задачи по управлению памятью. Однако, для неуправляемых ресурсов требуется явная очистка.

Основным типом неуправляемых ресурсов являются объекты, образующие упаковку для ресурсов операционной системы, такие как дескриптор файлов, дескриптор окна или сетевое подключение. Хотя сборщик мусора может отслеживать время жизни управляемого объекта, инкапсулирующего неуправляемые ресурсы, он не имеет определенных сведений о том, как освобождать эти ресурсы. При создании объекта, инкапсулирующего неуправляемый ресурс, рекомендуется предоставлять необходимый код для очистки неуправляемого ресурса в общем методе Dispose. Предоставление метода Dispose дает возможность пользователям объекта явно освобождать память при завершении работы с объектом.

Вопросы к разделу

1. Как происходит создание и удаление объектов?
2. В каких случаях используются закрытые конструкторы?
3. Возможна ли перегрузка конструкторов и деструкторов? Приведите примеры.
4. Какой метод вызывает сборщик мусора даже, если память ещё не переполнена?
5. В чем смысл использования оператора *using*?

Лабораторная работа

Задания на создание конструкторов, деструкторов, обращение к сборщику мусора. Время, необходимое на выполнение задания 75 мин.

Упражнение 9.1 В классе банковский счет, созданном в предыдущих упражнениях, удалить методы заполнения полей. Вместо этих методов создать конструкторы. Переопределить конструктор по умолчанию, создать конструктор для заполнения поля баланс, конструктор для заполнения поля тип банковского счета, конструктор для заполнения баланса и типа банковского счета. Каждый конструктор должен вызывать метод, генерирующий номер счета.

Упражнение 9.2 Создать новый класс BankTransaction, который будет хранить информацию о всех банковских операциях. При изменении баланса

счета создается новый объект класса BankTransaction, который содержит текущую дату и время, добавленную или снятую со счета сумму. Поля класса должны быть только для чтения (readonly). Конструктору класса передается один параметр – сумма.

В классе банковский счет добавить закрытое поле типа System.Collections.Queue, которое будет хранить объекты класса BankTransaction для данного банковского счета; изменить методы снятия со счета и добавления на счет так, чтобы в них создавался объект класса BankTransaction и каждый объект добавлялся в переменную типа System.Collections.Queue.

Упражнение 9.3 В классе банковский счет создать метод Dispose, который данные о проводках из очереди запишет в файл. Не забудьте внутри метода Dispose вызвать метод GC.SuppressFinalize, который сообщает системе, что она не должна вызывать метод завершения для указанного объекта.

Домашнее задание 9.1 В класс Song (из домашнего задания 8.2) добавить следующие конструкторы:

1) параметры конструктора – название и автор песни, указатель на предыдущую песню инициализировать null.

2) параметры конструктора – название, автор песни, предыдущая песня.

В методе Main создать объект mySong. Возникнет ли ошибка при инициализации объекта mySong следующим образом: Song mySong = new Song(); ?

Исправьте ошибку, создав необходимый конструктор.

ГЛАВА 10. НАСЛЕДОВАНИЕ В C#

Наследование – это свойство объектно-ориентированной системы наследовать данные и функциональность базового класса. Можно в класс-потомок к методам и полям родительского класса добавить необходимые поля и методы. Класс-потомок может замещать методы родительского класса. Надо помнить, при изменении родительского класса, класс-потомок может оказаться не рабочим.

Наследование от класса называется расширением базового класса. Если класс A наследует от класса B, то класс A называется потомком, а B – предком. Синтаксически это пишется следующим образом:

```
class A:B {…}
```

Класс-потомок наследует все элементы базового класса, кроме конструктора и деструктора. Все public элементы базового класса остаются неявно public в потомке, private элементы, хоть и наследуются, но доступны только для объектов базового класса. Класс-потомок не может быть более доступным, чем базовый класс.

```
class Example
{
        private class NestedBase { }
        public class NestedDerived: NestedBase { } // Ошибка
}
```

Класс-потомок имеет доступ ко всем protected полям и методам родительского класса, класс не являющийся потомком доступа к protected членам не имеет.

```
 class Token
{
        protected string name;
}
class CommentToken:Token
{
        public string Name()
        {
                return name;   // Доступ разрешен
        }
}
```

```
class CommentToken: Token
{
	void Fails(Token t)
	{
		Console.WriteLine(t.name); // Ошибка при компиляции
	}
}
```

Для вызова конструктора базового класса из класса-потомка используется ключевое слово base. Вызов конструктора базового класса пишется после заголовка конструктора класса-потомка через двоеточие:

```
A(список параметров): base(параметры для передачи в конструктор базового класса)
{
	//тело конструктора у класса потомка
}
```

Ключевое слово base обращается к базовому классу. Для конструктора по умолчанию вызов базового конструктора производится по умолчанию, то есть его можно не прописывать. Для конструкторов не по умолчанию необходимо явно вызывать конструктор базового класса.

```
class Token
{
	protected Token(string name) { ... }
}
class CommentToken: Token
{
	public CommentToken(string name) { ... } //ошибка при компиляции
}
```

В примере выше получим ошибку при компиляции, так как будет вызываться конструктор по умолчанию, но он отсутствует в классе Token. Для правильной работы необходимо записать конструктор в следующем виде:

```
public CommentToken(string name) : base(name) { ... }
```

Если конструктор базового класса private, то нельзя создавать конструктор класса потомка.

```
class NonDerivable
{
	private NonDerivable(){  }
```

```
}
class Impossible: NonDerivable
{
        public Impossible() {   }   // Ошибка при компиляции
}
```

В классе-потомке можно переопределять методы базового класса, если они для этого предназначены. Виртуальные методы можно полиморфно переопределять в классах-потомках. В C# по тому, содержит ли базовый класс виртуальные методы, можно определить, был ли этот класс разработан для наследования. Не виртуальный метод имеет только одно определение, одинаковое для всех потомков. Для объявления виртуального метода используется ключевое слово virtual. Виртуальный метод должен содержать тело в базовом классе. Нельзя объявлять виртуальными статичные и private методы. Статичные методы не могут быть виртуальными, так как полиморфизм – это свойство, относящееся к объектам, а не к классам.

Перегруженные методы могут создаваться только для виртуальных методов. Для задания перегруженных методов используется ключевое слово override. В базовом классе метод объявляется виртуальным virtual, в базовом классе есть тело у этого метода, в классе-потомке этот метод определяется перегруженным override и содержит свое тело – таким образом задается перегруженный метод.

```
class Token
{
        public virtual string Name() {…}
}
class CommentToken : Token
{
        public override string Name() {…}
}
```

Можно перегружать только абсолютно идентичные методы. Должны совпадать: имя метода, тип возвращаемого значения, список параметров, уровень доступа. Перегруженный метод должен быть virtual или override. Нельзя объявлять перегруженный метод одновременно виртуальным, т.е. override virtual – нельзя. Нельзя, чтобы перегруженные методы были статичными или private.

Можно скрыть наследуемый метод в иерархии классов, заменив его новым идентичным методом при помощи ключевого слова new. Метод

родительского класса не будет наследоваться потомком, и заменится на новый идентичный метод.

```
class Token
{
    public int LineNumber() {…}
}
class CommentToken : Token
{
    new public int LineNumber() {…}
}
```

Ключевое слово new прячет как виртуальные, так и невиртуальные методы, разрешает проблему совпадения имен, скрывает методы с одинаковой сигнатурой.

В C# можно объявить класс ненаследуемым, при помощи ключевого слова sealed.

```
public sealed class String {..}
public class MyStr:String; //ошибка – от класса String наследовать нельзя
```

Использование интерфейсов

Интерфейс – синтаксический и семантический шаблон, которого все классы-наследники должны придерживаться. Интерфейс говорит, что он умеет делать, классы определяют, как они это делают. Интерфейс представляет собой класс без какого-либо кода. Все интерфейсы по умолчанию public, модификатор доступа у интерфейсов не используется. У методов также не используется модификатор доступа, по умолчанию методы public. У методов в интерфейсе не должно быть тела, только заголовки методов.

```
interface IToken
{
    public int LineNumber( ){ … }; // Ошибка при компиляции: 1. Модификатор доступа у метода public //2. Есть тело метода
}
```

C# позволяет наследовать от одного класса и множества интерфейсов. Интерфейс может наследовать от многих интерфейсов.

```
interface IToken { ... }
interface IVisitable { ... }
```

```
interface IVisitableToken: IVisitable, IToken { ... }
class Token: IVisitableToken { ... }
```

Класс может быть более доступным, чем интерфейс

```
class Example
{
        private interface INested { }
        public class Nested: INested { } // Разрешено
}
```

Класс должен определить все методы всех интерфейсов, от которых он наследует как напрямую, так и косвенно. Метод интерфейса, определяемый классом должен быть идентичен, то есть должны совпадать параметр доступа, имя, возвращаемое значение и список параметров. Интерфейсные методы, реализуемые в классе, могут быть объявлены как virtual. В этом случае классы наследники могут перегружать эти методы в дальнейшем.

Другой способ реализации интерфейсных методов – явная реализация. При явной реализации необходимо указать полное имя метода: Имя_интерфейса.имя_метода. При явном определении метод не может быть виртуальным, должен отсутствовать модификатор доступа. При вызове метода к нему нет прямого доступа, только через интерфейс.

```
class Token: IToken, IVisitable
{
        string IToken.Name( )
        {
                ...
        }
        private void Example( )
        {
                Name( );                // Ошибка при компиляции
                ((IToken)this).Name( ); // Правильно
        }
        ...
}
```

Явная реализация позволяет:

- исключить определение интерфейса из класса, если он не интересен пользователям класса.
- обеспечивать классу несколько определений различных методов интерфейсов одинаковой сигнатуры.

```
interface IArtist
{
        void Draw();
}
interface ICowboy
{
        void Draw();
}
class ArtisticCowboy: IArtist, ICowboy
{
        void IArtist.Draw() {…}
        void ICowbowt.Draw() {…}
}
```

Использование абстрактных классов

Абстрактные классы используются для частичной реализации классов, которые могут быть полностью реализованы в конкретных классах-потомках. Абстрактный класс объявляется с помощью ключевого слова abstract. Правила создания абстрактного класса совпадают с правилами создания обычных классов. Однако, в абстрактных классах можно объявлять абстрактные методы. Нельзя создавать объекты абстрактного класса. Абстрактный класс может являться наследником неабстрактного класса. Все методы интерфейса, определяемого абстрактным классом, должны быть определены в абстрактном классе.

И абстрактные классы, и интерфейсы предназначены для наследования. Однако класс может наследовать только от одного абстрактного класса. Только абстрактные классы могут иметь абстрактные методы. У абстрактного метода отсутствует тело метода. Абстрактные методы – виртуальные, переопределенные абстрактные методы у классов-потомков будут override. Абстрактные методы могут переопределять virtual и override методы.

```
class Token
{
        public virtual string Name( ) { ... }
}
```

```
abstract class Force: Token
{
        public abstract override string Name( );
}
```

Вопросы к разделу

1. В чем отличие между public, private и protected полями?
2. Как переопределить метод базового класса у класса-потомка?
3. Что такое абстрактный класс?
4. Что такое интерфейс? В чем отличие интерфейса от абстрактного класса?
5. Допустимо ли множественное наследование?

Лабораторная работа

Задания на наследование, определение и использование интерфейсов, абстрактных классов, виртуальные методы. Время, необходимое на выполнение задания 60 мин.

Упражнение 10.1. Создать интерфейс ICipher, который определяет методы поддержки шифрования строк. В интерфейсе объявляются два метода encode() и decode(), которые используются для шифрования и дешифрования строк, соответственно.

Создать класс ACipher, реализующий интерфейс ICipher. Класс шифрует строку посредством сдвига каждого символа на одну «алфавитную» позицию выше. Например, в результате такого сдвига буква А становится буквой Б.

Создать класс BCipher, реализующий интерфейс ICipher. Класс шифрует строку, выполняя замену каждой буквы, стоящей в алфавите на i-й позиции, на букву того же регистра, расположенную в алфавите на i-й позиции с конца алфавита. Например, буква В заменяется на букву Э.

Написать программу, демонстрирующую функционирование классов.

Домашнее задание 10.1 Создать класс Figure для работы с геометрическими фигурами. В качестве полей класса задаются цвет фигуры, состояние «видимое/невидимое». Реализовать операции: передвижение геометрической фигуры по горизонтали, по вертикали, изменение цвета, опрос состояния (видимый/невидимый). Метод вывода на экран должен выводить состояние всех полей объекта.

Создать класс Point (точка) как потомок геометрической фигуры. Создать класс Circle (окружность) как потомок точки. В класс Circle добавить метод, который вычисляет площадь окружности. Создать класс Rectangle (прямоугольник) как потомок точки, реализовать метод вычисления площади прямоугольника.

Точка, окружность, прямоугольник должны поддерживать методы передвижения по горизонтали и вертикали, изменения цвета.

Подумать, какие методы можно объявить в интерфейсе, нужно ли объявлять абстрактный класс, какие методы и поля будут в абстрактном классе, какие методы будут виртуальными, какие перегруженными.

ГЛАВА 11. АГРЕГАЦИИ, ПРОСТРАНСТВА ИМЕН, СБОРКИ И МОДУЛИ

Использование внутренних (internal) классов, методов и данных

Модификаторы доступа определяют возможность доступа к элементам класса, таким как методы и свойства. При разработке класса необходимо явно указывать модификатор доступа у каждого члена класса. Модификаторы доступа:

- **Public** – элементы доступны всюду внутри области видимости;
- **Protected** – элементы доступны внутри класса и у всех потомков класса;
- **Private** – элементы доступны только внутри класса;
- **Internal** – элементы доступны внутри одной сборки Microsoft.NET (одного исполняемого файла или одного .dll файла), из других сборок доступ запрещён;
- **Protected internal** – элементы доступны внутри классов-потомков и во всех классах внутри одной сборки.

Public – слишком открытый доступ, private – слишком закрытый, protected – открыт только для потомков класса. Для создания промежуточного типа доступа был создан внутренний (internal) доступ. Типы доступа public и private – логические, то есть не зависят от физического размещения класса. Для internal классов или элементов размещение определяет область доступа: если класс находится в исполняемом файле, то он доступен для всех классов файла; если класс относится к конкретной сборке, то доступ разрешен только внутри данной сборки. В исполняемом файле может быть несколько сборок, тогда доступ к internal классу из другой сборки будет запрещен.

Когда класс определен как internal, доступ к нему имеют все классы из сборки. Классы и элементы protected internal доступны всей сборке, а также всем потомкам текущего класса.

Ограничения на использование модификаторов доступа:

1. Вне классов и пространств имен нельзя объявлять классы protected и private, эти типы доступа можно использовать только внутри какого-либо класса. Для элементов внутри класса модификатором доступа по

умолчанию будет private. В C# есть возможность внутри одного класса объявлять другой класс.

2. При объявлении типа в глобальной области видимости можно использовать модификаторы доступа public и internal. В глобальной области видимости или в пространстве имен модификатор доступа по умолчанию будет internal.

Использование агрегаций

Агрегации используются для группировки объектов вместе в иерархию объектов, которую можно неоднократно использовать. Агрегации определяют связь целое/часть между объектами, не классами. В одном случае в этой связи время жизни «целого» и «части» могут быть не связаны. В этом случае агрегация называется агрегацией по ссылке. В случае, когда время жизни «целого» и «части» связаны друг с другом, агрегация называется агрегацией по значению. В агрегации «целое» – это всего лишь класс, который используется для группировки «частей»-классов в единое целое, т.е. «целый» класс реально не существует. Например, что такое компьютер? Это всего лишь имя, которое используется для описания конкретных частей: CPU, монитор, клавиатура и т.д. Через это имя можно получить доступ к методам, которые работают с составными частями.

Сравнение агрегаций и наследования:

- Агрегации определяют связь на уровне объектов, наследование – на уровне классов. В случае агрегации у целого объекта может быть несколько объектов частей одного типа. Например, у агрегации компьютер может быть несколько мониторов, несколько процессоров. В случае наследования речь идет не о конкретных объектах, а о классах. В этом случае компьютер и монитор – это просто два разных класса, никак не связанные друг с другом.

- В агрегации при изменении методов «частей» методы «целого» автоматически не меняются. В наследовании же при изменении методов у предков, меняются и потомки.

- В агрегации в объект «целого» можно добавлять и удалять объекты «части» без каких-либо ограничений. В главном объекте сохраняются

ссылки на части, при необходимости они могу изменяться, можно создавать новые, удалять старые. В механизме наследования все статично: класс потомок всегда будет оставаться потомком.

Фабрики классов

Иногда бывает необходимо запретить создание объекта класса напрямую, разрешить создавать объекты только определенным объектам некоторого другого класса. Например, самостоятельно нельзя открыть счет в банке. Чтобы открыть счет, нужно пойти в банк и банковский работник открывает счет. В программировании, пусть есть два класса Bank и BankAccount. Открывать счет в банке – создавать объект класс BankAccount можно только внутри банка, вызвав внутри класса Bank нужный конструктор:

```
public class Bank
{
        public BankAccount OpenAccount( )
        {
                BankAccount opened = new BankAccount( );
                accounts[opened.Number( )] = opened;
                return opened;
        }
        private Hashtable accounts = new Hashtable( );
}
public class BankAccount
{
        internal BankAccount( ) { ... }
        public long Number( ) { ... }
        public void Deposit(decimal amount) { ... }
}
```

В примере для хранения всех открытых в банке счетов внутри класса Bank предусмотрено поле accounts в виде хэш-таблицы. Каждый открытый банковский счет сохраняется в этой хэш-таблице. В данном случае класс Bank будет являться фабрикой. Для того чтобы конструктор класса BankAccount был доступен внутри класса Bank, его объявили с модификатором доступа internal.

Пространства имен

Рассмотрим пример

```
public class Bank
{
        public class Account
        {
                public void Deposit(decimal amount)
                {
                        balance += amount;
                }
                private decimal balance;
        }
        public Account OpenAccount( ) { ... }
}
```

В этом примере четыре области видимости:

- глобальная область – в ней располагается класс Bank;
- область класса Bank – в ней класс Account и метод OpenAccount;
- область класса Account – в ней метод Deposit и поле balance;
- область метода Deposit – в ней объявляется параметр amount.

Если объект или метод находится вне области видимости или его имя скрыто, то вызвать его можно только при помощи полного пути.

```
class Top
{
        public void M(){…}
}
class Bottom:Top
{
        new public void M()
        {
                M();            // рекурсия
                base.M();      //обращение к скрытому методу
        }
```

```
}
```

При обращении к элементам базового класса используется ключевое слово base. Точно так же и для полей класса:

```
public struct Point
{
        public Point(int x, int y)
        {
                this.x = x;
                this.y = y;
        }
        private int x,y;
}
```

В сложных проектах, в которых используется большое количество классов, когда над проектом работает несколько программистов, возможна ситуация, что в одной области видимости окажутся два класса с одинаковым именем. В таком случае следует использовать пространства имен (namespaces). Пространства имен разделяют большой проект на несколько подсистем, в каждой подсистеме – своя глобальная область видимости. Рекомендуется внутри одного пространства имен размещать логически связанные друг с другом элементы.

```
namespace Outer
{
        namespace Inner
        {
                class Widget{…}
        }
}
```

В C# можно записать предыдущий класс короче.

```
namespace Outer.Inner
{
        class Widget{…}
}
```

Для того чтобы использовать класс из другого пространства имен, необходимо использовать полное его имя или подключить соответствующее пространство имен, используя директиву using. Например, для обращения к классу Widget из предыдущего примера из другого пространства имен можно написать

```
Outer.Inner.Widget w; //полный путь к классу Widget
```
или
```
using Outer.Inner; //подключаем пространство имен
Widget w;   //объявляем объект класса Widget без указания полного пути к классу.
```
Директива using должна располагаться перед объявлением каких-либо элементов либо в глобальной области видимости, либо внутри пространства имен. Директива разрешает доступ по имени к классам только данного пространства имен, то есть доступ к классам вложенных пространств имен по короткому имени запрещен.
```
namespace Microsoft.PowerPoint
{
    public class Widget { ... }
}
namespace VendorB
{
    using Microsoft; // но не Microsoft.PowerPoint
    class SpecialWidget: Widget { ... } // ошибка на этапе компиляции
}
```
Кроме того, если подключить при помощи директивы два пространства имен с несколькими классами с одним именем, то при вызове этого класса получим ошибку компиляции.
```
namespace Test
{
    using VendoreA;   // содержит класс Widget
    using VendoreB;   // также содержит класс Widget
    // Здесь ошибки нет

    class Application
    {
        static void Main()
        {
            Widget w = new Widget(); // ошибка при компиляции
        }
    }
}
```

Ошибка возникает, так как компилятор не знает, какой класс нужен. Надо явно прописать необходимое пространство имен, например

VendoreA.Widget w = new VendoreA.Widget();

Директиву using можно использовать для объявления коротких имён классов напрямую.

using Widget = VendoreA.SuiteB.Widget

В этом случае можно сразу обращаться по короткому имени к данному классу. Все ограничения на использование директивы остаются. Директиву using можно использовать в обеих её возможностях одновременно, единственное, что надо помнить, что действие директивы начинается с первого элемента пространства имен в которой она объявлена, то есть друг на друга директивы не действуют, и следующая конструкция даст ошибку:

```
using System;
using TheConsole = Console; //ошибка, хотя Console класс из System
```

Правильно будет:

```
using System;
using TheConsole = System.Console;
```

Модули и сборки

В C# присутствует возможность компилировать исходные .cs файлы в управляемый модуль – промежуточный язык MSIL, который содержит метаданные для описания модуля. Для этого в опциях компилятору надо задать параметр /target:module имя_файла.cs. При выполнении программы управляемый модуль преобразовывается в машинный код. При компиляции исходный файл компилируется в управляемый модуль с расширением .netmodule. Команда командной строки:

csc /target:module Bank.cs

На выходе получим файл Bank.netmodule.

Выполняемый файл может запускать модули только в составе какой-либо сборки. В сборке физически размещается группа взаимодействующих классов. Классы, находящиеся в одной сборке, имеют доступ к internal элементам друг друга. Для классов из других сборок доступ к этим элементам закрыт.

Сборка – это многократно-используемый, безопасный, и самоописываемый модуль, содержащий типы и ресурсы с поддержкой версий;

это первичный стандартный блок приложения .NET. Сборка состоит из двух логических частей:

- наборов типов и ресурсов, которые формируют некоторый логический модуль из функциональных возможностей,
- метаданные, которые описывают, как эти элементы связаны и от чего зависят их правильная работа.

Метаданные, которые описывают сборку, называют манифестом. В манифесте содержится следующая информация:

- Уникальность. Уникальность сборки включает в себя простое текстовое название, номер версии и дополнительный открытый ключ, который гарантирует уникальность названия и защищает от нежелательного использования.
- Содержание. Сборка содержит типы и ресурсы. Манифест перечисляет названия всех типов и ресурсов, которые видимы извне сборки. Содержит информацию о том, где они могут быть найдены во время трансляции.
- Ссылки. Каждая сборка явно описывает другие сборки, от которых она зависит.

В простейшем случае, сборка состоит из одного файла, который содержит код, ресурсы, метаданные и манифест. В более общем случае, сборка состоит из нескольких файлов, тогда сборка существует как автономный файл или содержится в одном из файлов PE, которые содержат типы и ресурсы.

Соответствующие команды компилятора:

csc /target:library /out:Bank.dll Bank.cs Account.cs

csc /t:library /addmodule:Account.netmodule /out:Bank.dll Bank.cs

Каждая сборка имеет свой уникальный номер версии. Две сборки, которые отличаются только номером, для исполняющей среды различны. Номер версии состоит из четырех частей:

старшая версия . младшая версия . номер компиляции . редакция

Например, номер 1.5.12540.0 означает, что старшая версия 1, младшая 5, номер компиляции 12540 и редакция 0.

Пространства имен – это логический механизм компиляции, он обеспечивает структуру имен сущностей исходного кода. При выполнении программы пространства имен не рассматриваются. Сборки – это физический

механизм выполнения, обеспечивающий структуру компонент при исполнении программы. Можно разместить классы одного пространства имен в разных сборках, также в одной сборке могут быть классы разных пространств имен. В сборку можно подключать внешние модули, тогда в сборке записывается именованная ссылка на управляемый модуль.

Вопросы к разделу

1. Пусть есть два cs файла. Файл alpha.cs содержит класс Alpha с internal методом Method. Файл beta.cs содержит класс Beta с internal методом с тем же названием Method. Может ли Alpha.Method вызвать Beta.Method, и наоборот?

2. Агрегации это связь объектов или классов?

3. Откомпилируется ли следующий код без ошибок?

```
namespace Outer.Inner
{
        class Wibble{}
}
namespace Test
{
        using Outer.Inner;
        class SpecialWible:Inner.Wible{}
}
```

Лабораторная работа

Модификатор доступа internal. Задание на агрегацию, пространство имен. Время, необходимое на выполнение задания 60 мин.

Упражнение 11.1 Создать новый класс, который будет являться фабрикой объектов класса банковский счет. Изменить модификатор доступа у конструкторов класса банковский счет на internal. Добавить в фабричный класс перегруженные методы создания счета CreateAccount, которые бы вызывали конструктор класса банковский счет и возвращали номер созданного счета. Использовать хеш-таблицу для хранения всех объектов банковских счетов в фабричном классе. В фабричном классе предусмотреть метод закрытия счета,

который удаляет счет из хеш-таблицы (методу в качестве параметра передается номер банковского счета).

Использовать утилиту ILDASM для просмотра структуры классов.

Упражнение 11.2 Разбить созданные классы, связанные с банковским счетом, и тестовый пример в разные исходные файлы. Разместить классы в одно пространство имен и создать сборку. Подключить сборку к проекту и откомпилировать тестовый пример со сборкой. Получить исполняемый файл, проверить с помощью утилиты ILDASM, что тестовый пример ссылается на сборку и не содержит в себе классов, связанный с банковским счетом.

Домашнее задание 11.1 Для реализованного класса из домашнего задания 7.1 создать новый класс Creator, который будет являться фабрикой объектов класса здания.

Для этого изменить модификатор доступа к конструкторам класса, в новый созданный класс добавить перегруженные фабричные методы CreateBuild для создания объектов класса здания. В классе Creator все методы сделать статическими, конструктор класса сделать закрытым. Для хранения объектов класса здания в классе Creator использовать хеш-таблицу. Предусмотреть возможность удаления объекта здания по его уникальному номеру из хеш-таблицы.

Создать тестовый пример, работающий с созданными классами.

Домашнее задание 11.2 Разбить созданные классы (здания, фабричный класс) и тестовый пример в разные исходные файлы. Разместить классы в одном пространстве имен. Создать сборку (DLL), включающие эти классы. Подключить сборку к проекту и откомпилировать тестовый пример со сборкой. Получить исполняемый файл, проверить с помощью утилиты ILDASM, что тестовый пример ссылается на сборку и не содержит в себе классов здания и Creator.

ГЛАВА 12. ОПЕРАЦИИ, ДЕЛЕГАТЫ, СОБЫТИЯ

Операции

Представим, что надо сложить между собой четыре комплексных числа класса Complex. Пусть в классе реализован статичный метод Add, который складывает два объекта типа Complex и возвращает объект типа Complex. Тогда для того, чтобы сложить четыре комплексных числа придется написать следующую строку:

Var = Complex.Add(Var1, Complex.Add(Complex.Add (Var2, Var3) , Var4);

Если бы для комплексных чисел (в классе Complex) была реализована операция сложения, код выглядел бы гораздо приятнее:

Var =Var1+Var2 +Var3 + Var4;

Операции в C# определяются как обычные методы. Компилятор и среда исполнения автоматически переводят выражения с операциями в соответствующую серию вызовов методов.

В C# есть большое количество предопределенных операций. Для различных типов данных одна и та же операция может реализовывать различные действия. Это свойство называется перегрузкой операций.

Операции следует определять только тогда, когда они упрощают выражения и когда класс или структура хранят какие-либо данные, для которых действие этой операции понятно. То есть, например, для класса Employee семантически не понятно действие оператора инкремента(emp++).

Все операции – это public static методы, их имена задаются по шаблону operatorop, где op – знак операции. Для операции сложения – operator+. Список параметров и их типы должны быть определены. Операции возвращают объект класса.

```
public static Time operator+(Time t1, Time t2)
{
        int newHours = t1.hours + t2.hours;
        int newMinutes = t1.minutes + t2.minutes;
        return new Time(newHours, newMinutes);
}
```

Перегрузка операций сравнения производится попарно: < и >, <= и =>, == и !=. При перегрузке одной операции из пары, обязательно надо определить и вторую. Кроме того, при перегрузке операций равенства и неравенства необходимо определить виртуальный метод Equals, наследуемый от класса Object, чтобы при сравнении объектов с помощью метода Equals не получить результат, противоположный сравнению операцией равенства (==). Для той же цели перегружается ещё один метод наследуемый от Object – GetHashCode, для равных объектов хеш-код также должен быть одинаков.

Для логических операций && и || нет прямой перегрузки, для этого используются побитовые операторы. Перегрузив операторы &, |, true и false, можно определить логическое И и логическое ИЛИ. Пусть x и y переменные типа T, тогда логические операторы определяются следующим образом:

- x && y – будет выражаться через T.false(x) ? x : T.&(x,y), что означает, что если x в терминах класса T является ложью, то результат равен x, иначе в результате получим побитовое И x и y в терминах класса T.

- x || y – выражается через T.true(x) ? x : T.|(x,y), то есть если x в терминах класса T является истинной, то результат равен x, иначе получим побитовое ИЛИ x и y в терминах класса T.

При задании операторов преобразования типов данных необходимо указать явным или неявным образом будет осуществляться преобразование. Для задания явного преобразования используется ключевое слово **explicit**, для неявного – **implicit**:

public static explicit operator Time (int minutes) – явное преобразование типа int в Time;

public static explicit operator Time (float minutes) – явное преобразование типа float к Time;

public static implicit operator int (Time t1) – неявное преобразование типа Time к типу int;

public static explicit operator float (Time t1) – явное преобразование типа Time к типу float;

public static implicit operator string (Time t1) – если для класса предусмотрена перегрузка преобразования в строку, то необходимо аналогично перегрузить метод ToString(), унаследованный от класса Object.

Операции можно перегружать несколько раз в зависимости от параметров:

public static Time operator+ (Time t1, int hours) {…}

public static Time operator+ (Time t1, float hours) {…}

Создание и использование делегатов

Делегаты позволяют вызывать методы неявно, не используя прямое обращение по имени. Фактически, делегаты предназначены для ситуаций, когда нужно передать метод другому методу. Примером одной из таких ситуаций может быть возникновение события. Например, в программировании графического интерфейса – пользователь нажал на кнопку, сгенерировал событие – необходимо предусмотреть обработку этого события. Написать метод и метод каким-то образом связать с этим событием.

Самый простой способ передать метод в качестве параметра:

void Method1() {…} //объявление Method1

MyMethod(Method1); //вызов MyMethod, в качестве параметра, которому передан указатель на Method1

Такой способ возможен для некоторых языков, например для C, C++. С точки зрения идеологии платформы .NET Framework, такой прямой подход вызывает проблемы с безопасностью типов и игнорирует тот факт, что в объектно-ориентированном программировании методы редко существуют в изоляции: для вызова метода обычно должен быть создан объект класса. В связи с этим, в C# такой подход синтаксически не допустим. Вместо этого, когда нужно передать метод, подробную информацию о нем следует поместить в специальную оболочку – делегат. Делегат содержит подробную информацию о методе.

Сначала дается определение делегата, который нужно будет использовать. Его определение означает сообщение компилятору, какого рода метод будет представлять делегат. Затем необходимо создать экземпляр этого делегата. Синтаксис объявления делегатов:

delegate void VoidOpetation(int x);

В данном случае определен делегат, каждый экземпляр которого может содержать ссылку на метод, принимающий один параметр типа int и возвращающий тип void. При определении делегата ему сообщается полная сигнатура метода, который он может представлять. Таким образом, экземпляр делегата может ссылаться на любой метод любого объекта, если сигнатура этого метода совпадает с сигнатурой делегата.

Синтаксис определения делегата подобен определению метода за исключением того, что за таким определением не следует тело метода, и перед определением используется ключевое слово delegate. Объявление делегата можно поместить в любое место, где может находиться объявление класса – то есть внутри какого-либо класса, либо вне любого класса, либо в пространстве имен объекта высшего уровня. Перед делегатом можно использовать любой модификатор доступа.

Пример:

```
private delegate string MyDelegate(int x);
private string MyMethod (int x){
        return x.ToString();
}
static void Main(string[] args)
{
        MyDelegate myD=new MyDelegate(MyMethod); //создаем делегат и связываем с MyMethod
        Console.WriteLine(myD(5)); //строка вызовет метод MyMethod с параметром 5
}
```

События

События позволяют объектам зарегистрироваться на интересующие его изменения в другом объекте. У события есть отправитель – тот, кто генерирует событие, и получатель. Получателем события может выступать любое приложение, объект или компонент, который нуждается в уведомлении, когда что-то происходит.

Отправитель события ничего не знает о том, кто его получает. Где-то внутри получателя есть метод, который отвечает за обработку события. Этот обработчик события должен запускаться всякий раз, когда возникает зарегистрированное для него событие. Как раз здесь и нужны делегаты. Поскольку отправитель не знает, кто будет получателем, между ними не может быть прямых ссылок, нужен посредник. Таким посредником является делегат. Отправитель определяет делегат, который будет использован получателем. Получатель регистрирует обработчик события, привязывает метод обработки к событию.

Пример. В windows приложении разместим на форме кнопку btnOne и создадим для нее обработчик события нажатия на кнопку. Для этого два раза нажмем на кнопку, откроется файл, в котором будет сгенерирован заголовок метода btnOne_Click:

private void btnOne_Click(object sender, EventArgs e){…},

в который необходимо вписать его реализацию. Также автоматически в коде появится строка

btnOne.Click+=new EventHandler(btnOne_Click);

Это означает, что когда произойдет событие Click для кнопки btnOne, то должен быть выполнен метод btnOne_Click. **EventHandler** – это делегат, который использует событие для назначения обработчика (btnOne_Click) событию (Click). Для добавления метода к списку делегатов использовался оператор +=.

Событию можно назначить больше одного обработчика. Однако нет никакой гарантии того, в каком порядке будут вызваны методы, с которыми связан делегат. С разными событиями можно связать один и тот же метод. Допустим, на форме есть кнопка btnTwo, свяжем событие Click этой кнопки с методом btnOne_Click:

btnTwo.Click+=new EventHandler(btnOne_Click);

Делегат EventHandler определен средой .NET и находится в пространстве имен System. Все события, определенные .NET, используют этот делегат. Делегат EventHandler использует два параметра object и EventArgs. Первый параметр – это объект, который сгенерировал событие, в данном примере – это кнопка btnOne или btnTwo. Внутри метода с помощью первого параметра можно узнать, кто именно сгенерировал событие. Второй параметр – это объект, содержащий другую полезную информацию, может быть любого типа, унаследованного от класса EventArgs.

Важно отметить, что обработчики событий всегда возвращают void.

Рассмотрим, как создавать свои собственные события.

Для определения события отправитель объявляет делегат и связывает его с событием.

public delegate void StartPumpCallback(object sender, CoreOverheatingEventArgs args);

private event StartPumpCallback CoreOverheating;

Здесь CoreOverheatingEventArgs – класс, наследник от класса System.EventArgs для передачи дополнительных параметров:

public class CoreOverheatingEventArgs : EventArgs

```
{
    private readonly int temperature;
    public CoreOverheatingEventArgs(int temperature)
    {
        this.temperature = temperature;
    }
    public int GetTemperature()
    {
        return temperature;
    }
}
```

Подписчики определяют метод, который будет вызываться при возникновении события. Если событие еще не создано, то подписчик определяет делегат, ссылающийся на метод при создании события. Если событие уже существует, подписчик добавляет делегат, вызывающий метод при возникновении события.

```
ElectricPumpDriver ed1 = new ElectricPumpDriver();
PneumaticPumpDriver pd1 = new PneumaticPumpDriver();
…
CoreOverheating = new StartPumpCallback(ed1.StartElectricPump); //делегат, который //указывает на метод StartElectricPump объекта ed1, подписывается на событие //CoreOverhating

CoreOverheating += new StartPumpCallback(pd1.SwitchOn); //делегат, который //указывает на метод SwitchOn объекта pd1, также подписывается на событие //CoreOverhating
```

Для уведомления подписчиков необходимо вызывать событие.

```
public void SwitchOnAllPumps()
{
    CoreOverheatingEventArgs e=new CoreOverheatingEventArgs(37);
    if(CoreOverheating != null) //если на событие кто-то подписан, то генерируем //событие

        CoreOverheating(this,e); //событие вызовет два метода //StaertElectricPump и SwithcOn, в которые передаст параметр this и e
```

}

Если событие происходит, то вызываются все делегаты. Заметим, что сначала надо проверить, есть ли хоть один подписчик, так как при отсутствии подписчиков вызов делегата сгенерирует исключение.

Кроме того, что произошло событие, подписчикам бывает интересно, при каких условиях оно произошло. Для этого события могут передавать параметры. Для передачи параметров событиями в C# выделен отдельный класс System.EventsArg.

Вопросы к разделу

1. Может ли арифметическое присваивание (+=, -=, *=, /= и %=) быть перегружено?
2. В каком случае операция преобразования типа должна быть явной?
3. Что такое делегат?
4. Как можно подписаться на событие?
5. Каким образом можно вызвать метод, подписавшийся на событие?

Лабораторная работа

Задания на определение операторов сложения, умножения, вычитания, деления, равенства, переопределение методов Equals(), ToString(), GetHashCode(); публикацию событий, передачу параметров событиям. Время, необходимое на выполнение задания 60 мин.

Упражнение 12.1 Для класса банковский счет переопределить операторы == и != для сравнения информации в двух счетах. Переопределить метод Equals аналогично оператору ==, не забыть переопределить метод GetHashCode(). Переопределить метод ToString() для печати информации о счете. Протестировать функционирование переопределенных методов и операторов на простом примере.

Упражнение 12.2 Создать класс рациональных чисел. В классе два поля – числитель и знаменатель. Предусмотреть конструктор. Определить операторы ==, != (метод Equals()), <, >, <=, >=, +, – , ++, --. Переопределить метод ToString() для вывода дроби.

Если останется время, то определить операторы преобразования типов между типом дробь, float, int. Определить операторы *, /, %.

Домашнее задание 12.1 На перегрузку операторов. Описать класс комплексных чисел. Реализовать операцию сложения, умножения, вычитания, проверку на равенство двух комплексных чисел. Переопределить метод ToString() для комплексного числа. Протестировать на простом примере.

Домашнее задание 12.2 Написать делегат, с помощью которого реализуется сортировка книг.

Книга представляет собой класс с полями Название, Автор, Издательство и конструктором.

Создать класс, являющийся контейнером для множества книг (массив книг). В этом классе предусмотреть метод сортировки книг. Критерий сортировки определяется экземпляром делегата, который передается методу в качестве параметра.

Каждый экземпляр делегата должен сравнивать книги по какому-то одному полю: названию, автору, издательству.

ГЛАВА 13. СВОЙСТВА И ИНДЕКСАТОРЫ

Свойства

Свойства – это наборы функций, которые могут быть доступны клиенту таким же способом, как открытые поля класса. Идея свойства заключается в методе или паре методов, которые ведут себя с точки зрения клиентского кода как поле. Чтобы определить свойство в C#, используется следующий синтаксис:

```
public string MyProperty
{
        get
        {
                return MyField; //возвращает значение поля MyField
        }
        set
        {
                MyField=value; //задаем значение поля MyField
        }
}
```

Средство доступа **get** не принимает никаких параметров и должно возвращать значение того типа, который объявлен для свойства. Для средства **set** в явном виде не передается никаких параметров, но компилятор предполагает, что оно принимает один параметр со значением value, относящийся к типу, указанному для свойства.

Внутри метода get и set можно написать дополнительный код, осуществляющий какую-либо обработку, допустим внутри set поместить код для проверки корректности введенных данных.

Свойство описывается как член класса, предоставляет доступ к полям класса. В соответствии с идеологией ООП имеет смысл поля делать private, а свойства более доступными: public, protected, internal (в зависимости от целей). Также рекомендуется, имена полей начинать с маленькой буквы, а имя свойства, предоставляющего доступ к полю, задавать таким же, как и у поля, но начинать с заглавной буквы.

Существует возможность создать свойство, доступное только для чтения. Для этого надо исключить set из определения свойства. Чтобы определить свойство, доступное только для записи, необходимо из его определения исключить get. Кроме того, для set и get можно применять различные модификаторы доступа, но надо помнить, что модификатор доступа одного из средств get или set должен совпадать с модификатором доступа всего свойства.

Индексаторы

Индексация позволяет индексировать объекты таким же способом, как массив или коллекцию, позволяет обращаться к членам класса как к массиву.

Пусть объект состоит из нескольких подэлементов (например, listbox состоит из нескольких строк). Индексация позволяет обращаться к подэлементам, как в массивах.

```
class StringList
{
        private string[] list;
        public string this[int index]
        {
                get{ return list[index];}
                set{ list[index] = value;}
        }
}
```

Индексатор – это свойство с именем this, квадратными скобками и индексом в них. Для использования индексатора, обращаемся к объекту:

```
StringList myList = new StringList();
myList[3] = “ok”;            // индексатор записывает
string myString = myList[3];    // индексатор читает
```

Индексаторы могут быть статичными элементами класса и их можно перегружать для различных типов индексов. Индексаторы не хранят значения и поэтому не могут быть переданы как ref и out параметры.

При объявлении индексаторов необходимо определить, по крайней мере, один параметр, определить значения для всех параметров. Можно определить несколько параметров для одного индексатора:

```
class MultipleParameters
{
        public string this[int one, int two]
        {
                get{…}
                set{…}
        }
}
…
MultipleParameters mp = new MultipleParameters();
string s = mp[2,3];
…
```

Вопросы к разделу

1. Каким образом можно объявить свойство только для чтения?
2. Может ли класс содержать индексатор, зависящий от двух параметров разного типа?
3. Как в классе объявить индексатор только для чтения?

Лабораторная работа

Задания на свойства, создание и использование индексаторов. Время, необходимое на выполнение задания 30 мин.

Упражнение 13.1 Из класса банковский счет удалить методы, возвращающие значения полей номер счета и тип счета, заменить эти методы на свойства только для чтения. Добавить свойство для чтения и записи типа string для отображения поля держатель банковского счета.

Изменить класс BankTransaction, созданный для хранений финансовых операций со счетом, - заменить методы доступа к данным на свойства для чтения.

Упражнение 13.2 Добавить индексатор в класс банковский счет для получения доступа к любому объекту BankTransaction.

Домашнее задание 13.1 В классе здания из домашнего задания 7.1 все методы для заполнения и получения значений полей заменить на свойства. Написать тестовый пример.

Домашнее задание 13.2 Создать класс для нескольких зданий. Поле класса – массив на 10 зданий. В классе создать индексатор, возвращающий здание по его номеру.

ГЛАВА 14. АТРИБУТЫ

Атрибуты – инструмент для добавления метаданных в классы, для изменения поведения объекта во время выполнения, для получения информации о транзакции объекта, также используется для передачи дополнительной информации, носящей описательный характер, разработчику. Атрибут – тэг для предоставления информации во время выполнения о поведении программных элементов, таких как классы, перечисления, сборки.

Информация об атрибутах хранится в метаданных, связанных с элементом. Атрибуты можно применять к разным программным элементам: сборкам, модулям, классам, структурам, перечислениям, конструкторам, методам, свойствам, полям, событиям, интерфейсам, параметрам, возвращаемым значениям и делегатам.

Синтаксис:

[атрибут (позиционные_параметры, именованные_параметры=значение, …)]

элемент, к которому применяется атрибут.

Сначала задается имя атрибута, затем в скобках передаются позиционные параметры, затем – именованные. Позиционные параметры используются для передачи важной информации, именованные – для передачи дополнительных сведений.

Для одного элемента можно задать несколько атрибутов в разных квадратных скобках [ ], или через запятую в одних квадратных скобках.

В .NET Framework определено большое количество атрибутов. Рассмотрим некоторые из них.

Условный атрибут используется при отладке кода. Вызывает условную компиляцию метода, для которого этот атрибут задан.

```
using System.Diagnostics;
class MyClass
{
    [Conditional(“DEBUGGING”)] //задаем условный атрибут для
    метода //MyMethod
    public static void MyMethod() { … }
}
```

Где идентификатор DEBUGGING определяется следующим образом:

```
#define DEBUGGING
Class AnotherClass
{
        public static void Test()
        {
                MyClass.MyMethod();
        }
}
```

Если параметр компиляции не задан, не задана строка #define DEBUGGING или соответствующая строка не указана в опции компилятора, то вызов MyMethod будет пропущен.

Метод, к которому применяется условный атрибут, должен возвращать void, не может быть override и не должен быть методом, определяющим интерфейс.

Атрибут DLLImport используется для вызова неуправляемого кода в программе C#. Неуправляемый код – это термин для кода разработанного вне среды .NET(например, программа на стандартном C откомпилированная в файл dll). При помощи этого атрибута можно вызывать внешние методы из неуправляемых dll. При вызове внешнего метода, общая среда выполнения определяет DLL, загружает её в память процесса, преобразует параметры, если это необходимо, и передает управление адресу начала неуправляемого кода.

```
using System.Runtime.InteropService;
…
public class MyClass()
{
        [DLLImport(“MyDll.dll”, EntryPoint=”MyFunction”)]
        public static extern int MyFunction(string param1);
        …
        int result = MyFunction(“Привет из неуправляемого кода”);
        …
}
```

У атрибута DLLImport два параметра: первый параметр – имя DLL библиотеки, второй – имя внешней функции.

Пользовательские атрибуты

В .NET можно создать свои собственные атрибуты. Также как и предопределенные атрибуты, они связаны с каким-то элементом программы, хранятся в их метаданных и обеспечены механизмом получения значения атрибута.

Для создания собственного атрибута создается пользовательский класс, наследник от класса System.Attribute. Перед классом указывается атрибут AttributeUsage, который определяет, к какому элементу класса будет привязан пользовательский атрибут. Если элементов несколько, то используется оператор | для их связи:

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Constructor)]
public class MyAttribute: System.Attribute
{
    …
}
```

Все классы атрибутов должны иметь конструктор, который использует позиционные параметры. Для работы с именованными параметрами задаются перегруженные конструкторы. Класс атрибута может содержать свойства для задания именованных параметров.

Например, атрибут DeveloperInfo содержит позиционный строковый параметр и именованный Date.

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Constructor)]
public class DeveloperInfoAttribute:System.Attribute
{
    public DeveloperInfoAttribute(string developer)
    {  …  }
    public string Date
{
        get { … }
        set { … }
    }
}

[DeveloperInfoAttribute(“Bert”, Date = “08-28-2009” )]
public class MyClass
```

```
{
    …
}
```

Во время компиляции при встрече атрибута происходит следующая последовательность действий:

1. Поиск класса атрибута.
2. Проверка области атрибута.
3. Проверка конструктора.
4. Создание объекта.
5. Проверка именованных параметров.
6. Установка значений именованных параметров.
7. Сохранение состояния объекта атрибута в метаданных связанного элемента.

Для того чтобы для одного элемента можно было задать несколько атрибутов, необходимо при объявлении класса атрибута установить параметр AllowMultiple=true:

```
[AttributeUsage(AttributeTergets.Class, AllowMultiple = true)]
```

Для получения информации, хранящейся в метаданных, среда .NET Framework использует механизм, называющийся рефлексией. Пространство имен System.Reflection содержит классы для извлечения метаданных. Класс **MemberInfo** используется для получения информации об атрибутах компонента и обеспечивает доступ к метаданным его членов. Для заполнения массива MemberInfo используется метод **GetMembers** объекта System.Type.

```
System.Reflection.MemberInfo[] memberInfoArray;
memberInfoArray = typeof(MyClass).GetMembers();
```

Для получения информации о пользовательских атрибутах есть специальный метод GetCustomAttributes объекта MemberInfo.

```
System.Reflection.MemberInfo typeInfo;
typeInfo = typeof(MyClass);
object[] attrs = typeInfo.GetCustomAttributes(false);
```

После получения массива атрибутов, можно проверить их значения.

```
foreach(Attribute atr in attrs){
    if(atr is DeveloperInfoAttribute){
        DeveloperInfoAttribute dia = (DeveloperInfoAttribute) atr;
        Console.WriteLine("{0} {1}", dia.Developer, dia.Date);
    }
}
```

Если нет ни одного атрибута для класса, то метод возвращает null. Кроме того, можно проверить наличие атрибута при помощи метода IsDefined.

```
Type devInfoAttrType = typeof(DeveloperInfoAttribute);
if(typeInfo.IsDefined(devInfoAttrType, false))
    object[] attrs = typeInfo.GetCustomAttributes(devInfoAttrType,false);
```

Вопросы к разделу

1. Можно ли отметить один объект класса используя атрибут?
2. Где хранятся значения атрибутов?
3. Какой механизм используется для определения значения атрибутов при выполнении приложения?

Лабораторная работа

Задания на использование условного атрибута(ConditionalAttribute), создание пользовательского атрибута. Время, необходимое на выполнение задания 45 мин.

Упражнение 14.1 Использование предопределенного условного атрибута для условного выполнения кода (указывает компиляторам, что при отсутствии символа условной компиляции, вызов метода или атрибут следует игнорировать).

В классе банковский счет добавить метод DumpToScreen, который отображает детали банковского счета. Для выполнения этого метода использовать условный атрибут, зависящий от символа условной компиляции DEBUG_ACCOUNT. Протестировать метод DumpToScreen.

Упражнение 14.2 Создать пользовательский атрибут DeveloperInfoAttribute. Этот атрибут позволяет хранить в метаданных класса имя разработчика и, дополнительно, дату разработки класса. Атрибут должен

позволять многократное использование. Использовать этот атрибут для записи имени разработчика класса рациональные числа (упражнение 12.2).

Домашнее задание 14.1 Создать пользовательский атрибут для класса из домашнего задания 13.1. Атрибут позволяет хранить в метаданных класса имя разработчика и название организации. Протестировать.

Учебное издание

Александрова Ирина Леонидовна
Тумаков Дмитрий Николаевич

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C#

Учебное пособие

Подписано в печать 14.11.2017 г.
Форм. бум. 60 × 84 1/16. Гарнитура "Таймс". Печать цифровая.
Печ. л. 8,25. Т.100. Заказ 12.

Лаборатория оперативной полиграфии Издательства КГУ 420008,
Казань, ул. пр Нужина, 1/37,
+7 (843) 233-73-28, +7 (843) 233-73-59